语法分析器描述文档

# 概述

本文档描述了一个语法分析器的设计及实现过程，实现了输入一个词法单元（Token）序列，分析后生成一个最左推导产生式序列。语法分析器使用LL(1)文法，在代码中输入文法（产生式组），程序自动生成LL(1)预测分析表。之后使用该分析表对第一步产生的Token序列进行分析，产生最左推导过程。语法分析器集成了上次实验中完成的词法分析器，最终实现了将源代码解析为token序列，然后再分析产生最左推导产生式序列。

# 需求分析

## 业务需求

对词法分析产生的token序列进行语法分析，生成最左推导的产生式序列。

## 功能需求

FR1 读入源程序代码

FR2 对源程序进行词法分析，生成token序列

FR3 自动生成预测分析表，对token序列进行语法分析

FR4 将语法分析产生的结果，最左推导产生式序列输出到文件中。

FR5 语法分析失败时，将错误信息打印到输出文件中。

# 主要实现步骤

## 设计非二义文法（CFG）

首先设计了带公共左因子的二义文法，然后对公共左因子进行合并，得到下面的非二义文法。

***/\*\****

***\* Created by Thpffcj on 2017/11/1.***

***\****

***\* START -> BASE T1***

***\* T1 -> START|~***

***\* BASE -> FOR|WHILE|IFELSE|SENTENCE;***

***\* WHILE -> while(CONDITION) BLOCK***

***\* FOR -> for(SENTENCE;CONDITION;SENTENCE) BLOCK***

***\* IFELSE -> if(CONDITION) BLOCK else BLOCK***

***\* BLOCK -> {SENTENCE;}***

***\* SENTENCE -> id=FACTOR***

***\* EXPRESSION -> FT2***

***\* T2-> +FACTOR|\*FACTOR|-FACTOR|/FACTOR|~***

***\* CONDITION -> FT3***

***\* T3-> ==FACTOR|>FACTOR|<FACTOR|>=FACTOR|<=FACTOR|!=FACTOR***

***\* FACTOR -> id|const***

***\*/***

说明：

1. 功能

可以识别带有代码块的for，while，if-else；表达式；条件判断；单个语句；

并可以对上述元素进行任意的组合，和任意嵌套。

|  |  |
| --- | --- |
| 简写中的非终结符（上述） | 完整写法的文法中的非终结符（输出结果） |
| *START* | Start (开始符) |
| T1 | T1 |
| BASE | Base |
| WHILE | While(while是终结符，区别有点小) |
| FOR | For(for是终结符) |
| IFELSE | IfElse |
| BLOCK | Block |
| SENTENCE | Sentence |
| EXPRESSION | Expression |
| CONDITION | Condition |
| FACTOR | Factor |
| T2 | T2 |
| T3 | T3 |

## 后续步骤

1. 实现First和Follow两个主要方法。
2. 使用First和Follow，根据文法生成预测分析表（PPT）。
3. 在语法分析器控制器中，维护一个token序列读头指针，一个符号栈；读入一个token，获取当前符号栈栈顶，然后到PPT中找到对应的最左推导产生式，加入到产生式序列中。
4. 若在上一步中找不到对应的产生式，则说明分析失败，源代码存在语法错误。输出错误信息。
5. 若没有出现4的情况，则按最左推导顺序输出全部产生式。

# 详细设计

## 各类职责

lexicalAnalyzer 包是实验一中实现的词法分析器，负责将源代码分析成为token序列做为语法分析器的输入。

syntaxAnalyzer 包是语法分析器的实现部分，其中：

syntax包为语法的模型

syntax.NonterminalType 枚举了所有的非终结符号（终结符号在lexicalAnalyzer.analyzer.DFA.TerminalType中进行了枚举）。

syntax.SymbolType 只是一个标识，是一个空接口，只表示符号的含义，被NonterminalType和TerminalType实现。

syntax.Symbol是符号接口，规定了符号（终结符模型，非终结符模型）的职责，包括判断自己是终结符还是非终结符，计算first，获取自身的类型。

Nonterminal是非终结符的模型，Symbol的实现类，实现了Symbol规定的所有职责。

Terminal是终结符的模型，Symbol的实现类，实现了Symbol规定的所有职责。

Production是产生式的模型，成员包括一个非终结符左部，一个符号序列右部，主要职责是计算右部的first。

Grammar是文法的模型，成员是Production序列，负责初始化文法；实现Follow算法，根据输入的文法，调用Production的first方法，和自身的follow生成PPT，供Parser使用。

Analyzer包 包含了语法分析器控制器和PPT模型，其中 PPT为预测分析表的模型，负责根据一个非终结符和一个终结符获得产生式。

Parser为语法分析器控制器，顺序读入token，维护符号栈，保存从PPT中获取到的产生式。

## 主要算法

1. 计算产生式右部First方法，根据编译原理（龙书）上的first算法实现。
2. 计算非终结符Follow的方法，根据编译原理（龙书）上计算follow的规则进行实现，最终这个方法实现为一个递归算法，若非终结符是产生式右部的最后，或非终结符后面部分包括空符号，则对产生式左部求follow，将结果加入上一步得到的结果中，直到不再出现空符号。
3. 生成PPT的方法，根据编译原理（龙书）上的算法实现。
4. 控制器的逻辑，根据编译原理（龙书）上的算法实现。

## 主要数据结构

PPT: Map<NonTerminalType,Map<TerminalType,List<Production>>>

预测分析表的二维结构采用两层嵌套的HashMap实现，这样可以根据一个终结符和一个非终结符快速定位对应的产生式，是表驱动的一种实现方式，通过这种技巧可以有效避免多重if-else的分支结构，代码效率更高，可读性更好，更易于维护。

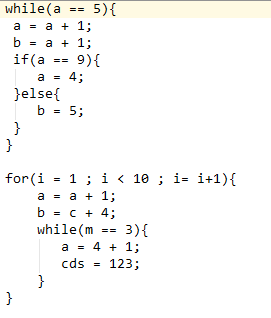
Production（产生式）: 左部 NonTerminal , 右部List<Symbol>

Symbol是终结符和非终结符的总称，因为右部可能同时包含两种类型的符号，使用list盛装右部的符号便于求first和follow。

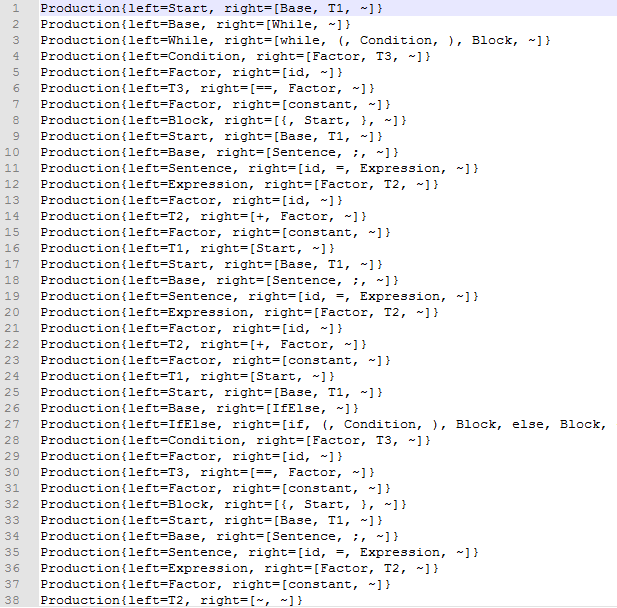
SymbolStack(符号栈): Stack<Symbol> 直接使用栈结构来实现符号栈，可以很好地模拟分析过程

# 测试用例

输入：见根目录下source.txt



输出：见根目录下 production.txt



其中，production.txt在输入的源代码没有错误时，保存最左推导产生式序列，否则保存错误信息提示。

# 错误处理

当出现错误时，分析终止，将错误信息输出到production.txt。

# 感受与总结

第二次实验，我选择了LL(1)方法进行语法分析。在完成本次实验时，我希望尽可能实现较为完备的语法，包括各种组合和嵌套关系，因此设计的文法也较为复杂，含有较多的产生式，手工消除左递归和公共左因子后，产生式数量更多了，如果手工绘制PPT，则需要花费很大的人力，且容易出错，因此我在程序中完成了根据文法自动构建PPT的逻辑，这样一来，我就可以随意调整文法，通过运行程序就可以判断文法是否是LL(1)文法，如果是就可以顺利的构造正确的PPT。不过期间有大量的繁琐初始化过程，不留意初始化文法就会添加错误符号造成分析结果和预期有差别，经过不断打断点看中间变量的值最终实现这个语法分析器，使我对语法分析过程加深了理解。