语法分析程序的设计与实现（LR 分析方法）

实验内容及要求

编写 LR 语法分析程序，实现对算术表达式的语法分析。要求所分析算数表达式由如下的文法产生。

E→ E+T | E–T | T

T→ T*F | T/F | F

F→ (E) | num

实验要求

在对输入的算术表达式进行分析的过程中，依次输出所采用的产生式。

实验方法要求

(1) 构造识别该文法所有活前缀的 DFA。

(2) 构造该文法的 LR 分析表。

(3) 编程实现算法 4.3，构造 LR 分析程序。

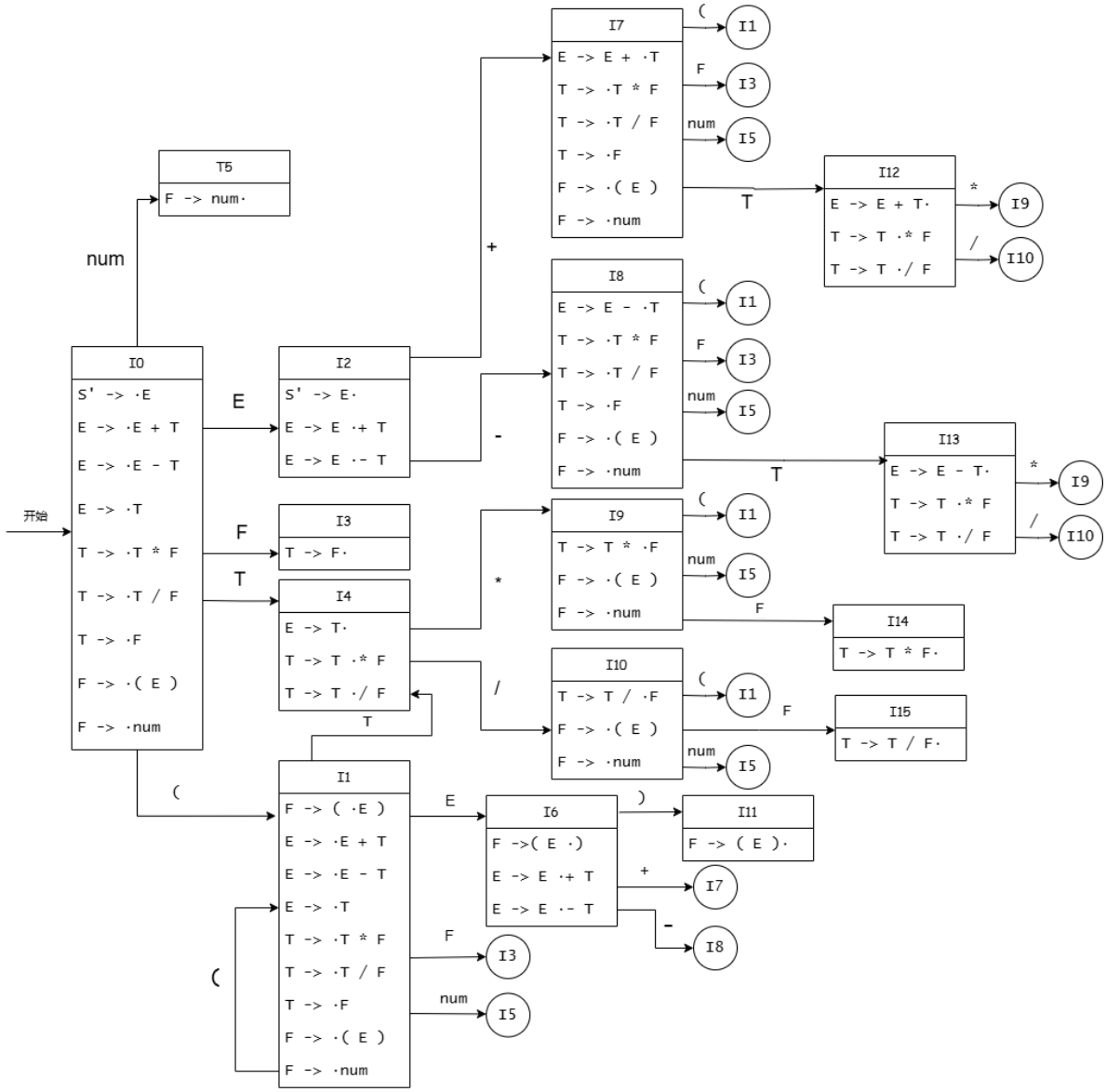
程序设计说明

首先对题目所给文法进行计算：

拓广文法如下：

```
S' -> E
E -> E + T
E -> E - T
E -> T
T -> T * F
T -> T/F
T -> F
F -> ( E )
F -> num
```

然后根据拓广文法构造LR(0)项目集规范族



由图可见，LR(0)文法存在移进-归约冲突，但是通过follow集可以向前看一个符号解决，因此文法为SLR(1)文法便足够完成任务，所以本实验本人构建的是SLR(1)文法。

	FOLLOW
S'	$
E	+, -,), $
T	*, /, +, -,), $
F	*, /, +, -,), $

预期分析表构造如下：

	+	-	*	/	(	)	num	$	S'	E	T	F
0					S1		S5			2	4	3
1					S1		S5			6	4	3
2	S7	S8						ACC				

	+	-	*	/	(	)	num	$	S'	E	T	F
3	R T->F	R T->F	R T->F	R T->F		R T->F		R T->F				
4	R E->T	R E->T	S9	S10		R E->T		R E->T				
5	R F->num	R F->num	R F->num	R F->num		R F->num		R F->num				
6	S7	S8				S11						
7					S1		S5				12	3
8					S1		S5				13	3
9					S1							14
10					S1		S5					15
11	R F->(E)	R F->(E)	R F->(E)	R F->(E)		R F->(E)		R F->(E)				
12	R E->E+T	R E->E+T	S9	S10		R E->E+T		R E->E+T				
13	R E->E-T	R E->E-T	S9	S10		R E->E-T		R E->E-T				
14	R T->T*F	R T->T*F	R T->T*F	R T->T*F		R T->T*F		R T->T*F				
15	R T->T/F	R T->T/F	R T->T/F	R T->T/F		R T->T/F		R T->T/F				

由上图我们可以看到自己计算的SLR(1)分析表没有冲突，可以识别题目所给语法。
接下来介绍程序的结构：

本程序的数据结构沿用了LL语法分析实验的数据结构，使用`Symbol`类储存文法中的各类符号，使用`Production`类储存文法的各类转换规则，`GrammarAnalyzer`类储存程序的主要类成员，包括符号，表达式，FIRST集，FOLLOW集，开始符以及分析表等，在这里不再做说明，在本程序中额外新增了`Item`类用来储存项目：

```
//项目类
class Item {
public:
    Production production;
    int dotPosition;

    Item(Production prod, int dot) : production(prod), dotPosition(dot) {}

    bool operator <(const Item& other) const {
        if(production.left.value != other.production.left.value) {
            return production.left.value < other.production.left.value;
        }
        if(dotPosition != other.dotPosition) {
            return dotPosition < other.dotPosition;
        }
        return production.right < other.production.right;
```

```
    }

    bool operator ==(const Item& other) const {
        return production.left.value == other.production.left.value
            && dotPosition == other.dotPosition
            && production.right == other.production.right;
    }
};
```

`production`用于储存项目中的表达式，dotPosition用于记录`·`的位置。
`ItemSet`类用于储存项目集：

```
class ItemSet {
public:
    set<Item> items;

    bool operator <(const ItemSet& other) const {
        return items < other.items;
    }
    bool operator ==(const ItemSet& other) {
        return items == other.items;
    }
};
```

通过`items`这个set来储存各项目集。

`closure`函数：
　　通过遍历输入表达式的右侧，对`·`右侧的符号进行判断，如果为非终结符，那么就在结果中查找，如果没有找到的话，就将以该非终结符为表达式左侧的表达式全部插入项目集中，并在表达式右侧开头加上`·`，代码实现如下：

```
ItemSet closure(ItemSet& itemset, vector<Production>& productions) {
    ItemSet result = itemset;
    bool changed = true;
    while(changed) {
        changed = false;
        for(auto& item : result.items) {
            auto rightSide = item.production.right;
            for(auto right : rightSide) {
                if(item.dotPosition < right.size()) {
                    Symbol B = right[item.dotPosition];
                    if(!B.isTerminal) {
                        for(auto& production : productions) {
                            if(production.left == B) {
                                Item newItem = Item(production, 0);
                                if(result.items.find(newItem) ==
result.items.end()) {
                                    result.items.insert(newItem);
                                    changed = true;
                                }
                            }
                        }
                    }
                }
```

```
            }
        }
    }
    return result;
}
```

`gotoSet`函数：

用于对输入的项目集和符号形成下一个项目集，通过遍历查找输入项目集中表达式的右侧`·`右侧的符号是否等于输入的符号X，若等于，则说明匹配成功，可以生成下一个项目集，那么就把匹配成功后`·`右移产生的新表达式插入到新的项目集中并对其调用closure函数对其求闭包，得到新的项目集，代码实现如下：

```
ItemSet gotoSet(ItemSet& itemset, Symbol X, vector<Production>& productions)
{
    ItemSet gotoSet;
    for(auto& item : itemset.items) {
        auto rightSide = item.production.right;
        for(auto right : rightSide) {
            if(item.dotPosition < right.size()) {
                if(right[item.dotPosition] == X) {
                    Production newProduction(item.production.left, {right});
                    Item newItem = Item(newProduction, item.dotPosition +
1);
                    gotoSet.items.insert(newItem);
                }
            }
        }
    }
    return closure(gotoSet, productions);
}
```

`generateDFA`函数：

通过调用上两个函数我们便可以生成可以识别输入文法的DFA，思路为首先生成拓广文法，为文法加上一个新的起始符，然后对项目集`I0`输入拓广文法的第一个表达式，然后对其求闭包，结果就是`I0`的全部内容，然后函数利用优先队列实现对状态集的广度搜索遍历，对每个项目集都调用`gotoSet`函数生成下一个项目集，最终遍历完成后就生成了可以识别所有活前缀的DFA，具体代码实现如下：

```
    void generateDFA() {
        //首先生成拓广文法
        symbols["S'"] = Symbol("S'",false);
        productions.emplace_back(symbols["S'"], vector<vector<Symbol>>
{{startSymbol}});
        startSymbol = symbols["S'"];

        //在状态0先输入拓广文法第一句，然后求闭包
        ItemSet startItemSet;
        startItemSet.items.insert(Item(productions.back(), 0));
        startItemSet = closure(startItemSet, productions);

        queue<ItemSet> itemQueue;
        itemQueue.push(startItemSet);
        itemSetMap[startItemSet] = 0;
        dfaStates.push_back(startItemSet);
```

```
        while(!itemQueue.empty()) {
            ItemSet currentSet = itemQueue.front();
            itemQueue.pop();
            int currentState = itemSetMap[currentSet];

            for(auto& s : symbols) {
                Symbol symbol = s.second;
                if(symbol.value == "ε" || symbol.value == "$") continue;
                ItemSet gotoSets = gotoSet(currentSet, symbol, productions);
                if(!gotoSets.items.empty()) {
                    if(itemSetMap.find(gotoSets) == itemSetMap.end()) {
                        itemSetMap[gotoSets] = dfaStates.size();
                        dfaStates.push_back(gotoSets);
                        itemQueue.push(gotoSets);
                    }
                    int nextState = itemSetMap[gotoSets];
                    cout << "状态" << currentState << "通过" << symbol.value
<< "转移到状态" << nextState << endl;
                }
            }
        }
        cout << "DFA生成完成" << endl;
    }
```

`generateParsingTable`函数：

本函数的功能是通过已经生成的DFA来生成SLR(1)分析表，首先介绍分析表的结构：

```
map<pair<int, Symbol>, tuple<string, int, Production>> parsingTable;
```

本分析表采用了`pair`对`tuple`的映射关系，`pair`表示表的行和列，`tuple`的第一个元素表示相应的操作，若为`S`则是Shift操作，若为`R`则是归约操作，若为`ACC`则表明识别成功，第二个元素则为下一步跳转的状态，第三个元素则是进行归约时归约对应的生成式。程序的思路为对项目集中的每个表达式的右侧都进行遍历，如果发现`·`已经在表达式末尾，则说明该表达式已经识别完成需要进行归约（若为开始表达式则说明已经识别完成，tuple第一个元素赋为ACC即可），那么根据课本上步骤需要对该表达式左侧非终结符的FOLLOW集进行遍历，对每个属于该FOLLOW的元素，查找分析表对应位置，将tuple第一个元素赋值为R，第二个元素赋值为-1即可，再将表达式插入到第三个元素中，由于是插入所有属于FOLLOW集的元素对应列，那么相当于对LR(0)的自动机，如果发生冲突，那么查找`·`后的终结符是否∈对应FOLLOW集的查询操作，如果不属于，那么tuple插入正常，冲突可以解决，如果属于，那么该项就无法赋值，则说明该冲突无法解决，不是SLR(1)文法；若`·`不在表达式末尾，则根据`·`后符号对tuple进行赋值，代码实现如下：

```
    void generateParsingTable() {
        for(int state = 0; state < dfaStates.size(); state++) {
            ItemSet itemSet = dfaStates[state];
            for(auto& item : itemSet.items) {
                auto rightSide = item.production.right;
                for(auto right : rightSide) {
                    if(item.dotPosition == right.size()) {
                        for(auto s : follow[item.production.left]) {
                            if(parsingTable.find({state, s}) !=
parsingTable.end()) {
                                cerr << "文法不是SLR文法" << endl;
```

```
                                return;
                            }
                            Production newProduction(item.production.left, {right});
                            parsingTable[{state, s}] = {"R", -1, newProduction};
                        }
                    }
                    else {
                        Symbol nextSymbol = right[item.dotPosition];
                        if(nextSymbol.isTerminal) {
                            ItemSet gotoSets = gotoSet(itemSet, nextSymbol, productions);
                            if(itemSetMap.find(gotoSets) != itemSetMap.end()) {
                                int nextState = itemSetMap[gotoSets];
                                parsingTable[{state, nextSymbol}] = {"S", nextState, Production()};
                            }
                            else {
                                cerr << "文法不是SLR文法" << endl;
                                return;
                            }
                        }
                        else {
                            int nextState = itemSetMap[gotoSet(itemSet, nextSymbol, productions)];
                            parsingTable[{state, nextSymbol}] = {"G", nextState, Production()};
                        }
                    }
                }
                if(item.production.left == startSymbol && item.dotPosition == item.production.right[0].size()) {
                    parsingTable[{state, symbols["$"]}] = {"ACC", -1, Production()};
                }
            }
        }
    }
```

`parsing`函数：

本函数是对算法4.3的实现，首先初始化，将`$`和初始符号`S'`依次压入符号栈中，然后将pos指向输入缓冲区的第一个符号，初始状态设为0，通过输入符号和状态在分析表中找到相应的操作，若为`S`则进行移入操作，将符号压入符号栈中，分析表中对应状态压入状态栈中，若为`R`则进行归约操作，从栈顶弹出`production.right.size()`个符号，然后把表达式左侧的符号压入符号栈中，并将分析表中状态压入状态栈，若为`ACC`则说明匹配成功，语法分析完成，具体代码实现如下：

```
void parsing() {
        string input;
        cout << "请输入要分析的字符串: ";
        getline(cin, input);
        vector<Symbol> inputSymbols;
        istringstream iss(input);
```

```
        string s;
        while (iss >> s) {  // Read each symbol separated by spaces
            if (symbols.find(s) == symbols.end()) {
                cerr << "输入包含未知符号 " << s << endl;
                return;
            }
            inputSymbols.push_back(symbols[s]);
        }

        vector<int> stateStack = {0};
        vector<Symbol> symbolStack = {symbols["$"]};
        int pos = 0;

        while(pos < inputSymbols.size()) {
            int currentState = stateStack.back();
            Symbol currentSymbol = inputSymbols[pos];
            if(parsingTable.find({currentState, currentSymbol}) ==
parsingTable.end()) {
                cerr << "无法识别的输入" << endl;
                return;
            }
            auto action = parsingTable[{currentState, currentSymbol}];
            if(get<0>(action) == "S") {
                stateStack.push_back(get<1>(action));
                symbolStack.push_back(currentSymbol);
                pos++;
                cout << "移进 " << currentSymbol.value << " Shift " << get<1>
(action) << endl;
            }
            else if(get<0>(action) == "R") {
                Production production = get<2>(action);
                for(int i = 0; i < production.right[0].size(); i++) {
                    stateStack.pop_back();
                    symbolStack.pop_back();
                }
                Symbol left = production.left;
                currentState = stateStack.back();
                stateStack.push_back(get<1>(parsingTable[{currentState,
left}]));
                symbolStack.push_back(left);
                cout << "规约 " << production.left.value << "->";
                for(auto& s : production.right[0]) {
                    cout << s.value;
                }
                cout << endl;
            }
            else if(get<0>(action) == "ACC") {
                cout << "分析成功" << endl;
                return;
            }
            else {
                cerr << "无法识别的输入" << endl;
                return;
            }
        }
```

```
59
60      }
```

测试结果与分析

测试一

1.1 输入

```
E,T,F
+,*,(,),id
E
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
.
id + id * id $
```

1.2 输出

```
状态0通过(转移到状态1
状态0通过E转移到状态2
状态0通过F转移到状态3
状态0通过T转移到状态4
状态0通过id转移到状态5
状态1通过(转移到状态1
状态1通过E转移到状态6
状态1通过F转移到状态3
状态1通过T转移到状态4
状态1通过id转移到状态5
状态2通过+转移到状态7
状态4通过*转移到状态8
状态6通过)转移到状态9
状态6通过+转移到状态7
状态7通过(转移到状态1
状态7通过F转移到状态3
状态7通过T转移到状态10
状态7通过id转移到状态5
状态8通过(转移到状态1
状态8通过F转移到状态11
状态8通过id转移到状态5
状态10通过*转移到状态8
DFA生成完成
请输入要分析的字符串: id + id * id $
移进 id Shift 5
规约 F->id
规约 T->F
规约 E->T
移进 + Shift 7
移进 id Shift 5
规约 F->id
规约 T->F
移进 * Shift 8
移进 id Shift 5
```

```
规约 F->id
规约 T->T*F
规约 E->E+T
分析成功
```

1.3 分析

本输入为考察程序初步运行的正确与否，采用的输入是课本例4.2的输入，该文法与题目要求文法类似，但是更简单一点，以此作为对照来初步检验程序的正确性，经过与课本上答案的对照可以看出程序运行正常，结果正确。

测试二

2.1 输入

```
E,L
(,),a
E
E -> ( L ) | a
L -> E L | E
.
( ( a ) a ( a a ) ) $
```

2.2 输出

```
状态0通过(转移到状态1
状态0通过E转移到状态2
状态0通过a转移到状态3
状态1通过(转移到状态1
状态1通过E转移到状态4
状态1通过L转移到状态5
状态1通过a转移到状态3
状态4通过(转移到状态1
状态4通过E转移到状态4
状态4通过L转移到状态6
状态4通过a转移到状态3
状态5通过)转移到状态7
DFA生成完成
请输入要分析的字符串: ( ( a ) a ( a a ) ) $
移进 ( Shift 1
移进 ( Shift 1
移进 a Shift 3
规约 E->a
规约 L->E
移进 ) Shift 7
规约 E->(L)
移进 a Shift 3
规约 E->a
移进 ( Shift 1
移进 a Shift 3
规约 E->a
移进 a Shift 3
规约 E->a
规约 L->E
```

```
规约 L->EL
移进 ) Shift 7
规约 E->(L)
规约 L->E
规约 L->EL
规约 L->EL
移进 ) Shift 7
规约 E->(L)
分析成功
```

2.3 分析

本次测试为从网络上找到的一道例题，其与课本上的习题4.11题目相同，因此采用这道题进行测试以验证程序能够普遍性地识别所有SLR(1)文法，经过校验对比

步骤	栈	输　入	动　作
(1)	State：0 Symbol：—	((a)a(aa))$	Shift 2
(2)	State：0 2 Symbol：—(	(a)a(aa))$	Shift 2
(3)	State：0 2 2 Symbol：—((	a)a(aa))$	Shift 3
(4)	State：0 2 2 3 Symbol：—((a	)a(aa))$	Reduce $E\rightarrow a$
(5)	State：0 2 2 5 Symbol：—((E	)a(aa))$	Reduce $L\rightarrow E$
(6)	State：0 2 2 4 Symbol：—((L	)a(aa))$	Shift 6
(7)	State：0 2 2 4 6 Symbol：—((L)	a(aa))$	Reduce $E\rightarrow(L)$
(8)	State：0 2 5 Symbol：—(E	a(aa))$	Shift 3
(9)	State：0 2 5 3 Symbol：—(E a	(aa))$	Reduce $E\rightarrow a$
(10)	State：0 2 5 5 Symbol：—(E E	(aa))$	Shift 2
(11)	State：0 2 5 5 2 Symbol：—(E E (	aa))$	Shift 3
(12)	State：0 2 5 5 2 3 Symbol：—(E E (a	a))$	Reduce $E\rightarrow a$
(13)	State：0 2 5 5 2 5 Symbol：—(E E (E	a))$	Shift 3

步骤	栈	输　入	动　作
(14)	State：0 2 5 5 2 5 3 Symbol：—(E E (E a	))$	Reduce $E\rightarrow a$
(15)	State：0 2 5 5 2 5 5 Symbol：—(E E (E E	))$	Reduce $L\rightarrow E$
(16)	State：0 2 5 5 2 5 7 Symbol：—(E E (E L	))$	Reduce $L\rightarrow EL$
(17)	State：0 2 5 5 2 4 Symbol：—(E E (L	))$	Shift 6
(18)	State：0 2 5 5 2 4 6 Symbol：—(E E (L)	)$	Reduce $E\rightarrow(L)$
(19)	State：0 2 5 5 5 Symbol：—(E E E	)$	Reduce $L\rightarrow E$
(20)	State：0 2 5 5 7 Symbol：—(E E L	)$	Reduce $L\rightarrow EL$
(21)	State：0 2 5 7 Symbol：—(E L	)$	Reduce $L\rightarrow EL$
(22)	State：0 2 4 Symbol：—(L	)$	Shift 6

	Symbol：$-(L$		
(23)	State：0 2 4 6 Symbol：$-(L)$	$	Reduce $E\rightarrow(L)$
(24)	State：0 1 Symbol：$-E$	$	接受

可以看到程序执行正常，语法分析步骤与答案相同，结果正确。

测试三

3.1 输入

```
E,T,F
+,-,*,/,(,),num
E
E -> E + T | E - T | T
T -> T * F | T / F | F
F -> ( E ) | num
.
num * ( num + ( num * ( num - ( num / num + ( num - num / num ) ) ) ) ) / num
- num + num $
```

3.2 输出

```
状态0通过(转移到状态1
状态0通过E转移到状态2
状态0通过F转移到状态3
状态0通过T转移到状态4
状态0通过num转移到状态5
状态1通过(转移到状态1
状态1通过E转移到状态6
状态1通过F转移到状态3
状态1通过T转移到状态4
状态1通过num转移到状态5
状态2通过+转移到状态7
状态2通过-转移到状态8
状态4通过*转移到状态9
状态4通过/转移到状态10
状态6通过)转移到状态11
状态6通过+转移到状态7
状态6通过-转移到状态8
状态7通过(转移到状态1
状态7通过F转移到状态3
状态7通过T转移到状态12
状态7通过num转移到状态5
状态8通过(转移到状态1
状态8通过F转移到状态3
状态8通过T转移到状态13
状态8通过num转移到状态5
状态9通过(转移到状态1
状态9通过F转移到状态14
状态9通过num转移到状态5
状态10通过(转移到状态1
状态10通过F转移到状态15
状态10通过num转移到状态5
```

```
状态12通过*转移到状态9
状态12通过/转移到状态10
状态13通过*转移到状态9
状态13通过/转移到状态10
DFA生成完成
请输入要分析的字符串: num * ( num + ( num * ( num - ( num / num + ( num - num / num ) ) ) ) ) / num - num + num $
移进 num Shift 5
规约 F->num
规约 T->F
移进 * Shift 9
移进 ( Shift 1
移进 num Shift 5
规约 F->num
规约 T->F
规约 E->T
移进 + Shift 7
移进 ( Shift 1
移进 num Shift 5
规约 F->num
规约 T->F
移进 * Shift 9
移进 ( Shift 1
移进 num Shift 5
规约 F->num
规约 T->F
规约 E->T
移进 - Shift 8
移进 ( Shift 1
移进 num Shift 5
规约 F->num
规约 T->F
移进 / Shift 10
移进 num Shift 5
规约 F->num
规约 T->T/F
规约 E->T
移进 + Shift 7
移进 ( Shift 1
移进 num Shift 5
规约 F->num
规约 T->F
规约 E->T
移进 - Shift 8
移进 num Shift 5
规约 F->num
规约 T->F
移进 / Shift 10
移进 num Shift 5
规约 F->num
规约 T->T/F
规约 E->E-T
移进 ) Shift 11
规约 F->(E)
规约 T->F
```

```
规约 E->E+T
移进 ) Shift 11
规约 F->(E)
规约 T->F
规约 E->E-T
移进 ) Shift 11
规约 F->(E)
规约 T->T*F
规约 E->T
移进 ) Shift 11
规约 F->(E)
规约 T->F
规约 E->E+T
移进 ) Shift 11
规约 F->(E)
规约 T->T*F
移进 / Shift 10
移进 num Shift 5
规约 F->num
规约 T->T/F
规约 E->T
移进 - Shift 8
移进 num Shift 5
规约 F->num
规约 T->F
规约 E->E-T
移进 + Shift 7
移进 num Shift 5
规约 F->num
规约 T->F
规约 E->E+T
分析成功
```

3.3 分析

本次输入为题目要求识别文法，所以也为最终测试，在输出中可以看到一共有15个项目集，同时其项目集规范族关系也与预期相同，最终识别过程也完全正确，程序运行正常，符合要求。

总结

本次实验我完成了LR语法分析程序，对题目所给文法进行了识别，由于数据结构都直接沿用了LL文法分析实验的代码，因此本次实验做起来轻松很多，在本次实验中我完成了对求项目闭包函数，生成下一项目集函数，构造SLR分析表以及算法4.2的实现，加深了我对LR语法分析的印象和理解，在实验中我也多次对不同文法的DFA和分析表进行了手工验算，使我对课程内容更加熟悉，也纠正了我对LR语法分析的一些错误认知，收获颇丰。