目录

[**1** **程序设计要求** 2](#_Toc102144780)

[**1.1** **词法分析要求** 2](#_Toc102144781)

[**1.2** **语法分析要求** 2](#_Toc102144782)

[**2** **程序支持功能** 2](#_Toc102144783)

[**2.1** **词法分析** 3](#_Toc102144784)

[**2.2** **语法分析** 3](#_Toc102144785)

[**3** **类设计** 3](#_Toc102144786)

[**3.1** **Symbol类** 3](#_Toc102144787)

[**3.2** **token类** 4](#_Toc102144788)

[**3.3** **NFAState类** 5](#_Toc102144789)

[**3.4** **DFAState类** 6](#_Toc102144790)

[**3.5** **Production类** 7](#_Toc102144791)

[**3.6** **FirstSet类** 8](#_Toc102144792)

[**3.7** **Grammar类** 8](#_Toc102144793)

[**3.8** **ProductionItem类** 9](#_Toc102144794)

[**3.9** **ProductionItemSet类** 11](#_Toc102144795)

[**3.10** **ActionItem类** 12](#_Toc102144796)

[**3.11** **GotoItem类** 12](#_Toc102144797)

[**4** **外部输入设计** 13](#_Toc102144798)

[**4.1** **parse.txt** 13](#_Toc102144799)

[**4.2** **production.txt** 13](#_Toc102144800)

[**4.3** **program.txt** 13](#_Toc102144801)

[**5** **词法分析** 13](#_Toc102144802)

[**5.1** **概述** 13](#_Toc102144803)

[**5.2** **readParseTXT()** 14](#_Toc102144804)

[**5.3** **regularGrammarToNFA ()** 14](#_Toc102144805)

[**5.4** **NFAToDFA()** 15](#_Toc102144806)

[**5.5** **readProgramTXT()** 17](#_Toc102144807)

[**5.6** **parseProgram()** 17](#_Toc102144808)

[**6** **语法分析** 19](#_Toc102144809)

[**6.1** **概述()** 19](#_Toc102144810)

[**6.2** **readProductionTXT()** 19](#_Toc102144811)

[**6.3** **Grammar.creat ()** 20](#_Toc102144812)

[**6.4** **generateProductioItemSets ()** 20](#_Toc102144813)

[**6.5** **createLR1Table()** 25](#_Toc102144814)

[**6.6** **mathTokenList()** 26](#_Toc102144815)

1. **程序设计要求**
   1. **词法分析要求**

**程序输入：**

1. 一组 3º型文法（正规文法）的产生式**。**
2. 含一组需要识别的字符串（程序代码）

**程序输出：**

token（令牌）表，包含该 token所在的行列、token类型及具体内容。该表由 6 种类型 token组成：**关键词**，**标识符**，**常量**，**限定符**，**界符**和**运算符**。

**过程简述：**

1. 根据用户输入的正规文法，生成 NFA。
2. 根据生成的NFA，对NFA进行确定化，获得DFA。
3. 根据 DFA 转换图，从头到尾从左至右识别用户输入的源代码，生成 token 列表（三元组：所在行号，类别，token 内容）。

**额外要求：**

1. 可以准确识别科学计数法形式的常量（如 0.314E+1）及复数常量（如 10+12i）。
2. 能检查整数常量的合法性，标识符的合法性（首字符不能为数字等）。
3. 尽量符合真实常用高级语言要求的规则。
   1. **语法分析要求**

**程序输入：**

1. 2º型文法（上下文无关文法）的产生式集合。
2. 词法分析程序输出的（生成的）token 令牌表。

**程序输出：**

1. TRUE（源代码字符串符合此2º型文法）或 FALSE （源代码字符串不符合此2º型文法）。
2. 如存在错误，则标示出错行号，并给出大致的出错原因。

**过程简述：**

1. 根据用户输入的 2º型文法，读取文法的终结符和非终结符，并对所有非终结符求First集。
2. 根据用户输入的2º型文法和求得的First集，生成项目集和项目转换图
3. 根据项目集转换图，生成 Action 及 Goto 表
4. 根据Action 和 Goto 表，判断 token 序列（用户输入的源程序转换）是否符合文法规则，并输出结果。
5. **程序支持功能**
   1. **词法分析**
6. 支持整数、小数、科学计数法和简单的复数
7. 支持字符串类型，以 “” 区分
8. 词法分析出错时，具有较好的错误提示信息
9. 对错误的数字输入具有一定的检错纠错能力，如：
   1. int a = 5.4e++4;

输出：

词法分析出错!

出错位置: 行:1, 列:14, 内容:+

出错原因:当前状态非终结状态!

期望匹配:{'0','1','2','3','4','5','6','7','8','9'}

* 1. double a = 5. ;（注：出现了小数点但小数点后没有任何数字）

输出：

出错位置: 行:1, 列:11, 内容:;

出错原因:当前状态非终结状态!

期望匹配:{'0','1','2','3','4','5','6','7','8','9'}

* 1. **语法分析**

1. 支持自定义类，并支持abstract关键字定义的抽象类，以class关键字进行类的声明。可以使用new关键字声明自定义变量。
2. 支持如下访问控制符：public, private, protected, final, static
3. 支持如下变量类型：int, long, float, double, char, short, string, auto, void(只能用在函数返回)和自定义变量类型
4. 支持自定义函数的声明与调用，使用func关键字进行函数声明。void类型的函数return语句可以省略，其他类型必须提供return语句。
5. 支持数组类型，声明时必须指定大小（可以为变量），如int[5] a;
6. 支持if, for, while, do while, continue, break等常见逻辑语句
7. 内置print语句，使用print(“输出内容”)进行调用
8. 支持与或非等逻辑运算和+=等赋值运算符和自增自减运算符。
9. 支持行注释及块注释
10. 语法分析出错时具有良好的提示信息，能指出出错位置即大致的出错原因。
11. **类设计**
    1. **Symbol类**

**作用：**

symbol类用于存储产生式中出现的文法符号。如产生式A->Ba，其中共有三个文法符号’A’, ‘B’, ‘a’。

**成员变量：**

1. private final String content; //用于存储文法符号的具体内容
2. private final boolean isEnd; //标记该符号是否是结束符号
3. private final boolean isVt; //标记该符号是否是非终结符，true表示终结符

**静态变量：**

1. public static final String *EPSILON* = "ε"; //存储ε符号，便于后续程序调用
2. public static final Symbol *END* = new Symbol("#",true); //文法的终结符

**关键函数：**

通过重写equals方法，将判断两个symbol对象相等的条件设置为**只判断其content的内容是否相等。**

public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Symbol symbol = (Symbol) o;  
 return isEnd == symbol.isEnd && Objects.*equals*(content, symbol.content);  
}

* 1. **token类**

**作用：**

存储词法分析的结果。词法分析将源程序根据文法规则分割成一个个文法符号，token负责将分割成的文法符号进行存储、分类、记录所在行列，为下一步语法分析做准备。

**成员变量：**

1. private int row; //行
2. private int col; //列
3. private String content; //该token的具体内容
4. private TokenType type; //该token的具体类型。TokenType为枚举类型，共有六种类型：KEYWORDS, IDENTIFIER, CONST, QUALIFIER, OPERATOR, SYMBOL。分别代表关键字、标识符、常量、限定符、操作符和界符。

**静态变量：**

静态变量以HashSet的形式存储关键字、标识符、限定符、界符的具体内容，便于后续对token类型的判断。

1. private static final Set<String> *KEYWORDS* = new HashSet<>(Arrays.*asList*(  
   "if", "while", "for", "continue", "break", "return", "do", "goto", "class", "func", "abstract", "new", "complex", "int", "char", "short", "string", "boolean", "long", "float", "double", "auto", "void","true", "false"));
2. private static final Set<String> *QUALIFIER* = new HashSet<>(Arrays.*asList*(  
   "public", "private", "protected", "final", "static"));
3. private static final Set<String> *OPERATOR* = new HashSet<>(Arrays.*asList*(  
   "+", "-", "\*", "/", "!", "~", "^", "|", "||", "&", "&&", "<", ">", "=", "<=", ">=", "==", "++", "--", "%", "+=", "-=", "\*=", "/="));
4. private static final Set<String> *SYMBOL* = new HashSet<>(Arrays.*asList*(

",", ";", ".", "[", "]", "{", "}", "(", ")", "\"", "'"));

**关键函数：**

在创建一个token对象时，构造函数中不需要指定token的tokenType的具体类型，改由在构造函数中调用genType()函数根据token内容自动生成类型。

genType()函数中，首先根据规则对判断token是否为const（常量），随后查HashSet判断是否为关键字、界符、操作符、限定符中的其中一种。如果都不是，则将token认定为标识符。

public TokenType genType(){  
 *//判断是否为常量* if((content.charAt(0) >= '0' && content.charAt(0) <= '9')  
 || (content.charAt(0) == '-' && content.length() >= 2 && content.charAt(1) != '-')  
 || (content.charAt(0) == '.' && content.length() > 1)  
 || (content.charAt(0) == 'e' && content.length() >= 2 && (content.charAt(1) == '+' || content.charAt(1) == '-')) || (content.charAt(0) == '"'))  
 return TokenType.*CONST*;  
 if(*KEYWORDS*.contains(content))  
 return TokenType.*KEYWORDS*;  
 if(*QUALIFIER*.contains(content))  
 return TokenType.*QUALIFIER*;  
 if(*SYMBOL*.contains(content))  
 return TokenType.*SYMBOL*;  
 if(*OPERATOR*.contains(content))  
 return TokenType.*OPERATOR*;  
 return TokenType.*IDENTIFIER*;  
}

* 1. **NFAState类**

**作用：**

在正规文法转NFA过程中，存储转换得到的NFA结点，并记录NFA结点之间的转换关系。

**成员变量：**

1. private Symbol id; //当前NFA结点对应的文法符号，相当于名字
2. private boolean isEnd = false; //当前NFA结点是否是终态
3. private boolean isStart = false; //当前NFA结点是否是初态
4. private Map<String, Set<NFAState>> edges; //存储NFA结点的转换关系，即当前NFA结点通过String类型的路径能转换到哪些NFA结点

**关键函数：**

在正规文法转NFA的过程中，每读取到一条正规文法，就调用addEdges()函数根据该正规文法为NFA结点添加一条边。

public void addEdges(Symbol symbol,NFAState nextNFAState){  
 Set<NFAState> stateSet = edges.get(symbol.getContent()); if(stateSet == null){  
 *//为空则创建* stateSet = new HashSet<>();  
 edges.put(symbol.getContent(), stateSet); *//添加到边的集合中* }  
 stateSet.add(nextNFAState); *//添加边*}

重写Comparable接口中的compareTo()函数，使NFA结点可以通过id的内容比较大小、排序。

public int compareTo(NFAState o) {  
 return id.getContent().compareTo(o.id.getContent());  
}

* 1. **DFAState类**

**作用：**

在NFA转DFA过程中，存储转换得到的DFA，并且存储该DFA包含的NFA结点。由于NFA转DFA过程的不确定性（DFA中添加NFA结点顺序不固定），为了避免重复生成相同的DFA，DFA中还包含了唯一的key，用于判断DFA是否相等。

**成员变量：**

1. private final Set<NFAState> NFAStates; //该DFA结点包含的NFA
2. private final Boolean isStart; //该DFA结点是否是开始结点
3. private final Boolean isEnd; //该DFA结点是否是结束结点
4. private final String key; //当前DFA结点的key

**静态变量：**

private static final Map<String,DFAState> *DFAMap* = new HashMap<>();

//以key(String类型)为键，DFAState为值的HashMap，将key与DFA结点一一对应。

**关键函数：**

在创建一个DFA结点时调用。传入参数为需要创建的DFA结点包含的NFA结点的集合，传出参数为对应的DFA结点。具体步骤如下：

1. 由于NFA集合是通过move，闭包等操作得到的，即使NFA集合的内容相同，但NFA结点的顺序不同，为判断是否是同一个DFA造成麻烦。因此该函数中先对NFA集合进行排序。
2. 根据排序后的NFA集合生成key。
3. 随后将生成的key查找DFAMap表，判断DFAMap中是否已存在该key对应的DFA结点（说明DFA结点已经创建过）。
4. a. 如果存在，则直接返回DFA结点，避免重复创建
5. b. 如果不存在，则遍历NFA集合，如果其中有一个NFA为开始状态或结束状态、则对应的DFA结点也为开始状态、结束状态。随后调用构造函数创建DFA结点，并 记录到DFAMap中。

public static DFAState create(Set<NFAState> NFAStates){  
 *//生成唯一key* StringBuffer stringBuffer = new StringBuffer();  
 NFAStates.stream().sorted().forEach(  
 nfaState -> stringBuffer.append(nfaState.getId().getContent()).append(",")  
 );  
 if(stringBuffer.length() > 0)  
 stringBuffer.deleteCharAt(stringBuffer.length() - 1);  
 *//如果hashmap中存在key，则表示相同的DFA已经创建过，直接返回即可* if(*DFAMap*.containsKey(stringBuffer.toString())){  
 return *DFAMap*.get(stringBuffer.toString());  
 }  
 *//如果NFAStates集合中有一个结点为开始/结束结点，则对应的DFA结点也是* boolean isStart = false;  
 boolean isEnd = false;  
 for(NFAState nfaState : NFAStates){  
 if(nfaState.isStart())  
 isStart = true;  
 if(nfaState.isEnd())  
 isEnd = true;  
 }  
 return new DFAState(NFAStates,isStart,isEnd,stringBuffer.toString());  
}

重写equals方法，将DFA结点是否相等的判断设置为**只判断key的值是否相等**。

public boolean equals(Object obj){  
 if(this == obj)  
 return true;  
 if(obj == null || obj.getClass() != getClass())  
 return false;  
 return key.equals(((DFAState) obj).key);  
}

* 1. **Production类**

**作用：**

存储产生式，并为每个产生式标号。

**成员变量：**

1. private final Symbol left; //产生式左部的文法符号
2. private final List<Symbol> right; //产生式右部的文法符号集合
3. private final boolean isEpsilon; //该产生式是否为空
4. private final int index; //该产生式的序号

**静态变量：**

1. public static int *count* = 0; //记录当前已经有多少个产生式，用于分配序号

**关键函数：**

在创建产生式时调用，在调用构造函数创建产生式之前先进行判断，避免左、右部为空的产生式被创建，造成后续程序分析出错。

注：产生式A->ε的右部为Symbol“ε“，而不是什么都没有。

public static Production create(Symbol left,List<Symbol> right){  
 if(left == null || right.isEmpty()){  
 throw new RuntimeException("产生式左/右部不能为空");  
 }  
 return new Production(left,right);  
}

* 1. **FirstSet类**

**作用：**

存储文法符号集合对应的First集合。

**成员变量：**

1. private final List<Symbol> start; //文法符号集合
2. private final Set<Symbol> set = new HashSet<>(); //该文法符号结合对应的First集合，Set用于消除重复
3. private boolean hasEpsilon; //First集合中是否包含空
   1. **Grammar类**

**作用：**

在语法分析时用到，在读取完所有三型文法后调用。能够对三型文法进行处理，得到每个非终结符的First集合，用于之后的语法分析。

**成员变量：**

1. private final Symbol start; //三型文法的开始符号
2. private final Set<Symbol> VtSet; //三型文法中的所有终结符集合
3. private final Set<Symbol> VnSet; //三型文法中的所有非终结符结合
4. private Map<Symbol,FirstSet> firstSetMap;//文法中的所有非终结符对应的First集合
5. private final LinkedHashMap<Symbol, List<Production>> productionMap; //以产生式左部的非终结符为键，其对应的产生式列表

**关键函数：**

创建Grammar时调用，输入参数为开始符号和产生式列表。在对产生式列表进行一定的判断后，生成产生式列表对应的productionMap，并调用构造函数创建Grammar对象。

public static Grammar creat(Symbol start,List<Production> productionList){  
 if(productionList == null || productionList.isEmpty()){  
 throw new RuntimeException("产生式为空");  
 }  
 if(!start.equals(productionList.get(0).getLeft())){  
 throw new RuntimeException("第一个产生式左部应该与开始符号相同！");  
 }  
 LinkedHashMap<Symbol,List<Production>> symbolListLinkedHashMap = new LinkedHashMap<>();  
 for(Production p : productionList){  
 Symbol left = p.getLeft();  
 *//获取左符号对应的所有产生式* List<Production> leftProductions = symbolListLinkedHashMap.computeIfAbsent(left, k -> new ArrayList<>()); *//添加产生式* leftProductions.add(p);  
 }  
 return new Grammar(start, symbolListLinkedHashMap);  
}

Grammar的私有构造函数，对传入的productionMap进行处理，得到产生式列表中的所有终结符和非终结符。

private Grammar(Symbol start,LinkedHashMap<Symbol, List<Production>> productionMap){  
 this.start = start;  
 this.productionMap = productionMap;  
 *//非终结符集合一定是产生式的键值* this.VnSet = productionMap.keySet();  
 this.VtSet = new HashSet<>();  
 *//终结符集合需要遍历获得* for(Map.Entry<Symbol,List<Production>> entry : productionMap.entrySet()){  
 for(Production production : entry.getValue()){  
 for(Symbol symbol : production.getRight()){  
 if(symbol.isVt())  
 this.VtSet.add(symbol);  
 }  
 }  
 }  
}

* 1. **ProductionItem类**

**作用：**

在语法分析中，存储项目集中的一个项目，如项目：A->a•b,a。

该类还实现了Comparable接口，根据content内容进行比较，为后续的排序提供便利。并重写了equals方法，通过比较content内容判断两个对象是否相等。

成员变量：

1. private final Production production; //该项目对应的产生式
2. private final int delimiterPos; //项目分隔符的位置
3. private final Symbol expect; //项目的展望符
4. private final String content; //该项目的字符串形式，用于输出和比较两个项目是否相等

**静态变量：**

public static final String *DELIMITER* = "•"; //存储分隔符

**关键函数：**

创建一个项目集的项目。输入参数：产生式，展望符，分割符的位置

函数根据分割符的位置和产生式，在项目的字符串形式中插入分割符。并将展望符也插入到项目的字符串形式中。最后调用构造函数创建一个项目集的项目并返回。

public static ProductionItem create(Production production, Symbol expect, int delimiterPos){  
 List<Symbol> rightList = production.getRight();  
 StringBuilder stringBuilder = new StringBuilder();  
 stringBuilder.append(production.getLeft().getContent()).append("->");  
 *//将分隔符插入到项目的字符串中* for(int i = 0; i < rightList.size(); ++i){  
 Symbol symbol = rightList.get(i);  
 if(i == delimiterPos){  
 *//插入分隔符* stringBuilder.append(*DELIMITER*);  
 }  
 stringBuilder.append(symbol.getContent());  
 }  
 *//分隔符在末尾的情况* if(delimiterPos == rightList.size()){  
 stringBuilder.append(*DELIMITER*);  
 }  
 *//插入展望符* stringBuilder.append(",").append(expect.getContent());  
 *//调用构造函数* return new ProductionItem(production, delimiterPos, expect, stringBuilder.toString());  
}

根据已有的项目创建该项目的下一个项目，即将分隔符右移一位。

函数中对分割符的位置进行了判断：如果分割符不在项目末尾，则将分隔符右移，调用create函数创建项目集的项目；如果分隔符在项目的末尾，则报错。

public static ProductionItem create(ProductionItem productionItem){  
 if(productionItem.delimiterPos >= productionItem.production.getRight().size()){  
 *//如果当前项目的分割符已经在项目末尾则创建失败* throw new RuntimeException("项目的分隔符已在项目末尾，无法创建");  
 }  
 return *create*(productionItem.production,productionItem.expect,productionItem.delimiterPos + 1);  
}

在求项目集的某个项目的闭包时调用。该函数将返回隔符往后的第二个元素及展望符，用于闭包中的获取first集。如产生式A->α•Bβ,a，返回βa。

public List<Symbol> getFirstSymbol(){  
 List<Symbol> result = new ArrayList<>();  
 List<Symbol> rightSymbols = production.getRight();  
 for(int i = delimiterPos + 1; i < rightSymbols.size(); ++i){  
 result.add(rightSymbols.get(i));  
 }  
 result.add(expect);  
 return result;  
}

* 1. **ProductionItemSet类**

**作用：**

LR1语法分析中的项目集。类重写了equals方法，通过比较content内容判断两个对象是否相等。

**成员变量：**

1. private final Set<ProductionItem> productionItemSet; //该项目集包含的具体项目
2. private final String content; //该项目集的内容的字符串形式
3. private final int index; //当前项目集的序号

**静态变量：**

1. public static final HashMap<String, ProductionItemSet> *itemSets* = new HashMap<>();

//所有项目集的HashMap，以项目集的内容content为键，映射对应的项目集

1. private static int *total* = 0; //用于给项目集编号

**关键函数：**

创建一个项目集时调用，参数为该项目集包含的所有项目Set。该函数的具体流程如下：

1. 对项目集包含的所有项目进行排序，确保顺序不同但内容相同的项目Set能获得一致的结果。
2. 根据排序后的结果，生成content。
3. 根据content查找所有项目集的HashMap，如果存在则直接返回对应的ProductionItemSet对象；如果不存在则调用构造函数创建一个新的ProductionItemSet对象并返回，同时记录到HashMap中。

public static ProductionItemSet create(Set<ProductionItem> productionItemSet){  
 StringBuilder stringBuilder = new StringBuilder();  
 *//排序，保证顺序不同但内容相同的set生成的stringBuilder是相同的* productionItemSet.stream().sorted().forEach(  
 productionItem -> stringBuilder.append(productionItem.getContent()).append(";")  
 );  
 String content = stringBuilder.toString();  
 *//通过content查找hashmap判断是否存在过* if(*itemSets*.containsKey(content)){  
 return *itemSets*.get(content); *//直接返回已有集合* }  
 else{  
 ProductionItemSet itemSet = new ProductionItemSet(productionItemSet,content,*total*);  
 *itemSets*.put(content,itemSet);  
 return itemSet;  
 }  
}

* 1. **ActionItem类**

**作用：**

LR1分析时，存储一个动作（移进、归约、ACC）。

**成员变量：**

1. private final String actionType; //动作的类型
2. private final ProductionItemSet productionItemSet;//移进时对应的下一个项目集
3. private final Production production; //归约时需要用到的产生式

**静态变量：**

1. public static final String *ACTION\_S* = "S"; //移进
2. public static final String *ACTION\_R* = "R"; //归约
3. public static final String *ACTION\_ACC* = "ACC"; //ACC
   1. **GotoItem类**

**作用：**

LR1分析时，存储Goto表项。

**成员变量：**

1. private final ProductionItemSet nextProductionItemSet; //下一个项目
2. private final int number; //下一个项目集对应的序号
3. **外部输入设计**
   1. **parse.txt**

parse.txt为词法分析时所需要用到的**正规文法**，且**必须为右线性文法**。产生式需要满足形如A->aB或A->a的形式，每行对应一个产生式。

此外，**终结符必须为一个字符**，非终结符没有字符长度限制。

* 1. **production.txt**

production.txt为LR1语法分析时所需要用到的三型文法，每行对应一个产生式。产生式右部的每个文法符号通过” ”（空格）进行分割，**所有非终结符必须以大写字母开头**。

如果输入的三型文法在LR1分析时存在冲突，不满足LR1分析的语法要求，可能导致最后结果出错。

* 1. **program.txt**

待分析的输入程序。

1. **词法分析**
   1. **概述**

调用ParserUtils类中的parse()函数进行词法分析。parse()函数的两个输入参数String parsePath, String programPath分别为正规文法的路径和待分析用户程序的路径。

parse()函数的总体流程如下图：



* 1. **readParseTXT()**

**参数：**

1. String filePath: 正规文法的路径

**相关变量：**

1. public final static List<Production> *productionList* = new ArrayList<>();//产生式列表

从正规文法中读取到的产生式将存储在这个变量中

**作用：**

根据正规文法路径读取产生式到productionList中

**函数逻辑伪代码：**

1. 根据输入路径读取文件
2. while(文件没读取完)
   1. 读取一整行
   2. 读取产生式左部，直到’->’为止
   3. 读取终结符
   4. if(字符串没读取到结尾)
      1. 读取非终结符
   5. 根据读取内容生成产生式
   6. pruductionList.add(读取到的产生式)
   7. **regularGrammarToNFA ()**

**相关变量：**

1. public final static List<Production> *productionList* = new ArrayList<>();//产生式列表
2. public final static Map<Symbol,NFAState> *nfaStateMap* = new HashMap<>(); //当前非终结对应的NFA结点
3. public static NFAState *startNFA*; //NFA的开始结点

**作用：**

将读取到的正规文法转换为NFA，并得到NFA转换图。

**函数逻辑伪代码：**

1. 将第一条产生式的左部作为开始符号startSymbol
2. 根据开始符号创建对应的NFA结点currentNFA，并设置为初态
3. startNFA = currentNFA; nfaStateMap.put(startSymbol,currentNFA);
4. 创建终态endState
5. nfaStateMap.put(END SYMBOL, endState);
6. 遍历产生式集合
   1. 获取产生式左部的符号leftSymbol
   2. 根据leftSymbol查找nfaStateMap
   3. 如果未找到：
      1. 创建currentNFA结点，添加到nfaStateMap中
   4. 如果产生式为A->aB的形式
      1. 调用addEdges()，添加一条从A->B的路径，且路径条件为a
   5. 如果产生式为A->a的形式
      1. 调用addEdges()，添加一条从A->终态的路径，且路径条件为a
   6. **NFAToDFA()**

#### closure()

**参数：**

1. Set<NFAState> nfaStates //需要求闭包的NFA结点集合

**返回：**

Set<NFAState> //求闭包后的结果集

**作用：**

对输入的NFA结点集合求空闭包。

**函数逻辑伪代码：**

1. 将输入的nfaStates添加到结果集resutlSet
2. 创建栈stack，将nfaState添加到栈中
3. while(栈不空)
   1. 获取栈顶的NFA结点 currentState，栈顶元素出栈
   2. 查找该结点的对于空符号的转换图，得到epsilonStateSet
   3. 如果epsilonStateSet中出现了新的NFA结点：
      1. 将新的NFA结点添加到resultSet
      2. 将新的NFA结点添加到栈中
4. 返回resultSet

#### move()

**参数：**

1. DFAState currentDFAState //当前的DFA结点
2. Symbol path //move的路径

**返回：**

经过move操作后的NFASet集合

**作用：**

根据path和currenDFAState，查找对应的NFA转换图，得到currenDFAState中的NFA集合经过path路径转换后得到的NFA集合。

**函数逻辑伪代码：**

1. for(currenDFAState中的每个NFA结点)
   1. 查找该NFA结点的转换图，判断是否包含转换路径
   2. 若包含：
      1. 将转换后的NFA结点添加到结果集
2. 返回结果集

#### NFAToDFA()

**参数：**

1. NFAState startNFA //NFA的开始状态

**相关变量：**

1. public final static Set<DFAState> *DFAStateSet* = new HashSet<>(); //DFA集合
2. public final static Map<DFAState,Map<String,DFAState>> *DFAGraph* = new HashMap<>();

//DFA转换图，第一个键为当前的DFA结点，通过其得到Map<String,DFAState>.

随后再以转换路径String为键，得到下一个DFA结点

1. public static DFAState *startDFA*; //DFA的开始状态

**作用：**

利用**子集法**将NFA转换图转换为DFA，并得到DFA转换图。

**函数逻辑伪代码：**

1. 对startNFA调用closure()求空闭包，得到startNFASet
2. startDFA = DFAState.create(startNFASet);
3. startDFA入栈
4. while(栈不空)：
   1. 获取栈顶的DFA结点currentDFA
   2. for(每个不是空的终结符symbol)
      1. 求move(currentDFA,symbol)，得到movedNFASet
      2. 求空闭包，closureNFASet = closure(movedNFASet)
      3. 如果closureNFASet不为空：
         1. 创建DFA，DFAState dfaState = DFAState.create(closureNFASet);
         2. 如果DFAStateSet中不包含dfaState：
            1. 添加到DFAStateSet
            2. 添加到栈
         3. 添加DFA转换图DFAGraph
   3. **readProgramTXT()**

**参数：**

1. String path //待识别用户程序的路径

**相关变量：**

1. public final static List<String> *sentences* //以行为单位，存储读取到的语句

**作用：**

以行为单位，读取待分析的用户语句，以string形式存储，为后续词法分析提供便利。

* 1. **parseProgram()**

**返回：**

词法分析正确与否。

**相关变量：**

1. public final static List<String> *sentences* //以行为单位，存储读取到的语句
2. public static DFAState *startDFA*; //DFA的开始状态
3. public final static Map<DFAState,Map<String,DFAState>> *DFAGraph*
4. //DFA转换图，第一个键为当前的DFA结点，通过其得到Map<String,DFAState>.

随后再以转换路径String为键，得到下一个DFA结点

1. public final static List<Token> *tokenList;* //存储分析得到的token集合

**作用：**

**词法分析核心函数。**根据DFA转换图，自动分析语句集合，得到token集合。

**函数逻辑伪代码：**

1. 设置DFA状态currentDFAState为开始状态startDFA
2. 设置当前为非注释状态
3. 当前行为1，当前列为1
4. for(每一条语句sentence)
   1. while(列≤语句长度)
      1. 如果当前为非注释状态：
         1. 如果匹配到”/\*”块注释语句开头：
            1. 如果存在token，则添加token
            2. 设置当前为注释状态
            3. 列加2
      2. 否则：
         1. 如果匹配到”\*/”块注释语句尾部：
            1. 设置为非注释状态
            2. 列加2
      3. 如果匹配到“//”行注释：
         1. 跳过该行，continue;
      4. 如果当前为非注释状态：
         1. 如果token不存在：
            1. 初始化token，设置token的行，列
            2. 设置当前DFA状态为开始状态
      5. 如果匹配到” ”空格：
         1. 空格表示token的结束，如果token存在：
            1. 添加token，tokenList.add(token)
            2. 将token设置为不存在
      6. 否则（表示其它字符）：
         1. 如果当前token不存在：
            1. 初始化token，设置token的行，列
            2. 设置当前DFA状态为开始状态
         2. 根据当前符号和当前DFA状态，查找DFAGraph，查找下一个DFA状态
         3. 如果未找到：
            1. 如果当前结点为终态结点：

添加token，tokenList.add(token)

将token设置为不存在

列减一，便于重新匹配当前符号

* + - * 1. 否则：

词法分析出错，输出错误信息和错误位置，返回false

* + - 1. 否则（已找到）：
         1. 转到下一个DFA状态
         2. 将当前符号添加到token内容中
    1. 列加1
  1. 行加1

1. 返回true
2. **语法分析**
   1. **概述()**

调用LR1Utils类的startLR1()函数进行语法分析。该函数具有两个参数：List<Token> tokenList, String productionPath，分别为词法分析后得到的token列表和语法分析所使用的三型文法的产生式文件的路径，程序将输出语法分析的结果。

以下为LR(1)语法分析的大致流程：



* 1. **readProductionTXT()**

**参数：**

1. String path; //三型文法产生式的路径

**返回：**

List<Production>; //读取到的产生式的列表

**作用：**

根据用户输入的三型文法路径，读取出文件中的所有产生式。要求产生式左部只能有一个非终结符，且产生式右部每个文法符号之间通过空格分割。非终结符必须以大写字母开头。

**函数逻辑伪代码：**

1. while(文件没有读取完)
   1. 读取产生式左部直到’->’为止
   2. while(当前行没有读取完)
      1. if(当前字符不是空格)
         1. 继续读取
      2. else(是空格或者读取到当前行末尾)
         1. 将文法符号添加到产生式右部
   3. **Grammar.creat ()**

**参数：**

1. Symbol start; //文法的开始符号
2. List<Production> productionList; //读取到的产生式列表

**返回：**

Grammar //创建好的文法对象

**作用：**

根据输入的产生式列表和文法的开始符号，判断产生式是否符合规则：产生式不能为空且第一个产生式左部必须是开始符号。若产生式符合规则，则将产生式进行整理。如A->a, A->b的产生式将整理为A->a | b。最后调用构造函数返回一个Grammar对象。

**函数逻辑伪代码：**

1. if(产生式为空 或 第一条产生式左部不是开始符号)
   1. 跳出
2. 整理产生式
3. 创建grammar对象
   1. **generateProductioItemSets ()**

#### getFirstSet()

**参数：**

1. Symbol VNSybmol; //非终结符

**相关变量：**

1. private Map<Symbol,FirstSet> firstSetMap; //非终结符对应的First集
2. private final LinkedHashMap<Symbol, List<Production>> productionMap; //以symbol为键，记录以symbol开头的产生式

**返回：**

该非终结符对应的First集合

**作用：**

求非终结符对应的First集合，并利用HashMap建立由非终结符到First集的映射，避免重复求解First集。

**函数逻辑伪代码：**

1. if(firstSetMap.containsKey(VNSymbol))
   1. 返回HashMap中的FirstSet
2. 获取该VNSymbol对应的List<Production>
3. for(每一条产生式)
   1. if(产生式不能推出空)
      * 1. 获取产生式右部的符号串
        2. if(当产生式包含左部符号，如A->aAb)
           1. 记录产生式到badProduction列表
        3. else:
           1. for(产生式右部的每个文法符号)

if(如果是终结符):

得到First集合，跳出for循环

else(如果是非终结符):

递归调用getFirstSet()

将getFirstSet的结果添加到结果集

if(getFirstSet的结果包含空):

继续求下一个符号的First集合

else:

跳出for循环

* + - * 1. if(产生式右侧的非终结符的First集合都包含空)

结果集包含空

* 1. else(产生式能直接推出空):
     + 1. 结果集包含空

1. for(badProduction列表中的每条产生式)
   1. for(产生式右部的每个文法符号)
      * 1. if(终结符):
           1. 直接添加到结果集
           2. 跳出for循环
        2. else(非终结符):
           1. if(形如A->Aa的形式)

if(A不能推出空)

直接跳出for循环

* + - * 1. if(形如A->Ba的形式)

递归调用getFirstSet，求B的first集

添加B的first集

if(B的first集能包含空)

继续查找下一个字符

else:

跳出for循环

1. 将结果集添加到firstSetMap中
2. 返回First集

#### getFirstSetMap()

**返回：**

Map<Symbol,FirstSet> //所有非终结符对应的First集

**相关参数：**

1. private Map<Symbol,FirstSet> firstSetMap; //非终结符对应的First集
2. private final LinkedHashMap<Symbol, List<Production>> productionMap;

**作用：**

每次调用时，先判断firstSetMap是否已经生成。如果已经生成则直接返回。

否则，通过查找Grammar中的所有非终结符集合VNSet，调用getFirstSet()函数获取每个非终结符的First集合，并存储到firstSetMap中。

#### getFirstSetBySymbols()

**参数：**

1. List<Symbol> symbols; //需要求first集的文法符号串

**返回：**

该文法符号串的First集

**作用：**

求当前文法符号串对应的First集合。

函数逻辑伪代码：

1. 调用getFirstSetMap()方法获取所有非终结符对应的First集
2. for(输入文法符号串中的每一个文法符号)
   1. if(为终结符)
      * 1. 结果集中添加当前符号
        2. 跳出for循环
   2. else(为非终结符)
      * 1. 查找firstSetMap，获得当前文法符号的First集
        2. 将查找到的First集添加到结果集中
        3. if(查找到的First集不包含空串)
           1. 跳出for循环
3. if(如果每个文法符号的First集包含空)
   1. 结果集包含空
4. 返回结果集

#### closure()

**参数：**

1. Set<ProductionItem> itemSet; //需要求闭包的项目
2. Grammar grammar; //文法对象

**返回：**

Set<ProcutionItem> //求闭包后的项目结果

**作用：**

对LR(1)项目集中的项目求闭包。规则如下：

1.假定I是一个项目集，I的任何项目都属于CLOSURE(I)  
2.若有项目A->α•Bβ,a属于CLOSURE(I),B->δ是文法中的产生式，β∈V\*，b∈FIRST(βa), 则B->•δ,b也属于CLOSURE(I)  
2.重复2，直到CLOSURE(I)不再增大为止

**函数逻辑伪代码：**

1. 将itemSet添加到结果集resultItems
2. 将itemSet中的每一个项目依次放入栈中
3. while(栈不空)
   1. 获取栈顶的文法项目，并出栈
   2. 获取分隔符后的下一个文法符号delimiterSymbol（若在末尾则设置为结束符号’#’）
   3. if(delimiterSymbol是非终结符)
      * 1. 获取该非终结符对应的产生式列表productionList
        2. 调用getFirstSetBySymbols，获取First(βa)
        3. for(productionList中的每一个产生式)
           1. for(First(βa)中的每一个文法符号)

创建新的项目，将文法符号作为新项目的展望符

if(结果集中不包含刚创建的项目)

将项目添加到结果集中

项目入栈

1. 返回结果集

#### Goto()

**参数：**

1. ProductionItemSet productionItemSet; //原项目集
2. Symbol symbol; //当前文法符号
3. Grammar grammar; //文法

**相关参数：**

1. public static final Map<ProductionItemSet,Map<Symbol,ProductionItemSet>> *GOTO\_MAP*  //存储项目集之间的转换关系，以ProductionItemSet原项目集为键，获得该项目集的转换表。转换表以Symbol转换符号为键，得到转换后的项目集

**返回：**

ProductionItemSet; //转换后的项目集

**作用：**

获得当前项目集面临文法符号后转换到的下一个项目集，并对每次转换的结果进行存储，避免重复计算。规则如下：

1.GOTO(I,X) = CLOSURE(J)

2.I为 LR(1)的项目集,X是文法符号,J={任何形如[A->αX•β,a]的项目 | [A->α•Xβ,a]∈I}

3.首先以[S'->•S,#]为初态集的初始项目，对其求闭包和转换函数，直到项目集不再增大为止

**函数逻辑伪代码：**

1. 根据输入的项目集和文法符号查找GOTO\_MAP
2. if(能找到对应的记录)
   1. 直接返回项目集
3. for(原项目集中的每条文法项目)
   1. 获得分隔符后的第一个文法符号nextSymbol
   2. if(nextSymbol与输入的文法符号相同)
      * 1. 将分隔符往后移动一位，创建新的文法项目，添加到Set<ProductionItem> currentProductionItemSet中
4. 对currentProductionItemSet调用closure()方法求闭包
5. 根据求闭包运算后的结果，调用ProductionItemSet的create()方法创建项目集
6. 将转换的路径和结果添加到GOTO\_MAP中
7. 返回项目集

#### generateProductionItemSet()

**参数：**

1. Grammar grammar; //文法符号

**返回：**

List<ProductionItemSet> //根据文法生成的所有的项目集的列表

**作用：**

根据输入的文法，从文法的开始符号对应的产生式开始，生成该文法对应的**所有的项目集。**

**函数逻辑伪代码：**

1. 获取文法的开始符号对应的项目集，添加到结果集中
2. 将项目集入栈
3. while(栈不空)
   1. 获取栈顶的项目集，栈顶元素出栈
   2. for(项目集中的每一个项目)
      * 1. 获取项目中分割符后的第一个文法符号，添加到nextSymbolList中。
   3. for(nextSymbolList中的每一个文法符号)
      * 1. 调用Goto()函数，获得该项目集和该文法符号对应的下一个项目集
        2. if(下一个项目集不为空且下一个项目集没有在结果集中)
           1. 添加到结果集
           2. 添加到栈中
4. 返回结果集
   1. **createLR1Table()**

**参数：**

1. Grammar grammar; **in参数** //文法
2. List<ProducionItemSet> productionItemSetList; **in参数** //所有项目集的列表
3. Map<ProductionItemSet, Map<Symbol, ActionItem>> actionTable; **out参数**

//action表，以ProductionItemSet当前项目集和Symbol面临的文法符号决定下一步的动作

1. Map<ProductionItemSet, Map<Symbol, GotoItem>> gotoTable; **out参数**

//goto表，以ProductionItemSet当前项目集和Symbol面临的文法符号决定下一步应该转向的状态

**作用：**

根据项目集和项目集之间的转换关系，得到action表和goto表。

**函数逻辑伪代码：**

1. for(对于项目集列表中的每一个项目集)
   1. for(对于项目集中的每一条项目)
      * 1. 获得当前项目分隔符后的第一次文法符号currentSymbol
        2. if（currentSymbol是非终结符）
           1. 调用Goto()函数，获得下一个项目集
           2. 将转换路径和结果添加到gotoTable中
        3. else(currentSymbol是终结符)
           1. if(currentSymbol是结束符号’#’，即形如A->b •,a)

获取当前项目对应的产生式

if(产生式为第一条产生式，即S’->S)

生成ACC Action

将转换路径和结果添加到actionTable中

else:

生成归约Action

将转换路径和结果添加到actionTable中

* + - * 1. else(如果不是结束符号)

生成移进Action

将转换路径和结果添加到actionTable中

* 1. **mathTokenList()**

#### matchTokenList()

**参数：**

1. List<Token> inputTokens //词法分析得到的token列表
2. ProductionItemSet startItemSet //开始项目集
3. Map<ProductionItemSet, Map<Symbol, ActionItem>> actionMap;

//action表，以ProductionItemSet当前项目集和Symbol面临的文法符号决定下一步的动作

1. Map<ProductionItemSet, Map<Symbol, GotoItem>> gotoMap;

//goto表，以ProductionItemSet当前项目集和Symbol面临的文法符号决定下一步应该转向的状态

**返回：**

Boolean; //语法分析是否成功

**作用：**

将inputTokens进行预处理。因为token类型为标识符和常量时，其具体内容对语法分析没有任何影响，所以直接设置为“id”和“const”。预处理后，调用match()函数进行语法分析。

**函数逻辑伪代码：**

1. 创建List<Symbol>对象
2. for(inputTokens中的每一个token)
   1. if(token类型为标识符)
      * 1. 生成内容为”id”的终结符，添加到List<Symbol>中
   2. if(token 类型为常量)
      * 1. 生成内容为“const”的终结符，添加到List<Symbol>中
   3. else:
      * 1. 生成内容为token实际内容的终结符，添加到List<Symbol>中
3. 调用match方法，获得返回值并返回。

#### match ()

**参数：**

1. List<Symbol> inputSymbols //输入的文法符号列表
2. ProductionItemSet startItemSet //开始项目集
3. Map<ProductionItemSet, Map<Symbol, ActionItem>> actionMap;

//action表，以ProductionItemSet当前项目集和Symbol面临的文法符号决定下一步的动作

1. Map<ProductionItemSet, Map<Symbol, GotoItem>> gotoMap;

//goto表，以ProductionItemSet当前项目集和Symbol面临的文法符号决定下一步应该转向的状态

**返回：**

Boolean; //语法分析是否成功

**作用：**

语法分析的核心函数。根据goto表和acton表，进行语法分析，并输入语法分析的过程。如果出现错误，提出大致的出错原因。

**函数逻辑伪代码：**

1. 向inputSymbols的最后放入结束符号
2. 将开始项目集startItemSet放入状态栈
3. 将结束符号放入到符号栈
4. while(true)
   1. if(状态栈为空)
      * 1. 输出错误信息——状态栈空，返回false
   2. 获取状态栈栈顶项目集，栈顶元素出栈
   3. 获取待匹配符号（inputSymbols中的第一个元素）
   4. 根据当前项目集和待匹配符号找到action表
   5. if(对应的action表项为空)
      * 1. 根据当前项目集用空符号查找aciton表
        2. if(对应的action表项不为空)
           1. 将当前待匹配符号设置为空
   6. if(action表为空)
      * 1. 输出错误信息，匹配错误，返回false
   7. else(action表不为空)
      * 1. if(action为移进项目)
           1. 将action表项对应的项目集添加到状态栈
           2. 向符号栈中添加当前符号
        2. if(action为ACC项目)
           1. if(如果当前符号为结束符号’#’)

输出“分析成功”，返回true

* + - * 1. else

输出错误信息，返回false

* + - 1. if(action为归约项目)
         1. 根据action表项获得归约用的产生式
         2. if(产生式右部的文法符号数量＞状态栈大小)

输出错误信息，返回false

* + - * 1. 根据产生式文法符号数量，将符号栈和状态栈中对应个数的元素出栈
        2. 将产生式左部添加到符号栈
        3. 根据当前项目集和文法符号，找到对应goto表项
        4. 根据goto表项，获取转到的项目集，将对应的项目集入栈
  1. 匹配下一个文法符号