（若发现问题，请及时告知）

A1　计算下列文法中每个非终结符的 First 集和 Follow 集，以及每个产生式的预测集合，并判断该文法是否 LL(1)文法（说明原因）：　　（冀伟清）

（1）文法 $G_1[S]$：

$S \to TS \mid Ta$
$T \to +ST \mid -T \mid \varepsilon$

（2）文法 $G_2[S]$：

$S \to TP$
$T \to +P \mid +T \mid \varepsilon$
$P \to (S) \mid a$

（3）文法 $G_3[S]$：

$S \to aaS \mid bbS \mid (S) \mid \varepsilon$

参考解答:

（1）计算非终结符的 FIRST 集和 FOLLOW 集，结果如下：

FIRST（S）= { +，-，*a* }　　FOLLOW（S）= { #，+，-，*a* }
FIRST（T）= { +，-，ε }　　FOLLOW（T）= { +，-，*a* }

PS($S \to TS$)={ +，-，*a* }

PS($S \to Ta$)={ +，-，*a* }

PS($T \to +ST$)={ + }

PS($T \to -T$)={ - }

PS($T \to \varepsilon$)={ +，-，*a* }

因为，PS($S \to TS$)∩ PS($S \to Ta$)= { +，-，*a* }所以，G(S)不是 LL(1)文法。

（2）计算非终结符的 FIRST 集和 FOLLOW 集，结果如下：

FIRST（S）= { +，(，a }　　FOLLOW（S）= { #，)}

FIRST（T）= { +，ε }　　FOLLOW（T）= { (，a }

FIRST（P）= { (，a }　　FOLLOW（P）= { #，+，(，a ，)}

PS($S \to TP$)={ +，(，a }

PS($T \to +P$)={ + }

PS($T \to +T$)={ + }

PS($T \to \varepsilon$)={ (，a，+}

PS($P \rightarrow (S)$)={ (}

PS($P \rightarrow$ a)={ a }

因为，PS($T \rightarrow +T$)∩ PS($T \rightarrow$ +P)= { + }，所以，G(S)不是 LL(1)文法。

（3）计算非终结符的 FIRST 集和 FOLLOW 集，结果如下：

FIRST（S）= { a，b，ε，(}　　FOLLOW（S）= { #，) }

PS($S \rightarrow aaS$)={ a }

PS($S \rightarrow bbS$)={ b}

PS($S \rightarrow (S)$)={ (}

PS($S \rightarrow$ ε)={ #，) }

因为， 任意两个产生式的预测集合交集为空，所以，G(S)是 LL(1)文法。

A2　按照第 4 讲介绍的消除左递归算法消除下面文法 $G[E]$中的左递归：　　（沈游人）

E → E+T | T
T → T*F | F
F → (E) | id

参考解答:

将非终结符排列为 ETF
对于产生式
E→ E+T | T
消除直接左递归：

E→ TE’
E’ → +TE’ | ε
对于产生式
T→ T*F | F
消除直接左递归
T→FT’
T’ →*FT’ | ε
关于 F 的产生式不含直接左递归，因此不产生变化。
因此消除左递归后的文法 G’[E] 为：

E→ TE’
E’ → +TE’ | ε
T→FT’
T’ →*FT’ | ε
F→ (E) | id

A3　对下面的文法 G[E]：　　（甄　艳　洁　）

E → T E '

E ' → + T E ' | ε

T → F T '

T' → *FT ' | ε

F → (E) | i

（1）构造它的预测分析表。

（2）指出 G[E]是否 LL(1)文法。

（3）对如上文法，给出输入终结符串 i*i+i 的表驱动 LL(1)分析过程（要求给出每一步的下推栈和余留符号串的内容以及每一步所用到的产生式）。

参考解答:

（1）

	i	+	*	(	)	#
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→ ε	E’→ ε
T	T→FT’			T→FT’		
T’		T’→ ε	T’→*FT’		T’→ ε	T’→ ε
F	F→i			F→(E)		

（2）根据预测表可以看出，每个表项最多有一个产生式，因此其是 LL(1)文法

（3）

步骤	下推栈	余留符号串	下一步动作
1	#E	$i_1*i_2+i_3\#$	应用产生式 E→TE'
2	#E'T	$i_1*i_2+i_3\#$	应用产生式 T→FT'
3	#E'T'F	$i_1*i_2+i_3\#$	应用产生式 F→i
4	#E'T'i	$i_1*i_2+i_3\#$	匹配栈顶和当前输入符号
5	#E'T'	$*i_2+i_3\#$	应用产生式 T'→*FT'
6	#E'T'F*	$*i_2+i_3\#$	匹配栈顶和当前输入符号
7	#E'T'F	$i_2+i_3\#$	应用产生式 F→i

8	# E'T'i	$i_2+i_3\#$	匹配栈顶和当前输入符号
9	# E'T'	$+i_3\#$	应用产生式 T'→ ε
10	# E'	$+i_3\#$	应用产生式 E'→+TE'
11	# E'T+	$+i_3\#$	匹配栈顶和当前输入符号
12	# E'T	$i_3\#$	应用产生式 T→FT'
13	# E'T'F	$i_3\#$	应用产生式 F→i
14	# E'T'i	$i_3\#$	匹配栈顶和当前输入符号
15	# E'T'	#	应用产生式 T'→ ε
16	# E'	#	应用产生式 E'→ ε
17	#	#	返回，分析成功

A4 （朱 倖 民 ）

分析表达式文法（Parsing Expression Grammar）是一个四元组(N, Σ, P, S)，其中

- N 为非终结符（nonterminal）集合；
- Σ 为终结符（terminal）集合；
- P 为产生式（parsing rules）集合；
- S 为开始符号（start symbol）。

各产生式形如 $A \rightarrow e$，其中 A 为非终结符，e 为分析表达式（parsing expression）。一个分析表达式分为原子表达式和复合表达式。

$$e ::= t \mid n \mid \varepsilon \mid e_1\, e_2 \mid e_1 \mid e_2 \mid e^* \mid e^+ \mid e^?$$

原子表达式可以是

- 任何一个终结符（$t \in \Sigma$）；
- 任何一个非终结符（$n \in N$）；
- 空串（ε）。

本题仅考虑下列典型的复合表达式：

- 两个分析表达式的序列（$e_1\, e_2$），表示先匹配 e_1 再匹配 e_2；
- 两个分析表达式的有序选择（$e_1 \mid e_2$），表示先尝试匹配 e_1，若成功则忽略 e_2，若失败才尝试匹配 e_2；
- 某个表达式匹配零次或多次（e^*）；
- 某个表达式匹配一次或多次（e^+）；
- 某个表达式匹配零次或一次（$e^?$）。

对于一个 PEG，我们可以采用类似递归下降的方法来分析。与 LL(1)不同的是，该分析过程，无需向后查看符号，而是根据定义好的顺序依次尝试，失败后允许回溯。例如分析

$$S \to e_1 \mid e_2$$

我们先用 $S \to e_1$ 来分析输入串，若分析失败，则尝试用 $S \to e_2$ 重新分析输入串。

根据以上设定，回答下列问题。

（1）分别用 LL(1)文法和 PEG 来表示 Decaf 语言的数组常量。除了, [ ] 为终结符外，为了简单，我们假定常量 *c* 也是终结符。

（2）对于 Decaf 语言中的条件语句，我们可以通过 PEG 的有序选择来消除二义性：

IfStmt → **`if`** Expr Stmt **`else`** Stmt | **`if`** Expr Stmt （A）

IfStmt → **`if`** Expr Stmt | **`if`** Expr Stmt **`else`** Stmt （B）

其中 **`if`** 和 **`else`** 为关键字，Stmt 和 Expr 分别对应于 Decaf 语言中的语句和表达式。分别用文法（A）和（B）分析如下程序片段，最终编译出的可执行代码对应的控制台输出是什么？

```
if (true) if (false) Print("T");
else Print("F");
```

（3）考查下列 CFG（其中 *x* 为终结符）：

$$S \to x\, S\, x \mid x$$

该文法对应的语言记作 *L*。

（3-1）是否可以直接对上述文法采用递归下降方法来分析？请说明理由。

（3-2）是否存在某个能做递归下降分析的 PEG，其对应的语言也为 *L*？若存在，请给出此 PEG；否则，请说明理由。

参考解答:

（1）LL(1):

$$S \to [\, T\,]$$
$$T \to c\, A \mid \varepsilon$$
$$A \to ,\, c\, A \mid \varepsilon$$

PEG:

$$S \to [\, T\,]$$
$$T \to c\, (,\, c)^* \mid \varepsilon$$

（2）文法（A）会将 ELSE 与其最近的 IF 进行匹配，即分析出的程序为

```
if (true) { if (false) Print("T");
else Print("F"); }
```

控制台打印 `F`。

文法（B）会优先丢掉 ELSE 字句，即分析出的程序为

```
if (true) { if (false) Print("T"); }
else Print("F");
```

控制台无输出。

（3）

（3-1）不能。语言 $L=\{x^n x x^n\}$为以 *x* 为中心的全 *x* 串，而在分析前我们无法找到中心点的 *x* 位于何处，因此我们无法决定何时采用 $S \to x\, S\, x$，何时采用 $S \to x$。

（3-2）该语言 $L=\{x^{2n+1}\}$，即奇数个 *x* 组成的串。因此可以设计如下接受 *L* 的 PEG：

$$S \to x\, (x\, x)^*$$

以下是 Lecture04 文档中的题目

1 设有文法 G[S]:

S → aSb | 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。

6 验证如下文法是 LL(1)文法，并基于该文法构造递归下降分析程序：

（2）文法 *G'*[*E*]：

E → [*F*] *E'*
E' → *E*| ε
F → *aF'*
F' → *aF'* | ε

参考解答:

（2）观察文法规则可知，可能产生规则选择冲突的规则只能是 E' → E| ε 和 F' → aF' | ε 。 我们只需要求出这四条规则的 PS 集合（预测集合）即可。欲求这四个 PS 集合，我们需要先求出：

First (E) = { [}　　Follow (E′) = {#}
First (aF′) = { a }　　Follow (F′) = {] }

从而

PS (E' → E) = First (E) = { [}
PS (E' → ε) = Follow (E′) = {#}
PS (F' → aF') = First (aF′) = { a }
PS (F' → ε) = Follow (F′) = {] }

因为

对于 E' → E| ε 有： *PS* (E' → E) ∩ *PS* (E' → ε) =Φ

对于 F’ → aF’ | ε 有： PS（F’ → aF’）∩ PS（F’ → ε）=Φ

所以，文法 G[E] 是 LL（1）文法。

用类似 C 语言写出 G[E]的递归子程序，其中 getToken()为取下一单词过程，变量 lookahead 为全局变量，存放当前单词。

```
void ParseE( ) {
    Match_Token ( [ );
    ParseF( );
    MatchToken ( ] );
    ParseE’ ( );
}

void ParseE’ ( ) {
    switch (lookahead)  {
        case [:
            ParseE( );
            break;
        case #:
            break;
        default:
            printf("syntax error \n" );
            exit(0);
    }
}

void ParseF( ) {
    MatchToken ( a );
    ParseF’  ( );
}

void ParseF’ ( ) {
    switch (lookahead)  {
        case a:
            MatchToken ( a );
            ParseF’  ( );
            break;
        case ]:
            break;
        case:
            printf("syntax error \n" );
            exit(0);
    }
}

void MatchToken(int expected) {//判别当前单词是否与期望的终结符匹配
    if (lookahead != expected){
            printf("syntax error \n" );
            exit(0);
    else     // 若匹配,消费掉当前单词并调用词法分析器读入下一个单词
```

```
        lookahead = getToken();
}
```

11 按照本讲介绍的消除一般左递归算法消除下面文法 $G[S]$中的左递归（要求依非终结符的排序 S、Q、P 执行该算法）：

$$S \to PQ \mid a$$
$$P \to QS \mid b$$
$$Q \to SP \mid c$$

参考解答:

按照非终结符的特定顺序排列各规则：

S →PQ | a
Q →SP | c
P →QS | b

第一步，得：

S →PQ | a
Q →PQP | aP | c
P →QS | b

第二步，得：

S →PQ | a
Q →PQP | aP | c
P →PQPS | aPS | cS | b

消去 P →PQPS 的左递归得：

S →PQ | a
Q →PQP | aP | c
P →aPSR | cSR | bR
R →QPSR | ε

经检查，此时得到的文法已经不含左递归，可结束消除左递归过程。

13 变换下列文法， 求出与其等价的一个文法，使变换后的文法不含左递归和左公因子：

（1）文法 $G[P]$：

$$P \to Pa \mid Pb \mid c$$

参考解答:

（1）消除左递归后，G［P］变换为等价的 G'[P］如下：

$$P \to \ c\,Q$$
$$Q \to a\,Q \mid b\,Q$$