LL(1)文法的语法分析器

2020-05-17 14:16:47

## 实验目的

1. 掌握生成语法分析器的方法，加深对语法分析原理的理解。
2. 掌握设计、编制并调试语法分析程序的思想和方法。
3. 本实验是高级语言程序设计、数据结构和编译原理中词法分析原理等知识的综合。

## 实验内容

1. 选择一种熟悉的高级语言（如C语言，C++，VB或VC等），设计、编写一个语法分析子程序。
2. 要求如下：

   输入：

      无二义性的上下文无关文法G.

      一段词法分析的输出记号流.

   输出：

      如果分析成功：一棵构造好的语法树.

      如果分析失败：抛出语法错误.

   要求：

      可以引入3.1.2中的错误处理机制，试图检查出所有可能存在的语法错误.

1. 编写的程序具有一定的查错能力。提交的实验报告中要有实验名称、实验目的、实验内容、实验程序清单、调试过程和运行结果，程序的主要部分作出功能说明，并有实验收获体会或改进意见等内容。

## 实验程序清单

根目录/src

   algorithm   FIRST集,FOLLOW集,预测分析表,驱动器算法的实现.

   cfg   CFG包,包括CFG的读取,消除左递归,消除公共左因子等算法.

   data   数据包

      cfg   存放CFG文本的包.

      test   存放测试用例的包

      log   存放输出日志的包

   util   工具包,实现读/写文件的操作.

   Parsing.java   主调函数

## 调试过程和运行结果

   调用src/Parsing.java中main函数即可.输出结果存放于src/log目录中.

   src/Parsing.java中指定了CFG,测试,输出文本的文件夹路径:

   请使用IDEA/ECLIPSE/VS CODE等IDE查看输出结果,不要使用记事本查看输出结果(格式会乱).

public class Parsing {  
 // 默认CFG路径  
 static String baseCfg = "src/data/cfg/";  
 // 默认测试文件路径  
 static String baseTest = "src/data/test/";  
 // 默认输出路径  
 static String baseOut = "src/log/";  
}

   主调函数parse结构如下:

/\*\*  
 \* 语法分析主调函数  
 \*   
 \* @param cfgPaths CFG文件  
 \* @param testPaths 测试文件  
 \* @param outputPaths 输出文件  
 \* @param num 分析数目  
 \*/  
 public static void parse(String[] cfgPaths, String[] testPaths, String[] outputPaths, int num)

   cfgPaths[i]指定第i个文法.

   testPaths[i]指定第i个文法的测试用例.

   outputPaths[i]指定该测试用例的输出文件.

   num指定i的最大值.

   在main函数中有如下示例:

public static void main(String[] args) {  
 String[] cfgPaths = { "cfg1.txt", "cfg2.txt" };  
 String[] testPaths = { "test1.txt", "test2.txt" };  
 String outputPaths[] = { "log1.txt", "log2.txt" };  
  
 parse(cfgPaths, testPaths, outputPaths, 2);  
 }

## 程序的主要部分及其功能说明

### 功能说明

#### 文法规范

1. 每个产生式独占一行.
2. 产生符号为->,中间不允许有空格.
3. 所有符号以空格分隔.
4. 以大写字母开头的符号为非终结符,其余均为终结符(ε除外).

# 为输入结束符,不要使用#定义文法.

1. 文本中无空行.

**示例:**

L -> E ; L  
 | ε  
E -> E + T  
 | E - T  
 | T  
T -> T \* F  
 | T / F  
 | T mod F  
 | F  
F -> ( E )  
 | id  
 | num

#### 测试用例规范

1. 每个测试用例独占一行.

# 代表终结符,测试用例中不能出现#.

1. 测试用例中不能出现ε.
2. 测试用例中所有符号以空格分隔.
3. 行首行尾无空格且文本中无空行.

**示例:**

id + id \* id ;  
id + id \* num - num / id ;  
id + num ; num - id ; id \* ( num ) ; num / ( id ) ;

#### 后缀

   在src/cfg/CFG.java中定义了后缀:

public class CFG implements Cloneable {  
 public static final char appendChar = '`';  
}

   消除左递归及公共左因子过程中产生的新产生式左部会不断添加该后缀直至无重复名.

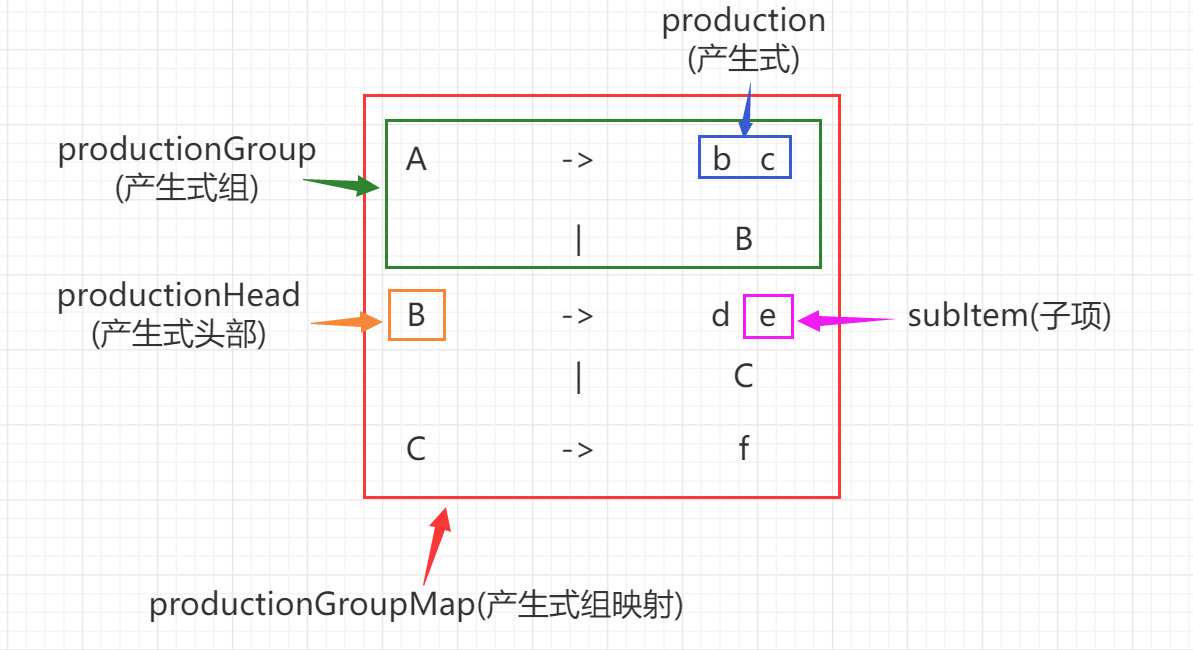
#### 系统环境要求

   JDK8及以上(支持lambda).

### 主要功能

#### CFG构造

   我们规定产生式结构描述如下:

[](https://imgchr.com/i/YhsBc9)

##### 数据结构

[YRQ8j1.png](https://imgchr.com/i/YRQ8j1)

   可以看到,数据结构的层次比较深.而且为了降低时间复杂度,需要查找的集合都使用了Hash(后续算法也是).

   因此程序中大量的使用了lambda表达式,例如:

// 遍历产生式  
cfg.productionGroupMap.forEach((key, val) -> {  
 // ...  
 val.productions.forEach(production -> {  
 // ...  
 });  
});

   如果不使用lambda表达式,代码的形式为:

for (Map.Entry<String,ProductionGroup> entry : cfg.productionGroupMap.entrySet()) {  
 String key=entry.getKey();  
 ProductionGroup val=entry.getValue();  
 // ...  
 for (Production production : val.productions) {  
 // ...  
 }  
 }

   如果使用JDK10及以上环境,可以使用var关键字简洁代码:

for (var entry : cfg.productionGroupMap.entrySet()) {  
 var key=entry.getKey();  
 var val=entry.getValue();  
 // ...  
 for (var production : val.productions) {  
 // ...  
 }  
 }

   显然,使用lambda表达式最为简洁.因此,系统环境要求JDK8及以上.

   但由于lambda表达式只能访问final类型的外部变量,所以要对外部的基本类型做一点处理.

   例如在src/cfg/CFG.java中:

// 如果循环后，没有匹配成功(mayExist为false)，则代表没有左公因子了  
 boolean[] mayExist = { true };

   标志位mayExist的类型为boolean[]而非boolen就是为了在后续的lambda表达式中进行访问/修改.

#### 消除左递归

   算法与教材相同,不再赘述.

##### 示例

S -> a A B e  
A -> b  
 | A b c  
B -> d

   消除左递归后的文法:

S -> a A B e   
  
A -> b A`   
  
B -> d   
  
A` -> b c A`   
A` -> ε

   由于顺序并未严格排序,故后续求FIRST集,FOLLOW集都采用递归求解.

#### 消除公共左因子

   算法与教材相同,不再赘述.

##### 示例

S -> A  
A -> a b  
 | a c d  
 | a c e

   消除公共左因子后的文法:

S -> A  
  
A -> a A`  
  
A` -> b  
A` -> c A``  
  
A`` -> d  
A`` -> e

#### FIRST集

   由于产生式非严格有序,采用递归求解FIRST集.算法与教材相同,不再赘述.

##### 数据结构

public class FirstSet {  
 public CFG cfg;  
 public Map<String, Set<String>> firstSet;  
 public boolean isUpdated = false;  
}

#### FOLLOW集

   由于产生式非严格有序,采用递归求解FOLLOW集.算法与教材相同,不再赘述.

##### 数据结构

public class FollowSet {  
 public static final String InputRightEndSym = "#";  
 public CFG cfg;  
 public Map<String, Set<String>> firstSet;  
 public Map<String, Set<String>> followSet;  
 public boolean isUpdated = false;  
}

##### 示例

CFG:  
开始符号 : S  
S -> a A B e   
  
A -> b A`   
  
B -> d   
  
A` -> b c A`   
A` -> ε   
  
FIRST集:  
S :[a]  
A :[b]  
B :[d]  
A` :[b, ε]  
  
FOLLOW集:  
S :[#]  
A :[d]  
B :[e]  
A` :[d]

##### 补充说明

   对于非LL(1)文法,程序可能死递归.以教材的文法G3.10'为例:

S -> i C t S S'  
S' -> e S  
 | ε   
C -> b  
  
教材给出的FOLLOW集:  
FOLLOW(S) = {#,e}  
FOLLOW(S') = {#,e}

   这里给出程序形成死递归的步骤:

1. 求解FOLLOW(S).
2. 根据S' -> e S,将FOLLOW(S')加入FOLLOW(S).
3. 尚未求得FOLLOW(S'),递归求解FOLLOW(S').
4. 根据S -> i C t S S',将FOLLOW(S)添加到FOLLOW(S').
5. 回到步骤1,形成死递归.

   消除死递归的方法也很简单,设置一个递归层数上限值(这个值跟产生式组数目相关)/给每个产生式组打上标记(类似DFS检测环).

  更好一点的做法是构造依赖树,检查有无循环依赖,且依赖树在消除文法二义性中也用得到.

  但由于实验要求明确输入为LL(1)文法,故程序未作处理.

#### 构造预测分析表

##### 数据结构

public class AnalyzeTable {  
 public CFG cfg;  
 public Map<String, Set<String>> firstSet;  
 public Map<String, Set<String>> followSet;  
 // 预测分析表  
 public Map<String, Map<String, List<String>>> table;  
 public boolean isUpdated;  
}

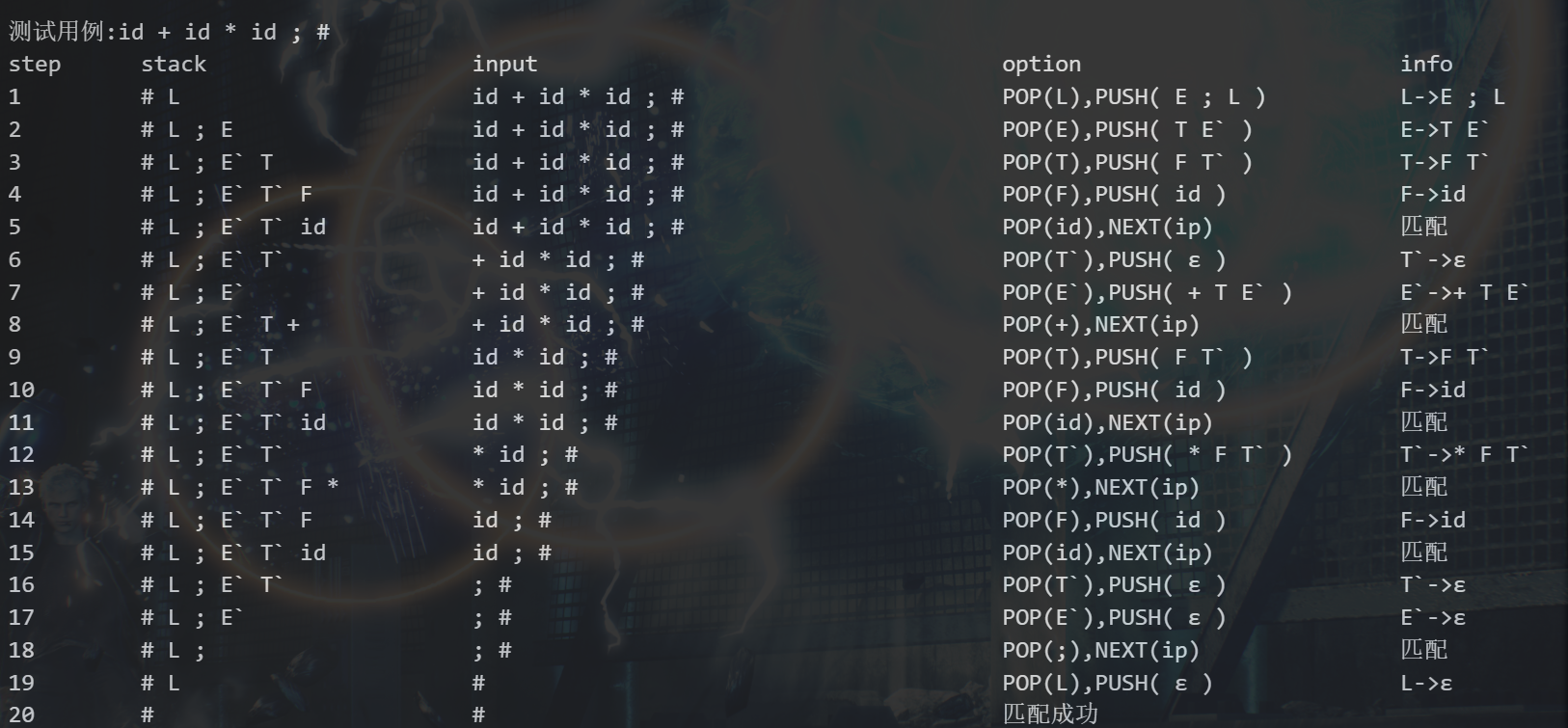
##### 示例

预测分析表:  
 a e b c d #   
S aABe   
A bA`   
B d   
A` bcA` ε

#### 驱动器算法

   能给出每一步的信息以及错误信息.

##### 示例

[](https://imgchr.com/i/tImGEF)

## 实验收获体会

1. 对LL(1)语法分析的算法有了更深的认识.
2. 能够与之前的词法分析实验结合,思考设计一个编译器时,各个模块负责的功能.

## 改进意见

1. 代码的安全性存在隐患,为节省时间,所有数据结构的属性均为public.
2. 错误分析有待进一步加强.
3. 对于某些非二义性算法,程序会死递归.可以添加二义性的检查,避免死递归.