北京交通大学

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

这段代码是一个简单的语法分析程序，它基于 LL(1) 文法，用于分析输入字符串是否符合给定的文法规则。程序包含了以下功能：

读取规则文件(rules.txt)，提取非终结符和终结符，构建文法规则。

计算每个非终结符的 FirstVT 集合和 LastVT 集合。

构建预测分析表。

对输入字符串(anaylsetxt.txt)进行语法分析，检查是否符合文法规则。

程序结构及功能：

getrule(): 从规则文件中读取文法规则，提取非终结符和终结符，构建文法规则。

getfirstvt(): 计算文法规则的 FirstVT 集合。

getlastvt(): 计算文法规则的 LastVT 集合。

gettable(): 根据 FirstVT 和 LastVT 集合构建预测分析表。

anaylse(): 对输入的字符串进行语法分析，检查是否符合文法规则。

main(): 程序入口点，调用以上函数完成整个语法分析过程。

程序的输入文件有：

rules.txt：包含文法规则。

anaylsetxt.txt：包含待分析的字符串。

程序的输出有：

progress.txt：记录语法分析的中间步骤。

firstvt.txt、lastvt.txt、table.txt、Vn&Vt.txt：分别记录计算得到的 FirstVT、LastVT、预测分析表以及非终结符和终结符。

最终的分析结果会在 progress.txt 中输出。

接下来是函数的详细说明：

1. 读取规则文件和构建文法规则 (getrule() 函数)

功能：

从名为 rules.txt 的文件中读取文法规则。

提取非终结符和终结符，并构建文法规则。

具体流程：

读取文件中的每一行，将文法规则存储在 gra 向量中。

根据规则提取非终结符和终结符，构建非终结符数组 Vn 和终结符数组 Vt，并使用 map 记录它们的索引。

输出提取得到的非终结符和终结符到文件 Vn&Vt.txt。

2. 计算 FirstVT 集合 (getfirstvt() 函数)

功能：

计算文法规则的 FirstVT 集合。

具体流程：

对每个文法规则进行处理，从左到右遍历每个产生式右部的第一个字符，将其加入相应的非终结符的 FirstVT 集合中。

循环迭代处理每个非终结符的 FirstVT 集合，直到集合不再变化。

输出计算得到的 FirstVT 到文件 firstvt.txt。

3. 计算 LastVT 集合 (getlastvt() 函数)

功能：

计算文法规则的 LastVT 集合。

具体流程：

对每个文法规则进行处理，从右到左遍历每个产生式右部的最后一个字符，将其加入相应的非终结符的 LastVT 集合中。

循环迭代处理每个非终结符的 LastVT 集合，直到集合不再变化。

输出计算得到的 LastVT 到文件 lastvt.txt。

4. 构建预测分析表 (gettable() 函数)

功能：

基于 FirstVT 和 LastVT 集合构建预测分析表。

具体流程：

使用 table 数组构建预测分析表，根据文法规则中的规律填充表格。

检查规则冲突，并在出现错误时输出到文件 table.txt。

5. 对输入字符串进行语法分析 (anaylse() 函数)

功能：

对输入的字符串进行语法分析，检查是否符合文法规则。

具体流程：

读取名为 anaylsetxt.txt 的文件中的待分析字符串。

使用预测分析表对输入字符串进行分析，执行移进和归约操作。

根据语法分析过程的每一步，输出中间步骤和结果到文件 progress.txt。

输出最终的分析结果，判断输入字符串是否符合文法规则。

6. 主函数 (main())

功能：

程序入口点。

调用以上各个函数，完成整个语法分析的过程。

二、 主要数据结构

变量及类型	用途
vector<string> gra	存储文法规则
vector<char> Vn	存储非终结符
vector<char> Vt	存储终结符
map<char, int> Vncot	将非终结符映射到文法规则的索引
map<char, int> Vtcot	将终结符映射到终结符集合的索引
vector<char> firstVt[100]	存储每个非终结符的 FirstVT 集合
vector<char> lastVt[100]	存储每个非终结符的 LastVT 集合
char table[100][100]	预测分析表格，存储分析过程中的移进和归约关系
vector<char> analst	存储语法分析的过程中的符号栈
string anastr	存储待分析的字符串

vector<string> gra：

用途：存储文法规则。每个字符串代表一个文法规则，字符串内部表示产生式的左部和右部。

vector<char> Vn：

用途：存储非终结符集合。该向量存放了文法规则中的所有非终结符。

vector<char> Vt：

用途：存储终结符集合。该向量存放了文法规则中的所有终结符。

map<char, int> Vncot：

用途：将非终结符映射到文法规则的索引。通过这个映射，可以快速获取特定非终结符对应的文法规则索引。

map<char, int> Vtcot：

用途：将终结符映射到终结符集合的索引。类似于 Vncot，用于快速获取特定终结符的索引。

vector<char> firstVt[100]：

用途：存储每个非终结符的 FirstVT 集合。用向量数组的形式保存每个非终结符的 FirstVT。

vector<char> lastVt[100]：

用途：存储每个非终结符的 LastVT 集合。类似于 firstVt，但存储的是 LastVT。

char table[100][100]：

用途：存储预测分析表格。用于构建分析过程中的移进和归约关系，帮助进行语法分析。

vector<char> analst：

用途：作为语法分析过程中的符号栈。存储分析时所使用的符号栈信息。

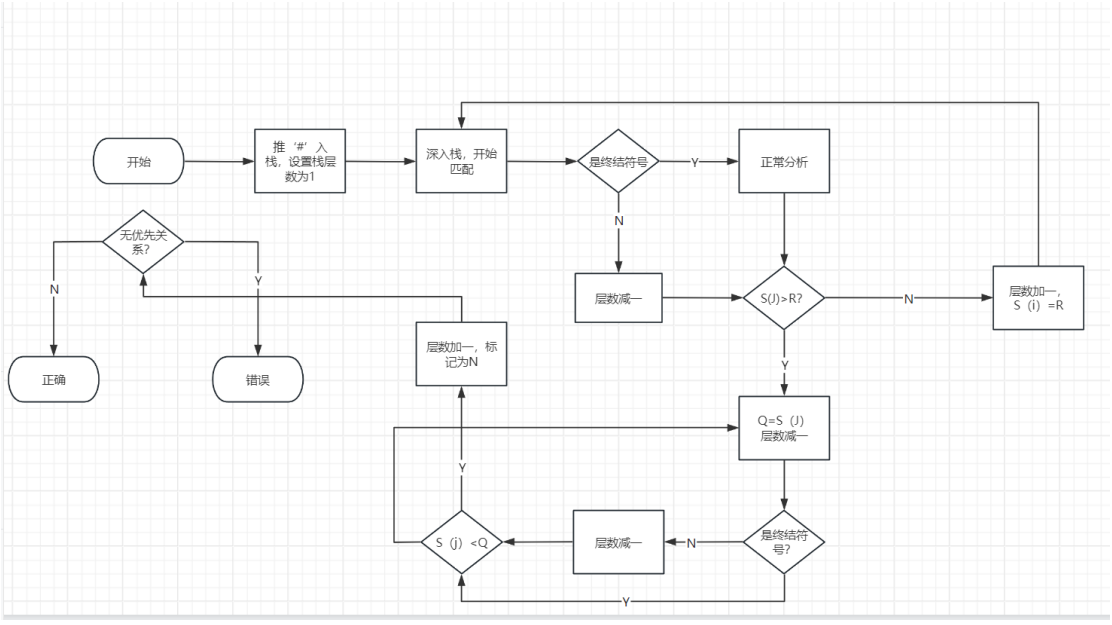
string anastr：

用途：存储待分析的字符串。在语法分析的过程中，存储需要进行分析的字符串信息。

三、 程序结构描述

（1）设计方法

程序流程图如下所示：



（2）函数定义及函数间的调用关系

函数名称	函数功能描述
getrule()	读取规则文件并解析文法规则，获取非终结符 Vn 和终结符 Vt 的集合
getfirstvt()	计算文法中非终结符的 FirstVT 集合
getlastvt()	计算文法中非终结符的 LastVT 集合
gettable()	根据文法规则和计算出的 FirstVT 和 LastVT 集合构建分析表
anaylse()	对输入的待分析字符串进行语法分析，判断是否符合文法规则
main()	主函数，程序入口点，负责调用其他函数执行整个语法

分析过程

以下是详细说明：

```
void getfirstvt()
{
    out2 << "计算FirstVT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 (stat == 0)
                {
                    firstVt[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] != firstVt[i].size())
            {
                flag = false;
                num[i] = firstVt[i].size();
            }
        }

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

        for (int i = 0; i < gra.size(); i++)
        {

            int flag = 0;

            for (int j = 3; j < gra[i].size(); j++)
            {
                if (isupper(gra[i][j]))
                {
                    if (flag == 0)
                    {
                        for (int k = 0; k < firstVt[Vncot[gra[i][j]]].size(); k++)
                        {
                            if (!inanalst(firstVt[i], firstVt[Vncot[gra[i][j]]][k]))
                            {
                                firstVt[i].push_back(firstVt[Vncot[gra[i][j]]][k]);
                            }
                        }
                    }
                }
                if (gra[i][j] == '|')
                {
                    flag = 0;
                    continue;
                }
                flag = 1;
            }
            out2 << gra[i][0] << ": ";
            for (int j = 0; j < firstVt[i].size(); j++)
            {
                out2 << firstVt[i][j] << " ";
            }
            out2 << "\n";
        }
    } while (true);

    out2 << "firstVT如下: \n";
    for (int i = 0; i < gra.size(); i++)
    {
        out2 << Vn[i] << ": ";
        for (int j = 0; j < firstVt[i].size(); j++)
        {
            out2 << firstVt[i][j] << " ";
        }
        out2 << '\n';
    }
    out2.close();
}
```

这段函数是 getfirstvt()，用于计算文法中非终结符的 FirstVT 集合。这个函数执行以下步骤：

初始化阶段：

在输出流 out2 中打印 "计算 FirstVT 集合:" 和 "第 1 轮:"。

设置迭代次数 iteration 为 1。

初始化数组 num，用于记录每个非终结符的 FirstVT 集合的大小。

遍历文法规则：

对于每个文法规则，按照产生式的右部字符将终结符添加到相应非终结符的 FirstVT 集合中。

在输出流 out2 中打印每个非终结符对应的终结符集合。

迭代计算 FirstVT：

使用迭代的方式计算 FirstVT 集合，直到没有新的终结符被添加。

每轮迭代中，遍历文法规则，并根据产生式右部的符号将终结符添加到相应的非终结符的 FirstVT 集合中。

若某轮迭代中 FirstVT 集合的大小没有变化，则结束迭代。

输出结果：

在最终结束迭代后，将计算得到的 FirstVT 集合输出到 out2 中。

输出每个非终结符对应的 FirstVT 集合。

总体来说，这个函数的目的是根据文法规则和产生式右部的终结符，计算每个非终结符的 FirstVT 集合，并通过迭代计算确保获取了完整的 FirstVT 集合。最终将计算结果输出到指定输出流中供进一步分析和使用。

```
void getlastvt()
{
    out3 << "计算lastVt集合:\n";
    out3 << "第1轮:\n";
    int iteration = 1;
    int num[100] = {0};
    for (int i = 0; i < gra.size(); i++)
    {
        out3 << 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 = gra[i].size() - 1; j >= 3; j--)
        {
            if (!isupper(gra[i][j]) && gra[i][j] != '|')
            {
                if (stat == 0)
                {
                    lastVt[i].push_back(gra[i][j]);
                    out3 << gra[i][j] << " ";
                    stat = 1;
                }
            }
            if (gra[i][j] == '|')
                stat = 0;
        }
        out3 << "\n";
    }
    iteration++;
    bool flag = false;

    do
    {
        out3 << "第" << iteration++ << "轮:\n";
        flag = true;

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

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

        for (int i = 0; i < gra.size(); i++)
        {

            int flag = 0;

            for (int j = gra[i].size() - 1; j >= 3; j--)
            {
                if (isupper(gra[i][j]))
                {
                    if (flag == 0)
                    {
                        for (int k = 0; k < lastVt[Vncot[gra[i][j]]].size(); k++)
                        {
                            if (!inanalst(lastVt[i], lastVt[Vncot[gra[i][j]]][k]))
                            {
                                lastVt[i].push_back(lastVt[Vncot[gra[i][j]]][k]);
                            }
                        }
                    }
                }
                if (gra[i][j] == '|')
                {
                    flag = 0;
                    continue;
                }
                flag = 1;
            }
            out3 << gra[i][0] << ": ";
            for (int j = 0; j < lastVt[i].size(); j++)
            {
                out3 << lastVt[i][j] << " ";
            }
            out3 << "\n";
        }
    } while (true);

    out3 << "lastVt如下: \n";
    for (int i = 0; i < gra.size(); i++)
    {
        out3 << Vn[i] << ": ";
        for (int j = 0; j < lastVt[i].size(); j++)
        {
            out3 << lastVt[i][j] << " ";
        }
        out3 << '\n';
    }
    out3.close();
}
```

这段代码定义了函数 getlastvt()，其功能是计算文法中非终结符的 lastVt 集合。以下是该函数的详细解释：

初始化阶段：

在输出流 out3 中打印 "计算 lastVt 集合:" 和 "第 1 轮:"。

初始化迭代次数 iteration 为 1。

初始化数组 num，用于记录每个非终结符的 lastVt 集合的大小。

遍历文法规则：

对于每个文法规则，按照产生式的右部字符将终结符添加到相应非终结符的 lastVt 集合中。

在输出流 out3 中打印每个非终结符对应的终结符集合。

迭代计算 lastVt：

使用迭代的方式计算 lastVt 集合，直到没有新的终结符被添加。

每轮迭代中，遍历文法规则，并根据产生式右部的符号将终结符添加到相应的非终结符的 lastVt 集合中。

若某轮迭代中 lastVt 集合的大小没有变化，则结束迭代。

输出结果：

在最终结束迭代后，将计算得到的 lastVt 集合输出到 out3 中。

输出每个非终结符对应的 lastVt 集合。

总体来说，这个函数是根据文法规则和产生式右部的终结符，计算每个非终结符的 lastVt 集合，并通过迭代计算确保获取了完整的 lastVt 集合。最终将计算结果输出到指定输出流中供进一步分析和使用。

```
void gettable()
{
    memset(table, '*', sizeof(table));
    for (int i = 0; i < gra.size(); i++)
    {
        for (int j = 3; j < gra[i].size(); j++)
        {
            if (j < gra[i].size() - 1 && (isupper(gra[i][j + 1])) && ((!isupper(gra[i][j])) && gra[i][j] != '|'))
            {
                int num1 = Vncot[gra[i][j + 1]];
                int num2 = Vtcot[gra[i][j]];
                for (int k = 0; k < firstVt[num1].size(); k++)
                {
                    if (table[num2][Vtcot[firstVt[num1][k]]] == '*' || table[num2][Vtcot[firstVt[num1][k]]] == '<')
                    {
                        table[num2][Vtcot[firstVt[num1][k]]] = '<';
                    }
                    else
                    {
                        out4 << "错误！ 错误发生在：\t";
                        out4 << i << "次序中\n";
                        out4 << gra[i][j] << "与" << firstVt[num1][k] << "关系不唯一！\n";
                        exit(-1);
                    }
                }
            }
            if (j > 3 && (isupper(gra[i][j - 1])) && (!isupper(gra[i][j])) && gra[i][j] != '|')
            {
                int num1 = Vncot[gra[i][j - 1]];
                int num2 = Vtcot[gra[i][j]];
                for (int k = 0; k < lastVt[num1].size(); k++)
                {
                    if (table[Vtcot[lastVt[num1][k]]][num2] == '*' || table[Vtcot[lastVt[num1][k]]][num2] == '>')
                    {
                        table[Vtcot[lastVt[num1][k]]][num2] = '>';
                    }
                    else
                    {
                        out4 << "错误！ 错误发生在：\t";
                        out4 << num1 << "次序中\n";
                        out4 << lastVt[num1][k] << "与" << gra[i][j] << "关系不唯一！\n";
                        exit(-1);
                    }
                }
            }
            if (j > 3 && j < gra[i].size() && (isupper(gra[i][j])) && ((!isupper(gra[i][j - 1])) && gra[i][j - 1] != '|') && ((!isupper(gra[i][j + 1])) && gra[i][j + 1] != '|'))
            {
                int num1 = Vtcot[gra[i][j - 1]];
                int num2 = Vtcot[gra[i][j + 1]];
                if (table[num1][num2] == '*' || table[num1][num2] == '=')
                {
                    table[num1][num2] = '=';
                }
                else
                {
                    out4 << "错误！ 错误发生在：\t";
                    out4 << i << "次序中\n";
                    out4 << gra[i][j - 1] << "与" << gra[i][j + 1] << "关系不唯一！\n";
                    exit(-1);
                }
            }
        }
    }
    Vt.push_back('#');
    int num3 = Vtcot['#'];
    table[num3][num3] = '=';
    for (int i = 0; i < firstVt[Vncot[gra[0][0]]].size(); i++)
    {
        int num4 = Vtcot[firstVt[Vncot[gra[0][0]]][i]];
        table[num3][num4] = '<';
    }
    for (int i = 0; i < lastVt[Vncot[gra[0][0]]].size(); i++)
    {
        int num4 = Vtcot[lastVt[Vncot[gra[0][0]]][i]];
        table[num4][num3] = '>';
    }
    out4 << " ";
    for (int i = 0; i < Vt.size(); i++)
    {
        out4 << Vt[i] << " ";
    }
    out4 << "\n";
    for (int i = 0; i < Vt.size(); i++)
    {
        out4 << "__";
    }
    out4 << "\n";
    for (int i = 0; i < Vt.size(); i++)
    {
        out4 << Vt[i] << " ";
        for (int j = 0; j < Vt.size(); j++)
        {
            out4 << table[i][j] << " ";
        }
        out4 << "\n";
    }
    out4.close();
}
```

这段代码定义了一个函数 gettable()，其目的是根据给定的文法规则，计算并生成分析表（分析表是用于语法分析的表格，通常用于 LL(1)分析器）。以下是该函数的详细解释：

初始化：

使用 memset 函数将名为 table 的二维字符数组初始化为 '*'

遍历文法规则：

对于每个文法规则，根据产生式右部的特定关系，填充分析表格 table。

填充分析表格：

分析文法规则的右部，根据右部符号之间的关系更新分析表 table：

如果是 A->aB 形式的产生式，a 是终结符，B 是非终结符，则将 table[A][a]

标记为 '<'。

如果是 A->Ba 形式的产生式，a 是终结符，B 是非终结符，则将 table[a][A] 标记为 '>'。

如果是 A->aBb 形式的产生式，a 和 b 是终结符，B 是非终结符，则将 table[a][b] 标记为 '='。

处理 '#'，开始和结束标记：

将 '#' 加入终结符集合 Vt 中，并且标记 table['#']['#'] 为 '='。

根据文法的起始符号，将其对应的 firstVt 和 lastVt 集合中的终结符与 '#' 的关系添加到分析表中。

输出分析表格：

将生成的分析表格输出到文件 out4 中，包括终结符、表格的行列标签和具体的分析表内容。

总体来说，这个函数根据文法规则和产生式右部的关系，填充分析表格 table，并将其输出到指定文件中供后续的语法分析使用。这是构建 LL(1) 分析器的关键步骤之一。

```
int getRule()
{
    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] != '|')
            {
                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
    out5.close();
}
```

这段代码定义了一个名为 getRule() 的函数，其目的是分析给定的文法规则，并进行以下操作：

检查非终结符和终结符：

遍历文法规则，检查文法规则的格式是否符合非终结符格式。

将符合格式的非终结符添加到 Vn 数组中，并使用非终结符映射到文法规则的索引，存储到 Vncot 中。

检查文法规则右侧的字符是否为终结符，将不在 Vt 数组中的终结符添加到

Vt 数组中，并使用终结符映射到位置的方式存储到 Vtcot 中。

结束符号 '#' 处理：

将结束符号 '#' 加入到 Vt 数组末尾，并且记录其位置到 Vtcot 中。

输出结果：

将分析后的非终结符和终结符输出到文件 out5 中。

函数返回值：

返回值为 1 表示成功执行。

```
void anaylse()
{
    ifstream input1("anaylsetxt.txt");
    if (!input1)
    {
        out1 << "无法打开！！！\n";
        exit(-1);
    }
    getline(input1, anastr);
    input1.close();
    anastr += '#';
    int cot, i = 0, step = 0;
    char c;
    analst.push_back('#');
    do
    {
        cot = analst.size() - 1;
        while (analst[cot] == 'N')
        {
            cot--;
        }
        if (table[Vtcot[analst[cot]]][Vtcot[anastr[i]]] == '>')
        {
            c = Vtcot[analst[cot]];
            while (analst[cot] == 'N' || Vtcot[analst[cot]] == c)
            {
                cot--;
            }
            for (int p = 0; p <= analst.size(); p++)
            {
                out1 << analst[p];
            }
            out1 << "    ";
            for (int p = i; p <= anastr.length(); p++)
            {
                out1 << anastr[p];
            }
            out1 << "\t执行归约\n";
            analst.erase(analst.begin() + cot + 1, analst.end());
            analst.push_back('N');
        }
        else if (table[Vtcot[analst[cot]]][Vtcot[anastr[i]]] == '<' || table[Vtcot[analst[cot]]][Vtcot[anastr[i]]] == '=')
        {
            analst.push_back(anastr[i]);
            i++;
            for (int p = 0; p <= analst.size(); p++)
            {
                out1 << analst[p];
            }
            out1 << "    ";
            for (int p = i; p <= anastr.length(); p++)
            {
                out1 << anastr[p];
            }
            out1 << "  ";
            if (anastr[i] != '#')
            {
                out1 << "\t执行移进"
                    << "\n";
            }
            else
                out1 << "\n";
        }
        else
        {
            out1 << "句子有误"
                << "\n";
            exit(0);
        }
    } while (anastr[i] != '#');

    cot = analst.size() - 1;
    while (cot > 0)
    {
        while (analst[cot] == 'N')
            cot--;
        if (table[Vtcot[analst[cot]]][Vtcot[anastr[i]]] == '>')
        {
            c = Vtcot[analst[cot]];
            while (analst[cot] == 'N' || Vtcot[analst[cot]] == c)
            {
                cot--;
            }
            analst.erase(analst.begin() + cot + 1, analst.end());
            analst.push_back('N');
            for (int p = 0; p <= analst.size(); p++)
            {
                out1 << analst[p];
            }
            out1 << "    ";
            for (int p = i; p <= anastr.length(); p++)
```

这段代码定义了一个名为 analyse() 的函数，其作用是对输入的字符串进行分析。这里将输入字符串表示为 anastr，并根据给定的算符优先关系表 table

进行分析。

主要流程如下：

文件输入操作：

打开名为 "anaylsetxt.txt" 的文件，读取其中的文本内容到字符串 anastr 中。如果无法打开文件，则输出错误信息并终止程序。

字符串处理：

在字符串末尾添加结束符号 '#'，同时将 '#' 添加到 analst 数组中作为初始分析栈的栈底。

分析过程：

进入 do-while 循环直到处理完整个输入字符串 anastr。

通过分析 anastr 中的字符与分析栈顶部的字符，根据算符优先关系表 table 中的规则执行归约或移进操作。

输出每一步分析后的栈情况和剩余待分析的字符情况。

句子分析结果输出：

根据分析结果进行输出：

若分析栈最终仅剩下 '#' 和一个 'N'，则表示句子符合语法规则，输出 "分析成功！符合语法！"。

若其他情况（例如：句子有误），则输出 "句子有误"。

文件输出操作：

将分析过程的输出结果写入到名为 outl 的文件中。

函数主要的工作是基于提供的算符优先关系表 table，使用一个分析栈 analst 和输入的字符串 anastr 来进行语法分析。如果输入的字符串符合语法规则，最终在分析栈中会保留 '#' 和一个 'N'。否则，如果分析过程中出现了与算符优先关系表不符的情况，会输出 "句子有误"。

四、 程序测试

由于程序的文法是给定的，所以除了分析过程之外文件的内容均一致，仅仅展示一个样例，更多样例参考文件附件

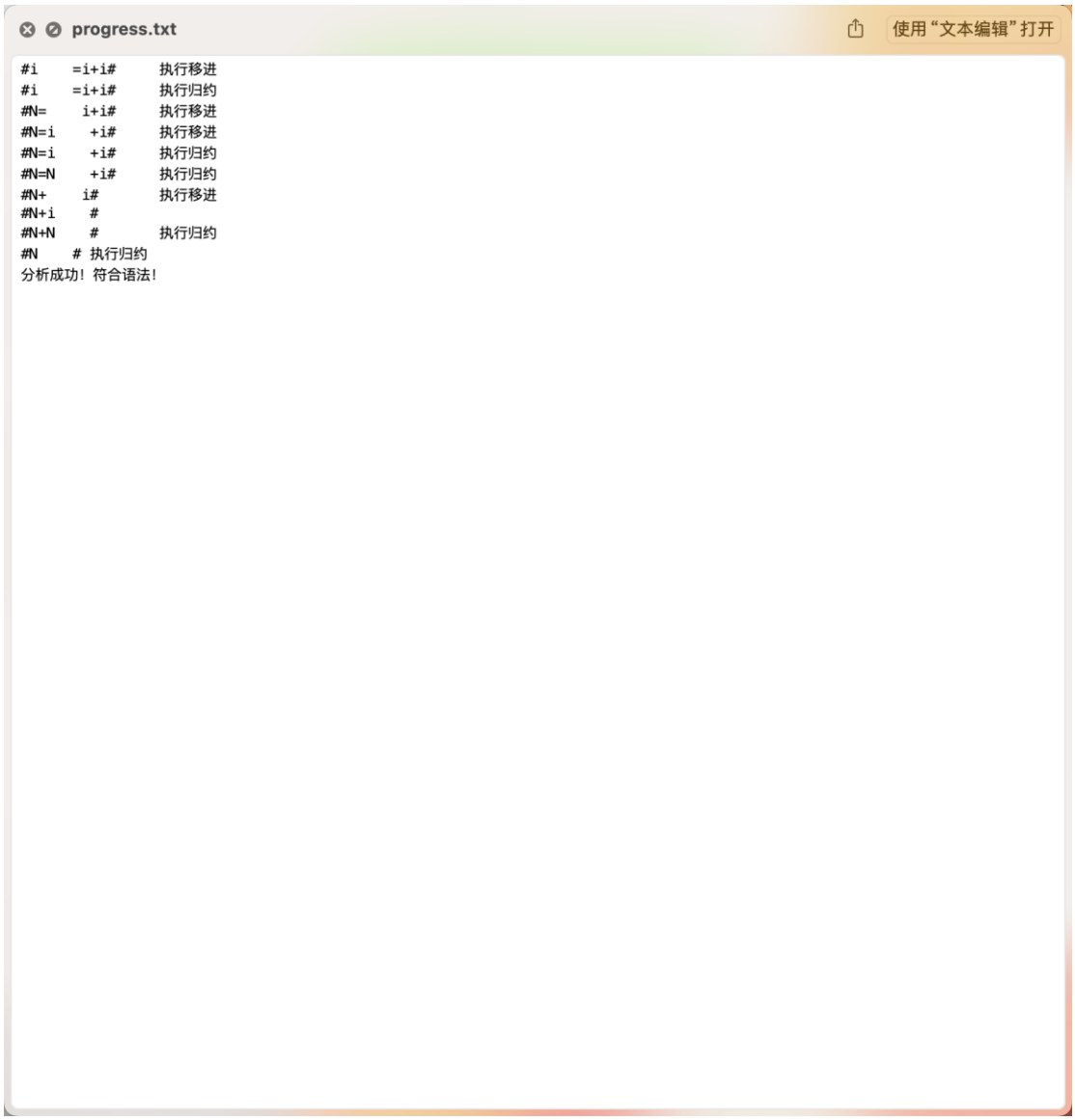

progress.txt
使用“文本编辑”打开
#i =i+i# 执行移进
#i =i+i# 执行归约
#N= i+i# 执行移进
#N=i +i# 执行移进
#N=i +i# 执行归约
#N=N +i# 执行归约
#N+ i# 执行移进
#N+i #
#N+N # 执行归约
#N # 执行归约
分析成功！符合语法！


table.txt
使用“文本编辑”打开
+ - * / ( ) i # #
+ > > < < < > < > *
- > > < < < > < > *
* > > > > < > < > *
/ > > > > < > < > *
(< < < < < = < * *
) > > > > * > * > *
i > > > > * > * > *
< < < < < * < = *
* * * * * * * * *

firstvt.txt

使用“文本编辑”打开

```
计算FirstVT集合:
第1轮:
E: + -
T: * /
F: ( i
第2轮:
E: + - * /
T: * / ( i
F: ( i
第3轮:
E: + - * / ( i
T: * / ( i
F: ( i
第4轮:
E: + - * / ( i
T: * / ( i
F: ( i
第5轮:
结束
firstVT如下:
E: + - * / ( i
T: * / ( i
F: ( i
```

lastvt.txt 使用“文本编辑”打开

```
计算lastVt集合:
第1轮:
E: - +
T: / *
F: i )
第2轮:
E: - + / *
T: / * i )
F: i )
第3轮:
E: - + / * i )
T: / * i )
F: i )
第4轮:
E: - + / * i )
T: / * i )
F: i )
第5轮:
结束
lastVt如下:
E: - + / * i )
T: / * i )
F: i )
```

Vn&Vt.txt

```
Vn:
E T F
Vt:
+ - * / ( ) i
```

*附件（源代码列表）

```cpp
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wdeprecated-declarations"
#include <iostream>
#include <cstring>
#include <fstream>
#include <map>
#include <vector>
#include <stack>

using namespace std;

vector<string> gra;
vector<char> Vn, Vt;
map<char, int> Vncot, Vtcot;
vector<char> firstVt[100], lastVt[100];
char table[100][100];
vector<char> analst;
string anastr;

ofstream out1("progress.txt", ios::app);
ofstream out2("firstvt.txt", ios::app);
ofstream out3("lastvt.txt", ios::app);
ofstream out4("table.txt", ios::app);
ofstream out5("Vn&Vt.txt", ios::app);

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

void getfirstvt()
{
    out2 << "计算 FirstVT 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 (stat == 0)
            {
                firstVt[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] != firstVt[i].size())
        {
            flag = false;
            num[i] = firstVt[i].size();
        }
    }

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

    for (int i = 0; i < gra.size(); i++)
    {
```

```
        int flag = 0;

        for (int j = 3; j < gra[i].size(); j++)
        {
            if (isupper(gra[i][j]))
            {
                if (flag == 0)
                {
                    for (int k = 0; k < firstVt[Vncot[gra[i][j]]].size(); k++)
                    {
                        if (!inanalst(firstVt[i], firstVt[Vncot[gra[i][j]]][k]))
                        {
                            firstVt[i].push_back(firstVt[Vncot[gra[i][j]]][k]);
                        }
                    }
                }
            }
            if (gra[i][j] == '|')
            {
                flag = 0;
                continue;
            }
            flag = 1;
        }
        out2 << gra[i][0] << ": ";
        for (int j = 0; j < firstVt[i].size(); j++)
        {
            out2 << firstVt[i][j] << " ";
        }
        out2 << "\n";
    }
} while (true);

out2 << "firstVT 如下：\n";
for (int i = 0; i < gra.size(); i++)
{
    out2 << Vn[i] << ": ";
    for (int j = 0; j < firstVt[i].size(); j++)
    {
        out2 << firstVt[i][j] << " ";
    }
    out2 << '\n';
}
```

```
    out2.close();
}

void getlastvt()
{
    out3 << "计算 lastVt 集合:\n";
    out3 << "第 1 轮:\n";
    int iteration = 1;
    int num[100] = {0};
    for (int i = 0; i < gra.size(); i++)
    {
        out3 << 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 = gra[i].size() - 1; j >= 3; j--)
        {
            if (!isupper(gra[i][j]) && gra[i][j] != '|')
            {
                if (stat == 0)
                {
                    lastVt[i].push_back(gra[i][j]);
                    out3 << gra[i][j] << " ";
                    stat = 1;
                }
            }
            if (gra[i][j] == '|')
                stat = 0;
        }
        out3 << "\n";
    }
    iteration++;
    bool flag = false;

    do
    {
        out3 << "第" << iteration++ << "轮:\n";
        flag = true;

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

```
    }

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

    for (int i = 0; i < gra.size(); i++)
    {

        int flag = 0;

        for (int j = gra[i].size() - 1; j >= 3; j--)
        {
            if (isupper(gra[i][j]))
            {
                if (flag == 0)
                {
                    for (int k = 0; k < lastVt[Vncot[gra[i][j]]].size(); k++)
                    {
                        if (!inanalst(lastVt[i], lastVt[Vncot[gra[i][j]]][k]))
                        {
                            lastVt[i].push_back(lastVt[Vncot[gra[i][j]]][k]);
                        }
                    }
                }
            }
            if (gra[i][j] == '|')
            {
                flag = 0;
                continue;
            }
            flag = 1;
        }
        out3 << gra[i][0] << ": ";
        for (int j = 0; j < lastVt[i].size(); j++)
        {
            out3 << lastVt[i][j] << " ";
        }
        out3 << "\n";
    }
} while (true);
```

```
    out3 << "lastVt 如下：\n";
    for (int i = 0; i < gra.size(); i++)
    {
        out3 << Vn[i] << ": ";
        for (int j = 0; j < lastVt[i].size(); j++)
        {
            out3 << lastVt[i][j] << " ";
        }
        out3 << '\n';
    }
    out3.close();
}

void gettable()
{
    memset(table, '*', sizeof(table));
    for (int i = 0; i < gra.size(); i++)
    {
        for (int j = 3; j < gra[i].size(); j++)
        {
            if (j < gra[i].size() - 1 && (isupper(gra[i][j + 1])) && ((!isupper(gra[i][j])) && gra[i][j] !=
 '|'))
            {
                int num1 = Vncot[gra[i][j + 1]];
                int num2 = Vtcot[gra[i][j]];
                for (int k = 0; k < firstVt[num1].size(); k++)
                {
                    if (table[num2][Vtcot[firstVt[num1][k]]] == '*' || table[num2][Vtcot[firstVt[num1][k]]]
 == '<')
                    {
                        table[num2][Vtcot[firstVt[num1][k]]] = '<';
                    }
                    else
                    {
                        out4 << "错误！ 错误发生在：\t";
                        out4 << i << "次序中\n";
                        out4 << gra[i][j] << "与" << firstVt[num1][k] << "关系不唯一！\n";
                        exit(-1);
                    }
                }
            }
            if (j > 3 && (isupper(gra[i][j - 1])) && (!isupper(gra[i][j])) && gra[i][j] != '|')
            {
                int num1 = Vncot[gra[i][j - 1]];
```

```
                int num2 = Vtcot[gra[i][j]];
                for (int k = 0; k < lastVt[num1].size(); k++)
                {
                    if (table[Vtcot[lastVt[num1][k]]][num2] == '*' || table[Vtcot[lastVt[num1][k]]][num2] =
= '>')
                    {
                        table[Vtcot[lastVt[num1][k]]][num2] = '>';
                    }
                    else
                    {
                        out4 << "错误！ 错误发生在：\t";
                        out4 << num1 << "次序中\n";
                        out4 << lastVt[num1][k] << "与" << gra[i][j] << "关系不唯一！\n";
                        exit(-1);
                    }
                }
            }
            if (j > 3 && j < gra[i].size() && (isupper(gra[i][j])) && ((!isupper(gra[i][j - 1])) && gra[i][
j - 1] != '|') && ((!isupper(gra[i][j + 1])) && gra[i][j + 1] != '|'))
            {
                int num1 = Vtcot[gra[i][j - 1]];
                int num2 = Vtcot[gra[i][j + 1]];
                if (table[num1][num2] == '*' || table[num1][num2] == '=')
                {
                    table[num1][num2] = '=';
                }
                else
                {
                    out4 << "错误！ 错误发生在：\t";
                    out4 << i << "次序中\n";
                    out4 << gra[i][j - 1] << "与" << gra[i][j + 1] << "关系不唯一！\n";
                    exit(-1);
                }
            }
        }
    }
    Vt.push_back('#');
    int num3 = Vtcot['#'];
    table[num3][num3] = '=';
    for (int i = 0; i < firstVt[Vncot[gra[0][0]]].size(); i++)
    {
        int num4 = Vtcot[firstVt[Vncot[gra[0][0]]][i]];
        table[num3][num4] = '<';
    }
```

```
    for (int i = 0; i < lastVt[Vncot[gra[0][0]]].size(); i++)
    {
        int num4 = Vtcot[lastVt[Vncot[gra[0][0]]][i]];
        table[num4][num3] = '>';
    }
    out4 << " ";
    for (int i = 0; i < Vt.size(); i++)
    {
        out4 << Vt[i] << " ";
    }
    out4 << "\n";
    for (int i = 0; i < Vt.size(); i++)
    {
        out4 << "__";
    }
    out4 << "\n";
    for (int i = 0; i < Vt.size(); i++)
    {
        out4 << Vt[i] << " ";
        for (int j = 0; j < Vt.size(); j++)
        {
            out4 << table[i][j] << " ";
        }
        out4 << "\n";
    }
    out4.close();
}

int getRule()
{
    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] != '|')
        {
            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";
out5.close();
return 1; // 返回1 表示成功
```

```
}

void getrule()
{
    ifstream input("rules.txt");
    if (!input)
    {
        cout << "ruleput went wrong!\n";
        exit(-1);
    }
    string s1;
    while (getline(input, s1))
        gra.push_back(s1);
    input.close();
    getRule();
    getfirstvt();
    getlastvt();
    gettable();
}

void anaylse()
{
    ifstream input1("anaylsetxt.txt");
    if (!input1)
    {
        out1 << "无法打开！！！\n";
        exit(-1);
    }
    getline(input1, anastr);
    input1.close();
    anastr += '#';
    int cot, i = 0, step = 0;
    char c;
    analst.push_back('#');
    do
    {
        cot = analst.size() - 1;
        while (analst[cot] == 'N')
        {
            cot--;
        }
        if (table[Vtcot[analst[cot]]][Vtcot[anastr[i]]] == '>')
        {
            c = Vtcot[analst[cot]];
```

```
            while (analst[cot] == 'N' || Vtcot[analst[cot]] == c)
            {
                cot--;
            }
            for (int p = 0; p < analst.size(); p++)
            {
                out1 << analst[p];
            }
            out1 << "    ";
            for (int p = i; p < anastr.length(); p++)
            {
                out1 << anastr[p];
            }
            out1 << "\t执行归约\n";
            analst.erase(analst.begin() + cot + 1, analst.end());
            analst.push_back('N');
        }
        else if (table[Vtcot[analst[cot]]][Vtcot[anastr[i]]] == '<' || table[Vtcot[analst[cot]]][Vtcot[anas
tr[i]]] == '=')
        {
            analst.push_back(anastr[i]);
            i++;
            for (int p = 0; p < analst.size(); p++)
            {
                out1 << analst[p];
            }
            out1 << "    ";
            for (int p = i; p < anastr.length(); p++)
            {
                out1 << anastr[p];
            }
            out1 << "  ";
            if (anastr[i] != '#')
            {
                out1 << "\t执行移进"
                     << "\n";
            }
            else
                out1 << "\n";
        }
        else
        {
            out1 << "句子有误"
                 << "\n";
```

```
            exit(0);
        }
    } while (anastr[i] != '#');

    cot = analst.size() - 1;
    while (cot > 0)
    {
        while (analst[cot] == 'N')
            cot--;
        if (table[Vtcot[analst[cot]]][Vtcot[anastr[i]]] == '>')
        {
            c = Vtcot[analst[cot]];
            while (analst[cot] == 'N' || Vtcot[analst[cot]] == c)
            {
                cot--;
            }
            analst.erase(analst.begin() + cot + 1, analst.end());
            analst.push_back('N');
            for (int p = 0; p < analst.size(); p++)
            {
                out1 << analst[p];
            }
            out1 << "    ";
            for (int p = i; p < anastr.length(); p++)
            {
                out1 << anastr[p];
            }
            out1 << "\t执行归约\n";
        }
        else
        {
            out1 << "句子有误"
                 << "\n";
            exit(0);
        }
    }
    if(analst.size() == 2 && analst[0] == '#' && analst[1] == 'N'){
        out1 << "分析成功！符合语法！\n";
    }
    out1.close();
}
int main()
{
    getrule();
```

```
    anaylse();
    return 0;
}
#pragma GCC diagnostic pop
```