（若发现问题，请及时告知）

A1. 以下是某简单语言的一段代码。该语言中不包含数据类型的声明，所有变量的类型默认为整型，变量声明以保留字var开头；语句块的括号为 'begin'和 'end'组合；过程声明保留字为procedure；赋值号为 ':='，不等号为'<>'。该语言支持嵌套的过程声明（类似于Pascal语言），但只能定义无参过程，且没有返回值。

```
（1）  var a0, b0, a2;
（2）  procedure fun1 ;
（3）          var a1, b1;
（4）          procedure fun2 ;
（5）                  var a2;
（6）                  begin
（7）                          a2 := a1 + b1;
（8）                          if(a0 <> b0) then call fun3;
  .                            ……    /*不含任何 call 语句和声明语句*/
  .                   end;
  .          begin
                       a1 := a0 - b0;
                       b1 := a0 + b0;
（x）                  If   a1 < b1   then   call fun2 ;
  .                    ……     /*不含任何 call 语句和声明语句*/
  .          end ;
  .  procedure fun3 ;
  .          var a3;
  .          begin
                  a3 := a0*b0 ;
（y）             if(a2 <> a3) call fun1 ;
  .                  ……    /*不含任何 call 语句和声明语句*/
  .          end ;
  .  begin
  .          a0 := 1;
  .          b0 := 2;
  .          a2 := a0 - b0 ;
  .          call fun3;
  .          ……    /*不含任何 call 语句和声明语句*/
.   end .
```

(a) 若该语言编译器的符号表类似于图 5 所示的那样，采用一个全局的单符号表栈结构。试指出：在分析至语句（x）和（y）时，当前开作用域分别有几个？各包含哪些符号？在分析至语句（y）时，所访问的 a2 是在哪行语句声明的？

(b) 若实现该语言时采用多符号表的组织和管理方式，即每个静态作用域均对应一个符

号表。假定采用多遍扫描机制，在静态语义检查之前每个作用域中的所有表项均已生成。实现静态语义分析时，为了体现作用域信息，须要维护一个作用域栈，假设栈中的元素是指向某个作用域的指针。试指出：在分析至语句（x）和（y）时，当前开作用域分别有几个？各包含哪些符号？

参考解答：

(a) 在分析至语句（x）时，当前开作用域：全局作用域（含符号 a0, b0, a2, fun1），fun1 作用域（含符号 a1, b1, fun2）。

在分析至语句（y）时，当前开作用域：全局作用域（含符号 a0, b0, a2, fun1, fun3），fun3 作用域（含符号 a3）；所访问的 a2 是在第 1 行语句声明的。

(b) 在分析至语句（x）时，当前开作用域：全局作用域（含符号 a0, b0, a2, fun1, fun3），fun1 作用域（含符号 a1, b1, fun2）。

在分析至语句（y）时，当前开作用域：全局作用域（含符号 a0, b0, a2, fun1, fun3），fun3 作用域（含符号 a3）。

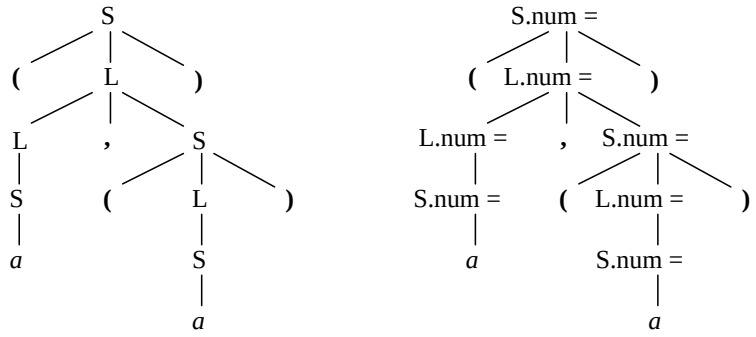
A2. 给定文法 *G*[*S*]：

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

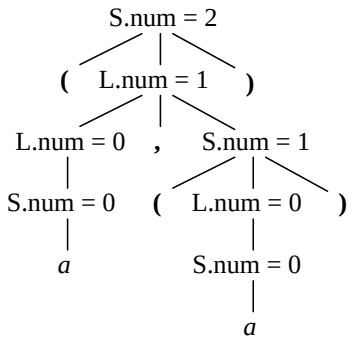
如下是相应于 *G*[*S*] 的一个属性文法（或翻译模式）：

$S \to （L）$　　{ *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; }

以下两个图分别是输入串（ *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）。



参考解答：



A3. 题 A2 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。如下是 $G[S]$ 的一个 *LR* 分析表：

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			

下图描述了输入串 $(a,(a))$ 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。

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			

参考解答：

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

A4. 不难判定，以下文法 $G[S]$ 是一个 LL(1) 文法：

$S \rightarrow a\ B\ c \mid b\ A\ B$
$A \rightarrow a\ A\ b \mid b$
$B \rightarrow b \mid \varepsilon$

1. 如下是以 $G[S]$ 为基础改造的一个 L 翻译模式：

   $S' \rightarrow S$ { $print(S.num)$; }
   $S \rightarrow a$ { $B.in\_num := 1$; } $B\ c$ {$S.num := B.num - 1$;}
   $S \rightarrow b$ { $A.in\_num := -1$; } A { $B.in\_num := A.num$; } B { $S.num := B.num$; }
   $A \rightarrow a$ { $A_1.in\_num := A.in\_num + 1$; } $A_1\ b$ { $A.num := A_1.num - 1$; }
   $A \rightarrow b$ { $A.num := A.in\_num - 1$; }
   $B \rightarrow b$ { $B.num := B.in\_num - 1$; }
   $B \rightarrow \varepsilon$ { $B.num := B.in\_num$; }

   对于输入串 *baabbb*，该翻译模式的语义计算结果是什么？（回答 print 语句的输出结果即可）

2. 如下是针对该翻译模式构造的一个自上而下的递归下降（预测）翻译程序，其中使用了与课程中所给相同的 MatchToken 函数。试补全程序中的(1)-(6)部分代码。

```
void main( )                    // 主函数
{
    S_num := ParseS();
    print(S_num);
}

int ParseS( )
{
    switch (lookahead)  {              // lookahead为下一个输入符号
        case 'a' :
            MatchToken('a');
            B_in_num := 1;
            B_num := ParseB(B_in_num);
            S_num := B_num-1;
            ____________________________(1)
            break;
```

```
            case ‘b’:
                MatchToken(‘b’);
                A_in_num := -1;
                A_num := ParseA(A_in_num);
                B_in_num := A_num;
                                                    (2)
                S_num := B_num;
                break;
            default:
                printf("syntax error \n")
                exit(0);
        }
                                                        (3)
}


int ParseA( int A_in_num )
{
    switch (lookahead)  {
            case ‘a’ :
                MatchToken(‘a’);
                A1_in_num := A_in_num + 1;
                A1_num := ParseA(A1_in_num);
                                                    (4)
                MatchToken(‘b’);
                break;
            case ‘b’:
                MatchToken(‘b’);
                A_num := A_in_num - 1;
                break;
            default:
                printf("syntax error \n")
                exit(0);
    }
    return A_num;
}

int ParseB( int B_in_num )
{
    switch (lookahead)  {
            case ‘b’ :
                MatchToken(‘b’);
                B_num := B_in_num – 1;
                break;
```

```
            case _____:_____(5)
                 _____(6)
                 break;
            default:
                 printf("syntax error \n")
                 exit(0);
        }
        return B_num;
    }
```

参考解答：

1. -2

2.

```
    void main( )                    //  主函数
    {
        S_num := ParseS();
        print(S_num);
    }

    int ParseS( )
    {
        switch (lookahead)  {           // lookahead为下一个输入符号
            case ‘a’ :
                MatchToken(‘a’);
                B_in_num := 1;
                B_num := ParseB(B_in_num);
                S_num := B_num-1;
                MatchToken(‘c’);                    (1)
                break;
            case ‘b’:
                MatchToken(‘b’);
                A_in_num := -1;
                A_num := ParseA(A_in_num);
                B_in_num := A_num;
                B_num := ParseB(B_in_num);          (2)
                S_num := B_num;
                break;
            default:
                printf("syntax error \n")
                exit(0);
        }
        return S_num;                               (3)
```

```
}


int ParseA( int A_in_num )
{
    switch (lookahead)  {
        case 'a' :
            MatchToken('a');
            A1_in_num := A_in_num + 1;
            A1_num := ParseA(A1_in_num);
            A_num := A1_num – 1;                    (4)
            MatchToken('b');
            break;
        case 'b':
            MatchToken('b');
            A_num := A_in_num - 1;
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
    return A_num;
}

int ParseB( int B_in_num )
{
    switch (lookahead)  {
        case 'b' :
            MatchToken('b');
            B_num := B_in_num – 1;
            break;
        case 'c', '#':                              (5)
            B_num := B_in_num;                      (6)
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
    return B_num;
}
```

其中使用了与课程中所给的 MatchToken 函数。

A5. 给定文法 $G[S]$：

$S \to ASy \mid Sa \mid \varepsilon$
$A \to Bx$
$B \to Bw \mid \varepsilon$

在文法 $G[S]$ 基础上设计如下翻译模式：

$S \to A \{ S_1.d := A.n \} S_1y \{ S.n := S_1.n \}$
$S \to \{ S_1.d := S.d \} S_1a \{ S.n := S_1.n + S.d \}$
$S \to \varepsilon \{ S.n := 0 \}$
$A \to Bx \quad \{ A.n := B.n * 2 \}$
$B \to B_1w \quad \{ B.n := B_1.n + 1 \}$
$B \to \varepsilon \quad \{ B.n := 0 \}$

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，例如可用 $v[i].n$ 访问栈中位置 i 所存放的属性值，不用考虑对 top 的维护）。

参考解答：

$S \to AS_1y \{ v[top - 2].n := v[top - 1].n \}$
$S \to S_1a \{ v[top - 1].n := v[top - 1].n + v[top - 2].n \}$
$S \to \varepsilon \{ v[top + 1].n := 0 \}$
$A \to Bx \quad \{ v[top - 1] := v[top - 1] * 2 \}$
$B \to B_1w \quad \{ v[top - 1].n := v[top - 1].n + 1 \}$
$B \to \varepsilon \quad \{ v[top + 1].n := 0 \}$

A6. 变换如下翻译模式，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确定的综合属性进行访问：

$D \to D_1 ; T \{ L.type := T.type; L.offset := D_1.width ; L.width := T.width \} L$
$\{D.width := D_1.width + L.num \times T.width \}$
$D \to T \{ L.type := T.type; L.offset := 0 ; L.width := T.width \} L$
$\{D.width := L.num \times T.width \}$
$T \to \underline{\text{integer}} \quad \{ T.type := int ; T.width := 4 \}$
$T \to \underline{\text{real}} \quad \{ T.type := real ; T.width := 8 \}$
$L \to \{ L_1.type := L.type ; L_1.offset := L.offset ; L_1.width := L.width ; \} L_1 , \underline{\text{id}}$
$\{ enter (\underline{\text{id}}.name, L.type, L.offset + L_1.num \times L.width) ; L.num := L_1.num + 1 \}$
$L \to \underline{\text{id}} \quad \{ enter (\underline{\text{id}}.name, L.type, L.offset) ; L.num := 1 \}$

参考解答：

$D \rightarrow D_1$; T { *L.type* := *T.type*; *L.offset* := D_1.*width* ; *L.width* := *T.width* } *L*
{*D.width* := D_1.*width* + *L.num* × *T.width* }

$D \rightarrow M\ N\ T$ { *L.type* := *T.type*; *L.offset* := *M.s* ; *L.width* := *T.width* } *L*
{*D.width* := *L.num* × *T.width* }

$T \rightarrow$ integer { *T.type* := *int* ; *T.width* := 4 }

$T \rightarrow$ real { *T.type* := *real* ; *T.width* := 8 }

$L \rightarrow$ { L_1. *type* := *L. type* ; L_1. *offset* := *L. offset* ; L_1. *width* := *L. width* ; } L_1 , id
{ *enter* (id.*name*, *L. type, L. offset* + L_1.*num* × *L. width*) ; *L.num* := L_1.*num* +1 }

$L \rightarrow$ id { *enter* (id.*name*, *L. type, L. offset*) ; *L.num* := 1}

$M \rightarrow \varepsilon$ { *M.s := 0* }

$N \rightarrow \varepsilon$ { }