1. 以下是某简单语言的一段代码。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（假设占用一个存储单元）。语句块的括号为‘begin’和‘end’组合；赋值号为‘:=’，不等号为‘<>’。每一个过程声明对应一个静态作用域（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）。该语言支持嵌套的过程声明，但只能定义无参过程，且没有返回值。

```
（1） var a0, b0;
（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;
  .                           ……    /*不含任何声明语句*/
  .                  end;
  .         begin
                      a1 := a0 - b0;
                      b1 := a0 + b0;
（x）                 If   a1 < b1   then   call fun2 ;
  .                   ……     /*不含任何声明语句*/
  .         end ;
  .  procedure fun3 ;
  .         var a3, b3;
  .         begin
                    a3 := a0*b0 ;
                    b3 := a0/b0 ;
（y）               if(a3 <> b3) call fun1 ;
.                   ……    /*不含任何声明语句*/
  .         end ;
  .  begin
  .         a0 := 1;
  .         b0 := 2;
  .         call fun3;
  .         ……    /*不含任何声明语句*/
  .  end .
```

若实现该语言时符号表的组织采用多符号表结构，即每个静态作用域均对应一个符号表。试指出：在分析至语句（x）时，当前开作用域有几个？分别包含哪些符号？在分析至语句（y）时，当前开作用域有几个？分别包含哪些符号？

参考解答：

在分析至语句（x）时，当前开作用域有2个；分别包含符号{ a0,b0,fun1, fun3}

和{ a1,b1,fun2}。在分析至语句（y）时，，当前开作用域有2个; 分别包含符号{ a0,b0,fun1, fun3}和{ a3,b3 }。

2. 以下是某简单语言的一段代码。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（假设占用一个存储单元）。语句块的括号为‘begin’和‘end’组合；赋值号为‘:=’，不等号为‘<>’。每一个过程声明对应一个静态作用域（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）。该语言支持嵌套的过程声明，但只能定义无参过程，且没有返回值。若实现该语言时符号表的组织采用单符号表结构，即所有静态作用域共用一个符号表。试补全，在分析至语句（x）时，线性表的内容。

```
（1） var a0, b0;
（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;
  .                           ……    /*不含任何声明语句*/
  .                  end;
  .          begin
                      a1 := a0 - b0;
                      b1 := a0 + b0;
（x）                 If   a1 < b1   then   call fun2 ;
  .                  ……     /*不含任何声明语句*/
  .          end ;
  .  procedure fun3 ;
  .          var a3, b3;
  .          begin
                      a3 := a0*b0 ;
                      b3 := a0/b0 ;
（y）                 if(a3 <> b3) call fun1 ;
.                     ……    /*不含任何声明语句*/
  .          end ;
  .  begin
  .          a0 := 1;
  .          b0 := 2;
  .          call fun3;
  .          ……    /*不含任何声明语句*/
  .  end .
```

NAME	*KIND*	*VAL / LEVEL*	*ADDR*	*SIZE*
a0	VARIABLE	LEV	DX	
b0	VARIABLE	LEV	DX+1	
fun1	PROCEDURE	（1）		CX+2
a1	VARIABLE	LEV+1	（2）	

b1	VARIABLE	LEV+1	（3）	
fun2	PROCEDURE	LEV+1		（4）

参考解答：

NAME	*KIND*	*VAL / LEVEL*	*ADDR*	*SIZE*
a0	VARIABLE	LEV	DX	
b0	VARIABLE	LEV	DX+1	
fun1	PROCEDURE	LEV		CX+2
a1	VARIABLE	LEV+1	DX	
b1	VARIABLE	LEV+1	DX+1	
fun2	PROCEDURE	LEV+1		CX+1

3 设有文法 G[S]：

$S \rightarrow aSb \mid aab$

若针对该文法设计一个自顶向下预测分析过程，则需要向前察看多少个输入符号？

参考解答：

需要向前察看 3 个单词。若向前察看 3 个单词是 aab 时，可选第 2 个分支；aaa 时，可选第 1 个分支。

4 给定文法 *G*[*S*]：

$S \rightarrow AB$
$A \rightarrow aA$
$A \rightarrow \varepsilon$
$B \rightarrow bB$
$B \rightarrow \varepsilon$

针对文法 *G*[*S*]，下表给出各产生式右部文法符号串的 *First* 集合，各产生式左部非终结符的 *Follow* 集合，以及各产生式的预测集合 *PS*。试填充其中空白表项的内容((1)-(6))，验证该文法是否 LL(1)文法（说明原因），并补全基于该文法构造的递归下降分析程序((7)-(10))，其中 yylex()为取下一单词过程，变量 lookahead 存放当前单词。

G 中的规则 *r*	*First* (*rhs*(*r*))	*Follow* (*lhs*(*r*))	*PS* (*r*)
$S \rightarrow AB$	（1）	#	（2）
$A \rightarrow aA$	*a*	（3）	*a*
$A \rightarrow \varepsilon$	ε	此处不填	（4）
$B \rightarrow bB$	（5）	#	（6）
$B \rightarrow \varepsilon$	ε	此处不填	#

```
void ParseS( )                    // 主函数
{
```

```
    ParseA( );
    ParseB( );
}
oid ParseA( )
{
    switch (lookahead)          // lookahead为下一个输入符号
    {
         case ′ a′ :
                 （7）    ;
              ParseA();
              break;
         case    （8）    :
              break;
         default:
              printf("syntax error \n")
              exit(0);
    }
    return A_num;
}
oid ParseB( )
{
    switch (lookahead)  {
         case    （9）    :
             MatchToken( ‘b’ );
                （10）    ;
              break;
         case ′ #′ :
              break;
         default:
              printf("syntax error \n");
              exit(0);
    }
}
void  Match_Token(int expected)
{
    if (lookahead != expected)
    {
              printf("syntax error \n")
              exit(0);
    else
         lookahead = getToken();
}
```

参考解答：

G 中的规则 r	$First(rhs(r))$	$Follow(lhs(r))$	$PS(r)$
$S \to AB$	a, b, ε	#	$a, b, \#$
$A \to aA$	a	$b, \#$	a
$A \to \varepsilon$	ε	此处不填	$b, \#$
$B \to bB$	b	#	b
$B \to \varepsilon$	ε	此处不填	#

PS（$A \to aA$）∩ PS（$A \to \varepsilon$）= {a }∩{b,# } = Φ

PS（$B \to bB$）∩ PS（$B \to \varepsilon$）= {b }∩{ # } = Φ

所以，G(S)是LL(1)文法。

用类似C语言写出G[E]的递归子程序,其中yylex()为取下一单词过程,变量lookahead存放当前单词。不需要考虑太多编程语言相关的细节。程序如下：

```
void ParseS( )                    // 主函数
{
    ParseA( );
    ParseB( );
}
void ParseA( )
{
    switch (lookahead)            // lookahead为下一个输入符号
    {
         case ′ a′ :
             MatchToken( ‘a’ );
             ParseA();
             break;
         case ′ b′ , ′ #′ :
             break;
         default:
             printf("syntax error \n")
             exit(0);
    }
    return A_num;
}
void ParseB( )
{
    switch (lookahead)  {
         case ′ b′ :
             MatchToken( ‘b’ );
             ParseB( );
             break;
         case ′ #′ :
             break;
         default:
```

```
                printf("syntax error \n");
                exit(0);
        }
}
void  Match_Token(int expected)
{
        if (lookahead != expected)
        {
                printf("syntax error \n")
                exit(0);
        else
             lookahead = getToken();
}
```

5 补全下列文法（与上一题相同）的分析表，根据分析表验证该文法是否 LL(1)文法（说明原因），并补全输入符号串 *aabb* 的表驱动 LL(1)分析过程（步骤 3、6、9）。

$S \to AB$
$A \to aA$
$A \to \varepsilon$
$B \to bB$
$B \to \varepsilon$

	a	*b*	#
S	$S \to AB$	（1）	$S \to AB$
A	（2）	（3）	$A \to \varepsilon$
B	*此处不填*	（4）	$B \to \varepsilon$

步骤	*下推栈*	余留符号串	*下一步动作*
1	*#S*	*aabb#*	*应用产生式* $S \to AB$
2	*#BA*	*aabb#*	*应用产生式* $A \to aA$
3			
4	*#BA*	*abb#*	*应用产生式* $A \to aA$
5	*#BAa*	*abb#*	*匹配栈顶和当前输入符号*
6			
7	*#B*	*bb#*	*应用产生式* $B \to bB$
8	*#Bb*	*bb#*	*匹配栈顶和当前输入符号*
9			
10	*#Bb*	*b#*	*匹配栈顶和当前输入符号*
11	*#B*	#	*应用产生式* $B \to \varepsilon$
12	#	#	*返回，分析成功*

参考解答：

	a	b	#
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aA$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

分析表中每个单元最多有一个产生式，所以是 LL(1)。

步骤	*下推栈*	余留符号串	*下一步动作*
1	*#S*	*aabb#*	*应用产生式 $S \to AB$*
2	*#BA*	*aabb#*	*应用产生式 $A \to aA$*
3	*#BAa*	*aabb#*	*匹配栈顶和当前输入符号*
4	*#BA*	*abb#*	*应用产生式 $A \to aA$*
5	*#BAa*	*abb#*	*匹配栈顶和当前输入符号*
6	*#BA*	*bb#*	*应用产生式 $A \to \varepsilon$*
7	*#B*	*bb#*	*应用产生式 $B \to bB$*
8	*#Bb*	*bb#*	*匹配栈顶和当前输入符号*
9	*#B*	*b#*	*应用产生式 $B \to bB$*
10	*#Bb*	*b#*	*匹配栈顶和当前输入符号*
11	*#B*	*#*	*应用产生式 $B \to \varepsilon$*
12	*#*	*#*	*返回，分析成功*

6 通过变换求出与下列文法 $G[S]$ 等价的一个文法，使其不含直接左递归：

$S \to AbB$
$A \to Aa \mid a$
$B \to Ba \mid Bb \mid b$

参考解答：

关于A、B的产生式含有直接左递归。根据转换方法，替换结果如下：

$S \to A\,b\,B$
$A \to a\,A'$
$A' \to a\,A' \mid \varepsilon$
$B \to b\,B'$
$B' \to a\,B' \mid b\,B' \mid \varepsilon$