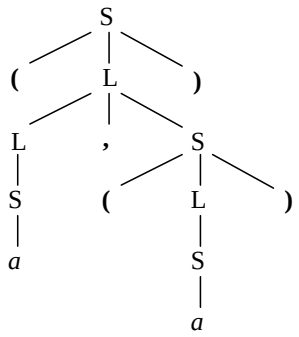
一．给定文法 $G[S]$：

$$S \to （L） \mid a$$
$$L \to L\,,S \mid S$$

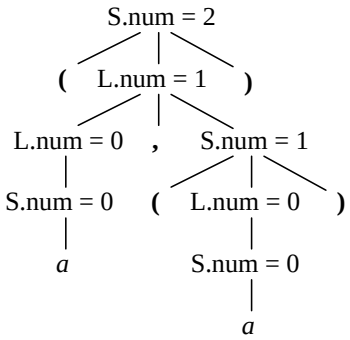
如下是相应于 $G[S]$ 的一个 S-属性文法：

$S \to （L）$　　{ $S.num := L.num + 1$; }
$S \to a$　　{ $S.num := 0$; }
$L \to L_1\,,S$　　{ L.num := L_1.num + S.num; }
$L \to S$　　{ L.num := S.num; }

1. 输入串（a，(a)）的语法分析树如下左图所示。请对该语法分析树进行标注，给出带标注的语法树。



参考解答：



2. 采用 LR 分析技术进行 S-属性文法的语义计算，可以在自底向上分析过程中，增加语义栈来计算属性值。图 1 是 G[S] 的一个 LR 分析表，图 2 描述了输入串（a，(a)）的分析和求值过程（语义栈中的值对应 S.num 或 L.num），其中，第 14，15 行没有给出，试补齐之。

状态	ACTION					GOTO	
	a	,	(	)	#	S	L
0	s_3		s_2			1	
1					acc		
2	s_3		s_2			5	4
3		r_2		r_2	r_2		
4		s_7		s_6			
5		r_4		r_4			
6		r_1		r_1	r_1		
7	s_3		s_2			8	
8		r_3		r_3			

图 1

步骤	状态栈	语义栈	符号栈	余留符号串
1）	0	–	#	(*a* , (*a*)) #
2）	02	– –	# (	*a* , (*a*)) #
3）	023	– – –	# (*a*	, (*a*)) #
4）	025	– – 0	# (S	, (*a*)) #
5）	024	– – 0	# (L	, (*a*)) #
6）	0247	– – 0 –	# (L ,	(*a*)) #
7）	02472	– – 0 – –	# (L , (	*a*)) #
8）	024723	– – 0 – – –	# (L , (*a*	)) #
9）	024725	– – 0 – – 0	# (L , (S	)) #
10）	024724	– – 0 – – 0	# (L , (L	)) #
11）	0247246	– – 0 – – 0 –	# (L , (L)	) #
12）	02478	– – 0 – 1	# (L , S	) #
13）	024	– – 1	# (L	) #
14）				
15）				
16）	接受			

图 2

参考解答：

14）	0246	– – 1 –	# (L)	#
15）	01	– 2	# S	#

二．以下是一个 S-翻译模式，其基础文法为 SLR(1)文法（开始符号为 S）：

$S \to E$ { *print*(*E.val*) }

$E \to E_1 + T$ { $E.val := E_1.val + T.val$ }

$E \to T$ { $E.val := T.val$ }

$T \to T_1 * F$ { $T.val := T_1.val \times F.val$ }

$T \to F$ { $T.val := F.val$ }

$F \to (\ E\ )$ { $F.val := E.val$ }

$F \to d$ { $F.val := d.lexval$ }

其中，$d.lexval$ 是由词法分析程序所确定的属性；$F.val$，$T.val$ 和 $E.val$ 都是综合属性；语义函数 $print(E.val)$ 用于显示 $E.val$ 的结果值。不难理解，这个属性文法描述了一个基于简单表达式文法进行算术表达式求值的语义计算模型。

由于是 S-翻译模式，所以在 LR 分析过程中根据该翻译模式进行自底向上语义计算时，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综合属性进行访问。试补全每个产生式归约时语义计算的代码片断（设语义栈由向量 v 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，可用 v[i].val 访问，不用考虑对 top 的维护）。

$S \to E$ { $print$(v[top]. val) }

$E \to E_1 + T$ { }

$E \to T$ { v[top]. val := v[top]. val }

$T \to T_1 * F$ { }

$T \to F$ { v[top]. val := v[top]. val }

$F \to (\ E\ )$ { }

$F \to d$ { }

参考解答：

$S \to E$ { $print$(v[top]. val) }

$E \to E_1 + T$ { v[top-2]. val := v[top-2]. val + v[top]. val }

$E \to T$ { v[top]. val := v[top]. val }

$T \to T_1 * F$ { v[top-2]. val := v[top-2]. val × v[top]. val }

$T \to F$ { v[top]. val := v[top]. val }

$F \to (\ E\ )$ { v[top-2]. val := v[top-1]. val }

$F \to d$ { v[top]. val := $d.lexval$ }

三．给定 LL(1)文法 $G[S]$：

$S \to A\ b\ B$

$A \to a\ A \mid \varepsilon$

$B \to a\ B \mid b\ B \mid \varepsilon$

如下是以 $G[S]$ 作为基础文法设计的一个 L 翻译模式：

$S \to A\ b$ {$B.in\_num := A.num$} B { $if\ B.num=0\ then$ $print(\text{“}Accepted!\text{”})$ else $print(\text{“}Refused!\text{”})$ }

$A \to aA_1$ {$A.num := A_1.num + 1$}

$A \to \varepsilon$ {$A.num := 0$}

$B \to a$ $\{B_1.in\_num := B.in\_num\}$ B_1 $\{B.\ num := B_1.num - 1\}$

$B \to b$ $\{B_1.in\_num := B.in\_num\}$ B_1 $\{B.\ num := B_1.num\ \}$

$B \to \varepsilon$ $\{B.\ num := B.in\_num\ \}$

针对该翻译模式构造相应的递归下降（预测）翻译程序。该翻译程序中 ①~⑥ 的部分未给出，试填写之。

```
void ParseS( )                          // 主函数
{
    A_num := ParseA();
    MatchToken('b');
    B_in_num := A_num;
    ____①____;
    if B_num =0   then   print("Accepted!")
                  else   print("Refused!");
}

int ParseA( )
{
    switch (lookahead)  {                   // lookahead为下一个输入符号
        case ′a′ :
            MatchToken('a');
            A1_num := ParseA();
            ____②____;
            break;
        case  ___③___ :
            A_num := 0;
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
    ____④____;
}

int ParseB( int in_num )
{
    switch (lookahead)  {
        case ′a′ :
            MatchToken('a');
            ____⑤____;
            B1_num := ParseB(B1_in_num);
            B_num := B1_num-1;
            break;
```

```
            case ′ b′ :
                MatchToken('b');
                B1_in_num := in_num;
                B1_num := ParseB(B1_in_num);
                B_num := B1_num;
                break;
            case ′ #′ :
                ⑥ ;
                break;
            default:
                printf("syntax error \n")
                exit(0);
        }
        return B_num;
    }
```

参考答案：

```
    void ParseS( )                          //  主函数
    {
        A_num := ParseA();                  //变量  A_num对应属性A.num
        MatchToken('b');
        B_in_num := A_num;                  //变量  B_in_num  对应属性B.in_num
        B_num := ParseB(B_in_num);                  //变量  B_num对应属性B.num
        if B_num =0    then    print("Accepted!")
                       else    print("Refused!");
    }

    int ParseA( )
    {
        switch (lookahead)   {                  // lookahead为下一个输入符号
            case ′ a′ :
                MatchToken('a');
                A1_num := ParseA();
                A_num := A1_num+1;
                break;
            case ′ b′ :
                A_num := 0;
                break;
            default:
                printf("syntax error \n")
                exit(0);
        }
        return A_num;
    }
```

```
int ParseB( int in_num )
{
    switch (lookahead)  {
        case ′ a′ :
            MatchToken('a');
            B1_in_num := in_num;          //变量 B1_in_num 对应属性B1.in_num
            B1_num := ParseB(B1_in_num);
            B_num := B1_num-1;
            break;
        case ′ b′ :
            MatchToken('b');
            B1_in_num := in_num;
            B1_num := ParseB(B1_in_num);
            B_num := B1_num;
            break;
        case ′ #′ :
            B_num := in_num;
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
    return B_num;
}
```

四．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. 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）

参考答案：

5

2. 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）$M \to \varepsilon$ { 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）

参考答案：

（1）$S \to M$ { $P.i := M$.s } P { 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 }
（6）$M \to \varepsilon$ { M.s := 0 }

3. 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）。

参考答案：

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

五.

1. 以下是一个 S-翻译模式片断，描述了某小语言相关的类型检查工作片断：

...
$T \to T_1$ [$\underline{int}$] { $T.type := if\ \underline{int}.lexval > 0\ then\ array(T_1.type)\ else$ $type\_error$ }
$S \to if\ (\ E\ )\ S1\ S2$ { $S.type := if\ E.type = int\ and\ S1.type = ok\ and\ S2.type = ok\ then\ ok$

else type_error }

$S \rightarrow$ *while (E) S1 { S.type := if E.type= int then S1.type else type_error }*

$S \rightarrow$ *S1 S2 { S.type := if S1.type = ok and S2.type = ok then ok else type_error }*

$S \rightarrow \ldots$

...

$E \rightarrow E_1\,[E_2]$ { *E.type* := if E_2.*type*= *int* and E_1.*type*= *array*(τ) then τ else *type_error* }

...

其中，type 属性以及类型表达式 *ok*，*type_error*，*int*，*array*(τ) 等的含义与讲稿中一致。

（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关于“数组迭代器”语句的产生式

foreach id in E_1 invariant E_2 do S_1

语句 foreach id in E_1 invariant E_2 do S_1 中，foreach、in、invariant 和 do 为保留字，**表达式 E_1 必须为数组类型，变量 id 的类型必须与数组元素的类型一致；同时 E_2 必须为 int类型的表达式，S_1 必须为合法的语句。**为实现这一类型检查任务，我们在上述翻译模式片段基础上，增加以下语义处理片断：

$S \rightarrow$ foreach id in E_1 invariant E_2 do S_1

{ *S.type* := if ＿＿＿①＿＿＿ and

＿＿＿②＿＿＿ and

＿＿＿③＿＿＿

then S_1.*type* else *type_error* }

试在 ①、② 和 ③ 处补充填写适当的内容。

参考解答：

$S \rightarrow$ foreach id in E_1 invariant E_2 do S_1

{*S.type* := if E_1.*type*=*array*(τ) and

lookup_type (id.*entry*) = τ and

E_2.*type* = *int*

then S_1.*type* else *type_error* }

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：

$S \rightarrow$ *if ({ E.true := newlabel; E.false := newlabel } E) { S_1.next := S.next } S_1*

{ S_2.next := S.next } S_2{ S.code := E.code || gen(E.true ‘:’) || S_1.code ||

gen(‘goto’ S.next) || gen(E.false ‘:’) ||S_2.code}

$S \rightarrow$ *while* ({ *E.true := newlabel;E.false := S.next* } *E*) { S_1*.next* := *newlabel* } 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 := newlabel;* S_1*.break := S.break* } S_1

{ S_2*.next := S.next;* S_2*.break := S.break* } S_2

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

$S \rightarrow \ldots$

...

$E \rightarrow E_1 [E_2]$ { *E.place* := *newtemp*; *E.code* := E_1*.code* || E_2*.code* || *gen* (*E.place* ':=' E_1*.place* '[' E_2*.place* ']') }

$E \rightarrow$ int { *E.place* := *newtemp*; *E.code* := *gen* (*E.place* ':=' int *.val*) }

...

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：继承属性 *E.true* 和 *E.false* 分别代表 *E* 为真和假时控制要转移到的程序位置，即标号；综合属性 *E.code* 分别表示对*E* 进行求值的 *TAC* 语句序列；综合属性 *S.code* 表示对应于 *S* 的 *TAC* 语句序列；继承属性 *S.next*代表退出 *S* 时控制要转移到的语句标号。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关于"数组迭代器"语句的产生式

$$S \rightarrow \text{foreach } \underline{\text{id}} \text{ in } E_1 \text{ invariant } E_2 \text{ do } S_1$$

如第1小题中所述，foreach、in、invariant 和 do 为保留字，表达式 E_1 必须为一维数组类型，变量 id 的类型必须与数组元素的类型一致；同时 E_2 必须为 int 类型的表达式，S_1 必须为合法的语句。

语句 foreach id in E_1 invariant E_2 do S_1 的执行语义为：对数组 $\boldsymbol{E_1}$，从下标为1的第一个元素开始，将其每一个元素分别赋值给变量 id，依次执行语句 S_1 一次，但执行之前先要判断由 $\boldsymbol{E_2}$ 描述的条件是否成立，成立则执行，不成立则退出这个"数组迭代器"语句的执行。为产生相应的 *TAC* 语句序列，我们在上述 *L*-翻译模式片段基础上，增加针对"数组迭代器"语句的语义处理内容如下：

$S \rightarrow$ foreach id in E_1 invariant
{ E_2*.true* := ① ; E_2*.false* := ② } E_2 do
{ S_1*.next* := ③ } S_1
{ *p* := *newtemp*;
n := *getArrayLen*(E_1); //获得数组 E_1 的大小保存至 *n*
S.code := *gen*(*p*':='1) ||

gen(S_1.*next* ':') || *gen* ('if' *p* '>' *n* 'goto' *S.next*) ||
E_1.*code* || *gen*(*id* .*place* ':=' E_1.*place* '[' ④ ']') ||
⑤ || *gen*(E_2.*true* ':') || ⑥ ||
gen (*p* ':=' *p* '+' 1) || *gen*('*goto*' S_1.*next*) }

试在 ①、②、③、④、⑤ 和 ⑥ 处补充填写适当的内容。

注：假设已经通过了静态语义检查（第1小题中已完成），并且略去了相关语义属性的计算（如 type）；重新设计翻译逻辑也是允许的，但必要时应给出适当的解释。

参考解答：

$S \rightarrow$ foreach id in E_1 invariant
{ E_2.*true* := *newlabel*; E_2.*false* := *S.next* } E_2 do
{ S_1.*next* := *newlabel* } S_1
{ *p* := *newtemp*;
n := *getArrayLen*(E_1); //获得数组 E_1 的大小保存至 *n*
S.code := *gen*(*p*':=' *0)* ||
gen(S_1.*next* ':') || *gen* (*p* ':=' *p* '+' 1) || *gen* ('if' *p* '>' *n* 'goto' *S.next*) ||
E_1.*code* || *gen*(*id* .*place* ':=' E_1.*place* '[' *p* ']') ||
E_2.*code* || *gen*(E_2.*true* ':') || S_1.*code* ||
gen('*goto*' S_1.*next*) }

`