（若发现问题，请及时告知）

1　下面的文法 $G[S']$ 描述由布尔常量 *false*、*true* ，联结词 ∧（合取）、∨（析取）、¬（否定）构成的不含括号的二值布尔表达式的集合：

$S' \to S$
$S \to S \vee T \mid T$
$T \to T \wedge F \mid F$
$F \to \neg F \mid$ *false* $\mid$ *true*

试设计一个基于 $G[S']$ 的属性文法，它可以计算出每个二值布尔表达式的取值。如对于句子 ¬ *true* ∨ ¬ *false* ∧ *true*，输出是 *true*。

参考解答：

$S' \to S$	{ print S.val}
$S^1 \to S^2 \vee T$	{ if (S^2.val= *true* ∨T.val= *true*)then S^1.val:= *true* else S^1.val:= *false* }
$S \to T$	{S.val:= T.val }
$T^1 \to T^2 \wedge F$	{ if (T^2.val= *false* ∧ F.val= *false*) then T^1.val:= *false* else T^1.val:= *true* }
$T \to F$	{T.val:= F.val }
$F^1 \to \neg F^2$	{ if (F^2.val= *false* then F^1.val:= *true* else F^1.val:= *false* }
$F \to$ *false*	{F.val:= *false* }
$F \to$ *true*	{F.val:= *true* }

其中，|| 表示逻辑“或”运算，= 表示关系运算“相等”，其余程序符号不多解释。

2　给定文法 $G[S]$：

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

如下是相应于 $G[S]$ 的一个属性文法（或翻译模式）：

$S \to$（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; }

图 18 分别是输入串（a，(a) ）的语法分析树和对应的带标注语法树，但后者的属性值没有标出，试将其标出（即填写右下图中符号 “=” 右边的值）。

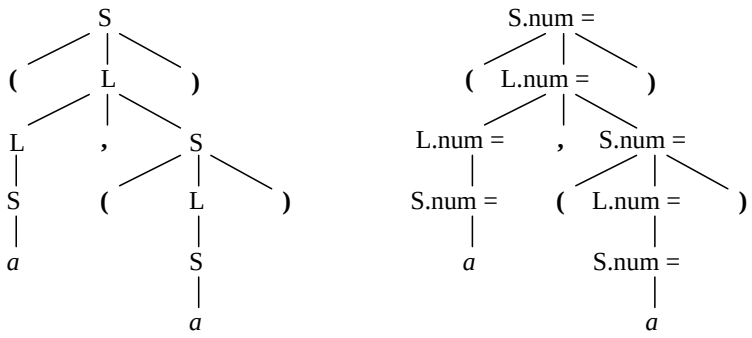
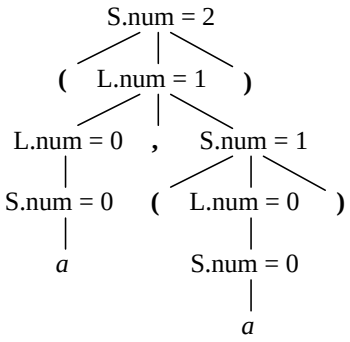


图 18　题 2 的语法分析树和带标注语法树

参考解答：



4.　题 2 中所给的 *G*[*S*] 的属性文法是一个 *S*-属性文法，故可以在自底向上分析过程中，增加语义栈来计算属性值。图 19 是 *G*[*S*] 的一个 *LR* 分析表，图 20 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 19　题 4 的 LR 分析表

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 20　题 4 的分析和求值过程

参考解答：

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

5. 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翻译模式：

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

$A \to aA_1$　{*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* }

试针对该翻译模式构造相应的递归下降（预测）翻译程序（参考例 9）。（可直接使用 2.3 中的 MatchToken 函数）

参考解答：

```
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;
    }
```

8. 给定文法 $G[S]$：

$S \to M\ A\ b\ B$
$A \to A\ a \mid \varepsilon$
$B \to B\ a \mid B\ b \mid \varepsilon$
$M \to \varepsilon$

在文法 $G[S]$ 基础上设计如下翻译模式：

$S \to M$ { $A.in\_num := M.num$ }
$A\ b$ { $B.in\_num := A.num$ }
B { *if* $B.num=0$ *then* $S.accepted := true$ else $S.accepted := false$ }
$A \to$ { $A_1.in\_num := A.in\_num$ } $A_1\ a$ { $A.num := A_1.num - 1$ }
$A \to \varepsilon$ { $A.num := A.in\_num$ }
$B \to$ { $B_1.in\_num := B.in\_num$ } $B_1\ a$ { $B.num := B_1.num - 1$ }
$B \to$ { $B_1.in\_num := B.in\_num$ } $B_1\ b$ { $B.num := B_1.num$ }
$B \to \varepsilon$ { $B.num := B.in\_num$ }
$M \to \varepsilon$ { $M.num :=$ 100 }

不难看出，嵌在产生式中间的语义动作集中仅含复写规则，并且在自底向上的语法分析过程中，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综合属性唯一确定地进行访问。试写出在按每个产生式归约时语义计算的一个代码片断（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].num 或 v[i]. $accepted$ 访问，不用考虑对 top 的维护）。

参考解答：

结果之一：

$S \to MAbB$ { *if* v[top].$num=0$ *then* v[top-3]. $accepted := true$)
else v[top-3]. $accepted := false$) }
$A \to A_1\ a$ { v[top-1] . num := v[top-1] . num - 1}
$A \to \varepsilon$ { v[top+1] . num := v[top].num }
$B \to B_1\ a$ { v[top-1] . num := v[top-1] . num - 1}
$B \to B_1\ b$ { v[top-1] . num := v[top-1] . num }
$B \to \varepsilon$ { v[top+1] . num := v[top-1] . num }
$M \to \varepsilon$ { v[top+1] . num := 100 }

11 如下翻译模式，其基础文法是 $G[S]$：

$S \to A\ b$ {*B.in_num := A .num* + 100}
　　B { *if B.num=0 then S.accepted := true*
　　　　else S.accepted := false }
$S \to A\ b\ b$ {*B.in_num := A .num* + 50}
　　B { *if B.num=0 then S.accepted := true*
　　　　else S.accepted := false }
$A \to A_1\ a$ {*A. num* := A_1.*num* + 1}
$A \to \varepsilon$ {*A. num* := 0}
$B \to$ {B_1.*in_num* := *B.in_num*} $B_1\ a$ {*B. num* := B_1.*num* - 1}
$B \to \varepsilon$ {*B. num* := *B.in_num* }

（*a*）变换该翻译模式，使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。

（*b*）如果在 LR 分析过程中根据（*a*）所得到的新翻译模式进行自底向上的语义计算，试写出在按每个产生式归约时语义计算的一个代码片断（假设语义栈由向量 *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 *x* 都只对应一个语义值，可用 *v*[i].*x* 访问，不用考虑对 *top* 的维护）。

参考解答：

（*a*）结果之一：

$S \to A\ b$ {*M.i := A .num*} M {*B.in_num := M.s* }
　　B { *if B.num=0 then S.accepted := true*
　　　　else S.accepted := false }
$S \to A\ b\ b$ { *N.i := A .num* } N {*B.in_num := N.s* }
　　B { *if B.num=0 then S.accepted := true*
　　　　else S.accepted := false }
$A \to A_1\ a$ {*A. num* := A_1.*num* + 1}
$A \to \varepsilon$ {*A. num* := 0}
$B \to$ {B_1.*in_num* := *B.in_num*} $B_1\ a$ {*B. num* := B_1.*num* - 1}
$B \to \varepsilon$ {*B. num* := *B.in_num* }
$M \to \varepsilon$ { *M.s := M.i* + 100 }
$N \to \varepsilon$ { *N.s := N.i* + 50}

（*b*）结果之一：

$S \to A\ b\ M\ B$ { *if* v[top].*num =0 then* v[top-3].*accepted := true*
　　　　else v[top-3].*accepted := false* }
$S \to A\ b\ b\ N\ B$ { *if* v[top].*num =0 then* v[top-4].*accepted := true*
　　　　else v[top-4].*accepted := false* }
$A \to A_1\ a$ { v[top-1].*num* := v[top-1].*num* + 1}
$A \to \varepsilon$ { v[top+1].*num* := 0}

$B \rightarrow B_1\, a$ { v[top-1].*num* := v[top-1].*num* - 1}

$B \rightarrow \varepsilon$ { v[top+1].*num* := v[top].*s* }

$M \rightarrow \varepsilon$ { v[top+1].*s* := v[top-1].*num* + 100 }

$N \rightarrow \varepsilon$ { v[top+1].*s* := v[top-2].*num* + 50}