1. 基于前面给出的数据结构，就 LL 语法分析写出下列功能函数的实现代码：

    1） 产生式有左递归的判断以及左递归的消除实现；

    2） 产生式有左公因子的判断，以及左公因子的提取实现；

    3） 产生式的 FIRST 函数求解；

    4） 非终结符的 FIRST 函数求解；

    5） 非终结符的 FOLLOW 函数求解；

    6） 文法是为 LL(1)文法的判断；

    7） LL(1)语法分析表的填写。

```
void First(Production* pProduction) {
    NonTerminalSymbol* pHeader = (NonTerminalSymbol*)pProduction->pHeader;
    if (pHeader->pFirstSet == nullptr) {
        pHeader->pFirstSet = new set<TerminalSymbol*>();
    }

    set<TerminalSymbol*>* pFirstSet = new set<TerminalSymbol*>();
    int index = 0;
    for (auto it = pProduction->pBodySymbolTable->begin(); it != pProduction->pBodySymbolTable->end(); it++) {
        GrammarSymbol* pSymbol = (*it);
        switch (pSymbol->type) {
            case TERMINAL:
                pFirstSet->insert((TerminalSymbol*)pSymbol);
                index++;
                break;
            case NONTERMINAL:
                First((NonTerminalSymbol*)pSymbol);
                set<TerminalSymbol*>* pFirst = ((NonTerminalSymbol*)pSymbol)->pFirstSet;
                pFirstSet->insert(pFirst->begin(), pFirst->end());
                if (pFirst->find(NULL_TERMINAL) == pFirst->end()) {
                    goto END_OF_COMPUTATION;
                }
                index++;
                break;
            case NULL_TERMINAL:
                pFirstSet->insert(NULL_TERMINAL);
                index++;
                break;
            default:
                break;
        }
    }
END_OF_COMPUTATION:
    pProduction->pFirstSet = pFirstSet;
}

void First(NonTerminalSymbol* pNonTerminal) {
    if (pNonTerminal->pFirstSet == nullptr) {
        pNonTerminal->pFirstSet = new set<TerminalSymbol*>();
    }
    for (auto it = pNonTerminal->pProductionTable->begin(); it != pNonTerminal->pProductionTable->end(); it++) {
        Production* pProduction = (*it);
        set<TerminalSymbol*>* pFirst = pProduction->pFirstSet;
        if (pFirst == nullptr) {
            First(pProduction);
            pFirst = pProduction->pFirstSet;
        }
        pNonTerminal->pFirstSet->insert(pFirst->begin(), pFirst->end());
    }
}
```

```cpp
void Follow(NonTerminalSymbol* pSymbol) {

    pSymbol->pFollowSet = new std::set<TerminalSymbol*>();


    if (pSymbol->name == startSymbol->name) {
        pSymbol->pFollowSet->insert(terminalSymbolTable["$"]);
    }

    for (int i = 0; i < nonTerminalSymbolTable.size(); i++) {
        NonTerminalSymbol* pNonTerminalSymbol = nonTerminalSymbolTable[i];
        for (int j = 0; j < pNonTerminalSymbol->numOfProduction; j++) {
            Production* pProduction = pNonTerminalSymbol->pProductionTable->at(j);
            for (int k = 0; k < pProduction->bodySize; k++) {
                GrammarSymbol* pSymbolInProduction = pProduction->pBodySymbolTable->at(k);
                if (pSymbolInProduction->type == NONTERMINAL && pSymbolInProduction == pSymbol) {
                    if (k < pProduction->bodySize - 1) {
                        GrammarSymbol* pNextSymbol = pProduction->pBodySymbolTable->at(k + 1);
                        if (pNextSymbol->type == TERMINAL) {
                            pSymbol->pFollowSet->insert((TerminalSymbol*)pNextSymbol);
                        } else {
                            std::set<TerminalSymbol*>* pFirstSetOfNextSymbol =
                                ((NonTerminalSymbol*)pNextSymbol)->pFirstSet;
                            for (std::set<TerminalSymbol*>::iterator it = pFirstSetOfNextSymbol->begin();
                                    it != pFirstSetOfNextSymbol->end(); it++) {
                                if ((*it) != nullSymbol) {
                                    pSymbol->pFollowSet->insert((*it));
                                }
                            }
                        }
                    } else {
                        std::set<NonTerminalSymbol*>* pDependentSetInFollow = pSymbol->pDependentSetInFollow;
                        for (std::set<NonTerminalSymbol*>::iterator it = pDependentSetInFollow->begin();
                                it != pDependentSetInFollow->end(); it++) {
                            Follow((*it));
                            std::set<TerminalSymbol*>* pFollowSetOfDependent = (*it)->pFollowSet;
                            for (std::set<TerminalSymbol*>::iterator it2 = pFollowSetOfDependent->begin();
                                    it2 != pFollowSetOfDependent->end(); it2++) {
                                pSymbol->pFollowSet->insert((*it2));
                            }
                        }
                    }
                }
            }
        }
    }
}
```

2. 基于前面给出的数据结构，就 LR 语法分析写出下列功能函数的实现代码：

    1) 一个项集中 LR(0)核心项的闭包求解，即实现函数：
    void getClosure(ItemSet *itemSet)；

    2) 穷举一个 LR(0)项集的变迁，其中包括驱动符的穷举，下一项集的创建，下一项集中核心项的确定，下一项集是否为新项集的判断。即实现函数：
    void exhaustTransition(ItemSet *itemSet)；

    3) 文法的 LR(0)型 DFA 求解；

    4) 文法是否为 SLR(1)文法的判断；

    5) LR 语法分析表的填写；

1) 一个项集中 LR(0)核心项的闭包求解，即实现函数：

*void getClosure(ItemSet *itemSet);*

```
void getClosure(ItemSet *itemSet) {
    queue<LR0Item *> closure;
    for (int i = 0; i < itemSet->pItemTable->size(); i++) {
        LR0Item *item = itemSet->pItemTable->get(i);
        if (item->type == CORE) {
            closure.push(item);
        }
    }
    while (!closure.empty()) {
        LR0Item *item = closure.front();
        closure.pop();
        if (item->dotPosition == item->production->pBodyTable->size()) {
            continue;
        }
        GrammarSymbol *symbol = item->production->pBodyTable->get(item->dotPosition);
        if (symbol->symbolType == TERMINAL) {
            continue;
        }
        NonTerminalSymbol *nonTerminalSymbol = (NonTerminalSymbol *)symbol;
        List<Production *> *pProductionList = pGrammarSymbolTable->find(nonTerminalSymbol->name)->pProductionList;
        for (int i = 0; i < pProductionList->size(); i++) {
            Production *production = pProductionList->get(i);
            LR0Item *newItem = new LR0Item();
            newItem->nonTerminalSymbol = nonTerminalSymbol;
            newItem->production = production;
            newItem->dotPosition = 0;
            newItem->type = NONCORE;
            if (itemSet->find(newItem) == -1) {
                itemSet->pItemTable->add(newItem);
                closure.push(newItem);
            } else {
                delete newItem;
            }
        }
    }
```

2) 穷举一个 LR(0)项集的变迁，其中包括驱动符的穷举，后继项集的创建，后继项集中核心项的确定，后继项集是否为新项集的判断。即实现函数：

*void exhaustTransition(ItemSet *itemSet);* 19

```
void exhaustTransition(ItemSet *itemSet) {
    for (int i = 0; i < itemSet->pItemTable->Count(); i++) {
        LR0Item *item = itemSet->pItemTable->Get(i);
        GrammarSymbol *symbol = item->GetSymbolAfterDot();
        if (symbol == NULL || symbol->IsTerminal()) {
            continue;
        }
        NonTerminalSymbol *nonTerminalSymbol = dynamic_cast<NonTerminalSymbol *>(symbol);
        ItemSet *newItemSet = CreateNewSet(itemSet, nonTerminalSymbol);
        newItemSet = GetClosure(newItemSet);
        if (!IsExistItemSet(pItemSetTable, newItemSet)) {
            newItemSet->stateId = pItemSetTable->Count();
            pItemSetTable->Add(newItemSet);
            exhaustTransition(newItemSet);
        }
        TransitionEdge *edge = new TransitionEdge();
        edge->driverSymbol = nonTerminalSymbol;
        edge->fromItemSet = itemSet;
        edge->toItemSet = newItemSet;
        itemSet->pTransitionTable->Add(edge);
    }
}
```

3）文法的 LR(0)型 DFA 求解；

```
void DFA* Grammar::constructLR0DFA()
{
    ItemSet* initItemSet = new ItemSet();
    initItemSet->stateId = 0;
    initItemSet->pItemTable = new List<LR0Item*>();
    LR0Item* initItem = new LR0Item();
    initItem->nonTerminalSymbol = RootSymbol;
    initItem->production = RootSymbol->pFirstProduction;
    initItem->dotPosition = 0;
    initItem->type = CORE;
    initItemSet->pItemTable->add(initItem);
    getClosure(initItemSet);

    List<ItemSet*>* itemSetTable = new List<ItemSet*>();
    itemSetTable->add(initItemSet);
    List<TransitionEdge*>* edgeTable = new List<TransitionEdge*>();
    int nextStateId = 1;


    for (int i = 0; i < itemSetTable->count(); i++)
    {
        ItemSet* curItemSet = itemSetTable->get(i);


        exhaustTransition(curItemSet);


        for (int j = 0; j < curItemSet->pEdgeTable->count(); j++)
        {
            TransitionEdge* curEdge = curItemSet->pEdgeTable->get(j);
            ItemSet* nextItemSet = new ItemSet();
            nextItemSet->stateId = nextStateId++;


            for (int k = 0; k < curItemSet->pItemTable->count(); k++)
            {
                LR0Item* curItem = curItemSet->pItemTable->get(k);
                if (curItem->dotPosition < curItem->production->bodyLength()
                    && curItem->production->pBody[curItem->dotPosition] == curEdge->driverSymbol)
                {
                    LR0Item* nextItem = new LR0Item();
                    nextItem->nonTerminalSymbol = curItem->nonTerminalSymbol;
                    nextItem->production = curItem->production;
                    nextItem->dotPosition = curItem->dotPosition + 1;
                    nextItem->type = CORE;
                    nextItemSet->pItemTable->add(nextItem);
                }
            }
            getClosure(nextItemSet);

            if (nextItemSet->pItemTable->count() > 0)
            {
                itemSetTable->add(nextItemSet);
                TransitionEdge* nextEdge = new TransitionEdge();
                nextEdge->driverSymbol = curEdge->driverSymbol;
                nextEdge->fromItemSet = curItemSet;
                nextEdge->toItemSet = nextItemSet;
                edgeTable->add(nextEdge);
            }
        }
```

4) 文法是否为 *SLR*(1)文法的判断；

```
bool isSLR1(Grammar *grammar) {
    if (!isLR0(grammar)) {
        return false;
    }
    DFA *dfa = constructLR0DFA();
    for (int i = 0; i < grammar->num_productions; i++) {
        Production *prod = grammar->productions[i];
        set<int> followSet = getFollowSet(grammar, prod->left);
        State *state = dfa->states[prod->left];
        set<int> lookahead = getLookahead(grammar, state, prod);
        for (auto l : lookahead) {
            if (followSet.find(l) == followSet.end()) {
                return false;
            }
        }
    }
    return true;
}
```

5）LR 语法分析表的填写；

```
void fillLRTable(Grammar *grammar, LRTable *lrTable) {
    int n = grammar->getNumOfSymbols();
    int m = lrTable->getNumOfStates();
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            ItemSet *itemSet = lrTable->getItemSet(i);
            LRAction action = lrTable->getAction(i, j);
            if (action.type != LRActionType::UNDEFINED) {
                continue;
            }
            Symbol *symbol = grammar->getSymbol(j);
            ItemSet *nextItemSet = lrTable->getGoto(itemSet, symbol);
            if (nextItemSet != nullptr) {
                int k = lrTable->getStateIndex(nextItemSet);
                if (symbol->isTerminal()) {

                    action.type = LRActionType::SHIFT;
                    action.param = k;
                } else {
                    action.type = LRActionType::GOTO;
                    action.param = k;
                }
            } else {
                Production *production = lrTable->getReduceProduction(itemSet, symbol);
                if (production != nullptr) {
                    action.type = LRActionType::REDUCE;
                    action.param = production->getIndex();
                } else {
                    action.type = LRActionType::UNDEFINED;
                }
            }
            lrTable->setAction(i, j, action);
        }
    }
}
```

3. 对于如下两个文法 ，其中 S 为非终结符，其他为终结符。

文法 1	$S \rightarrow +SS \mid *SS \mid a$	输入串+*aaa
文法 2	$S \rightarrow SS+ \mid SS* \mid a$	输入串 aa+a*

1）基于最左推导，写出对输入串的推导过程，只要求写出每步推导后的句型；

2）基于最右推导，写出对输入串的推导过程，只要求写出每步推导后的句型；

3）画出输入串的语法分析树。

3. 1) 最左推导

文法1： +SS

+*SSS

+*aSS

+*aaS

+*aaa

文法2： SS*

SS+S*

aS+S*

aa+S*

aa+a*

2) 最右推导

文法1： +SS

+Sa

+*SSa

+*Saa

+*aaa

文法2： SS*

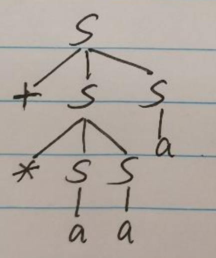
Sa*

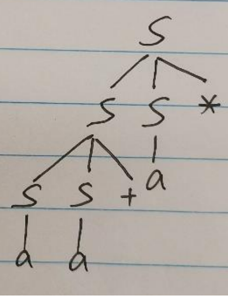
SS+a*

Sa+a*

aa+a*

3)





4. 对于如下三个文法，非终结符仅只有 S，其他为终结符。如果有左递归则先消除左递归，如果有左公因子则先提取左公因子。然后求每个非终结符的 FIRST 和 FOLLOW 函数值。分别判断它们是否为 LL(1)文法？如果是 LL(1)文法，则填出其 LL（1）语法分析表。

文法 1	文法 2	文法 3
S→+SS S→*SS S→a	S→$S(S)S$ S→ ε	S→S+S S→SS S→(S) S→S* S→a

4. 文法1

First(S) = {'+', '*', 'a'}

Follow(S) = {'$'} + (First(S) - 'ε')

= {'+', '*', 'a', '$'}

非终结符	输入符 +	*	a	$
S	S→+SS	S→*SS	S→a	

文法1是 LL(1)文法

LL(1)语法分析表如左图

文法2

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

$\Rightarrow S \to \varepsilon \cdot S'$

$S' \to (S)SS' \mid \varepsilon$

即 $S \to S'$

$S' \to (S)SS' \mid \varepsilon$

First(S') = {'(', 'ε'}

First(S) = {'(', 'ε'}

Follow(S) += First(S') - 'ε'

= {'(', '$'}

又 ε ∈ First(S')

∴ Follow(S) += Follow(S')

Follow(S) += ')' = {'(', ')', '$'}

Follow(S') += Follow(S)

综上 Follow(S) = {'(', ')', '$'}

Follow(S') = {'(', ')', '$'}

非终结符	输入符 (	)	$
S	S→S'		
S'	S'→(S)SS' S'→ε		

一个单元格内填了两个产生式

故文法2不是 LL(1)文法

文法3.

消除左递归

$S \to (S)S' \mid aS'$

$S' \to +SS' \mid SS' \mid *S' \mid \varepsilon$

$First(S') = \{+, *, \varepsilon\} + First(S)$

$First(S) = \{'(', 'a'\}$

$\therefore First(S') = \{'+', '*', '\varepsilon', '(', 'a'\}$

$Follow(S) += \{')', '\$'\}$

$Follow(S) += First(S') - \varepsilon$

$= \{'+', '*', '\varepsilon', '(', ')', 'a', '\$'\}$

$Follow(S) += Follow(S')$

$Follow(S') += Follow(S)$

综上 $Follow(S) = \{'+', '*', '\varepsilon', '(', ')', 'a', '\$'\}$

$Follow(S') = \{'+', '*', '\varepsilon', '(', ')', 'a', '\$'\}$

非终结符	输入符 +	*	(	)	a	\$
S			$S \to (S)S'$		$S \to aS'$	
S'	$S' \to +SS'$, $S' \to \varepsilon$	$S' \to *S'$				

一个单元格内填了两个产生式

故文法3不是LL(1)文法

5. 对于如下两个文法，非终结符仅只有 *S*，其他为终结符。分别画出其 LR(0)项集的状态转换图（即 DFA）。分别判断它们是否为 LR(0)文法？是否为 SLR(1)文法，并给出理由。如果是 LR(0)文法或者 SLR(1)，则填出其 LR 语法分析表。

文法 1	文法 2
S→*SS*+ S→*SS** S→a	S→a*S*a S→aa

5. 文法一

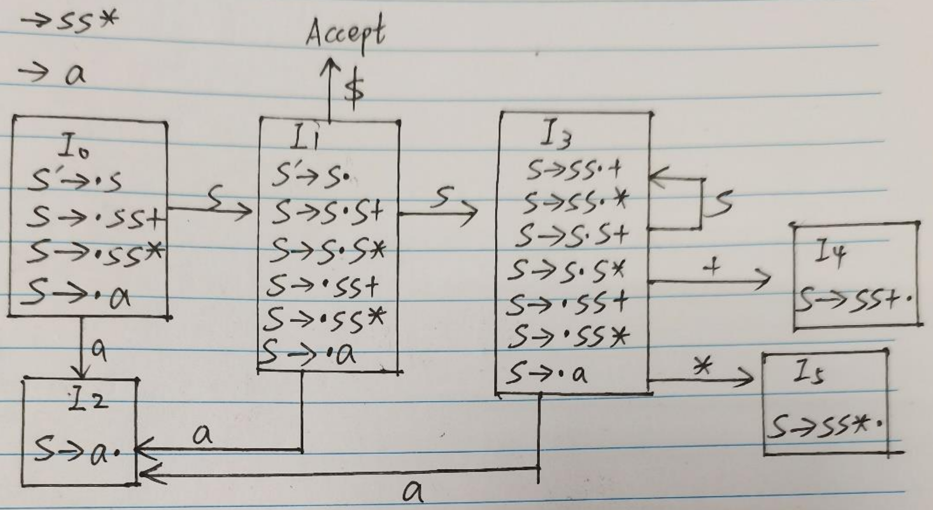
① S'→S

② S→SS+

③ S→SS*

④ S→a

Follow(S) = {'+', '*', '$', 'a'}



给文法编号

①S→SS+

②S→SS*

③S→a

状态	ACTION a	ACTION +	ACTION *	ACTION $	GOTO S
0	s2				1
1	s2			acc	3
2	r3	r3	r3	r3	
3	s2	s4	s5		3
4	r1	r1	r1	r1	
5	r2	r2	r2	r2	

该文法是LR(0)文法

因为语法分析表中没有重复的条目

无移入-归约冲突

文法二.

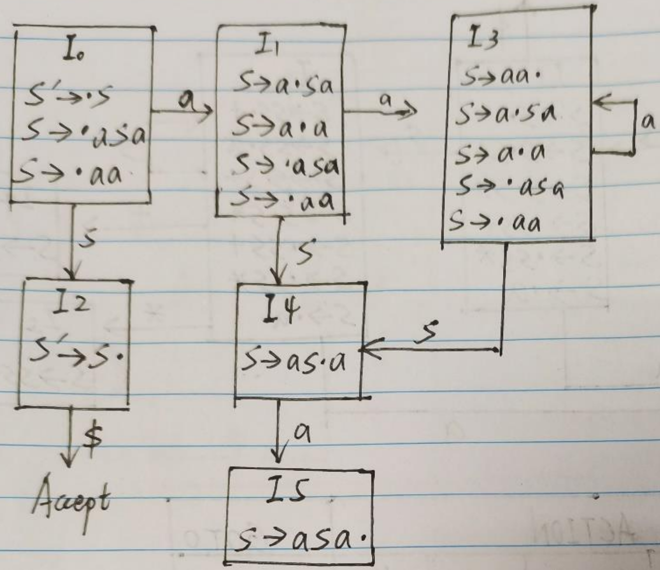
$S' \to S$

① $S \to aSa$

② $S \to aa$.

$First(S) = \{'a'\}$

$Follow(S) = \{'a', '\$'\}$



状态	ACTION		GOTO
	a	$	S
0	s1		2
1	s3		4
2		acc	
3	s3/r2	r2	4
4	s5		
5	r1	r1	

该文法不是LR(0)和SLR(1)文法

因为文法出现了二义性，出现了移入一归约冲突

且无法根据Follow(S)解决，一个单元格内出现了两个内容

6. 说明如下文法是 LR(0) 文法。其中 S 和 A 为非终结符。

$S \rightarrow SA \mid A$

A→a

6.

$S' \rightarrow S$

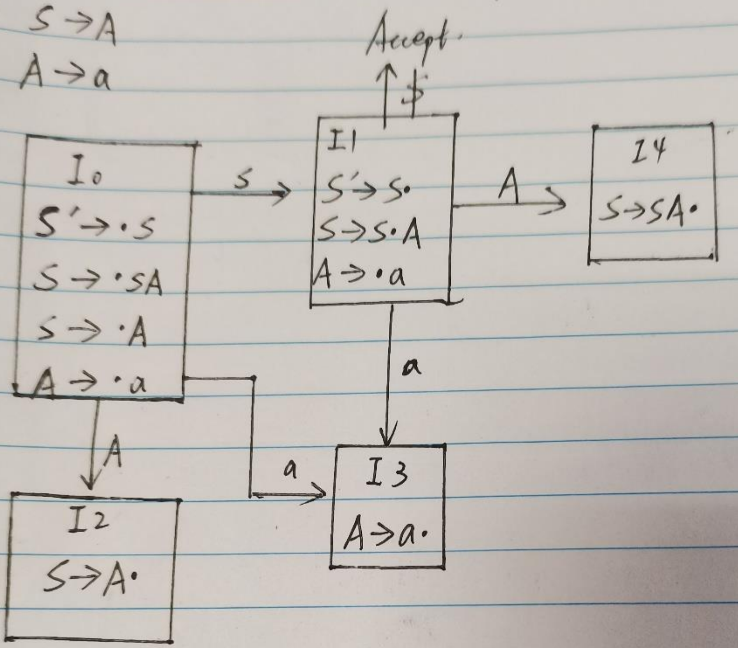
① $S \rightarrow SA$

② $S \rightarrow A$

③ $A \rightarrow a$

Follow(S) = {'a', '$'}

Follow(A) = {'a', '$'}



状态	ACTION		GOTO	
	a	$	S	A
0	s3		1	2
1	s3	acc		4
2	r2	r2		
3	r3	r3		
4	r1	r1		

不存在移入-归约的冲突

因此是 LR(0) 文法

7. 说明如下文法是 LALR(1)文法，但不是 SLR(1)文法。其中 S 和 A 为非终结符。

$S \to A$a | bAc| dc | bda

$A \to$d

7.

$S' \to S$

① $S \to Aa$

② $S \to bAc$

③ $S \to dc$

④ $S \to bda$

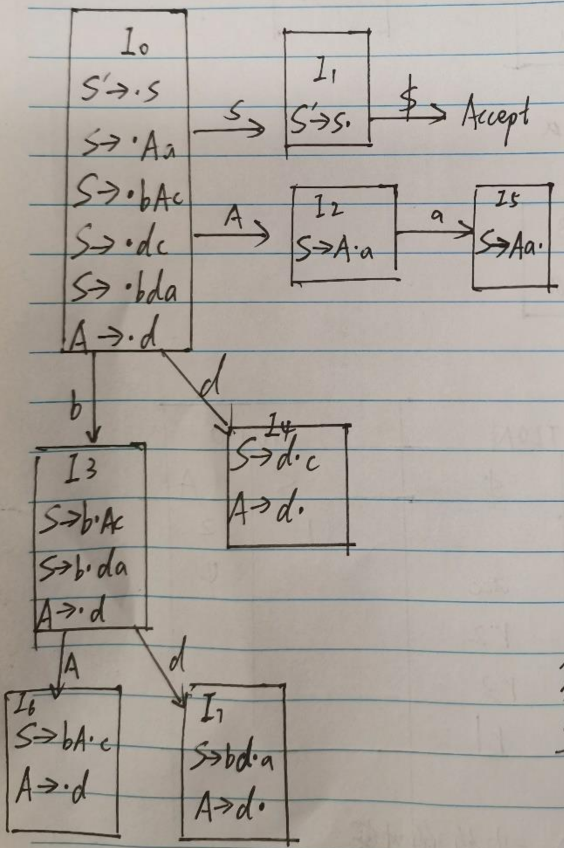
⑤ $A \to d$

First(S)={b,d}

First(A)={d}

Follow(S)={\$}

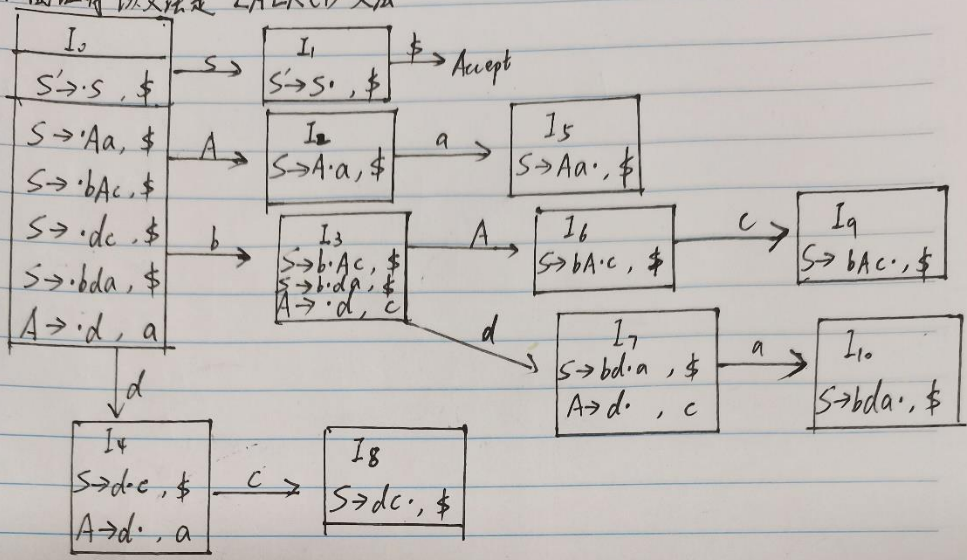
Follow(A)={a,c}



左图已画出部分
DFA图，
可以看出在I_4和I_7
已经出现了移入-归约冲突
对于I_4，有$A \to d\cdot$
若$c \notin Follow(A)$，则可以利用$S \to d\cdot c$
移入，但是 $c \in Follow(A)$
SLR(1)无法解决移入-归约冲突
对于I_7，有$A \to d\cdot$
若$a \notin Follow(A)$，则可以利用$S \to bd\cdot a$
移入，但是$a \in Follow(A)$

综上不是SLR(1)文法.

下面证明该文法是 LALR(1) 文法



状态	ACTION					GOTO	
	a	b	c	d	$	S	A
0		s3		s4		1	2
1					acc		
2	s5						
3				s7			6
4	r5		s8				
5					r1		
6			s9				
7	s10		r5				
8					r3		
9					r2		
10					r4		

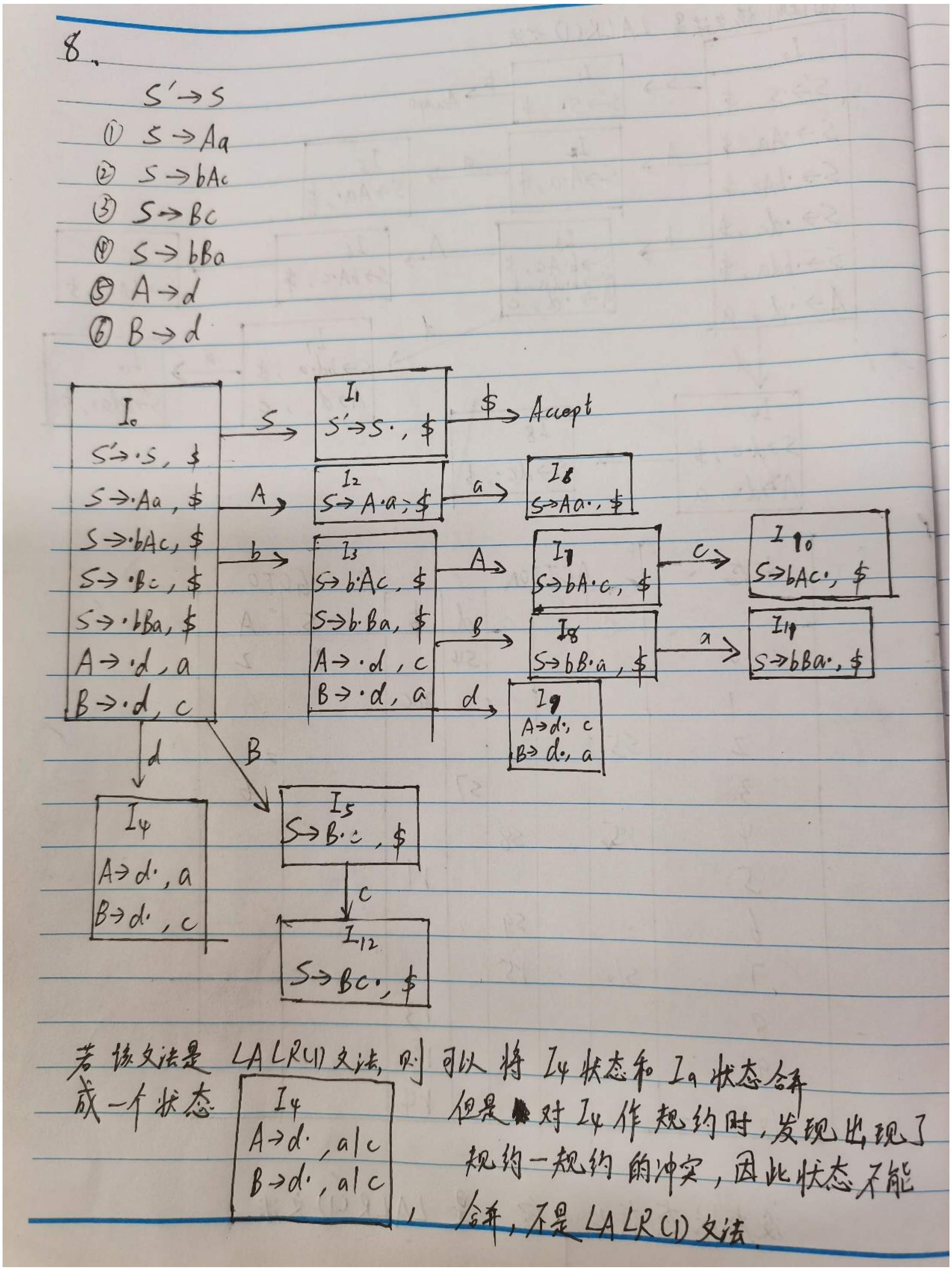
没有状态存在冲突，是 LALR(1) 文法.

8. 说明如下文法是 LR(1)文法，但不是 LALR(1)文法。其中 S、A、B 为非终结符。

$S \to A\mathrm{a} \mid \mathrm{b}A\mathrm{c} \mid B\mathrm{c} \mid \mathrm{b}B\mathrm{a}$

$A \to \mathrm{d}$

$B \to \mathrm{d}$



状态	ACTION					GOTO		
	a	b	c	d	$	S	A	B
0		s3		s4		1	2	5
1					acc			
2	s6							
3				s9			7	8
4	r5		r6					
5			s12					
6					r1			
7			s10					
8	s11							
9	r6		r5					
10					r2			
11					r4			
12					r3			

无状态存在冲突，是LR(1)文法.