北京交通大學

BEIJING JIAOTONG UNIVERSITY

《编译原理》实验报告

实验名称：专题四算符优先语法分析设计原理与实现

学　　号：22281188

姓　　名：江家玮

学　　院：计算机科学与技术学院

日　　期：2024 年 12 月 12 日

目录

1. 程序功能描述 1

1.1 定义文法规则 1

1.2 计算 FirstVT 和 LastVT 集合 1

1.3 构造算符优先关系分析表（OPM） 1

1.4 算符优先分析 1

1.5 输入与输出 2

2. 主要数据结构描述 2

2.1 集合类数据结构 2

2.2 映射类数据结构 2

2.3 二维向量 2

2.4 字符串处理 3

3. 程序结构描述 3

3.1 calculateFirstVtSet() 3

3.2 calculateLastVtSet() 3

3.3 parseTable() 4

3.4 analyzeExpression() 4

3.5 主函数 main() 5

4. 程序测试（包括选做内容） 6

4.1 算符优先关系矩阵和 FIRSTVT 和 LASTVT 集合构造结果 6

4.2 简单表达式 6

4.3 带括号的表达式 7

4.4 无效表达式 7

5. 附录：完整代码 8

1. 程序功能描述

实现了算符优先分析算法，用于处理包含算符和操作数的表达式的语法分析。具体功能如下：

1.1 定义文法规则

通过 grammar 定义了一个简化的拓广文法，规则包括：

S -> #E#（起始符号 S）；

E -> E+T | E-T | T；

T -> T*F | T/F | F；

F -> (E) | i（i 表示操作数）。

1.2 计算 FirstVT 和 LastVT 集合

FirstVT：计算每个非终结符号的 First 集合，即一个产生式右部的第一个终结符。

LastVT：计算每个非终结符号的 Last 集合，即一个产生式右部的最后一个终结符。

1.3 构造算符优先关系分析表（OPM）

parseTable() 函数基于文法规则构建了算符优先关系表，用于表示各种符号之间的优先关系（<, =, >）。

1.4 算符优先分析

analyzeExpression() 函数负责读取输入表达式，执行 算符优先分析。

它根据算符优先分析表判断输入表达式是否符合文法，并进行“移进”（Push）和“规约”（Apply production）操作，直到完成分析。

1.5 输入与输出

输入：通过文件读取一个二元表达式的符号序列，转换为字符串。

输出：打印出 FirstVT 集合 和 LastVT 集合。输出算符优先关系表，显示 <, =, > 的优先关系。进行分析并输出分析过程，最后输出分析结果（接受或拒绝）。

2. 主要数据结构描述

2.1 集合类数据结构

set<string> Vt：用于存储终结符集合，确保无重复且高效查找。

set<string> Vn：用于存储非终结符集合。

map<string, set<string>> FirstVt：存储每个非终结符的 FirstVT 集合。

map<string, set<string>> LastVt：存储每个非终结符的 LastVT 集合。

作用：这些集合帮助构造算符优先关系，通过 FirstVT 和 LastVT 推导产生式的优先级关系。

2.2 映射类数据结构

map<string, vector<string>> grammar：表示文法规则的映射，其中键是非终结符，值是该非终结符对应的产生式集合。

map<string, int> VtIndex：用于将终结符映射到其在分析表中的下标，用于快速索引分析表。

作用：grammar 用于定义文法规则，VtIndex 在分析表构造时便于按终结符索引。

2.3 二维向量

vector<vector<string>> table：构造的 算符优先分析表 (OPM)，表中存储 <、>、= 等优先级关系。

作用：用于记录终结符之间的优先级关系，后续在表达式分析过程中决定栈操作。

2.4 字符串处理

string shizi 和 string like：分别用于存储输入表达式和其简化形式。

char buf[1024]：用于读取外部文件内容。

3. 程序结构描述

3.1 calculateFirstVtSet()

功能分析：

该函数用于计算每个非终结符的 FirstVT 集合，即计算文法中每个非终结符推导出的可能的终结符集合。

=

初始化：

函数首先遍历所有文法规则，检查每个非终结符的第一个产生式符号。如果该符号是终结符，则将其加入 FirstVt 集合中。

迭代计算：

函数继续迭代，直到没有新的终结符加入到任何 FirstVt 集合中为止。对每个非终结符的产生式，检查右边符号的 FirstVT 集合，并将终结符加入当前非终结符的集合中。

如果产生式的第一个符号是非终结符，则递归查找该非终结符的 FirstVT 集合，直到遇到终结符。

终止条件：

当所有非终结符的 FirstVT 集合不再变化时，算法终止。

3.2 calculateLastVtSet()

功能分析：

该函数用于计算每个非终结符的 LastVT 集合，即计算文法中每个非终结符推导出的可能的终结符集合。

初始化：

函数首先遍历所有文法规则，检查每个非终结符的所有产生式的最后一个符号。如果该符号是终结符，则将其加入 LastVt 集合中。

迭代计算：

对于每个非终结符的产生式，检查其右边从最后一个符号向前推导，直到遇到终结符或非终结符。如果遇到非终结符，则将该非终结符的 LastVT 集合中的终结符加入当前非终结符的集合。

如果产生式的右边有多个非终结符，则从右到左依次进行递归。

终止条件：

当所有非终结符的 LastVT 集合不再变化时，算法终止。

3.3 parseTable()

功能分析：

该函数用于构建 算符优先分析表，分析表是一个二维表，用于表示符号之间的优先关系。

初始化：

分析表是一个二维矩阵，表格的大小由终结符的数量决定。初始状态下，所有表项均为空字符串。

填充分析表：

通过遍历文法规则，对产生式的每个符号对填充分析表：

如果产生式的符号是终结符，并且该符号出现在产生式的最右边，则设置该表项为 "="，表示该符号等价于该非终结符。

如果符号不在最右边，则设置为 "＜"，表示该符号优先于非终结符。

3.4 analyzeExpression()

功能分析：

该函数用于读取输入的表达式并进行语法分析，检查表达式是否符合文法。

读取输入：

从指定文件中读取表达式，按行处理。

栈操作：

利用栈来模拟算符优先法的分析过程。每次遇到符号时，根据分析表检查是否需要推导、压栈或者应用产生式。

接受与拒绝：

如果整个表达式被成功分析且符合文法规则，输出 "ACCEPT"，否则输出 "REJECT"。

3.5 主函数 main()

加载文法：

首先，程序从文件 grammar.txt 中加载文法规则，并将其解析为非终结符及其对应的产生式。

计算 FirstVT 和 LastVT 集合：

利用文法规则计算出每个非终结符的 FirstVT 集合和 LastVT 集合，这些集合对语法分析至关重要。

输出集合：

程序将计算得到的 FirstVT 集合和 LastVT 集合输出到控制台，以便检查和调试。

生成并输出分析表：

调用 parseTable() 构建算符优先分析表，并将其输出，以便进一步分析符号之间的优先关系。

表达式分析：

最后，程序读取 expression.txt 文件中的表达式，利用已生成的分析表判断表达式是否符合文法，输出结果。

4. 程序测试（包括选做内容）

4.1 算符优先关系矩阵和 FIRSTVT 和 LASTVT 集合构造结果

```
----------------------------------------
Vn集合: E F N S T
Vt集合: # ( ) * + - / i
FirstVt集合:
E: "(" "*" "+" "-" "/" "i"
F: "(" "i"
S: "#"
T: "(" "*" "/" "i"
LastVt集合:
E: ")" "*" "+" "-" "/" "i"
F: ")" "i"
S: "#"
T: ")" "*" "/" "i"
------------OPM分析表------------------
----------------------------------------
        |#      |(      |)      |*      |+      |-      |/      |i      |
#       |=      |<      |       |<      |<      |<      |<      |<      |
(       |       |<      |=      |<      |<      |<      |<      |<      |
)       |>      |       |>      |>      |>      |>      |>      |       |
*       |>      |<      |>      |>      |>      |>      |>      |<      |
+       |>      |<      |>      |<      |>      |>      |<      |<      |
-       |>      |<      |>      |<      |>      |>      |<      |<      |
/       |>      |<      |>      |>      |>      |>      |>      |<      |
i       |>      |       |>      |>      |>      |>      |>      |       |
```

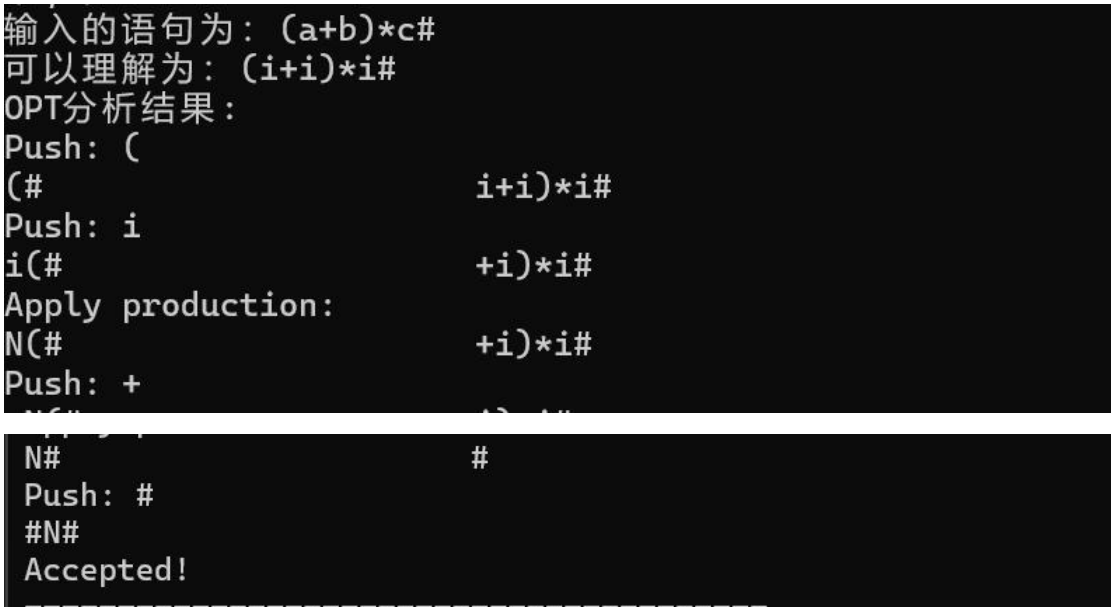
4.2 简单表达式

a*b+c。程序会展现逐步的分析过程，包括使用的规则以及每一步的新产生式，完整过程如下。最终此例会输出 Accepted 表示识别成功。

```
输入的语句为: i*i+i#
可以理解为: i*i+i#
OPT分析结果:
Push: i
i#                      *i+i#
Apply production:
N#                      *i+i#
Push: *
*N#                     i+i#
Push: i
i*N#                    +i#
Apply production:
N*N#                    +i#
Apply production:
N#                      +i#
Push: +
+N#                     i#
Push: i
i+N#                    #
Apply production:
N+N#                    #
Apply production:
N#                      #
Push: #
#N#
Accepted!
```

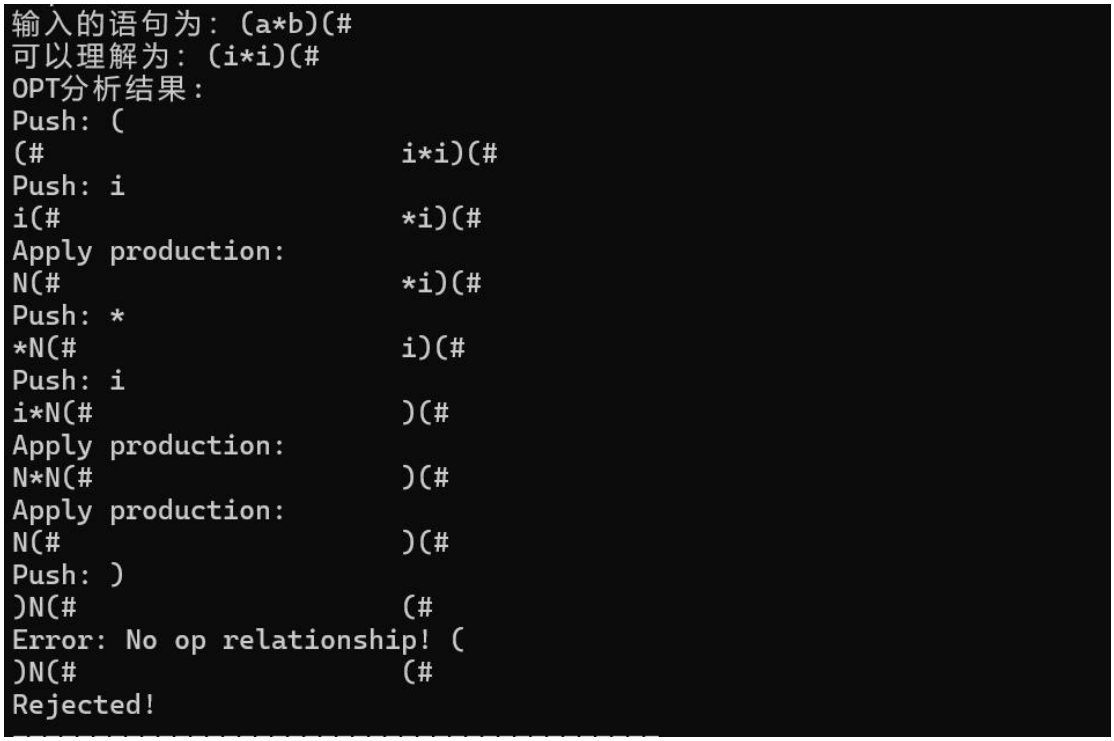
4.3 带括号的表达式

(a+b)*c，符合文法规则，输出 Accepted。



4.4 无效表达式

(a*b)(，是无效表达式，会输出 Rejected。



5. 附录：完整代码

```
#define _CRT_SECURE_NO_WARNINGS 1
#include <iostream>
#include <fstream>
#include <set>
#include <map>
#include <vector>
#include <string>
#include<cstring>

using namespace std;

set<string> Vt;
set<string> Vn;
map<string, set<string>> FirstVt;
map<string, set<string>> LastVt;
map<string, int>VtIndex;
vector<vector<string>> table;

// 定义文法规则(拓广文法)
map<string, vector<string>> grammar = {
    {"S", {"#E#"}},
    {"E", {"E+T","E-T","T"}},
    {"T", {"T*F","T/F","F"}},
    {"F",{"(E)","i"}}
};

// 判断是否是非终结符
bool isVn(const string& symbol) {
    return Vn.count(symbol);
}

// 判断是否是终结符
bool isVt(const string& symbol) {
    return Vt.count(symbol);
}

// 计算 FirstVT 集
map<string, set<string>> calculateFirstVtSet()
{
    map<string, set<string>> firstSet;

```

```
42.  // 初始化非终结符的 First 集
43.  for (map<string, vector<string>>::iterator entry = grammar.begin(); entry != grammar.end(); entry++)
44.  {
45.      firstSet[entry->first] = {};
46.      for (vector<string>::iterator production = entry->second.begin(); production != entry->second.end(); production++) //若每个右部第一个或第二个字母是终结符则直接加入
47.      {
48.          string s = *production;
49.          if (isVt(s.substr(0, 1)))
50.              firstSet[entry->first].insert(s.substr(0, 1));
51.          else if (isVt(s.substr(1, 1)))
52.              firstSet[entry->first].insert(s.substr(1, 1));
53.      }
54.  }
55.
56.  bool changes = true;
57.  while (changes)
58.  {
59.      changes = false;
60.
61.      for (map<string, vector<string>>::iterator entry = grammar.begin(); entry != grammar.end(); entry++)
62.      {
63.          string nonTerminal = entry->first;
64.          int originalSize = firstSet[nonTerminal].size();
65.          for (vector<string>::iterator it = entry->second.begin(); it != entry->second.end(); it++)
66.          {
67.              string production = *it; // 一个产生式右部
68.              string vn = production.substr(0, 1);
69.              if (isVn(vn))// 如果是非终结符
70.              {
71.                  for (set<string>::iterator fi = firstSet[vn].begin(); fi != firstSet[vn].end(); fi++)
72.                      firstSet[nonTerminal].insert(*fi);
73.              }
74.
75.          }
76.          if (firstSet[nonTerminal].size() != originalSize) //若first 改变，则继续迭代
77.          {
78.              changes = true;
79.          }
```

```
80.      }
81.
82.   }
83.
84.   return firstSet;
85.}
86.
87.// 计算 LastVt 集
88.map<string, set<string>> calculateLastVtSet()
89.{
90.   map<string, set<string>> LastSet;
91.
92.   // 初始化非终结符的 LastVt 集
93.   for (map<string, vector<string>>::iterator entry = grammar.begin(); entry != grammar.e
      nd(); entry++)
94.   {
95.      LastSet[entry->first] = {};
96.      for (vector<string>::iterator production = entry->second.begin(); production != entry
      ->second.end(); production++) //若每个右部第一个或第二个字母是终结符则直接加
      入
97.      {
98.         string s = *production;
99.         int len = s.length() - 1;
100.               if (isVt(s.substr(len, 1)))
101.                  LastSet[entry->first].insert(s.substr(len, 1));
102.               else if (len > 1 && isVt(s.substr(len - 1, 1)))
103.                  LastSet[entry->first].insert(s.substr(len - 1, 1));
104.            }
105.         }
106.
107.         bool changes = true;
108.         while (changes)
109.         {
110.      changes = false;
111.
112.      for (map<string, vector<string>>::iterator entry = grammar.begin(); entry != gram
      mar.end(); entry++)
113.      {
114.         string nonTerminal = entry->first;
115.         int originalSize = LastSet[nonTerminal].size();
116.         for (vector<string>::iterator it = entry->second.begin(); it != entry->second.end();
       it++)
117.         {
118.            string production = *it; // 一个产生式右部
```

```
119.            string vn = production.substr(production.length() - 1, 1);
120.                if (isVn(vn))// 如果是非终结符
121.                {
122.                    for (set<string>::iterator fi = LastSet[vn].begin(); fi != LastSet[vn].end
    (); fi++)
123.                        LastSet[nonTerminal].insert(*fi);
124.                }
125.
126.            }
127.            if (LastSet[nonTerminal].size() != originalSize) //若first 改变，则继续迭代
128.            {
129.                changes = true;
130.            }
131.        }
132.
133.    }
134.
135.    return LastSet;
136.  }
137.
138.  // 构造 OPM
139.  void parseTable()
140.  {
141.    for (map<string, vector<string>>::iterator it = grammar.begin(); it != grammar.en
    d(); it++)
142.    {
143.        for (vector<string>::iterator tmp = it->second.begin(); tmp != it->second.end();
     tmp++)
144.        {
145.            string str = *tmp; // 右部
146.            for (int i = 0; i + 1 < str.size(); i++)
147.            {
148.                if (i + 2 < str.size())
149.                {
150.                    if (isVt(str.substr(i, 1)) && isVn(str.substr(i + 1, 1)) && isVt(str.substr
    (i + 2, 1)))
151.                        table[VtIndex[str.substr(i, 1)]][VtIndex[str.substr(i + 2, 1)]] = "=";
152.                }
153.                if (isVt(str.substr(i, 1)))
154.                {
155.                    if (isVt(str.substr(i + 1, 1)))
156.                        table[VtIndex[str.substr(i, 1)]][VtIndex[str.substr(i + 1, 1)]] = "=";
157.                    else if (isVn(str.substr(i + 1, 1)))
158.                    {
```

```
159.                         for (set<string>::iterator fi = FirstVt[str.substr(i + 1, 1)].begin(); fi !
    = FirstVt[str.substr(i + 1, 1)].end(); fi++)
160.                             table[VtIndex[str.substr(i, 1)]][VtIndex[*fi]] = "<";
161.                     }
162.                 }
163.                 else if (isVn(str.substr(i, 1)))
164.                 {
165.                     if (isVt(str.substr(i + 1, 1)))
166.                     {
167.                         for (set<string>::iterator la = LastVt[str.substr(i, 1)].begin(); la != La
    stVt[str.substr(i, 1)].end(); la++)
168.                             table[VtIndex[*la]][VtIndex[str.substr(i + 1, 1)]] = ">";
169.                     }
170.                 }
171.             }
172.         }
173.     }
174. }
175.
176. // 读取二元式文件并进行 OPT 分析
177. void analyzeExpression(const char* expressionFile)
178. {
179.     cout << "读取文件可知输入语句：" << endl;
180.     FILE* fp;
181.     char buf[1024];
182.     string shizi, like;
183.     if ((fp = fopen(expressionFile, "r")) != NULL)
184.     {
185.         while (fgets(buf, 1024, fp) != NULL)
186.         {
187.             int len = strlen(buf);
188.             buf[len - 1] = '\0';  /*去掉换行符*/
189.             printf("%s \n", buf);
190.
191.             if (buf[1] == '2') // 说明为标识符
192.             {
193.                 like += 'i';
194.             }
195.             for (int i = 3; i < len - 2; i++)
196.             {
197.                 shizi = shizi + buf[i];
198.                 if (buf[1] != '2')
199.                 {
200.                     like += buf[i];
```

```
                }
            }
        }
    }
    shizi += '#';
    like += '#';
    fclose(fp);

    cout << "输入的语句为：" << shizi << endl;
    cout << "可以理解为：" << like << endl;

    cout << "OPT 分析结果:" << endl;

    string expression = like;
    string result = "ACCEPT";
    string topOfStack = "#";

    while ((expression.size() > 0) && result == "ACCEPT")
    {
        string nextSymbol = expression.substr(0, 1);
        if (!isVt(nextSymbol) && !isVn(nextSymbol))
        {
            result = "REJECT";
            cout << "Error: undefined symbol! :\"" << nextSymbol << "\"" << endl;
            break;
        }
        if (isVt(topOfStack.substr(0, 1)))
        {
            string op = table[VtIndex[topOfStack.substr(0, 1)]][VtIndex[nextSymbol]];
            if (op == "=" || op == "<")
            {
                cout << "Push: " << nextSymbol << endl;
                expression = expression.substr(1);
                topOfStack = nextSymbol + topOfStack;
            }
            else if (op == ">")
            {
                cout << "Apply production: " << endl;
                int j = 1;
                string Q = topOfStack.substr(0, 1);
                while (isVn(topOfStack.substr(j, 1)) || (isVt(topOfStack.substr(j, 1)) && t
able[VtIndex[topOfStack.substr(j, 1)]][VtIndex[Q]] != "<"))
                {
                    if (isVn(topOfStack.substr(j, 1)))
```

```
                    j++;
                else
                {
                    Q = topOfStack.substr(j, 1);
                    j++;
                }
            }
            topOfStack = "N" + topOfStack.substr(j);
        }
        else
        {
            result = "REJECT";
            cout << "Error: No op relationship! " << nextSymbol << endl;
        }
    }
    else
    {
        string op = table[VtIndex[topOfStack.substr(1, 1)]][VtIndex[nextSymbol]];
        if (op == "=" || op == "<")
        {
            cout << "Push: " << nextSymbol << endl;
            expression = expression.substr(1);
            topOfStack = nextSymbol + topOfStack;
        }
        else if (op == ">")
        {
            cout << "Apply production: " << endl;
            int j = 2;
            while (isVn(topOfStack.substr(j, 1)) || (isVt(topOfStack.substr(j, 1)) && table[VtIndex[topOfStack.substr(j, 1)]][VtIndex[topOfStack.substr(0, 1)]] != "<"))
                j++;
            topOfStack = "N" + topOfStack.substr(j);
        }
        else
        {
            result = "REJECT";
            cout << "Error: No op relationship! " << nextSymbol << endl;
        }
    }
    cout << topOfStack << "\t\t\t" << expression << endl;
}

if (topOfStack.size() == 3 && expression.size() == 0 && result == "ACCEPT")
{
```

```
287.             cout << "Accepted!" << endl;
288.         }
289.         else
290.         {
291.             cout << "Rejected!" << endl;
292.         }
293.
294.         cout << "---------------------------------------" << endl;
295.     }
296.
297.     int main()
298.     {
299.         int cntVt = 0;
300.
301.         // 记录 Vn 集合和 Vt 集合及其在分析表中的下标
302.         for (map<string, vector<string>>::iterator it = grammar.begin(); it != grammar.en
    d(); ++it)
303.         {
304.             pair<string, vector<string>> temp = *it;
305.
306.             Vn.insert(temp.first);
307.
308.             vector<string> production = temp.second;
309.             for (vector<string>::iterator iit = production.begin(); iit != production.end(); iit
    ++)
310.             {
311.         string ss = *iit;
312.                 for (int j = 0; j < ss.length(); j++)
313.                 {
314.                     //记录产生式中的非终结符和终结符
315.                     if (ss[j] >= 'A' && ss[j] <= 'Z')
316.                     { // 大写字母
317.                         if (ss[j + 1] == '\'')
318.                         {       // 有'则读入俩作为一个非终结符
319.                             Vn.insert(ss.substr(j, 2));
320.                             j++;
321.                         }
322.                         else
323.                         {
324.                             Vn.insert(ss.substr(j, 1));
325.                         }
326.                     }
327.                     else
328.                     {
```

```
                    Vt.insert(ss.substr(j, 1)); // 是终结符
                    string v = ss.substr(j, 1);
                    if (v != "" && VtIndex[v] == 0)
                        VtIndex[v] = cntVt++;
                }
            }

        }
    }

    table.resize(Vt.size(), vector<string>(Vt.size(), ""));
    Vn.insert("N");

    // 求 FirstVt 集合
    FirstVt = calculateFirstVtSet();
    // 求 LastVt 集合
    LastVt = calculateLastVtSet();

    // 构造 OPM 分析表
    parseTable();

    // 读取二元式文件并进行 OPT 分析
    analyzeExpression("D:/lexical_analysis_output.txt");


    // 输出 Vn 集合
    cout << "Vn 集合: ";
    for (set<string>::iterator it = Vn.begin(); it != Vn.end(); it++)
        cout << *it << ' ';
    cout << endl;

    // 输出 Vt 集合
    cout << "Vt 集合: ";
    for (set<string>::iterator it = Vt.begin(); it != Vt.end(); it++)
        cout << *it << ' ';
    cout << endl;

    // 输出 FirstVt 集合
    cout << "FirstVt 集合: " << endl;
    for (map<string, set<string>>::iterator it = FirstVt.begin(); it != FirstVt.end(); it+
+)
    {
        cout << it->first << ": ";
```

```
371.            for (set<string>::iterator it2 = it->second.begin(); it2 != it->second.end(); it2+
    +) {
372.                cout << '\"' << *it2 << '\"' << " ";
373.            }
374.            cout << endl;
375.        }
376.
377.        // 输出 LastVt 集合
378.        cout << "LastVt 集合: " << endl;
379.        for (map<string, set<string>>::iterator it = LastVt.begin(); it != LastVt.end(); it+
    +) {
380.            cout << it->first << ": ";
381.            for (set<string>::iterator it2 = it->second.begin(); it2 != it->second.end(); it2+
    +) {
382.                cout << '\"' << *it2 << '\"' << " ";
383.            }
384.            cout << endl;
385.        }
386.
387.
388.
389.        // 输出 OPM 分析表
390.        cout << "-------------OPM 分析表------------------" << endl;
391.        cout << "---------------------------------------" << endl;
392.        cout << "\t|";
393.        vector<string> vtt;
394.        for (set<string>::iterator vt = Vt.begin(); vt != Vt.end(); vt++)
395.        {
396.            cout << *vt << "\t|";
397.            vtt.push_back(*vt);
398.        }
399.        cout << endl;
400.        for (set<string>::iterator vt = Vt.begin(); vt != Vt.end(); vt++)
401.        {
402.            cout << *vt << "\t|";
403.            for (int j = 0; j < Vt.size(); j++)
404.            {
405.                cout << table[VtIndex[*vt]][VtIndex[vtt[j]]] << "\t|";
406.            }
407.            cout << endl;
408.        }
409.
410.
411.
```

```
412.         return 0;
413.
414.     }
```