编 译 原 理

实验名称

学　　院：詹天佑

专　　业：计算机科学与技术

学生姓名：程维森

学　　号：21231264

北京交通大学

2023 年 11 月

一、程序功能描述

文法处理：

从文件中读取文法规则，并提取非终结符和终结符列表。

计算文法的 First 集合和 Follow 集合。

基于 First 集合和 Follow 集合构建分析表。

LL(1) 分析器：

使用 LL(1)分析方法解析输入字符串。

对输入的字符串进行语法分析，查看是否符合文法规则。

输出和记录：

将文法分析过程中的详细信息输出到文件中，包括 First 集合、Follow 集合、分析表、分析栈、输入剩余部分等。

代码中各个函数的主要功能如下：

getfirst()：计算文法规则的 First 集合。

computeFirstFollow()：计算文法规则的 Follow 集合。

printFollowSets()：输出 Follow 集合的计算结果。

updateFollowSets()：更新 Follow 集合并输出结果。

printTerminalList() 和 processGrammarSymbols()：构建并输出文法规则的分析表。

gettable()：生成分析表并输出到文件。

getRule()：获取文法规则中的非终结符和终结符列表。

getrule()：整体执行从文件读取文法规则、计算 First 集合、Follow 集合和分析表的过程。

checktopVn()、checktopVt()、checkok() 和 printanalst()：辅助函数，用于语法分析中的栈操作、规则检查和输出。

main()：程序的入口，读取输入、执行文法分析和 LL(1)分析，将分析结果输出到文件中。

总的来说，这个代码实现了对给定文法规则的分析、First 集合、Follow 集合和分析表的计算，以及 LL(1)语法分析器的实现。同时将分析过程中的详细信息记录到输出文件中。

二、 主要数据结构

变量和类型	用途
string anastr	存储输入的字符串
vector<string> gra	存储文法规则
vector<char> Vn	存储非终结符列表
vector<char> Vt	存储终结符列表
map<char, int> Vncot	将非终结符映射到文法规则的索引
map<char, int> Vtcot	将终结符映射到 Vt 数组中的索引
vector<char> table[20][20]	存储 LL(1) 分析表
vector<char> analst	存储 LL(1) 分析器中的分析栈
vector<char> first[10]	存储文法规则的 First 集合
vector<char> follow[10]	存储文法规则的 Follow 集合
ofstream objects	用于输出到文件的 ofstream 对象

string anastr: 存储输入的字符串，即待分析的符号串。

vector<string> gra: 存储文法规则，每个元素是一个字符串，表示一个文法规则。

vector<char> Vn: 存储非终结符列表，用于表示文法中的所有非终结符。

vector<char> Vt: 存储终结符列表，用于表示文法中的所有终结符。

map<char, int> Vncot: 将非终结符映射到文法规则的索引，用于查找非终结符在 gra 中的位置。

map<char, int> Vtcot: 将终结符映射到 Vt 数组中的索引，方便查找终结符在 Vt 数组中的位置。

vector<char> table[20][20]: 存储 LL(1) 分析表，用二维向量表示，用于 LL(1) 文法分析过程中的分析表。

vector<char> analst: 存储 LL(1) 分析器中的分析栈, 用于模拟分析过程的栈结构。

vector<char> first[10]: 存储文法规则的 First 集合，用于 LL(1) 文法分析的 First 集合计算。

vector<char> follow[10]: 存储文法规则的 Follow 集合，用于 LL(1) 文法分析的 Follow 集合计算。

ofstream 对象：ofstream 类型的对象，用于输出到文件。在代码中有多个 ofstream 对象用于输出进度信息、First 集合、Follow 集合、LL(1) 分析表等到文件中。

三、 程序结构描述

（1）设计方法

以下是对每个函数设计的简要解释:

1. bool inanalst(const vector<char> &vec, char target): 检查字符 target 是否存在于给定的字符向量 vec 中。

2. void getfirst(): 计算文法规则的 First 集合。该函数实现了对每个非终结符的 First 集合的计算，并将计算结果输出到文件中。

3. void computeFirstFollow(): 计算文法规则的 Follow 集合。该函数实现了对每个非终结符的 Follow 集合的计算，并将计算结果输出到文件中。

4. void printFollowSets(): 打印 Follow 集合的结果。将计算好的 Follow 集合输出到文件中。

5. void updateFollowSets(): 更新 Follow 集合。对 Follow 集合进行进一步的计算和更新，并将更新后的结果输出到文件中。

6. void printTerminalList(): 打印终结符列表。将文法中的终结符列表输出到文件中。

7. void processGrammarSymbols(): 处理文法符号。该函数根据文法规则和 First 集合，构建 LL(1) 分析表，并将构建的 LL(1) 分析表输出到文件中。

8. void gettable(): 获取 LL(1) 分析表。调用 printTerminalList() 和 processGrammarSymbols() 函数，用于生成并输出 LL(1) 分析表。

9. int getRule(): 获取文法规则。该函数解析给定的文法规则，获取其中的非终结符和终结符，并将结果输出到文件中。

10. void getrule(): 获取文法规则并计算 First 和 Follow 集合以及 LL(1) 分析表。该函数调用 getRule()、getfirst()、getfollow() 和 gettable() 函数，实现了对文法规则的解析和相关集合的计算。

11. bool checktopVn(int i): 检查栈顶元素是否为非终结符。

12. bool checktopVt(int i): 检查栈顶元素是否为终结符。

13. bool checkok(char c): 检查栈顶的非终结符与输入串当前位置的终结符是否匹配。

14. void printanalst(int n): 输出分析栈和剩余输入。输出当前分析栈和剩余待分析的输入串。

15. int main(): 主函数，实现了对输入的符号串进行 LL(1) 文法分析的过程。

这些函数在代码中各自负责不同的功能，包括计算文法规则的 First 和 Follow 集合、构建 LL(1) 分析表以及模拟 LL(1) 文法分析器的过程等。

程序流程图如图 3.4 所示：

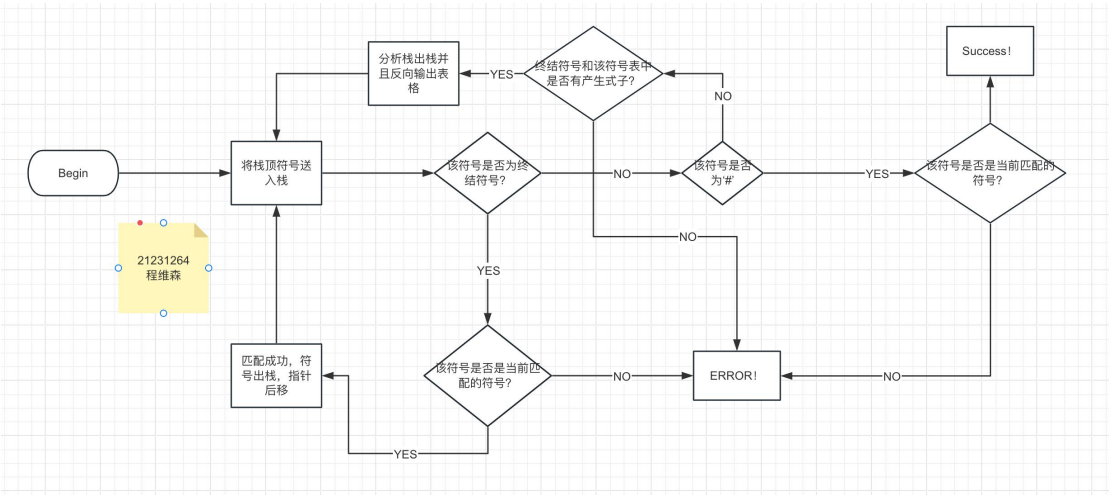

图 3.4 程序流程图

(2) 函数定义及函数间的调用关系

函数详细分析:

1.查找函数

```
bool inanalst(const vector<char> &vec, char target)
{
    for (char element : vec)
    {
        if (element == target)
        {
            return true;
        }
    }
    return false;
}
```

这个函数实现了从 vector 中遍历查找所需的字符的功能

2. 求 first 集合函数:

```
void getfirst()
{
    out2 << "计算First集合:\n";
    out2 << "第1轮:\n";
    int iteration = 1;
    int num[100] = {0};
    for (int i = 0; i < gra.size(); i++)
    {
        out2 << gra[i][0] << ": "; // 打印当前处理的非终结符

        int stat = 0; // stat 用于标记是否处理到产生式的右部
        for (int j = 3; j < gra[i].size(); j++)
        {
            if (!isupper(gra[i][j]) && gra[i][j] != '|' && stat == 0)
            {
                if (!inanalst(first[Vncot[gra[i][0]]], gra[i][j])) // inanalst(first[Vncot[gra[i][0]]],gra[i][j])
                {
                    first[Vncot[gra[i][0]]].push_back(gra[i][j]);
                    out2 << gra[i][j] << " ";
                }
            }
            stat = 1;
            if (gra[i][j] == '|')
                stat = 0;
        }
        out2 << "\n";
    }
    iteration++;
    bool flag = false;

    do
    {
        out2 << "第" << iteration++ << "轮:\n";
        flag = true;

        for (int i = 0; i < gra.size(); i++)
        {
            if (num[i] != first[i].size())
            {
                flag = false;
                num[i] = first[i].size();
            }
        }

        if (flag == true)
        {
            out2 << "结束\n";
            break;
        }

        for (int i = 0; i < gra.size(); i++)
        {
            int nonTerminalIndex = Vncot[gra[i][0]];
            out2 << gra[i][0] << ": ";
            int flag = 0;

            for (int j = 3; j < gra[i].size(); j++)
            {
                if (isupper(gra[i][j]))
                {
                    for (int k = 0; k < first[Vncot[gra[i][j]]].size(); k++)
                    {
                        if (first[Vncot[gra[i][j]]][k] != 'e' && !inanalst(first[nonTerminalIndex], first[Vncot[gra[i][j]]][k]) && flag == 0) //
                        {
                            first[nonTerminalIndex].push_back(first[Vncot[gra[i][j]]][k]);
                        }
                    }
                }
                flag = 1;
                if (gra[i][j] == '|')
                {
                    flag = 0;
                }
            }

            for (int j = 0; j < first[nonTerminalIndex].size(); j++)
            {
                out2 << first[nonTerminalIndex][j] << " ";
            }
            out2 << "\n";
        }
    } while (true);

    out2.close();
}
```

打印计算 First 集合的信息和第一轮的开始信息。

遍历文法规则，对每个非终结符计算其 First 集合，并将结果输出到文件中。

进行迭代计算，直到 First 集合不再更新。

在每一轮迭代中，根据文法规则的右部字符更新对应非终结符的 First 集合。

当 First 集合不再更新时，输出结束信息并关闭文件流。

具体步骤包括：

首先，对每个产生式左部非终结符，检查其右部字符，并将非终结符的 First 集合初始化为其右部首个非终结符。

之后，对每个非终结符根据右部字符的情况，更新 First 集合。

如果 First 集合在一轮更新后不再改变，结束迭代计算，关闭输出文件流。

整体来说，这个函数对文法规则中每个非终结符的 First 集合进行了计算和更新，并将计算结果输出到文件中。

3. 求 follow 集合函数：

```
void computeFirstFollow()
{
    // 初始的Follow集合计算
    follow[Vncot[gra[0][0]]].push_back('#'); // 初始化起始符号的Follow集合为#
    out3 << "计算Follow集合:\n";
    out3 << "第1轮:\n";

    for (int i = 0; i < gra.size(); i++)
    {
        for (int j = 3; j < gra[i].size(); j++)
        {
            if ((isupper(gra[i][j])))
            {
                int num1 = Vncot[gra[i][j]];
                if (j < gra[i].size() - 1)
                {
                    if ((!isupper(gra[i][j + 1])) && gra[i][j + 1] != '|')
                    {
                        if (!inanalst(follow[num1], gra[i][j + 1]))
                        {
                            follow[num1].push_back(gra[i][j + 1]);
                        }
                    }
                    else if (isupper(gra[i][j + 1]))
                    {
                        int num2 = Vncot[gra[i][j + 1]];
                        for (int k = 0; k < first[num2].size(); k++)
                        {
                            if (first[num2][k] != 'e' && (!inanalst(follow[num1], first[num2][k])))
                            {
                                follow[num1].push_back(first[num2][k]);
                            }
                        }
                    }
                }
                else
                {
                    if (inanalst(follow[num1], 'e'))
                    {
                        follow[num1].push_back('e');
                    }
                }
            }
        }
    }
}
```

初始化起始符号的 Follow 集合为 #。

输出 Follow 集合的计算信息和第一轮的开始信息。

遍历文法规则，对每个非终结符的右部字符进行分析，并根据规则计算 Follow 集合。

更新 Follow 集合，将结果存储在 follow 数据结构中。

具体步骤包括：

对于每个产生式右部的非终结符，进行如下操作：

如果当前字符是一个非终结符，并且不在产生式的最后位置，根据其后继字符的情况更新 Follow 集合：

若后继字符不是非终结符且不是 |，且不在当前非终结符的 Follow 集合中，则加入其 Follow 集合。

若后继字符是一个非终结符，则将其 First 集合中不为 'e' 的元素加入当前非终结符的 Follow 集合。

如果当前字符在产生式右部的最后位置且其 Follow 集合中包含 'e'，则将 'e' 加入当前非终结符的 Follow 集合。

总体来说，这个函数实现了对文法规则中每个非终结符的 Follow 集合进行计算和更新的功能。

```
void printFollowSets()
{
    // 打印Follow集合结果
    for (int i = 0; i < gra.size(); i++)
    {
        out3 << gra[i][0] << ": ";
        for (char c : follow[Vncot[gra[i][0]]])
        {
            out3 << c << " ";
        }
        out3 << "\n";
    }
}
```

这个函数实现了对现在 follow 容器内的所有元素的输出

```
void updateFollowSets()
{
    // 更多轮次的Follow集合计算
    int iteration = 2;
    int num[100] = {0};

    while (true)
    {
        out3 << "第" << iteration++ << "轮:\n";
        bool sign = true;

        // 检查是否有变化，若无变化则结束循环
        for (int i = 0; i < gra.size(); i++)
        {
            if (num[i] != follow[i].size())
            {
                sign = false;
                num[i] = follow[i].size();
            }
        }

        if (sign == true)
        {
            out3 << "结束\n";
            break;
        }

        for (int i = 0; i < gra.size(); i++)
        {
            for (int j = 3; j < gra[i].size(); j++)
            {
                if ((gra[i][j] >= 'A' && gra[i][j] <= 'Z'))
                {
                    if (j < gra[i].size() - 1)
                    {
                        if (gra[i][j + 1] == '|')
                        {
                            int num3 = Vncot[gra[i][0]];
                            int num4 = Vncot[gra[i][j]];
                            for (int k = 0; k < follow[num3].size(); k++)
                            {
                                if (!inanalst(follow[num4], follow[num3][k]))
                                {
                                    follow[num4].push_back(follow[num3][k]);
                                }
                            }
                        }
                        else if (isupper(gra[i][j + 1]))
                        {
                            int num5 = Vncot[gra[i][j + 1]];
                            if (inanalst(first[num5], 'e'))
                            {
                                int num4 = Vncot[gra[i][j]];
                                for (int k = 0; k < follow[num5].size(); k++)
                                {
                                    if (!inanalst(follow[num4], follow[num5][k]))
                                    {
                                        follow[num4].push_back(follow[num5][k]);
                                    }
                                }
                            }
                        }
                    }
                    else if (j == gra[i].size() - 1)
                    {
                        int num3 = Vncot[gra[i][0]];
                        int num4 = Vncot[gra[i][j]];
                        for (int k = 0; k < follow[num3].size(); k++)
                        {
                            if (!inanalst(follow[num4], follow[num3][k]))
                            {
                                follow[num4].push_back(follow[num3][k]);
```

初始化迭代计数 iteration 和 num 数组，用于跟踪 Follow 集合是否发生变化。

使用一个无限循环来执行 Follow 集合的更新，直到 Follow 集合不再发生变化为止。

在循环内部:

输出当前是第几轮迭代。

检查 Follow 集合是否发生变化，若无变化，则跳出循环。

遍历文法规则中的每个产生式右部字符，并根据规则更新 Follow 集合。

根据不同情况更新 Follow 集合：处理非终结符情况和当前字符位置是否在产生式右部末尾。

打印每一轮的 Follow 集合结果。

总体来说，这个函数的目的是通过多次迭代来计算文法规则中每个非终结符的 Follow 集合，直到 Follow 集合不再发生变化为止。

4. 制作表格函数

```
void printTerminalList()
{
    out4 << "表格如下: \n ";

    // 打印终结符列表
    for (const auto &terminal : Vt)
    {
        out4 << terminal << "   ";
    }
    out4 << "\n";
    for (int j = 0; j < Vt.size(); j++)
    {
        out4 << "___";
    }
    out4 << "\n";
}
```

这个函数打印了所有的终结符列表并且打印了分割线

```
void processGrammarSymbols()
{
    // 对每个文法符号循环
    for (int i = 0; i < gra.size(); i++)
    {
        out4 << gra[i][0] << ": ";   // 打印当前文法符号
        int num1 = Vncot[gra[i][0]]; // 获取文法符号在 Vn 数组中的位置

        // 对于每个终结符循环
        // 对于每个终结符循环
        for (int j = 0; j < Vt.size(); j++)
        {
            // 如果当前终结符在文法符号的 FIRST 集合中
            if (inanalst(first[num1], Vt[j]))
            {
                int stat = 0;
                // 遍历文法产生式右部
                for (int k = 3; k < gra[i].size(); k++)
                {
                    // 如果当前字符是非终结符
                    if (isupper(gra[i][k]))
                    {
                        if (stat == 0)
                        {
                            int num2 = Vncot[gra[i][k]];
                            // 如果当前非终结符的 FIRST 集合包含当前终结符
                            if (inanalst(first[num2], Vt[j]))
                            {
                                // 将文法产生式右部加入相应的表格中
                                while (k < gra[i].size() && gra[i][k] != '|')
                                {
                                    table[num1][j].push_back(gra[i][k]);
                                    k++;
                                }
                                break;
                            }
                        }
                    }
                    // 如果当前字符是终结符
                    if (gra[i][k] == Vt[j])
                    {
                        if (stat == 0)
                        {
                            // 将文法产生式右部加入相应的表格中
                            while (k < gra[i].size() && gra[i][k] != '|')
                            {
                                table[num1][j].push_back(gra[i][k]);
                                k++;
                            }
                            break;
                        }
                    }
                    stat = 1;
                    if (gra[i][k] == '|')
                    {
                        stat = 0;
                    }
                }
            }
            else if (inanalst(first[num1], 'e') &&
                     inanalst(follow[num1], Vt[j]))
            {
                // 如果空串在文法符号的 FIRST 集合中，并且当前终结符在 FOLLOW 集合中
                table[num1][j].push_back('e'); // 将空串加入相应的表格中
            }
        }

        // 打印构建的表格
        for (int j = 0; j < Vt.size(); j++)
        {
            if (!table[num1][j].empty())
            {
                for (char c : table[num1][j])
                {
                    out4 << c;
                }
                out4 << "  ";
            }
            else
            {
                out4 << "** "; // 如果表格为空，则打印 "** "
            }
        }
        out4 << "\n";
    }
}
```

对于每个文法符号：

输出当前文法符号。

获取文法符号在非终结符数组 Vn 中的位置 num1。

针对每个终结符 Vt:

如果当前终结符在文法符号的 FIRST 集合中:

遍历文法产生式右部，根据情况将相应的字符添加到表格中。

否则，如果空串在文法符号的 FIRST 集合中，并且当前终结符在 FOLLOW 集合中:

将空串 'e' 添加到表格中。

根据构建的表格输出结果:

打印构建的表格，其中有符号表示该位置可推导出的符号序列，没有符号则打印 "**"。

该函数主要目的是根据文法符号的 FIRST 集合和 FOLLOW 集合构建文法分析表格，以便在语法分析过程中使用。

5. 获取文法函数

```
    out5 << "Vn:\n";

    // 循环遍历文法规则
    for (int i = 0; i < gra.size(); i++)
    {
        // 检查文法规则的开头是否符合非终结符的格式
        if ((isupper(gra[i][0])) && gra[i][1] == '-' && gra[i][2] == '>')
        {
            // 如果是非终结符，则将其添加到Vn数组中
            Vn.push_back(gra[i][0]);
            Vncot[gra[i][0]] = i;      // 使用非终结符映射到文法规则的索引
            out5 << gra[i][0] << ' '; // 打印非终结符
        }
        else
        {
            // 如果不符合非终结符格式，则打印当前文法规则索引并返回-1
            out5 << i << "\n";
            return -1;
        }
    }
    out5 << "\n";

    out5 << "Vt:\n";
    int num1 = 0;
    // 循环遍历文法规则
    for (int i = 0; i < gra.size(); i++)
    {
        for (int j = 3; j < gra[i].size(); j++)
        {
            // 检查文法规则右侧的字符是否为终结符，并且不是 '|' 或 'e'
            if ((!isupper(gra[i][j])) && gra[i][j] != '|' && gra[i][j] != 'e')
            {
                int t, flag = 0;
                // 检查字符是否已存在于Vt数组中
                for (t = 0; t < Vt.size(); t++)
                {
                    if (gra[i][j] == Vt[t])
                    {
                        flag = 1;
                        break;
                    }
                }
                // 如果字符不在Vt数组中，则将其添加到Vt数组中，并记录其位置到Vtcot中
                if (!flag)
                {
                    Vtcot[gra[i][j]] = num1++;
                    Vt.push_back(gra[i][j]);
                    out5 << gra[i][j] << " "; // 打印终结符
                }
            }
        }
    }
    // 将结束符号 '#' 添加到Vt数组末尾，并记录其位置到Vtcot中
    Vt.push_back('#');
    Vtcot['#'] = num1;
    out5 << "\n";
    return 1; // 返回1表示成功
}
```

打印出非终结符和终结符集合，这些集合将在语法分析中使用。

检查文法规则的格式并提取非终结符、终结符集合。

函数内部逻辑：

Vn 表示非终结符集合，Vt 表示终结符集合。

Vncot 和 Vtcot 是非终结符和终结符的位置映射。

首先，检查文法规则的开头是否符合非终结符的格式。如果符合，将非终结符添加到 Vn 数组中，并使用 Vncot 映射非终结符到文法规则的索引。

然后，遍历文法规则的右侧字符，检查是否为终结符，不是 '|' 或 'e'，并将其添加到 Vt 数组中。对于每个不在 Vt 中的终结符，将其位置记录在 Vtcot 中。

最后，将结束符号 '#' 添加到 Vt 数组末尾，并记录其位置到 Vtcot 中。

函数返回值为 1，表示成功执行。如果文法规则开头不符合非终结符的格式，函数会输出当前文法规则索引并返回 -1。

6. 总函数

```
int main()
{
    ifstream open("input.txt");
    if (!open)
    {
        cout << "files wrong!"
             << "\n";
        exit(-1);
    }
    string begin = "i";
    getline(open, anastr);
    getrule();
    anastr += '#';
    analst.push_back('#');
    analst.push_back(gra[0][0]);
    int len = 0;
    while (!analst.empty() && len != anastr.length())
    {
        printanalst(len);
        if (checktopVn(len))
        {
            if (checkok(anastr[len]))
            {
                char check = analst.back();
                out1 << check << " 更改规则为: ";
                analst.pop_back();
                char check2 = anastr[len];
                if (table[Vncot[check]][Vtcot[check2]][0] != 'e')
                {
                    for (int j = table[Vncot[check]][Vtcot[check2]].size() - 1; j >= 0; j--)
                    {
                        out1 << table[Vncot[check]][Vtcot[check2]][j] << " ";
                        analst.push_back(table[Vncot[check]][Vtcot[check2]][j]);
                    }
                    out1 << "\n";
                }
                else
                {
                    out1 << "e\n";
                }
            }
            else
            {
                break;
            }
        }
        else if (checktopVt(len))
        {
            char c = anastr[len];
            if (c != analst.back())
            {
                break;
            }
            else
            {
                analst.pop_back();
                len++;
            }
        }
    }
    /*cout << len << " " << anastr.length() << "\n";
    for(auto it : analst){
        cout << it << "\n";
    }
    */
    if (analst.empty() && len == anastr.length())
        cout << "success!"
             << "\n";
    else
        cout << "failed!"
             << "\n";
    open.close();
```

主要逻辑如下：

打开文件 "input.txt" 以读取文法规则，如果文件打开失败，则输出错误信息并退出程序。

读取文法规则并获取非终结符、终结符集合等信息。

对输入的字符串进行语法分析，使用分析栈 analst 和输入字符串 anastr，并在分析过程中不断检查、比较、修改栈内元素。

执行分析过程中，根据文法规则的表格信息，不断检查栈顶元素和输入字符串的对应字符，并根据文法规则进行栈的变化。

如果分析成功，即分析栈为空并且字符串处理结束，输出 "success!"，否则输出 "failed!"。

最后关闭文件并返回 0。

这段代码涉及使用了分析栈、文法规则的表格信息等内容，以及一系列对栈和输入字符串的处理过程，通过查表和逐步匹配来判断输入字符串是否符合给定的文法规则。

7. 一些检查函数

```
bool checktopVn(int i)
{
    char top = analst.back();
    return (inanalst(Vn, top));
}

bool checktopVt(int i)
{
    char top = analst.back();
    return (inanalst(Vt, top));
}

bool checkok(char c)
{
    char top_non_terminal = analst.back();
    int non_terminal_index = Vncot[top_non_terminal];
    int terminal_index = Vtcot[c];
    return !table[non_terminal_index][terminal_index].empty();
}

void printanalst(int n)
{
    out1 << "栈内容: ";
    for (int i = 0; i < analst.size(); i++)
    {
        out1 << analst[i] << " ";
    }
    out1 << "   剩余输入: ";
    for (int i = n; i < anastr.size(); i++)
    {
        out1 << anastr[i] << " ";
    }
    out1 << "\n";
}
```

checktopVn(int i) 检查分析栈顶部的字符是否是非终结符。

checktopVt(int i) 检查分析栈顶部的字符是否是终结符。

checkok(char c) 检查当前非终结符的栈顶字符对应于输入的终结符 c 是否存在合法的推导规则。

printanalst(int n) 输出当前栈的内容以及剩余的输入字符，用于调试和跟踪语法分析的执行过程。

这些函数主要是用于语法分析过程中对栈顶字符和输入字符进行判断和输出, 以帮助理解分析过程中栈和输入的变化。checkok 函数用于检查给定非终结符的栈顶字符和输入字符是否有匹配的推导规则。

四、 程序测试

仅在测试一中展示 first 和 follow 以及表格

测试一：

first.txt 使用“文本编辑”打开

```
计算First集合:
第1轮:
S:
E:
R: e
T:
Y: e
F: ( i
A: + -
M: * /
V: i
第2轮:
S: i
E:
R: e + -
T: ( i
Y: e * /
F: ( i
A: + -
M: * /
V: i
第3轮:
S: i
E: ( i
R: e + -
T: ( i
Y: e * /
F: ( i
A: + -
M: * /
V: i
第4轮:
S: i
E: ( i
R: e + -
T: ( i
Y: e * /
F: ( i
A: + -
M: * /
V: i
第5轮:
结束
```

follow.txt

使用“文本编辑”打开

```
计算Follow集合:
第1轮:
S: #
E: )
R:
T: + -
Y:
F: * /
A: ( i
M: ( i
V: =
第2轮:
S: #
E: ) #
R: ) #
T: + - ) #
Y: + - ) #
F: * / + - ) #
A: ( i
M: ( i
V: =
第3轮:
S: #
E: ) #
R: ) #
T: + - ) #
Y: + - ) #
F: * / + - ) #
A: ( i
M: ( i
V: =
第4轮:
结束
```

table.txt

表格如下:

	=	(	)	i	+	-	*	/	#
S:	**	**	**	V=E	**	**	**	**	**
E:	**	TR	**	TR	**	**	**	**	**
R:	**	**	e	**	ATR	ATR	**	**	e
T:	**	FY	**	FY	**	**	**	**	**
Y:	**	**	e	**	e	e	MFY	MFY	e
F:	**	(E)	**	i	**	**	**	**	**
A:	**	**	**	**	+	-	**	**	**
M:	**	**	**	**	**	**	*	/	**
V:	**	**	**	i	**	**	**	**	**

progress.txt 使用“文本编辑”打开

```
栈内容: # S    剩余输入: i = ( i + i ) * i #
S 更改规则为: E = V
栈内容: # E = V    剩余输入: i = ( i + i ) * i #
V 更改规则为: i
栈内容: # E = i    剩余输入: i = ( i + i ) * i #
栈内容: # E =    剩余输入: = ( i + i ) * i #
栈内容: # E    剩余输入: ( i + i ) * i #
E 更改规则为: R T
栈内容: # R T    剩余输入: ( i + i ) * i #
T 更改规则为: Y F
栈内容: # R Y F    剩余输入: ( i + i ) * i #
F 更改规则为: ) E (
栈内容: # R Y ) E (    剩余输入: ( i + i ) * i #
栈内容: # R Y ) E    剩余输入: i + i ) * i #
E 更改规则为: R T
栈内容: # R Y ) R T    剩余输入: i + i ) * i #
T 更改规则为: Y F
栈内容: # R Y ) R Y F    剩余输入: i + i ) * i #
F 更改规则为: i
栈内容: # R Y ) R Y i    剩余输入: i + i ) * i #
栈内容: # R Y ) R Y    剩余输入: + i ) * i #
Y 更改规则为: e
栈内容: # R Y ) R    剩余输入: + i ) * i #
R 更改规则为: R T A
栈内容: # R Y ) R T A    剩余输入: + i ) * i #
A 更改规则为: +
栈内容: # R Y ) R T +    剩余输入: + i ) * i #
栈内容: # R Y ) R T    剩余输入: i ) * i #
T 更改规则为: Y F
栈内容: # R Y ) R Y F    剩余输入: i ) * i #
F 更改规则为: i
栈内容: # R Y ) R Y i    剩余输入: i ) * i #
栈内容: # R Y ) R Y    剩余输入: ) * i #
Y 更改规则为: e
栈内容: # R Y ) R    剩余输入: ) * i #
R 更改规则为: e
栈内容: # R Y )    剩余输入: ) * i #
栈内容: # R Y    剩余输入: * i #
Y 更改规则为: Y F M
栈内容: # R Y F M    剩余输入: * i #
M 更改规则为: *
栈内容: # R Y F *    剩余输入: * i #
栈内容: # R Y F    剩余输入: i #
F 更改规则为: i
栈内容: # R Y i    剩余输入: i #
栈内容: # R Y    剩余输入: #
Y 更改规则为: e
栈内容: # R    剩余输入: #
R 更改规则为: e
栈内容: #    剩余输入: #
```

Vn&Vt.txt

```
Vn:
S E R T Y F A M V
Vt:
= ( ) i + - * /
```

测试二:

progress.txt 使用“文本编辑”打开

```
栈内容: # S    剩余输入: i = i + i #
S 更改规则为: E = V
栈内容: # E = V    剩余输入: i = i + i #
V 更改规则为: i
栈内容: # E = i    剩余输入: i = i + i #
栈内容: # E =    剩余输入: = i + i #
栈内容: # E    剩余输入: i + i #
E 更改规则为: R T
栈内容: # R T    剩余输入: i + i #
T 更改规则为: Y F
栈内容: # R Y F    剩余输入: i + i #
F 更改规则为: i
栈内容: # R Y i    剩余输入: i + i #
栈内容: # R Y    剩余输入: + i #
Y 更改规则为: e
栈内容: # R    剩余输入: + i #
R 更改规则为: R T A
栈内容: # R T A    剩余输入: + i #
A 更改规则为: +
栈内容: # R T +    剩余输入: + i #
栈内容: # R T    剩余输入: i #
T 更改规则为: Y F
栈内容: # R Y F    剩余输入: i #
F 更改规则为: i
栈内容: # R Y i    剩余输入: i #
栈内容: # R Y    剩余输入: #
Y 更改规则为: e
栈内容: # R    剩余输入: #
R 更改规则为: e
栈内容: #    剩余输入: #
```

错误测试一:

progress.txt 使用“文本编辑”打开

```
栈内容: # S    剩余输入: i = i + ( #
S 更改规则为: E = V
栈内容: # E = V    剩余输入: i = i + ( #
V 更改规则为: i
栈内容: # E = i    剩余输入: i = i + ( #
栈内容: # E =    剩余输入: = i + ( #
栈内容: # E    剩余输入: i + ( #
E 更改规则为: R T
栈内容: # R T    剩余输入: i + ( #
T 更改规则为: Y F
栈内容: # R Y F    剩余输入: i + ( #
F 更改规则为: i
栈内容: # R Y i    剩余输入: i + ( #
栈内容: # R Y    剩余输入: + ( #
Y 更改规则为: e
栈内容: # R    剩余输入: + ( #
R 更改规则为: R T A
栈内容: # R T A    剩余输入: + ( #
A 更改规则为: +
栈内容: # R T +    剩余输入: + ( #
栈内容: # R T    剩余输入: ( #
T 更改规则为: Y F
栈内容: # R Y F    剩余输入: ( #
F 更改规则为: ) E (
栈内容: # R Y ) E (    剩余输入: ( #
栈内容: # R Y ) E    剩余输入: #
```

错误测试二:

progress.txt

使用“文本编辑”打开

```
栈内容: # S    剩余输入: i = ( #
S 更改规则为: E = V
栈内容: # E = V    剩余输入: i = ( #
V 更改规则为: i
栈内容: # E = i    剩余输入: i = ( #
栈内容: # E =    剩余输入: = ( #
栈内容: # E    剩余输入: ( #
E 更改规则为: R T
栈内容: # R T    剩余输入: ( #
T 更改规则为: Y F
栈内容: # R Y F    剩余输入: ( #
F 更改规则为: ) E (
栈内容: # R Y ) E (    剩余输入: ( #
栈内容: # R Y ) E    剩余输入: #
```