**语法分析程序的设计与实现**

**目录**

[实验内容及要求 3](#_Toc118746563)

[实验环境 3](#_Toc118746564)

[程序设计说明 4](#_Toc118746565)

[一、总体设计 4](#_Toc118746566)

[1.1、总体思路 4](#_Toc118746567)

[1.2、类设计 4](#_Toc118746568)

[1.2.1、Grammar类 5](#_Toc118746569)

[1.2.2、Production类 6](#_Toc118746570)

[1.2.3、To类 7](#_Toc118746571)

[1.2.4、LL1类 9](#_Toc118746572)

[1.2.5、LR1类 10](#_Toc118746573)

[1.2.6、Item类 12](#_Toc118746574)

[1.2.7、Arith类 13](#_Toc118746575)

[1.3、流程设计 14](#_Toc118746576)

[二、主要模块设计 15](#_Toc118746577)

[2.1、FIRST/FOLLOW集计算 15](#_Toc118746578)

[2.1.1、FIRST集计算 15](#_Toc118746579)

[2.1.2、文法FIRST表计算 16](#_Toc118746580)

[2.1.3、FOLLOW集计算 17](#_Toc118746581)

[2.2、消除左递归 18](#_Toc118746582)

[2.3、提取左公因子 19](#_Toc118746583)

[2.4、DFA构建 19](#_Toc118746584)

[2.4.1、项目集构建及闭包计算 19](#_Toc118746585)

[2.4.2、规范族构建及DFA生成 20](#_Toc118746586)

[2.5、分析表构建 21](#_Toc118746587)

[2.5.1、LL1分析表构建 21](#_Toc118746588)

[2.5.2、LR1分析表构建 23](#_Toc118746589)

[2.6、递归调用预测分析模式 24](#_Toc118746590)

[2.6.1、文法与状态机 24](#_Toc118746591)

[2.6.2、识别程序 25](#_Toc118746592)

[2.7、词法分析器调用 26](#_Toc118746593)

[2.7.1、控制台输入 26](#_Toc118746594)

[2.7.2、记号流输出 26](#_Toc118746595)

[运行及分析 28](#_Toc118746596)

[一、运行方式 28](#_Toc118746597)

[1.1、输入源程序 28](#_Toc118746598)

[1.2、运行参数 28](#_Toc118746599)

[二、运行结果与分析 29](#_Toc118746600)

[2.1、LL1运行结果 29](#_Toc118746601)

[2.1.1、文法信息 29](#_Toc118746602)

[2.1.2、LL1分析表 29](#_Toc118746603)

[2.1.2、LL1分析结果 29](#_Toc118746604)

[2.1.3、错误处理 30](#_Toc118746605)

[2.2、LR1运行结果 31](#_Toc118746606)

[2.2.1、文法信息 31](#_Toc118746607)

[2.2.2、项目集规范族及DFA 32](#_Toc118746608)

[2.2.3、LR1分析表 35](#_Toc118746609)

[2.2.4、LR1分析结果 36](#_Toc118746610)

[2.2.5、LR1错误处理 38](#_Toc118746611)

[2.3、递归调用预测分析运行结果 39](#_Toc118746612)

[2.3.1、分析结果 39](#_Toc118746613)

[2.3.2、错误处理 39](#_Toc118746614)

[LEX + YACC实现 40](#_Toc118746615)

[一、LEX与YACC连接 40](#_Toc118746616)

[1.1、LEX 40](#_Toc118746617)

[1.2、YACC 41](#_Toc118746618)

[1.3、连接 42](#_Toc118746619)

[二、YACC语法分析程序执行 42](#_Toc118746620)

实验内容及要求

* 编写语法分析程序，实现对算数表达式的语法分析。
* 算术表达式文法如下：

**E** 🡪 **E+T | E-T | T**

**T** 🡪 **T\*F | T/F | F**

**F** 🡪 **(E) | num**

* 在对输入的算术表达式进行分析的过程中，依次输出所采用的产生式。
* 实现递归调用分析程序
* 构造LL1预测分析表及预测分析程序
* 构造LR1预测分析表及预测分析程序
* 调用LEX自动生成的词法分析程序，利用YACC自动生成语法分析程序。

实验环境

|  |  |  |
| --- | --- | --- |
| **环境/工具** | **说明** | **版本/网址** |
| **Windows11** | **OS** | **220916** |
| **TDM-gcc** | **编译器** | **10.3.0** |
| **Flex** | **LEX** | **2.5.4** |
| **Bison** | **YACC** | **2.4.1** |

程序设计说明

一、总体设计

1.1、总体思路

**复用词法分析程序单词符号的数据结构**作为语法分析程序的基本结构，同时将运用词法分析的读取模式nextSymbol()进行文法定义文件的读取、记号流的生成。

通过将文法数据结构**拆分为产生式、产生式右端等更具体的数据结构**，降低各模块间的耦合性，同时也使得程序处理流程更为清晰。

**该语法分析程序通过文件读入文法，支持自定义不同文法及符号；并能够自动进行语法变换，包括消除直接左递归、提取左公因子以及文法拓展，能自动构建LL1分析表、LR1分析的DFA及分析表，并能够持续从控制台读入符号串，同时支持通过分号一次输入多个人物流并自动分隔处理。**

1.2、类设计

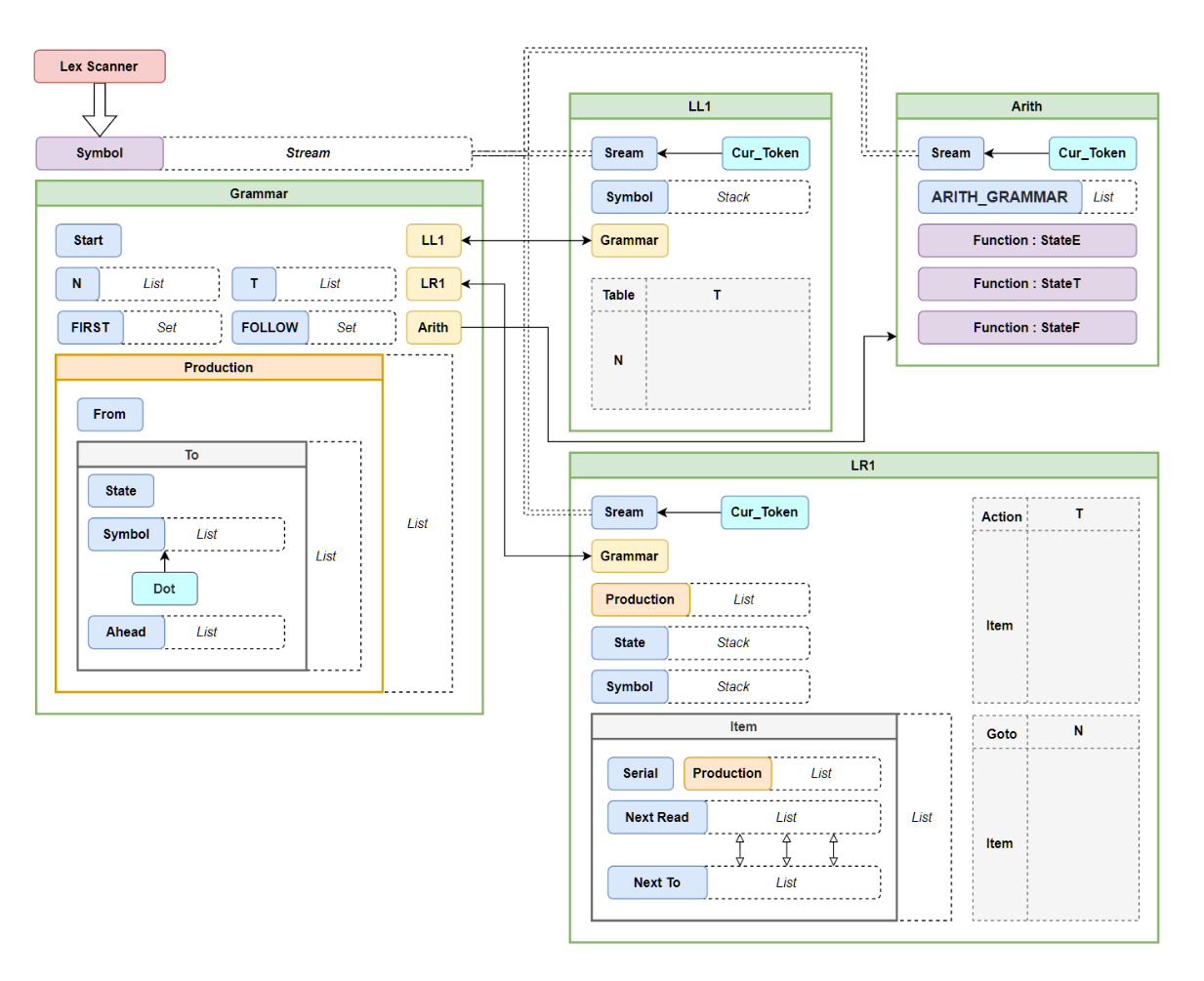
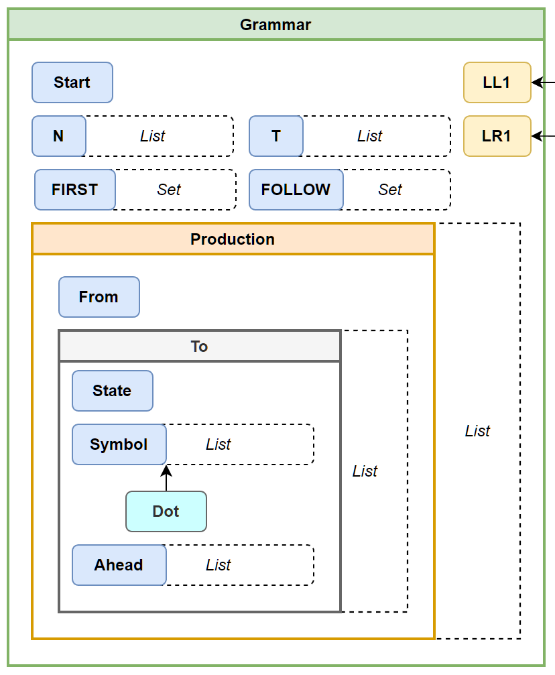


图 1 文法类设计概览

1.2.1、Grammar类

* **类结构**



* **类说明**

记录文法的**详细信息**，包括：

* + - **开始符号**
    - **终结符集合**
    - **非终结符集合**
    - **产生式集合**
    - **FIRST集合**
    - **FOLLOW集合**
* **类定义**

/\* 部分主要文法定义 \*/

class Grammar{

/\* LL1/LR1 处理程序子结构 \*/

class GrammarLL1;

class GrammarLR1;

public:

    /\* 产生式子结构 \*/

    class Production;

public:

/\* Variable \*/

    std::string start;                             // 开始符号

    std::vector<std::string> T;                    // 终结符

    std::vector<std::string> N;                    // 非终结符

    std::vector<Production> P;                     // 产生式

    std::vector<std::set<std::string>> FIRST;      // FIRST集

    std::vector<std::set<std::string>> FOLLOW;     // FOLLOW集

/\* Function \*/

    /\* 非终结符操作 \*/

    bool setStart(std::string);

    bool addN(std::string);

    /\* 插入新产生式 \*/

    void insert(std::string from, std::vector<Symbol> to);

    /\* FIRST FOLLOW集计算\*/

    void computeFirstTable();

    void computeFollowTable();

    std::set<std::string> findFirst(std::vector<Symbol>);   // 通过记符号串

    /\* LL1 \*/

    void eliminateLeftRecursion();  // 消除左递归

    void extractLeftCommonFactor(); // 提取左公因子

    /\* LR1 \*/

    void expandGrammar();           // 拓展文法

    /\* main \*/

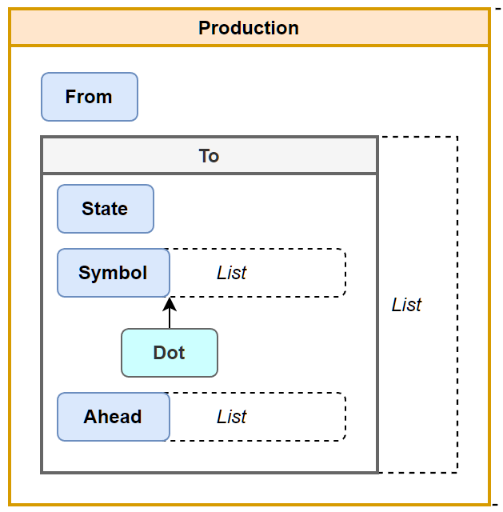
    void runLL1();                  // 运行LL1分析程序

    void runLR1();                  // 运行LR1分析程序

};

1.2.2、Production类

* **类结构**

****

* **类说明**

作为文法类的子结构，记录每个产生式的详细信息，包括产生式的**左部符号和右部结构**。

* **类定义**

/\* 部分主要产生式子结构定义 \*/

class Grammar::Production{

public:

    /\* 产生式右部子结构 \*/

    class To;

public:

/\* Variable \*/

    std::string from;               // 产生式左部

    std::vector<To> to;             // 产生式右部集合

/\* Function \*/

    /\* 插入新产生式右部 \*/

    bool insert(std::vector<Symbol> to);

    /\* 产生式处理 \*/

    bool existLeftRecursion();                  // 左递归判断

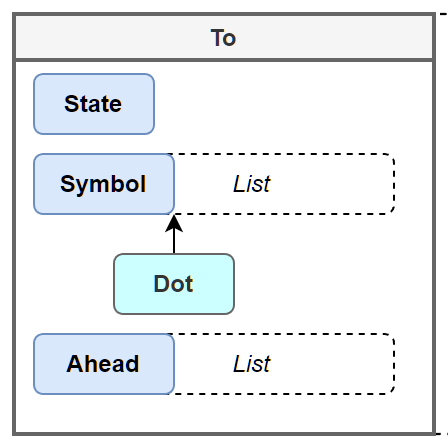
    Production eliminateLeftRecursion();        // 消除左递归, 返回新产生式

    std::set<Symbol> existLeftCommonFactor();   // 左公因子判断, 返回重复因子集合

};

1.2.3、To类

* **类结构**

****

* **类说明**

作为产生式类的子结构，记录每条产生式右部记录的详细信息，包括：

* + **产生式右部符号集合**
  + **向前看符号集合**
  + **点号位置**
  + **当前项目状态**

其中，当前项目状态有：

/\* LR1分析，项目状态定义 \*/

enum class State{

    SHIFT,                  // 移进

    REDUCTION\_EXPECTING,    // 待约

    REDUCTION,              // 归约

};

* **类定义**

/\* 部分主要产生式右部结构定义 \*/

class Grammar::Production::To{

public:

/\* LR1分析，项目状态定义 \*/

enum class State{

    SHIFT,                  // 移进

    REDUCTION\_EXPECTING,    // 待约

    REDUCTION,              // 归约

};

private:

    int \_dot = 0;                           // 点号位置

public:

/\* Variable \*/

    State state;                            // 项目状态

    std::vector<Symbol> elements;           // 产生式右部符号串

    std::set<std::string> ahead;            // 向前看符号集合

/\* Function \*/

    void append(Symbol&);

    /\* LR1 操作 \*/

    int getDot();                           // 获取点号位置

    State getState();                       // 获取当前项目状态

    State next();                           // 向前移动点号

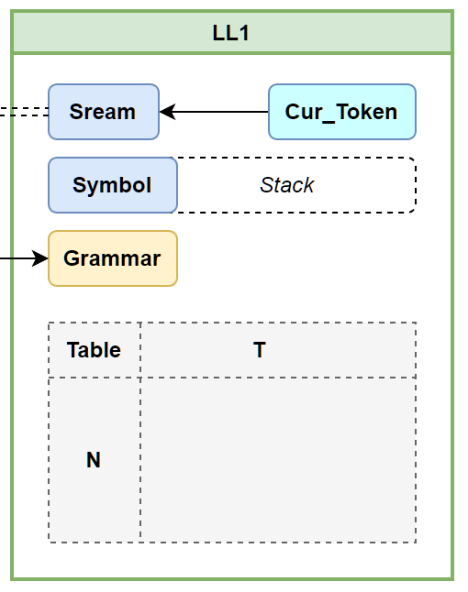
    Symbol getDotSymbol();                  // 获取点号后符号

    std::vector<Symbol> getAheadStream();   // 获取向前看符号计算的符号流

};

1.2.4、LL1类

* **类结构**

****

* **类说明**

LL1类用于LL1分析模式，通过指针连接到文法结构和输入的记号流，存储有LL1分析所需的所有结构：

* + **记号流指针**
  + **文法结构指针**
  + **分析栈**
  + **预测分析表**
* **类定义**

/\* 部分主要LL1 分析结构定义 \*/

class Grammar::GrammarLL1{

const static int ERROR = -1;            // 错误代码

const static int SYNCH = -2;            // 同步代码

private:

/\* Variable \*/

    std::vector<Symbol>\* \_stream;       // 记号流

    std::vector<Symbol> \_state\_stack;   // 分析栈

/\* Function \*/

    /\* 分析操作 \*/

    void \_analyze();                    // 预测分析

    void \_error();                      // 错误处理

public:

/\* Variable \*/

    std::vector<std::vector<int>> table;// 预测分析表

/\* Fucntion\*/

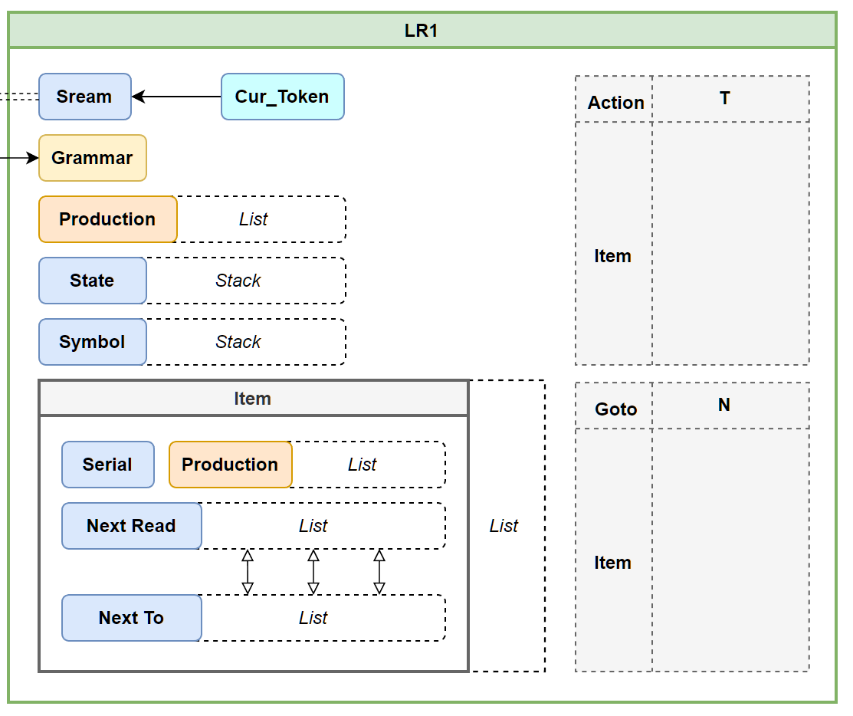
    bool initTable();                   // 构建分析表

    void analyze(std::vector<Symbol>& s\_strem); // 传入记号流分析

};

1.2.5、LR1类

* **类结构**

****

* **类说明**

LR1类用于LR1分析，定义有子结构Item类用于存储**所有项目集规范族**及**识别其所有活前缀的DFA**。同样的，通过指针连接到文法结构及输入的记号流。其也存储了LR1分析所需的所有结构：

* + **带标号的单个产生式**
  + **状态分析栈**
  + **符号分析栈**
  + **Action表**
  + **Goto表**
  + **DFA**

其中，Action表和Goto表的列与文法的终结符/非终结符列表有**对应关系**，通过在文法Grammar结构中找到到终结符/非终结符的**下标**，就能通过该下标指定对应行。

同样的，每一个规范族的**标号Serial**和其在Item数组中的下标也是一致的，Action表和Goto表的行也和Item表具有全对应关系。

* **类定义**

class Grammar::GrammarLR1{

/\* 部分主要项目集规范组子结构定义 \*/

class Item;

/\* Action类型 \*/

enum class ActionType{

    ERROR,      // 错误

    SHIFT,      // 移进

    REDUCTION,  // 归约

    ACCEPT,     // 接受

};

private:

/\* Variable \*/

    /\* LR1分析结构 \*/

    std::vector<Item\*> \_colletion;          // 项目集规范组

    std::vector<int> \_state\_stack;          // 状态分析栈

    std::vector<Symbol> \_symbol\_stack;      // 符号分析栈

    /\* 输入流操作 \*/

    std::vector<Symbol>\* \_stream;           // 记号流

/\* Function \*/

    /\* 分析处理 \*/

    Symbol& \_nextToken();                   // 下一记号

    void \_analyze();                        // 预测分析

    void \_error();                          // 错误处理

public:

/\* Variable \*/

    std::vector<Production> P;          // 带标号的单一文法

    vector<vector<pair<ActionType,int>>> action\_table; // Action分析表

    std::vector<std::vector<int>> goto\_table; // Goto分析表

/\* Function \*/

    /\* 操作 \*/

    void generateDFA();     // 生成DFA

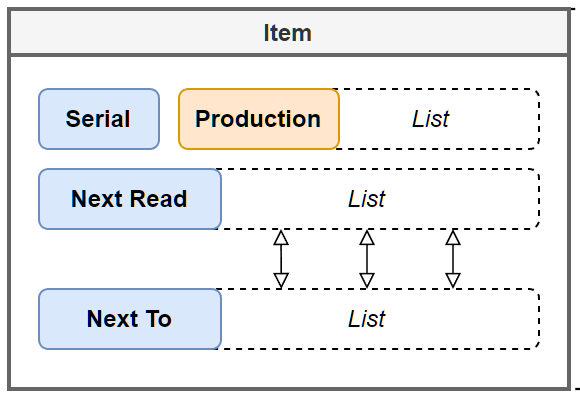
    bool initTable();       // 生成分析表

    void analyze(std::vector<Symbol>& s\_table); // 传入记号流分析

};

1.2.6、Item类

* **类结构**



* **类说明**

Item类作为LR1类的子结构，用于存储**项目集规范族**及通过指针存储**识别其所有活前缀后的规范族**。其中存储有项目集规范组的必要信息：

* + 规范族标号
  + 当前产生式列表
  + 活前缀识别列表
  + 识别后状态列表

其中，Next Read和Next To数组的内容是一一对应的，通过在Next Read中找到活前缀下标，就能通过该下标在Next To数组中定位到识别该活前缀后的规范组标号。

* **类定义**

/\* 部分主要项目集规范组子结构定义 \*/

class Grammar::GrammarLR1::Item{

private:

    void \_computeAhead(Production&,Production&);    // 向前看符号计算

public:

/\* Variable \*/

    int \_serial;                           // 标号

    std::vector<Production> \_productions;   // 当前产生式列表

    std::vector<Symbol> \_nextRead;         // 识别活前缀列表

    std::vector<int> \_nextTo;               // 识别后下一规范组列表

/\* Function \*/

    /\* 操作 \*/

    int insert(Production&);    // 插入产生式

    void closure();             // 计算闭包

    void next();                // 活前缀识别及添加新规范组

};

1.2.7、Arith类

* **类结构**

图形用户界面, 图示

描述已自动生成

* **类说明**

Arith类作为**专用于算数表达式递归调用预测分析**的结构，定义有算数运算符号作为终结符，以及存储有输入的记号流指针。需要另外说明的是，由于该类专用于算数表达式的文法识别，因此各非终结符和递归预测识别函数已经提前定义为函数。

* **类定义**

class Arith{

/\* 非终结符 \*/

enum class State{

    E,T,F

};

/\* 终结符定义 \*/

const Symbol \_\_ADD = Symbol("+",Symbol::Type::OPERATOR);

const Symbol \_\_SUB = Symbol("-",Symbol::Type::OPERATOR);

const Symbol \_\_MUL = Symbol("\*",Symbol::Type::OPERATOR);

const Symbol \_\_DIV = Symbol("/",Symbol::Type::OPERATOR);

const Symbol \_\_LBK = Symbol("(",Symbol::Type::LIMIT);

const Symbol \_\_RBK = Symbol(")",Symbol::Type::LIMIT);

const Symbol ENDING = Symbol("$",Symbol::Type::END);

private:

/\* Variable \*/

    std::vector<Symbol>\* \_stream;   // 记号流

    Symbol \_cur\_token;              // 当前记号

/\* Function \*/

    Symbol& \_nextToken();           // 下一记号

    void \_outputPair(State,int);    // 输出分析情况

    void \_error();                  // 错误处理

    void \_analyze();                // 分析

    bool \_stateE();                 // 处理非终结符E的产生式

    bool \_stateT();                 // 处理非终结符T的产生式

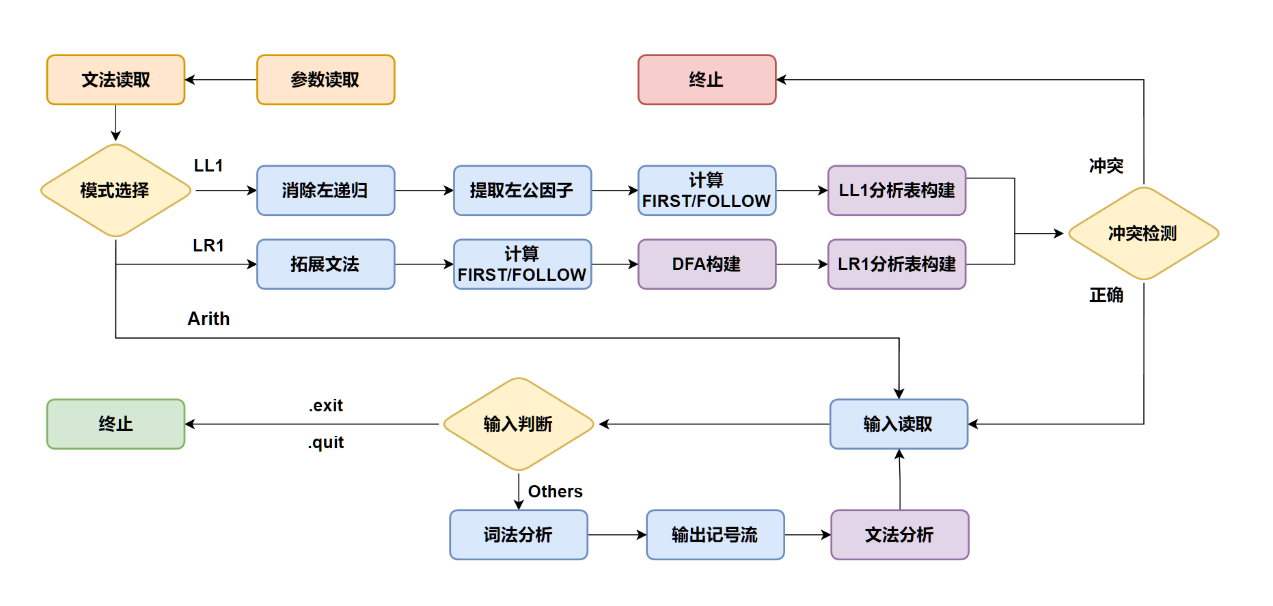
    bool \_stateF();                 // 处理非终结符F的产生式

public:

    void analyze(std::vector<Symbol>&); // 传入记号流分析

};

1.3、流程设计

****

二、主要模块设计

2.1、FIRST/FOLLOW集计算

2.1.1、FIRST集计算

对于文法符号串的FIRST集合，采用以下计算规则：

* 如果，则
* 如果，且 ，其中，则把加入中。

如果，则把 加入。

* 如果，且，则把中所有非元素加入到中。

若，对于某个，到都含有，即，则把中的所有非元素加入到中。

若所有都含有，则把加入中。

**将该规则表述于程序设计，流程如下：**

图示

描述已自动生成

部分主要的程序代码如下：

set<string> Grammar::findFirst(vector<Symbol> to)

{

    std::set<std::string> first;  // 定义容器

    for(auto ele:to){/\* 遇到终结符 \*/

        if(ele.attr() != Type::NOT\_TERMINAL){

            first.insert(ele.content());    // 将其加入FIRST

            epsilon\_flag = false;           // 最终不推出epsilon

            break;

        }else{/\* 遇到非终结符 \*/

            int pos = \_findN(ele);    // 定位该非终结符下标

            bool has\_epsilon = false;   // 该非终结符是否推出epsilon

            for(auto ele : FIRST[pos]){ // 对该非终结符的每一个FIRST集元素

                if(ele != EPSILON\_STR)

                    first.insert(ele); // 将其非epsilon元素插入现有容器

                else has\_epsilon = true;  // 能推出epsilon，设置该记号

            }

            if(!has\_epsilon){             // 该非终结符不推出epsilon

                epsilon\_flag = false;   // 最终不推出epsilon

                break;                 // 停止继续向下遍历

            }

        }

    }

    if(epsilon\_flag)                       // 所有符号都能推出epsilon

        first.insert(EPSILON\_STR);         // 将epsilon加入现有容器

    return first; // 返回计算结果

}

2.1.2、文法FIRST表计算

**参照符号串FIRST集合的计算，对于文法的FIRST表的构建，部分主要程序设计如下：**

void Grammar::computeFirstTable(){

    while(run\_flag){

        run\_flag = false;                     // 设置循环记号

        for(int i = 0; i < N.size(); i++){     // 对于每一个非终结符

            set<string>\* first = &FIRST[i];   // 通过指针记录

            int pre\_size = first->size();               // 记录原大小

            /\* p : FROM -> a|bX|cY \*/

            for(auto& P\_right : P[i].to)   // 对于每个非终结符的对应产生式

            {   /\* P\_right : bX \*/

                for(auto& ele : P\_right.elements) // 对于每个产生式右部的元素

                {   /\* ele : X \*/

                    if(ele.attr()!= Type::NOT\_TERMINAL){//非终结符

                        first->insert(ele.content());// 将其加入FIRST

                        break; //停止向下遍历

                    }

                    epsilon\_flag = false;

                    // 是非终结符，获取其FIRST集合所有元素

                    for(auto eles : FIRST[ele])

                        if(eles != EPSILON\_STR) //该非终结符不推出epsilon

                            first->insert(eles);

                        else epsilon\_flag = true;

                    if(!epsilon\_flag) // 如果不包含epsilon, 停止向下遍历

                        break;

                }

                if(epsilon\_flag) // 所有的产生式右部都能推出epsilon

                    first->insert(EPSILON\_STR); // 将epsilon加入该非终结符的FIRST集

            }

            if(first->size() > pre\_size) // FIRST集大小检测

                run\_flag = true;  // 如果某个FIRST集合的大小有变化，则继续循环

        }

    }

}

2.1.3、FOLLOW集计算

对于文法每个非终结符的FOLLOW集合，计算规则如下：

* 对文法开始符号，置于中。
* 如果有产生式，则把中所有非元素加入到中。
* 如果有产生式，或有，但，则把中所有元素加入到中。

**将该规则表述于程序设计，流程如下：**

void Grammar::computeFollowTable()

{

    FOLLOW[start].insert(ENDING\_STR); //将$插入文法开始符号的FOLLOW集合

    while(run\_flag)

    {

        run\_flag = false;

        for(int i = 0; i < FOLLOW.size(); i++){// 对于每个非终结符, 记为A

            set<string>\* follow = &FOLLOW[i]; // 指针记录

            for(auto& P\_right : P[i].to){    // 对A的产生式的所有右部

                vector<Symbol> ele = P\_right.elements;

                for(int j = 0; j < ele.size(); j++){//对每个产生式所有符号

                    if(ele[j].attr() != Type::NOT\_TERMINAL)//终结符,跳过

                        continue;

                    /\* 遇到非终结符, 计算该终结符对应的FOLLOW, 记为X \*/

                    int pos = \_findN(ele[j]);// 定位该终结符X

                    int pre\_size = FOLLOW[pos].size();// 记录初始大小

                    bool end\_flag = true; // 记录是否X后部符号串能推出空串

                    for(int k=j+1;k<ele.size();k++){//对X后的每个符号

                        if(ele[k].attr() != Type::NOT\_TERMINAL)

                        {// 遇到终结符, 将其加入X的FOLLOW集

                        FOLLOW[pos].insert(ele[k]);

                            end\_flag = false;// X后部的符号串不推出空串

                            break;

                        }

                        /\* 遇到非终结符, 记为Y, 找到其FIRST集 \*/

                        for(auto& it\_str : findFirst(ele[k])){

                            if(it\_str == EPSILON\_STR) // Y能推出epsilon

                                epsilon\_flag = true;

                            else // 将非epsilon元素加入X的FOLLOW集

                                FOLLOW[pos].insert(it\_str);

                        }

                        if(!epsilon\_flag){ // 如果Y不能推出epsilon

                            end\_flag = false; // X后部的符号串不推出空串

                            break;             // 停止向下遍历

                        }

                    }

                    if(end\_flag)   // X后部的符号串能推出epsilon

                        FOLLOW[pos].insert(follow); //将A的FOLLOW集合加入X的

                    if(FOLLOW[pos].size() > pre\_size)

                        run\_flag = true; // 如果X的FOLLOW集大小有更新，则进行新一轮循环

                }

            }

        }

    }

}

2.2、消除左递归

对于LL1分析方法，需要进行消除直接左递归操作。通过检所有产生式的右部，将右部第一个符号和产生式左部的符号进行比较，如果存在直接左递归，则将该产生式进行修改，并返回新生成的产生式。循环检测，直到所有产生式都不存在直接左递归。

部分原代码如下：

/\* 消除直接左递归 \*/

Grammar::Production Grammar::Production::eliminateLeftRecursion(){

    Production new\_p(from + "'");                   // 定义新产生式

    for(auto it = to.begin();it != to.end();){      // 对所有产生式右部

        if((\*it)[0].content() == from){

            vector<Symbol> s = (\*it).elements; // 复制该条产生式右部内容

            it = to.erase(it);      // 从原产生式重删除该条产生式右部

            /\* add new\_satate -> xx \*/

            s.erase(s.begin());     // 删除左部直接左递归符号

            s.emplace\_back(Symbol(new\_p.from)); // 在最后插入新符号

            new\_p.insert(s);        // 将其和epsilon插入新的产生式结构

            new\_p.insert({Symbol(EPSILON\_STR,Symbol::Type::EPSILON)});

        }else{

            (\*it).append(new\_p.from);   // 修改不含直接左递归的产生式右部

            it++;

        }

    }

    return new\_p;

}

void Grammar::eliminateLeftRecursion(){

    while(run\_flag){// 循环检测直接左递归, 直到所有产生式都不存在直接左递归

        run\_flag = false;

        for(auto it = P.begin(); it != P.end(); it++){

            if((\*it).existLeftRecursion()){ // 如果存在直接左递归

               Production p = (\*it).eliminateLeftRecursion();//修改原产生式

               insert(p);      //插入消除该直接左递归后的新产生式

               run\_flag = true;    // 设置循环标志, 继续检测

               break;

            }

        }

    }

}

2.3、提取左公因子

对于LL1分析方法，需要进行提取左公因子操作。为了简化程序设计，这里没有进行最长公共子串的检测，只**针对各条产生式右部的最左部符号**进行检测，**循环检测并提取**左公因子，生成并插入新的产生式，**直到所有产生式右部都不存在左公因子**。

\*\*由于源代码较长，此处不展示。

2.4、DFA构建

2.4.1、项目集构建及闭包计算

因为对于产生式右部的数据结构设计相对完善，包括了点号位置、符号内容和当前移进-规约状态，因此只需要对每个项目集中的文法检测其状态，针对待约项目进行闭包计算。同时循环检测该项目集中的所有产式，直到所有产生式都不再变化。

部分源代码如下：

/\* compute closure \*/

void Grammar::GrammarLR1::Item::closure(){

    while(run\_flag){

        run\_flag = false;

        for(int i = 0; i < \_productions.size(); i++){   // 对所有产生式

            Production\* p = &\_productions[i];

            if((\*p)[0].state == State::REDUCTION\_EXPECTING){

            /\* 检测待约项目 \*/

                Symbol s = (\*p)[0].getDotSymbol();

                /\* 加入对应产生式 \*/

                for(auto& t : G[s].to){

                    Production temp(s.content()); // 定义新产生式

                    temp.insert(t); // 插入对应产生式右部

                  \_computeAhead(temp,\_productions[i]); //计算向前看符号集合

                    int res = insert(temp); // 插入新产生式

                    if(res == 1) run\_flag = true;

                }

            }

        }

    }

}

2.4.2、规范族构建及DFA生成

根据上述构建项目集的算法，通过检查项目集中的所有产生式，针对所有移进/待约项目计算活前缀，并生成新的闭包，将其插入项目集规范组并与当前项目集通进行连接。

部分源代码如下：

/\* 活前缀读取 \*/

void Grammar::GrammarLR1::Item::next(){

    /\* 对于所有移进/待约项目 \*/

    for(auto& p : \_productions){

        if(p[0].state != State::REDUCTION){

            Symbol s = p[0].getDotSymbol();

/\* 构建活前缀集合 \*/

            for(auto& ele : \_nextRead)

                if(ele == s){ found = true; break;}

            if(!found) \_nextRead.emplace\_back(s);//插入非重复的符号

        }

    }

    /\* 构建活前缀识别后的项目集 \*/

    Item new\_item(\_LR1);

    for(int i = 0; i < \_nextRead.size(); i++){ // 遍历所有活前缀

        // 匹配该活前缀

        for(auto& p : \_productions)

            if(p[0].getDotSymbol() == \_nextRead[i])

                new\_item.insert(p); // 在新项目集中插入该产生式

        for(auto& p : new\_item.\_productions)

            p[0].next(); // 产生式右部点号向后移动

        new\_item.closure(); // 计算新闭包

\_LR1->insert(new\_item); // 插入新闭包

        \_nextTo[i] = new\_item.serial; // 连接新闭包

    }

}

2.5、分析表构建

2.5.1、LL1分析表构建

LL1预测分析表的构建思想大致为：

* 如果有产生式位于栈顶，如果当前输入符号，则应当被选作的唯一合法表代表去执行分析任务，即用去展开，所以表项中应放入产生式。
* 如果，并且当前输入符号，这里，则就认为自动得到匹配，因此应当把产生式放入表项中。
* 对于非终结符号，终结符号，如果表项，为空，则填入，代表从栈顶弹出。

部分源代码如下：

bool Grammar::GrammarLL1::initTable(){

    for(int p = 0; p < G->N.size(); p++){    // 对每个非终结符

        for(int to\_serial=0; to\_serial < G[p].to.size(); to\_serial++){

            /\* A -> alpha \*/

            /\* FIRST(alpha) \*/

           set<string> first = G.findFirst(G[p].to[to\_serial].elements);

            for(auto fi : first){

                if(fi == EPSILON\_STR){  // FIRST集合中含有epsilon

/\* 对于该非终结符的每个FOLLOW符号 \*/

                    for(auto fo : G.FOLLOW[p])

                    {/\*将该非终结符所推出的产生式的序号填入分析表 \*/

                        int t = G.\_findT(fo);

                        if(table[p][t] == GrammarLL1::ERROR)

                            table[p][t] = to\_serial;

                        else return false; // 分析表冲突检测

                    }

                }else{ // FIRST集合中不含有epsilon

                    int t = G.\_findT(fi);

                    if(table[p][t] == GrammarLL1::ERROR)

                        table[p][t] = to\_serial;

                    else return false; // 分析表冲突检测

                }

            }

        }

        /\* 填入同步化信息 \*/

        for(auto f : G->FOLLOW[p]){

            int\* num = &table[p][G.\_findT(f)];

            if(\*num == GrammarLL1::ERROR)

                \*num = GrammarLL1::SYNCH;

        }

    }

    return true;

}

**最终得到的LL1分析表格式输出如下：**

图形用户界面, 文本

描述已自动生成

**如果该文法不是LL1文法，程序会直接终止。**

文本

描述已自动生成

2.5.2、LR1分析表构建

LR1分析表的构造流程大致为：

* 拓广文法、构造LR1项目集规范族
* 对于状态（代表项目集），分析动作如下
  + 若，且，则置
  + 若，且，则置，是产生式的编号
  + 若，则置
* 对于非终结符，有，则置

部分源代码如下：

bool Grammar::GrammarLR1::initTable(){

    for(int i = 0; i < \_colletion.size(); i++){      // 对每一个状态

        Item\* item = \_colletion[i];

        /\* 遍历每一个识别的活前缀 \*/

        for(int j = 0; j < item->\_nextRead.size(); j++){

            Symbol\* s = &item->\_nextRead[j]; // 活前缀

            int to\_num = item->\_nextTo[j]; // 下一状态

            if(s->attr() == Symbol::Type::NOT\_TERMINAL){ // 非终结符

                int pos = (\*\_G)(\*s);

                goto\_table[i][pos] = to\_num;    // 填写goto

            }else{// 终结符

               int pos = G.findT(s); // 填写Sx到action

                action\_table[i][pos].first = ActionType::SHIFT;

                action\_table[i][pos].second = to\_num;

            }

        }

        /\* 遍历每一个产生式 \*/

        for(auto& p : item->\_productions){

            if(p[0].state == Production::To::State::REDUCTION)

            {// 找到所有归约项目

                int p\_num = \_findProduction(p);

                for(auto& str : p[0].ahead){ // 对每一个向前看符号

                    int pos = G.\_findT(str);

                    if(action\_table[i][pos].first != ActionType::ERROR)

                        return false; // 冲突

                    if(str == ENDING\_STR && p.from == G.start)

                        action\_table[i][pos].first = ActionType::ACCEPT;

                    else{// 填写Rx到action

                       action\_table[i][pos].first = ActionType::REDUCTION;

                        action\_table[i][pos].second = p\_num;

                    }

                }

            }

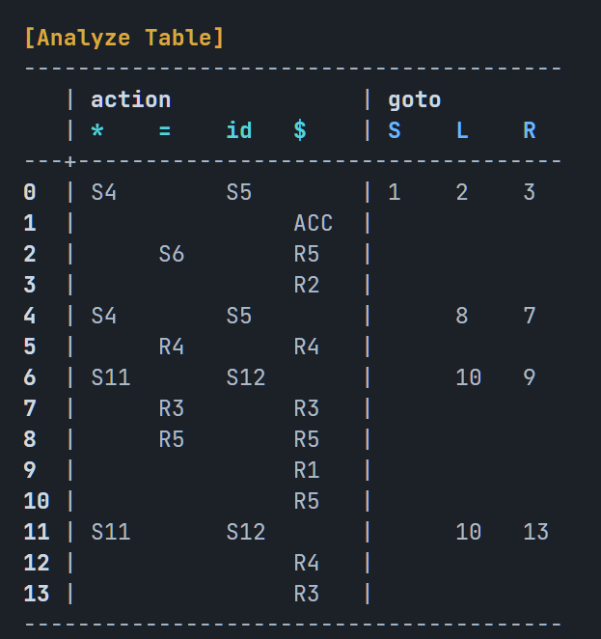
        }

    }

    return true;

}

**最终得到的LR1分析表格式输出如下：**

****

2.6、递归调用预测分析模式

2.6.1、文法与状态机

对于算数表达式的文法，经过消除直接左递归并带入化简后，可以构造如下文法识别DFA：

**图示

描述已自动生成**

2.6.2、识别程序

**根据状态机，可以构建识别程序：**

/\* 处理非终结符E \*/

bool Arith::\_stateE(){

    if(!\_stateT())

        return false;

    /\* 判断 + - \*/

    if(\_cur\_token == \_\_ADD || \_cur\_token == \_\_SUB){

        \_outputPair(State::E,\_cur\_token == \_\_ADD ? 0 : 1);

        \_nextToken();    // 移动指针

        return \_stateE();

    }

    \_outputPair(State::E,2);    // 输出匹配项

    return true;

}

/\* 处理非终结符T \*/

bool Arith::\_stateT(){

    if(!\_stateF()){

        //\_error();

        return false;

    }

    /\* 判断 \* / \*/

    if(\_cur\_token == \_\_MUL || \_cur\_token == \_\_DIV){

        \_outputPair(State::T,\_cur\_token == \_\_MUL ? 0 : 1);

        \_nextToken();   // 移动指针

        return \_stateT();

    }

    \_outputPair(State::T,2);

    return true;

}

/\* 处理非终结符F \*/

bool Arith::\_stateF(){

    /\* 判断数字 \*/

    if(\_cur\_token.attr() == Symbol::Type::NUMBER){

        \_outputPair(State::F,1);

        \_nextToken();

        return true;

    }

    /\* 判断左右括号 \*/

    if(\_cur\_token == \_\_LBK){

        \_outputPair(State::F,0);

        \_nextToken();

        if(!\_stateE())

            return false;

        if(\_cur\_token == \_\_RBK){

            \_outputPair(State::F,0);

            \_nextToken();

            return true;

        }

    }

    \_error();

    return false;

}

2.7、词法分析器调用

2.7.1、控制台输入

语法分析程序引入了词法分析实验中的Scanner类，由于Scanner类默认通过文件读取，因此这里通过将控制台中的标准输入内容写入到特定tmp文件，再调用Lexer模块的Scanner从该tmp文件中读取，以达到从控制台输入的效果。

**部分源代码：**

void Scanner::input(const std::string in\_str){

    \_ofs.open(\_\_tmp\_filename,std::ios::out|std::ios::trunc);    // 写入tmp文件

    if(!\_ofs.is\_open())

        EXIT\_ERROR("system error");

    \_ofs << in\_str;

    \_ofs.close();

    close();

    open(\_\_tmp\_filename.c\_str());   // 调用open读取文件流

}

2.7.2、记号流输出

在Lexer实验中，源程序将记号流输出到控制台标准输出及文件中，为了将记号流导入语法分析，针对词法分析模块进行了修改，使其能够**将完整记号流以vector形式保存在内存中**，并可传入语法分析程序进行分析。

**部分源代码如下：**

std::vector<Symbol> Scanner::tokenStream(){

    std::vector<Symbol> symbols;    // 记号流

    /\* 生成记号流 \*/

    while((++s).attr() != Symbol::Type::END){

        if(s.attr() == Symbol::Type::ERROR)

            error\_flag = true;

        else if(s.attr() != Symbol::Type::COMMENT)  // 跳过注释

            symbols.emplace\_back(s);

    }

    if(error\_flag){

        symbols.clear();

        LOG\_INFO("invalid input");  // 错误检测

    }

    symbols.emplace\_back(Symbol("end",Symbol::Type::END)); //插入结束记号

    return symbols;

}

运行及分析

一、运行方式

1.1、输入源程序

输入文法为识别算术表达式的文法，具体如下：

#S E

#N E T F

#T num + - \* / ( )

#P

@ E -> E+T # E-T # T

@ T -> T\*F # T/F # F

@ F -> (E) # num

其中，通过#分割文法的各个部分以及不同的产生式右部，通过@标明产生式识别的开始。S代表开始符号，N代表非终结符号，T代表终结符号，P代表产生式。

1.2、运行参数

语法分析程序定义了文法文件路径及分析模式：

.\out.exe [-I <file\_name = scripts/grammar.in>]

[-m <analyze\_mode = LL1/LR1/Arith>]

通过-I指定输入文法路径，通过-m指定分析模式（LL1 / LR1 / Arith）。分别代表**LL1分析**、**LR1分析**以及**指定算数表达式分析**。

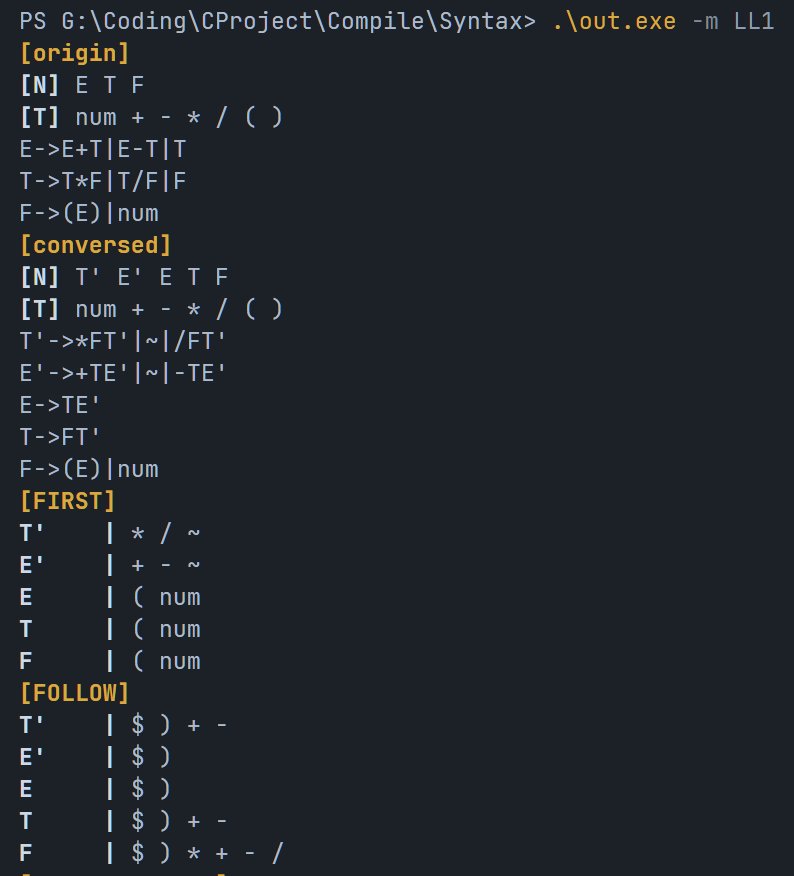
二、运行结果与分析

2.1、LL1运行结果

将输入的算数表达式文法传入文法分析程序，运行指令为

.\out.exe -m LL1

2.1.1、文法信息



2.1.2、LL1分析表

图形用户界面, 文本

描述已自动生成

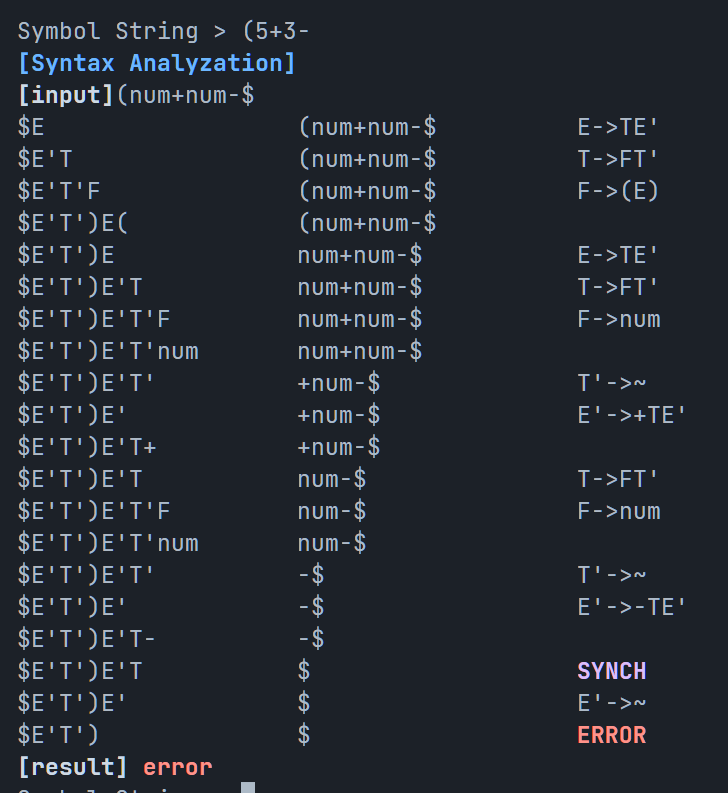
2.1.2、LL1分析结果

根据LL1预测分析表，进行LL1预测分析可以得到如下结果，这里进行了正确性测试和错误处理的测试。输入支持通过分号表示多条分析任务。

**图形用户界面, 文本

描述已自动生成**

2.1.3、错误处理

****

2.2、LR1运行结果

将输入的算数表达式文法传入文法分析程序，运行指令为

.\out.exe -m LR1

2.2.1、文法信息



可以看到程序输出了原文法、拓展后的文法、拓展后文法的FIRST/FOLLOW集。

2.2.2、项目集规范族及DFA

图片包含 文本

描述已自动生成文本

描述已自动生成图片包含 文本

描述已自动生成文本

描述已自动生成

文本

描述已自动生成图片包含 文本

描述已自动生成

文本

描述已自动生成

2.2.3、LR1分析表

日历

描述已自动生成

2.2.4、LR1分析结果

根据LR1预测分析表，参考LL1分析的输入串进行LR1预测分析，可以得到如下结果：

**文本

描述已自动生成**

**文本

描述已自动生成**

**文本

描述已自动生成**

2.2.5、LR1错误处理

文本

描述已自动生成

2.3、递归调用预测分析运行结果

运行指定的算数表达式语法分析，运行指令为

.\out.exe -m Arith

可以得到如下结果：

2.3.1、分析结果

文本

描述已自动生成

2.3.2、错误处理

屏幕上有字

描述已自动生成

LEX + YACC实现

一、LEX与YACC连接

1.1、LEX

为识别算数表达式，根据词法分析实验结果，设计了基于正则表达式的LEXER，识别数字类型、算数运算符以及左右括号。

**和独立的词法分析程序不同，由于需要LEX程序传递记号到YACC语法分析程序，因此在识别到特定字符后，需要将该记号返回，即需要在识别后加上return关键字。**

LEX程序定义如下：

%{/\*2022-11-07\*/

#include <stdlib.h>

#include <stdio.h>

#include "y.tab.h"

void yyerror(char \*);

%}

NUM         [0-9]

digit       ({NUM}\*(".")?{NUM}\*([Ee][+-]|[Ee])?{NUM}+)|({NUM}+)

ADD         "+"

SUB         "-"

MUL         "\*"

DIV         "/"

LBK         "("

RBK         ")"

LINE        [\r\n]

%%

{digit}     {return digit;}

{LBK}       {return LBK;}

{RBK}       {return RBK;}

{ADD}       {return ADD;}

{SUB}       {return SUB;}

{MUL}       {return MUL;}

{DIV}       {return DIV;}

{LINE}      {return LINE;}

.           {}

%%

1.2、YACC

**YACC（Yet Another Compiler Compiler）**，即编译器的编译器，使用BNF定义文法，能处理上下文无关文法。

根据YACC的语法规范，定义YACC程序如下：

其中，需要通过%token定义从LEXER词法分析程序接受的终结符。另外，在各条产生式被识别时，还加入了输出选项，便于观测语法分析过程。

%{

#include <stdio.h>

#include <ctype.h>

int yylex(void);

void yyerror(char \*);

%}

%token digit ADD SUB MUL DIV LBK RBK LINE

%%

S : E LINE      {}

  ;

E : E ADD T     {printf("E->E+T\n");}

  | E SUB T     {printf("E->E-T\n");}

  | T           {printf("E->T\n");}

  ;

T : T MUL F     {printf("T->T\*F\n");}

  | T DIV F     {printf("T->T/F\n");}

  | F           {printf("T->F\n");}

  ;

F : LBK E RBK   {printf("F->(E)\n");}

  | digit       {printf("F->digit\n");}

  ;

;

%%

int main(){

    return yyparse();

}

int yywrap(){

    return 1;

}

void yyerror(char\* s){

    printf("[result] %s\n",s);

}

1.3、连接

将YACC程序和LEX程序连接，需要将**.y**文件通过**YACC**指令，将**.l**文件通过**LEX**指令分别生成对应的**.c**文件，而后通过**gcc**将二者**联合编译**生成可执行程序。

其中，由于使用的是GNU Bison和GNU Flex，因此指令分别为bison 和 flex。

文本

中度可信度描述已自动生成

二、YACC语法分析程序执行

直接运行上述编译后得到的语法分析程序的可执行文件，并**输入符号串**，即可以看到分析过程和分析结果。

对于正确识别的符号串，YACC会返回**end**，而若识别错误，会调用到**yyerror()函数**进行错误输出或处理。

**正确符号串分析：**

文本

描述已自动生成

**错误符号串分析：**

文本

描述已自动生成