编译原理实验专题三：LL(1)语法分析设计原理与实现

一、目标任务

编程实现下面的文法的 LL(1)分析过程： G[S]: S→V=E E→TE′ E′→ATE′|ε T→FT′ T′→MFT′|ε F→(E)|i A→+|- M→*|/ V→i

G[S]:S→V=E
E→TE′
E′ →ATE′ | ε
T→FT′
T′ →MFT′ | ε
F→ (E)|i
A→+|-
M→*|/
V→i

二、设计说明

2.1 背景介绍

在实验一、实验二所建工程的基础上继续进行实验。

> 背景介绍：实验一我完成了词法分析器，程序能够自动解析正则表达式定义的词源并自动生成NFA以完成词法分析。实验二我完成了扩展的巴克斯淖尔范式的解析，以及最左公因子提取和左递归的消除，并初步实现了First集、Follow集和Select集的求解以及LL1文法的判定。实验三（本次实验），我将在前两次实验的基础上实现LL1预测分析表法的语法分析。

2.2 词法设计

首先根据实验要求，设计词法分析器所需要的定义如下（`lab3.lex`）：

```
PATTERN ${
    BLANK       \s+
    IDENTIFIER [\a_][\w]*
    SEPARATOR  [\+\-\*/]
$}

IGNORE ${
    BLANK
$}
```

该词法定义描述了，词法分析器可以识别空白、标识符和分隔符。词法分析器将会把读入文件中符合正则表达式所识别的单词封装成tokens序列传递给后续文法分析程序。

2.3 文法设计

而后根据实验要求，设计文法分析器所需要的定义如下（`lab3.stx`）：

```
GRAMMAR ${
    S*  ::= $V `=` E;
    E   ::= T E';
    E'  ::= A T E' | \e ;
    T   ::= F T';
    T'  ::= M F T' | \e ;
    F   ::= `(` E `)` | $V;
    A   ::= `+` | `-`;
    M   ::= `*` | `/`;
$}

MAPPING ${
    $V     -->     @IDENTIFIER ;
$}
```

该文法定义使用扩展的巴克斯淖尔范式（由于题目所给文法过于简单，且为了方便与正确答案对比，此处刻意没有使用EBNF的高级语法）定义了题目所给文法（具体定义说明请参见实验二报告）。

值得注意的是，文法额外定义了将词法分析程序所识别的`IDENTIFIER`类型映射到`$V`上，这样便实现了将任意标识符作为运算对象`i`的替代，而不会使文法分析器只能识别`i`。

2.4 测试用例

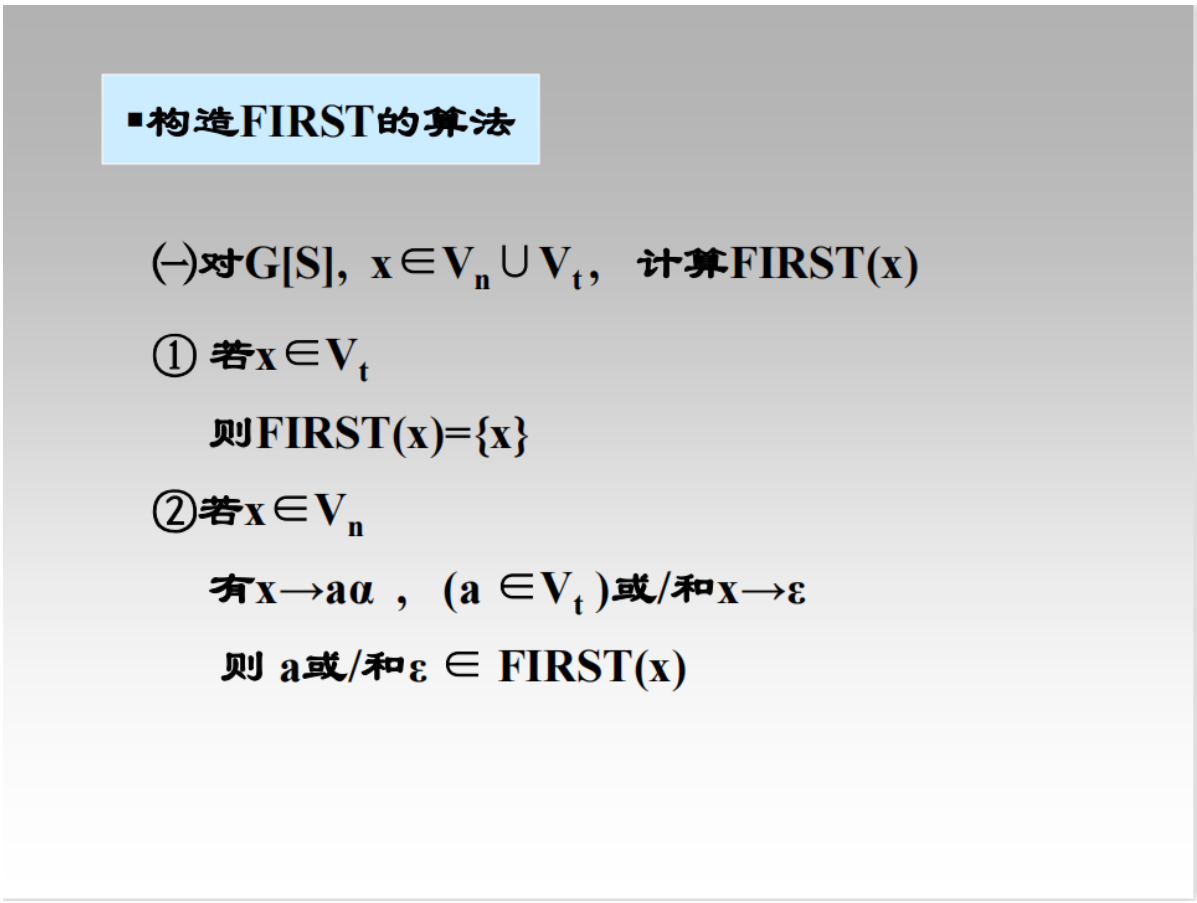
根据2.3所述，我的分析程序可以自动完成词法类型到文法终结符类型的映射，故此特意将测试程序中的`i`用一些合法的标识符代替，参考如下（`lab3.txt`）：

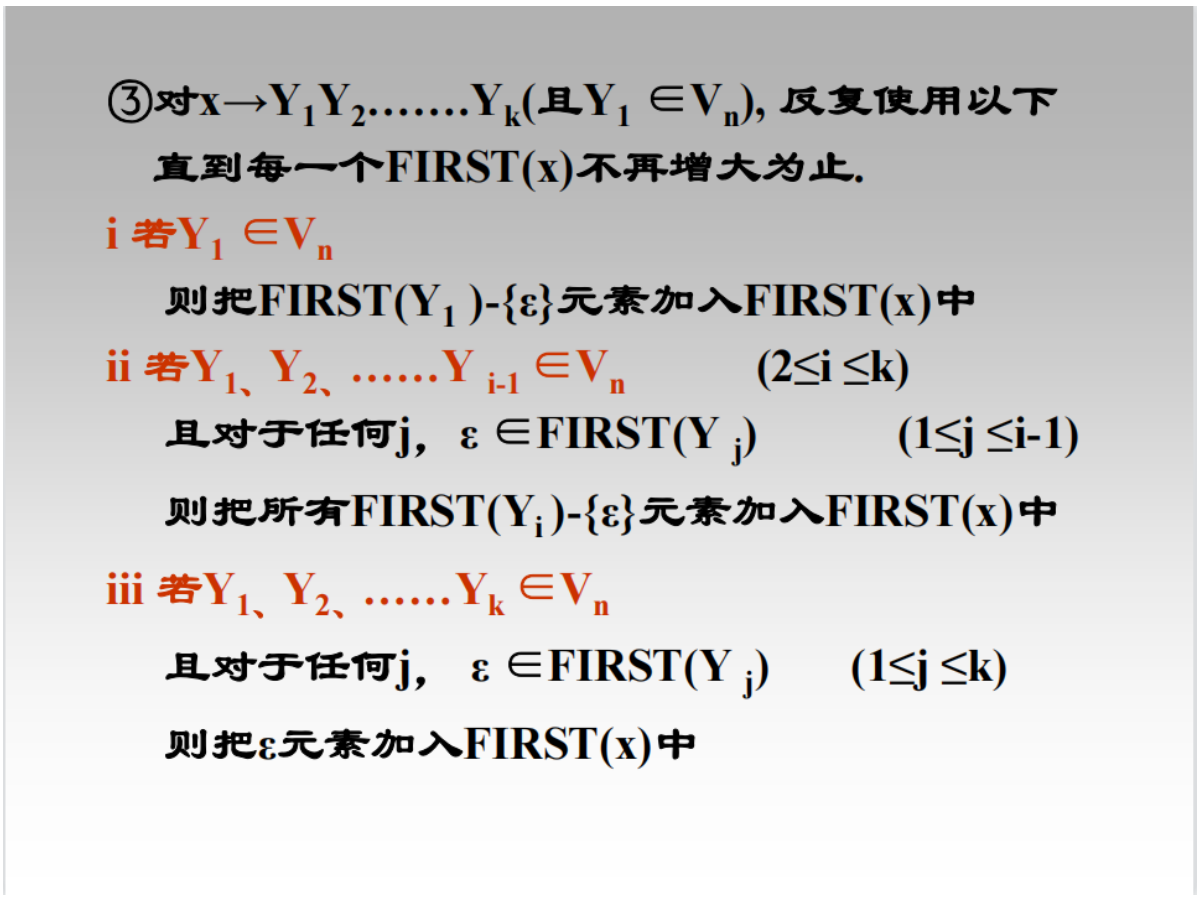
```
res=a*b+(c/d-e)
```

三、实验步骤

3.1 First集、Follow集的求解

原理解析：





实现代码：

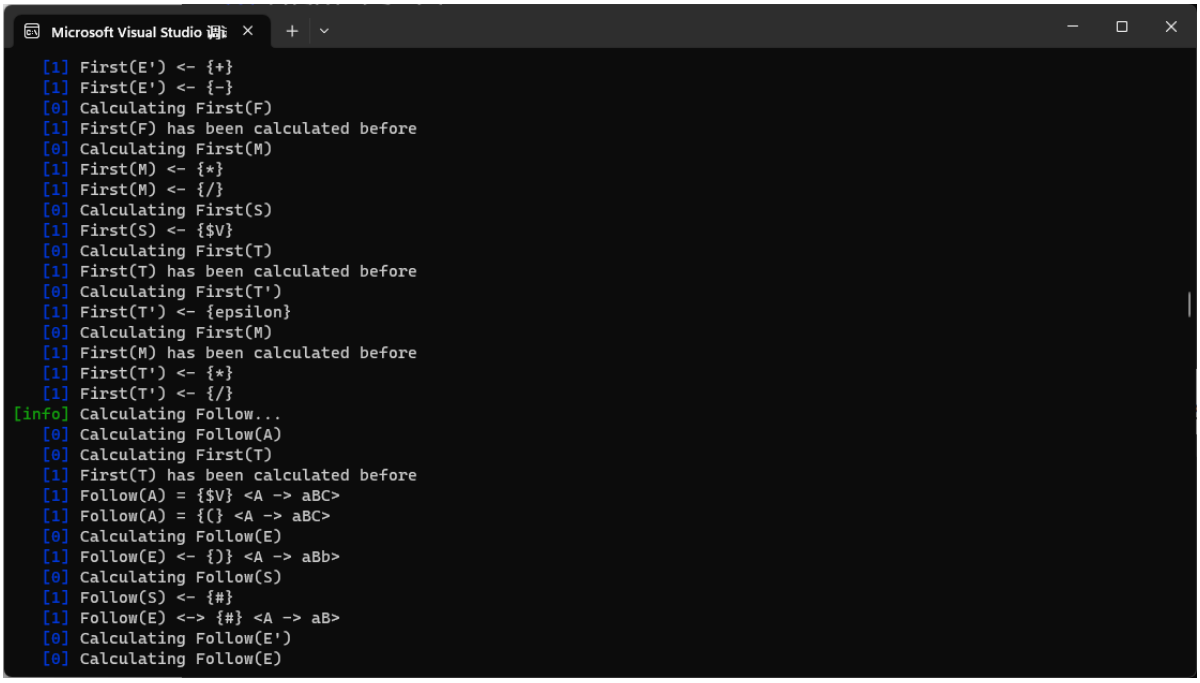
```
set<term> Grammar::calcFirstOf(term t)
{
    debug(0) << "Calculating First(" << t << ")" << endl;
    if (first[t].size() > 0)
    {
        debug(1) << "First(" << t << ") has been calculated before" <<
endl;
        return first[t]; // 已经计算过
    }
    // 对于每个产生式，计算first集
    for (auto &right : rules[t])
    {
        // 空产生式，加入epsilon
        if (right.size() == 0)
        {
            first[t].insert(EPSILON);
            debug(1) << "First(" << t << ") <- {epsilon}" << endl;
        }
        else
        {
            // 对于产生式右部的每个符号
            bool allHaveEpsilon = true;
            for (auto &symbol : right)
            {
                if (_find(terminals, symbol))
                {
                    // 终结符，直接加入first集
                    first[t].insert(symbol);
```

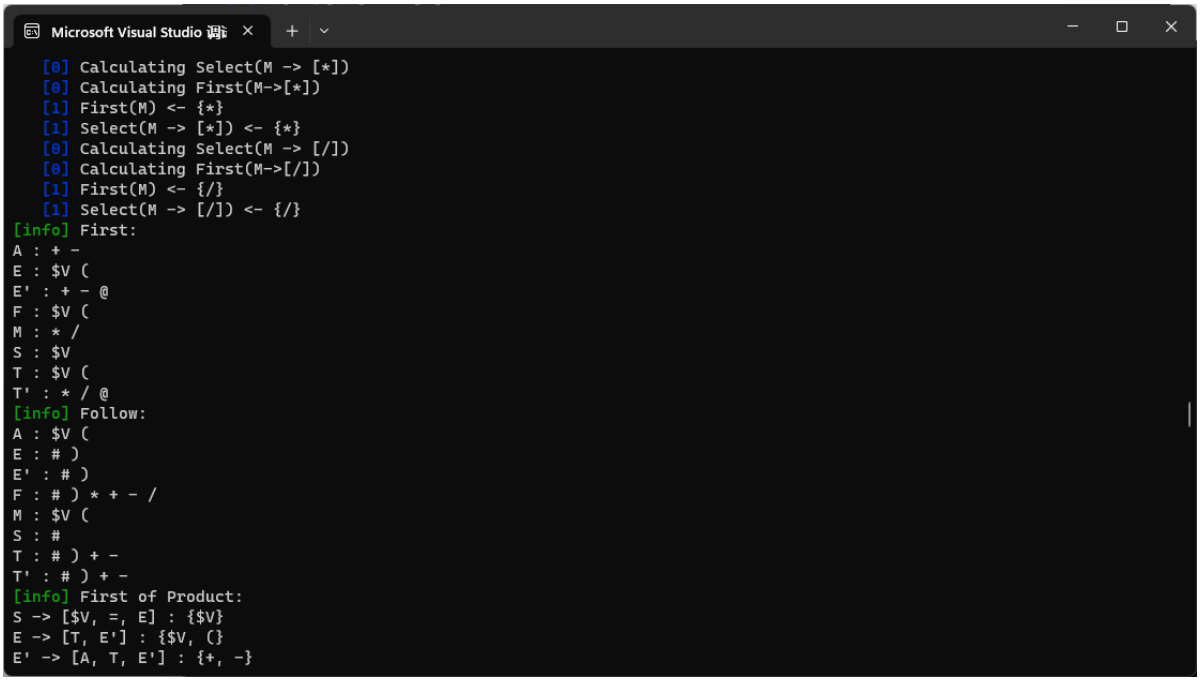
```
                    debug(1) << "First(" << t << ") <- {" << symbol << "}"
<< endl;
                    allHaveEpsilon = false;
                    break; // 终结符后面的符号不再计算
                }
                else
                {
                    set<term> resFirst = calcFirstOf(symbol);
                    set<term> tmpFirst = resFirst;
                    tmpFirst.erase(EPSILON);
                    first[t].insert(tmpFirst.begin(), tmpFirst.end());
                    for (auto &f : tmpFirst)
                    {
                        debug(1) << "First(" << t << ") <- {" << f << "}"
<< endl;
                    }
                    if (!_find(resFirst, EPSILON))
                    {
                        // 该符号的first集不含epsilon，后面的符号不再计算
                        allHaveEpsilon = false;
                        break;
                    }
                }
            }
            if (allHaveEpsilon)
            {
                first[t].insert(EPSILON);
                debug(1) << "First(" << t << ") <- {epsilon}" << endl;
            }
        }
    }
    return first[t];
}

set<term> Grammar::calcFollowOf(term t)
{
    debug(0) << "Calculating Follow(" << t << ")" << endl;
    if (follow[t].size() > 0)
    {
        debug(1) << "Follow(" << t << ") has been calculated before" <<
endl;
        return follow[t]; // 已经计算过
    }
    if (t == startTerm)
    {
        follow[t].insert(SYM_END);
        debug(1) << "Follow(" << t << ") <- {#}" << endl;
    }
    for (auto &rule : rules)
    {
        for (auto &right : rule.second)
        {
            auto symIt = right.begin();
            symIt = find(symIt, right.end(), t);
            while (symIt != right.end())
            {
```

```
81                  if (symIt == right.end() - 1)
82                  {
83                      // A -> aB
84                      if (rule.first != t)
85                      {
86                          set<term> resFollow = calcFollowOf(rule.first);
87                          follow[t].insert(resFollow.begin(),
resFollow.end());
88                          for (auto &f : resFollow)
89                          {
90                              debug(1) << "Follow(" << t << ") <-> {" << f <<
"} <A -> aB>" << endl;
91                          }
92                      }
93                  }
94                  else if (_find(terminals, *(symIt + 1)))
95                  {
96                      // A -> aBb
97                      follow[t].insert(*(symIt + 1));
98                      debug(1) << "Follow(" << t << ") <- {" << *(symIt + 1)
<< "} <A -> aBb>" << endl;
99                  }
100                 else
101                 {
102                     // A -> aBC
103                     set<term> resFirst;
104                     for (auto tmpIt = symIt + 1; tmpIt != right.end();
tmpIt++)
105                     {
106                         resFirst = calcFirstOf(*tmpIt);
107                         set<term> tmpFirst = resFirst;
108                         tmpFirst.erase(EPSILON);
109                         follow[t].insert(tmpFirst.begin(), tmpFirst.end());
110                         for (auto &f : tmpFirst)
111                         {
112                             debug(1) << "Follow(" << t << ") = {" << f <<
"} <A -> aBC>" << endl;
113                         }
114                         if (!_find(resFirst, EPSILON))
115                         {
116                             break;
117                         }
118                     }
119                     if (resFirst.size() == 0 || _find(resFirst, EPSILON))
120                     {
121                         // A -> aBC, First(C) 含有epsilon
122                         if (rule.first != t)
123                         {
124                             set<term> resFollow = calcFollowOf(rule.first);
125                             follow[t].insert(resFollow.begin(),
resFollow.end());
126                             for (auto &f : resFollow)
127                             {
128                                 debug(1) << "Follow(" << t << ") = {" << f
<< "} <A -> aBC(e)>" << endl;
129                             }
```

```
                        }
                    }
                }
                symIt = find(symIt + 1, right.end(), t);
            }
        }
    }
    return follow[t];
}
```

实验结果：

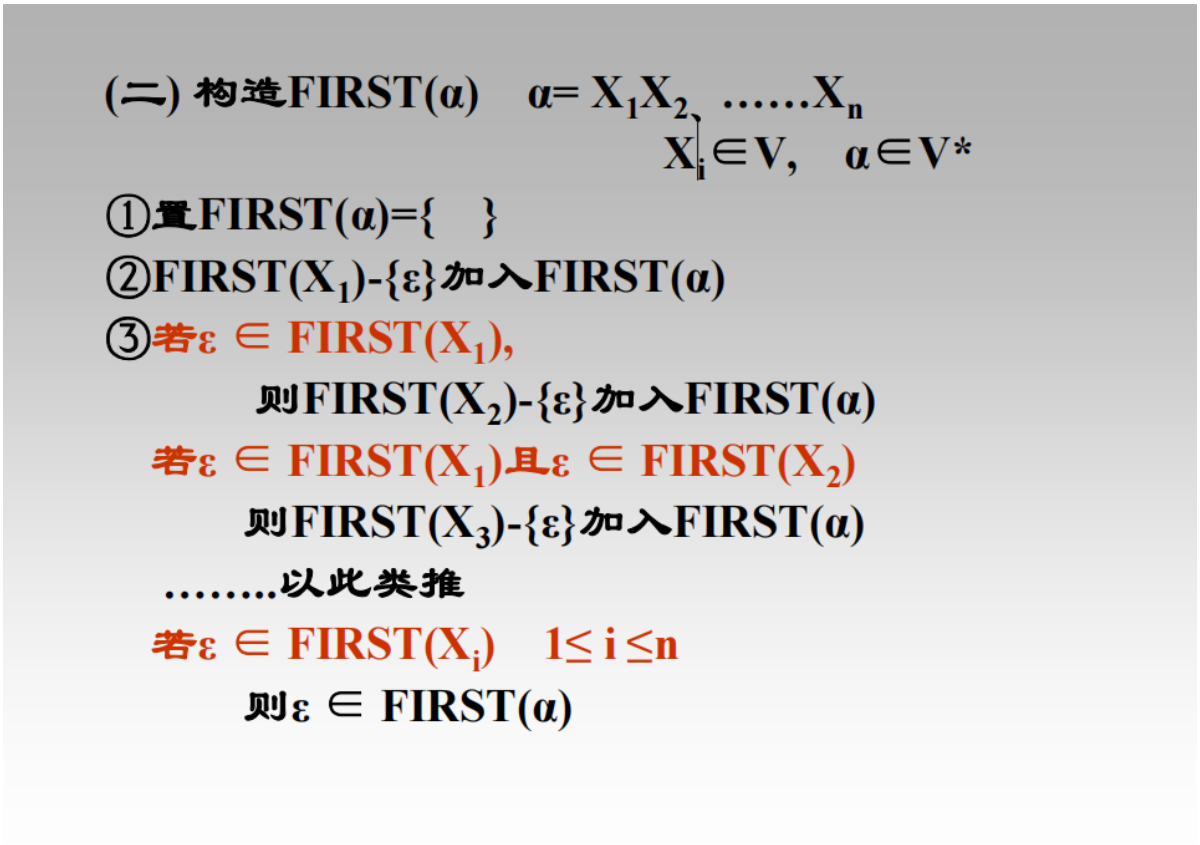




3.2 Select集的求解和LL1文法的判定

原理解析：

在编译原理中，Select集是用于构建LL(1)分析表的关键概念。Select集的本质是在构造自顶向下的语法分析器时，用于预测输入串中的
下一个符号应该选择哪一条产生式进行推导。Select集帮助我们确定在某个非终结符的推导过程中，应该选择哪一条产生式来进行推导。
它与LL(1)预测分析表有密切关系，因为构建LL(1)预测分析表的过程就是计算各个产生式的Select集。

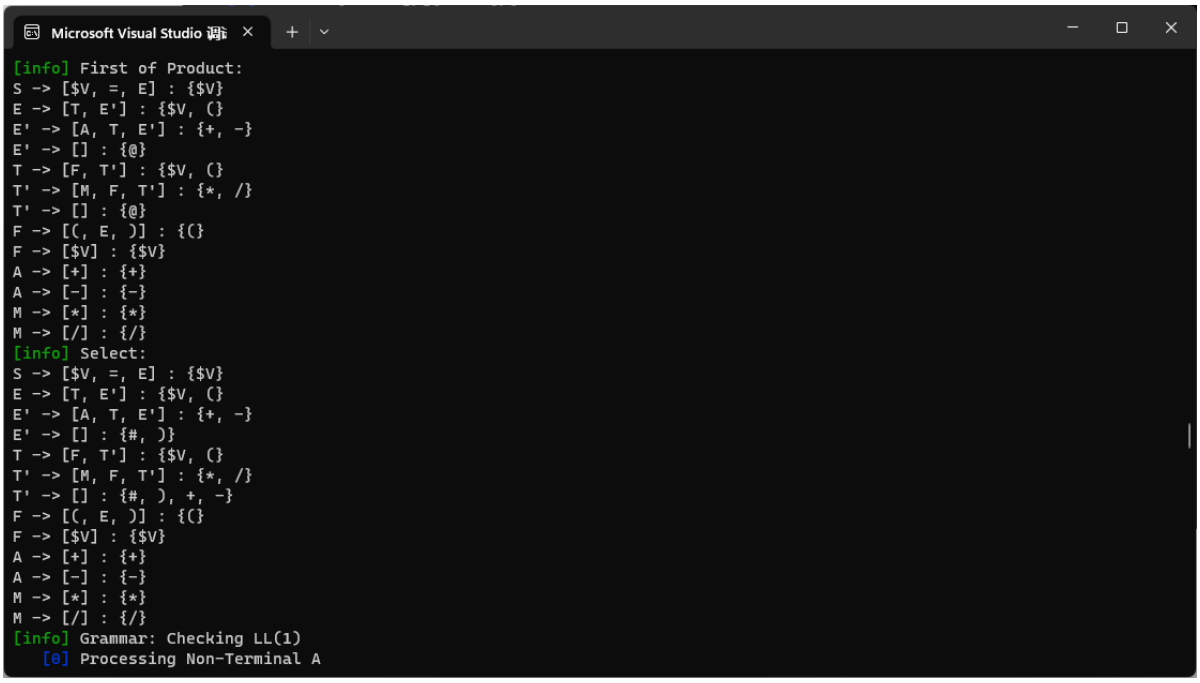


实现代码：

```
set<term> Grammar::calcFirstOf(production product)
{
    debug(0) << format("Calculating First($->$)", product.first,
vec2str(product.second)) << endl;
    if (firstP[product].size() > 0)
    {
        debug(1) << format(
            "First($->$) has been calculated before.\n",
            product.first,
            vec2str(product.second));
        return firstP[product]; // 已经计算过
    }
    set<term> resFirst;
    bool allHaveEpsilon = true;
    for (auto &symbol : product.second)
    {
        if (_find(terminals, symbol))
        {
            // 终结符，直接加入first集
            resFirst.insert(symbol);
            debug(1) << "First(" << product.first << ") <- {" << symbol <<
"}" << endl;
            allHaveEpsilon = false;
```

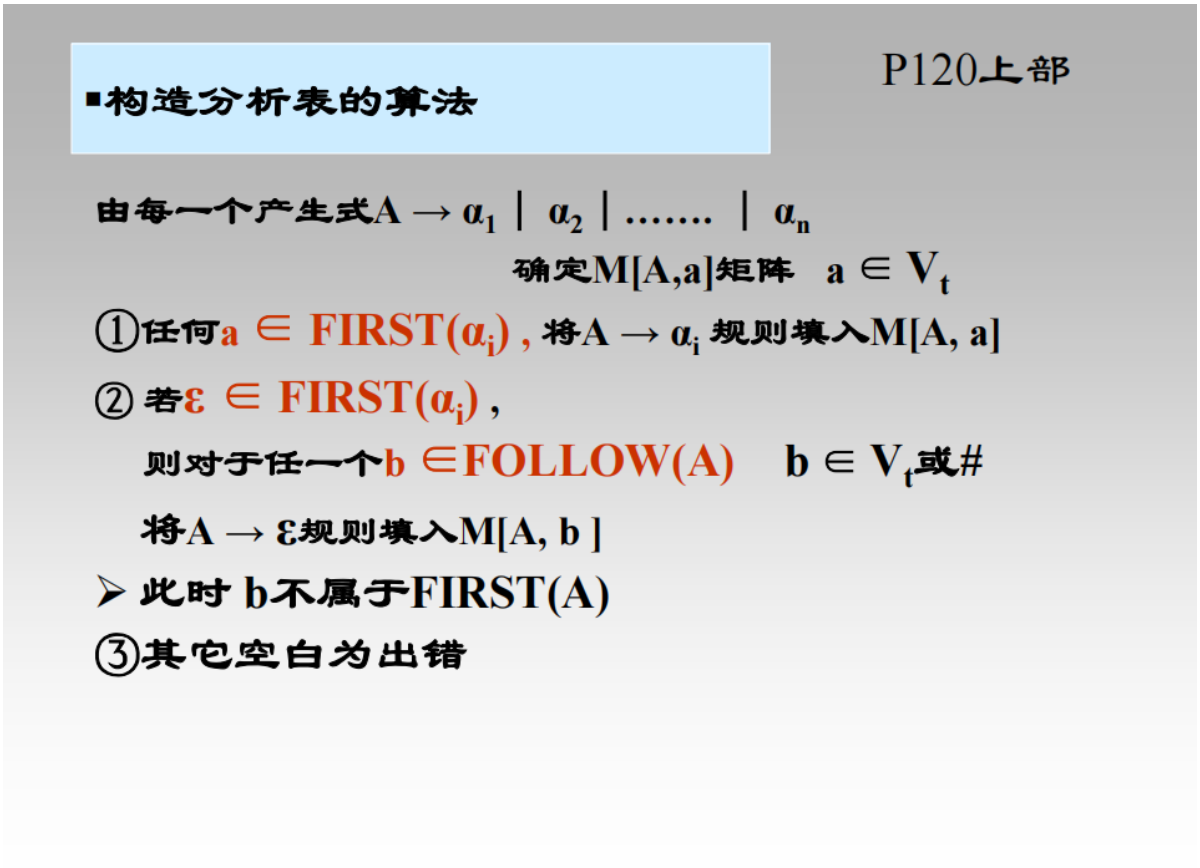
```
            break; // 终结符后面的符号不再计算
        }
        else
        {
            set<term> tmpFirst = calcFirstOf(symbol);
            resFirst.insert(tmpFirst.begin(), tmpFirst.end());
            debug(1) << "First(" << product.first << ") <- " <<
container2str(tmpFirst) << endl;
            if (!_find(tmpFirst, EPSILON))
            {
                // 该符号的first集不含epsilon，后面的符号不再计算
                allHaveEpsilon = false;
                break;
            }
        }
    }
    if (allHaveEpsilon)
    {
        resFirst.insert(EPSILON);
        debug(1) << "First(" << product.first << ") <- { epsilon }" << endl;
    }
    firstP[product] = resFirst;
    return resFirst;
}

set<term> Grammar::calcSelectOf(production product)
{
    debug(0) << "Calculating Select(" << product.first;
    debug_u(0) << " -> " << vec2str(product.second) << ")" << endl;
    set<term> resSelect;
    set<term> resFirst = calcFirstOf(product);
    set<term> tmpFirst = resFirst;
    tmpFirst.erase(EPSILON); // First(A) - {epsilon}
    resSelect.insert(tmpFirst.begin(), tmpFirst.end());
    debug(1) << "Select(" << product.first << " -> " <<
vec2str(product.second);
    debug_u(1) << ") <- " << set2str(tmpFirst) << endl;
    if (_find(resFirst, EPSILON))
    {
        set<term> resFollow = calcFollowOf(product.first);
        resSelect.insert(resFollow.begin(), resFollow.end());
        debug(1) << "Select(" << product.first << " -> " <<
vec2str(product.second);
        debug_u(1) << ") <- " << set2str(resFollow) << endl;
    }
    return resSelect;
}
```

实验结果：



3.3 LL1预测分析表的构建

原理解析：



实现代码：

```
#define _find(s, t) (s.find(t) != s.end())

void PredictiveTableAnalyzer::calcPredictTable()
{
    for (auto pdt : grammar.products)
    {
```

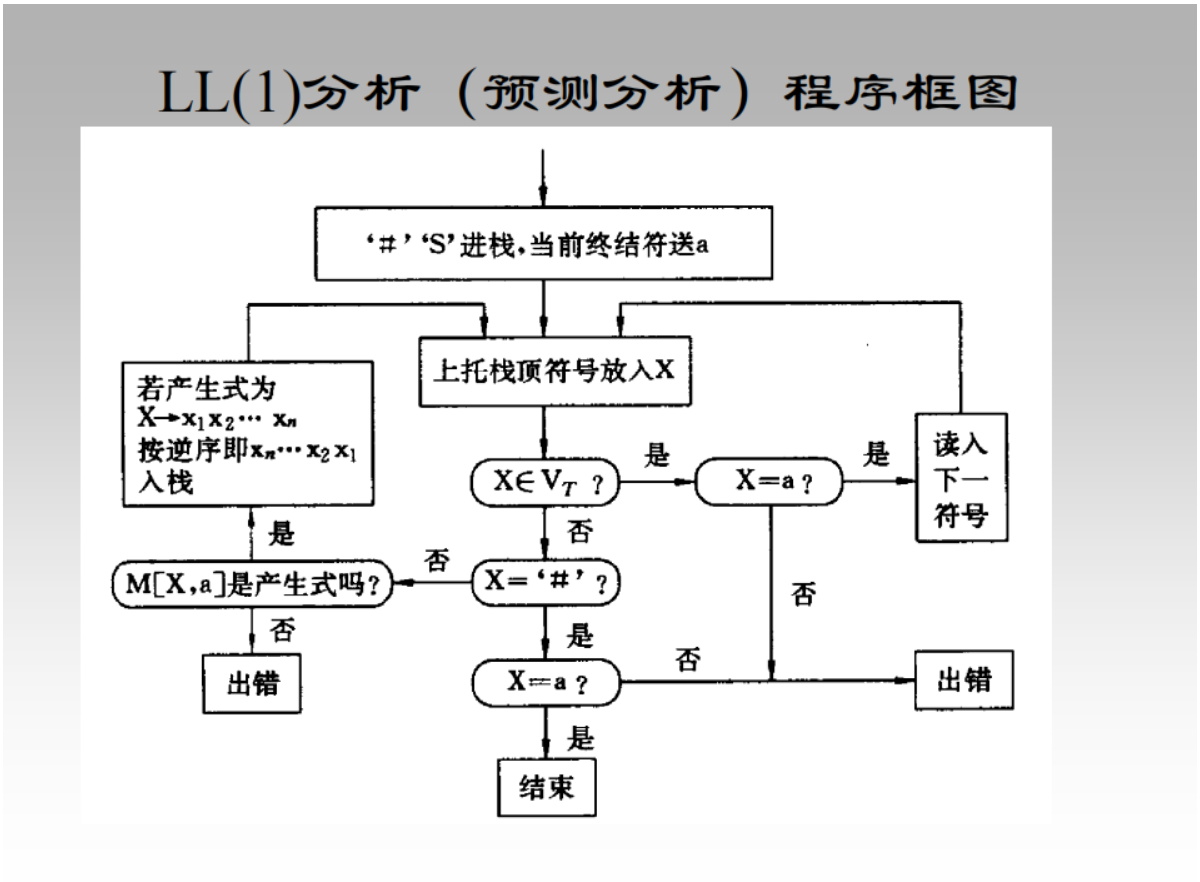
```
            auto first = grammar.firstP[pdt];
            for (auto t : first)
            {
                if (t != EPSILON)
                {
                    predict[pdt.first][t] = pdt.second;
                }
            }
            if (_find(first, EPSILON))
            {
                auto follow = grammar.follow[pdt.first];
                for (auto t : follow)
                {
                    if (_find(grammar.terminals, t))
                        predict[pdt.first][t] = pdt.second;
                }
            }
        }
}
```

实验结果：

```
[info] Predictive Table:
| Non-Term | #          | $V         | (          | )          | *          | +          | -          | /          | =          |
| A        |            |            |            |            |            | A -> [+]   | A -> [-]   |            |            |
| E        |            | E -> [T, E'] | E -> [T, E'] |            |            |            |            |            |            |
| E'       | E' -> []   |            |            | E' -> []   |            | E' -> [A, T, E'] | E' -> [A, T, E'] |            |            |
| F        |            | F -> [$V]  | F -> [(, E, )] |            |            |            |            |            |            |
| M        |            |            |            |            | M -> [*]   |            |            | M -> [/]   |            |
| S        |            | S -> [$V, =, E] |            |            |            |            |            |            |            |
| T        |            | T -> [F, T'] | T -> [F, T'] |            |            |            |            |            |            |
| T'       | T' -> []   |            |            | T' -> []   | T' -> [M, F, T'] | T' -> []   | T' -> []   | T' -> [M, F, T'] |            |
```

3.4 预测分析的实现和测试

原理解析：

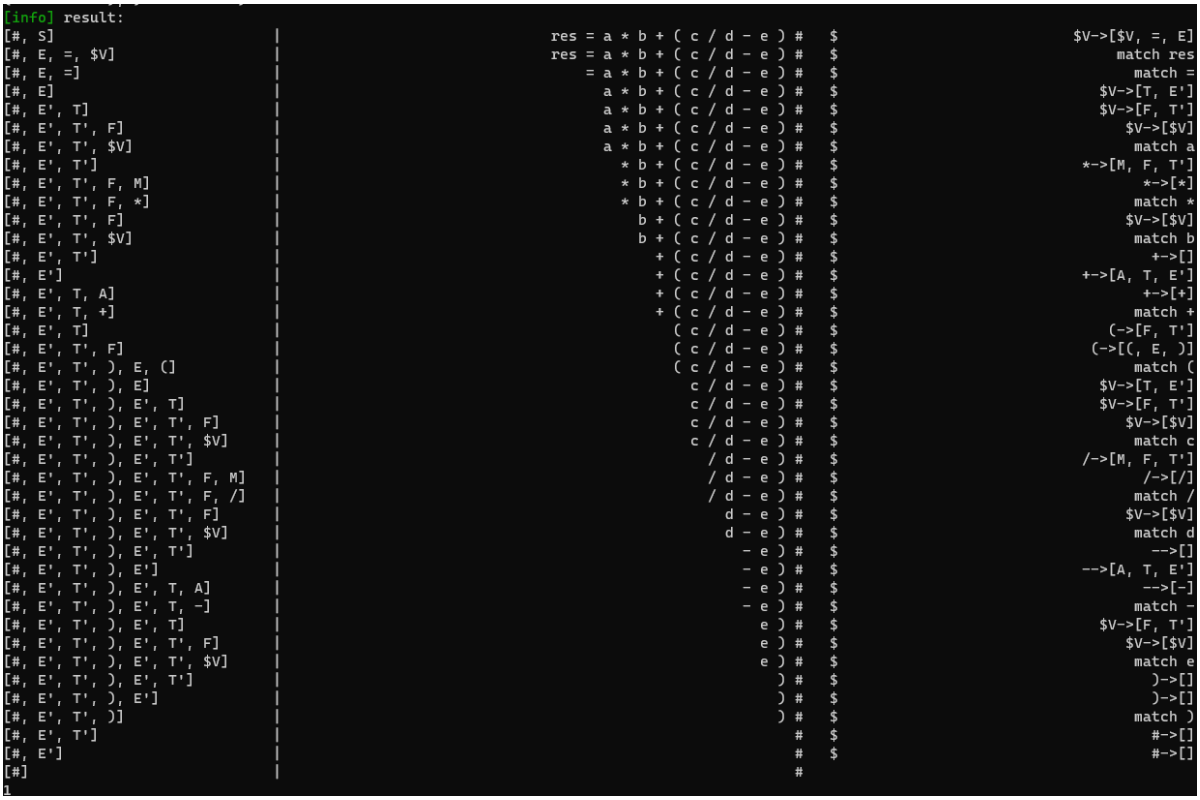


实现代码:

```
bool PredictiveTableAnalyzer::analyze(vector<token> input)
{
    stack<term> s;
    input.push_back(token(make_shared<term>(SYM_END), SYM_END, 0, 0));
    s.push(SYM_END);
    s.push(grammar.startTerm);
    int i = 0;
    while (!s.empty() && s.top() != SYM_END && i < input.size())
    {
        std::cout << setw(8) << std::left;
        printStack(s);
        std::cout << " | ";
        std::cout << setw(60) << std::right;
        printVecFrom(input, i);
        assert(_find(grammar.terminals, *(input[i].type)));
        if (s.top() == *(input[i].type))
        {
            s.pop();
            i++;
            cout << setw(4) << right << " $ ";
            stringstream ss;
            ss << "match " << input[i - 1].value;
            cout << setw(40) << right << ss.str();
            cout << endl;
        }
        else if (_find(grammar.nonTerms, s.top()))
        {
            auto it = predict[s.top()].find(*(input[i].type));
            if (it != predict[s.top()].end())
```

```
            {
                s.pop();
                for (auto it1 = it->second.rbegin(); it1 != it->second.rend(); it1++)
                {
                    s.push(*it1);
                }
                cout << setw(4) << right << " $ ";
                stringstream ss;
                ss << it->first << "->" << vec2str(it->second);
                cout << setw(40) << right << ss.str();
                cout << endl;
            }
            else
            {
                error << "Error: " << input[i].line << ":" << input[i].col << ": "
                      << "Unexpected token: " << input[i].type << std::endl;
                return false;
            }
        }
        else
        {
            error << "Error: " << input[i].line << ":" << input[i].col << ": "
                  << "Unexpected token: " << input[i].type << std::endl;
            return false;
        }
    }
    std::cout << setw(8) << std::left;
    printStack(s);
    std::cout << " | ";
    std::cout << setw(60) << std::right;
    printVecFrom(input, i);
    std::cout << std::endl;
    return true;
}
```

实验结果：



四、实验心得

本次实验我完成了文法规则的解析，并让程序自动求出 First 和 Follow 集然后构造出 LL(1)分析表进行 LL(1)文法分析。我尽量使程序的抽象程度足够高，因而导致代码极其复杂，关于代码的具体逻辑我会在后续的报告中逐步解释。谢谢老师！

我的所有实验共享同一个项目结构，代码已经开源：

Github开源链接：https://github.com/Kogler7/SatoriCompiler

参考资料：

(77条消息) 编译原理——自顶向下分析中SELECT集的计算与预测分析表的构造_pureZer的博客-CSDN博客