编译原理实习二

——LL(1)语法分析

班级：
姓名：
学号：

AnDJ

一、 提交文件

Main.java

二、 编译及运行环境

JDK 1.8.0

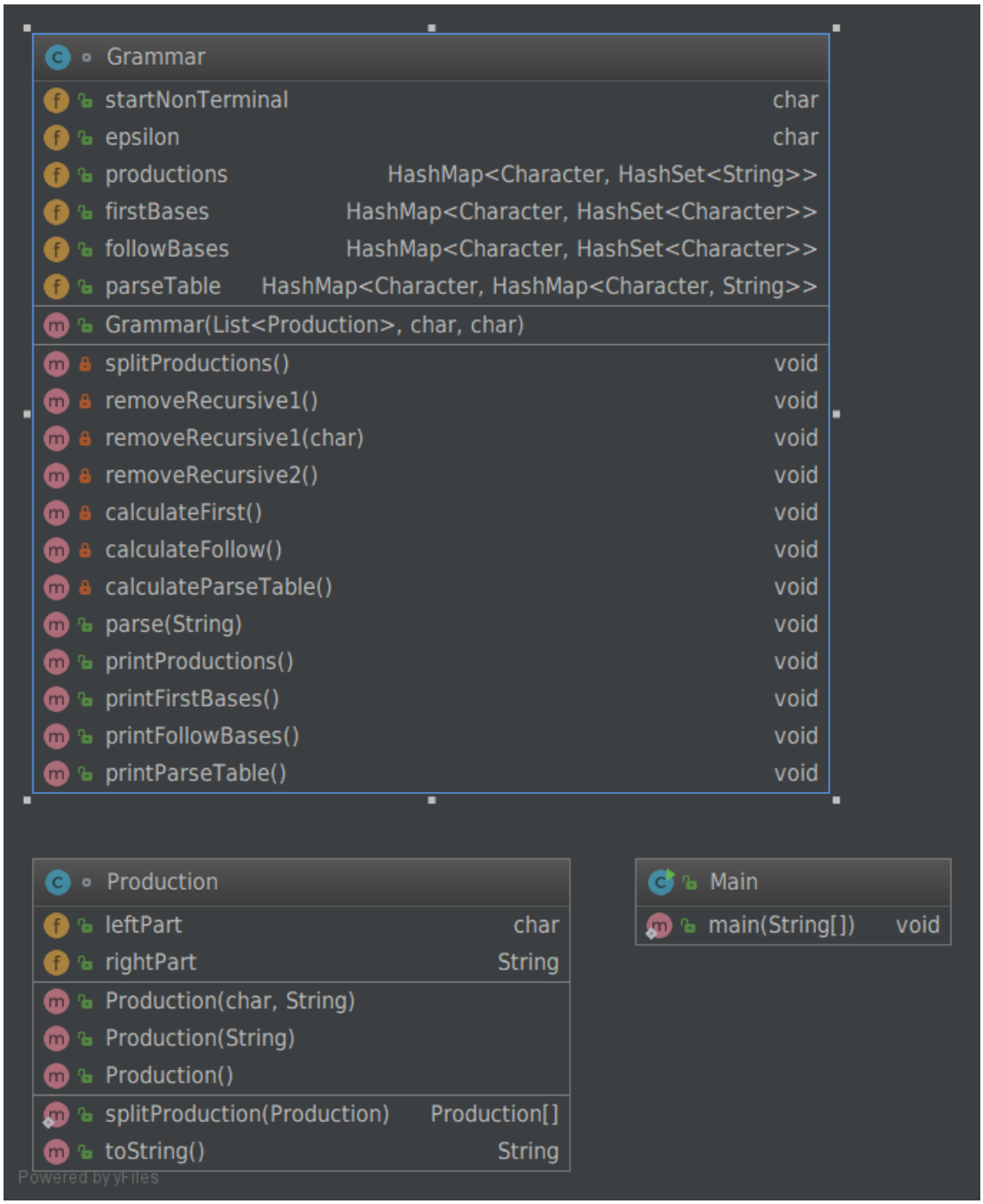
三、 执行方式

1） javac Main.java
2） java Main
3） 提示 1：输入起始非终结符。
4） 提示 2：输入多条语法规则，**添加完成后输入 end 结束。**
5） 提示 3：输入待解析的句子。
6） 过程中输出所有产生式、first 基和 follow 基、LL(1)表及解析句子的过程和结果。
7） 空字这里默认为“ε”，键盘无法键入可以复制粘贴提示中给出的该字符。

四、 运行约束

产生式约束：
1） 输入产生式中的字符必须是可见 ASCII 字符（ε除外）。
2） 产生式输入格式为：XXX->YYY，连接左部和右部的符号是->，两个字符都是英文字符‘-’，‘>’。

五、 代码文件类图



六、 实现过程

1. 输入语法产生式的解析与保存
   1）产生式是逐条输入的，在进入分析之前需要进行简单的判断和保存。产生式的输入形式为“XXX->YYY”，不满足该形式的字符串视为无效字符串。同时，根据->进行左右部划分，并简单保存下来。
   2）产生式的保存形式封装为 Production 类，左部为 char 字符型，右部为字符串。构造函数通过合法的字符串构建该对象。
   3）通过逐条读取合法产生式字符串，构造 Production 集合，该集合为初次判别后的产生式集合，且左右部已经分开，用于接下来的语法分析。
2. 分析语法前的保存形式

1） 构建 Grammar 类，针对语法解析函数封装。该类的 UML 图详见总图。该类的属性和函数封装如下：

属性：

startNonTerminal， char 类型，保存语法的起始非终结符。该变量对后面的 Follow 基的求解是必要的。

Epsilon，char 类型，保存 epsilon 的具体表示形式，默认为 ε，可以是其他字符。允许使用该代码的开发者基于源码修改。

Productions， HashMap<Character,HashSet<String>>类型，保存为 HashMap。这里有必要解释一下，该语法所有产生式归结起来可以看作是一系列右部集合，每一个右部集合内的所有右部，它们的左部是相同的。换种说法，该语法的所有产生式可以表示为：

X -> aYb | dad | nSa | mA | …
Y -> dsaN |dad | dsa | …
S -> … | … | … | … | … | …
…

可以通过“|”针对每条产生式的右部进行分割，分割的结果即一个集合。当进行递归消除的算法进行时，可以归结为针对集合的操作。

firstBases，HashMap<Character,HashSet<Character>>类型，单个非终结符的 first 基为字符集合，因此保存为该形式。

followBases，HashMap<Character,HashSet<Character>>类型，单个非终结符的 follow 基为字符集合，因此保存为该形式。

parseTable,HashMap<Character,HashMap<Character,String>>类型,后面解释。

函数：

splitProdutions(),分割右部存在“|”的产生式。

removeRecursive1()，消除直接左递归。

removeRecursive2()，消除间接左递归。

calculateFirst()，计算 first 基。

calculateFollow()，计算 follow 基。

Parse(String)，解析输入的字符串。

2） Grammar 类构造包含产生式集合，起始非终结符，epsilon 代替字符。Grammar 类根据产生式集合构造 Producitons，方法是通过 splitProduction 函数将每条产生式进行切割，构造对应左部的右部集合，保存至 productions。

3. 消除直接左递归和间接左递归

1） 消除直接左递归的算法：

原文法: E --> E a1 | E a2 | ... | E an | b1 | b2 | ... | bn
消除后: E --> b1 E’ | b2 E’ | ... | bn E’
E’--> a1 E’ | a2 E’ | ... | an E’ | epsilon

即首先将单个非终结符的右部集合划分成“该非终结符开头”“非该终结符开头”两个集合，然后根据上面的算法构造一个新的集合，进行集合操作向两个集合添加对应的右部集。算法实现为 removeRecursive1()函数。

2） 消除间接左递归的算法：

a)把所有非终结符号按一定序列排序为 E1, E2, ... En;

b) for i=1 to n do /*依次处理每个非终结符号*/
for j=1 to i-1 do /*处理第 1 个到 i-1 个*/

若存在 Ei --> Ej r
则改为 Ei --> S1 r | S2 r | ... | Sk r
其中 Ej --> S1 | S2 | ... | Sk

对 Ei 消除直接左递归

按照如上算法进行集合操作，先针对 Ei 集合的每一个 Ejr 的 Ej 进行替换，然后对 Ei 进行消除直接左递归算法即可。算法实现为 removeRecursive2()函数。

4. 求解 First 基

求解 First 基的算法为：

```
Do
    For every 非终结符
        For every 该非终结符对应右部集合中每个右部，即(…|…|.)中的一个
            For 该右部[1] to 该右部[n]
                If like aD  //终结符开头
                    将 a 放入该终结符的 first 基集合中
                    Break
                If like Bd  //非终结符开头
                    将 first(B)中非 epsilon 元素放入 first(该非终结符)中
                    If (  first(B)中包含 epsilon 元素 )
                        Continue
                    Else
                        Break
While(has first update)
```

该算法的实现为 calculateFirst()函数。

5. 求解 Follow 基

求解 Follow 基的算法为：

```
Follow(起始非终结符)添加 '$'
Do
    For every 非终结符
        For every 该非终结符的右部集合中的元素
            For 该元素[1] to 该元素[n]
                Char c=该元素[i]
                If C 为非终结符
                    If C 为最后一个元素，即 left->…C
                        follow(C) 添加 follow(left)所有元素
                    Else
                        If 该元素[i+1]为终结符，即 left->…Ca…
                            将该元素[i+1]加入 follow(C)
                        Else 即 left->…CA…
                            将 first(A)中非空字符放入 follow(C)
                            如果 first(A)中包含空字符
                                将 follow(A)放入 follow(C)
While(has follow update)
```

该算法的实现为 calculateFollow()函数。

6. 求解 LL(1)表

LL(1)表的存储形式为 HashMap<char,HashMap<char,string>>，即如下的形式：

非终结符\终结符	a	b	c	d
A	右部string	右部string	右部string	右部string
B	右部string	右部string	右部string	右部string
C	右部string	右部string	右部string	右部string
D	右部string	右部string	右部string	右部string

即外层 HashMap 存储键值对（A，HashMap），内层键值对为(a,右部 string)

构造该表的算法为：

```
For every 非终结符
    For every 该非终结符的右部集合中的元素（字符串）
        If 该字符串[0]为终结符且不为 epsilon
            LL(1)Table[left][该字符串[0]]=该右部字符串
        Else
            //该字符串[0]为 epsilon 或非终结符
            If 该字符串[0]为 epsilon
                For every element in follow(left)
                    LL(1)Table[left][element]=该右部字符串
                Continue
            //该字符串[0]为非终结符
            For element in first(该字符串[0])
                If element 非空字符
                    LL(1)Table[left][element]=该右部字符串
            If 该字符串[0]→* epsilon
                For element in follow(left)
                    LL(1)Table[left][element]=该右部字符串
```

该算法实现函数为 calculateParseTable()函数。

7. 解析待解析的字符串

该算法为：

1） 构建栈 stack，input 字符串，针对其采取对应的 Action

2） 将 $、始非终结符压入栈。

3） 根据 LL(1)表查找下一步，根据课上讲的算法计算，Action 如下：

Terminal、Replace、epsilon、Accept。具体过程这里从简，实现体为 parse 函数。

七、 测试

1. 用例 1 测试过程

```
1.INPUT THE START NON_TERMINAL:
>E
2.INPUT THE GRAMMAR:(copy ε from there)
>E->TX
>T->(E)|intY
>X->+E|ε
>Y->*T|ε
>end
```

```
----------------------------------all productions
T:
    [intY, (E)]
E:
    [intYX, (E)X]
X:
    [ε, +E]
Y:
    [ε, *T]
-------------------------------------------first
T:
    [(, i]
E:
    [(, i]
X:
    [ε, +]
Y:
    [ε, *]
-------------------------------------------follow
T:
    [$, ), +]
E:
    [$, )]
X:
    [$, )]
Y:
    [$, ), +]
---------------------------------------parse table
T:
    {(=(E), i=intY}
E:
    {(=(E)X, i=intYX}
X:
    {$=ε, )=ε, +=+E}
Y:
    {$=ε, )=ε, *=*T, +=ε}
3.INPUT THE SENTENCE:
>int*(int+int)
-----------------------------------------parse
STACK               INPUT               ACTION
[$, E]              int*(int+int)$      intYX
[$, X, Y, t, n, i]  int*(int+int)$      Terminal
[$, X, Y, t, n]     nt*(int+int)$       Terminal
[$, X, Y, t]        t*(int+int)$        Terminal
```

```
[$, X, Y]           *(int+int)$         *T
[$, X, T, *]        *(int+int)$         Terminal
[$, X, T]           (int+int)$          (E)
[$, X, ), E, (]     (int+int)$          Terminal
[$, X, ), E]        int+int)$           intYX
[$, X, ), X, Y, t, n, i]int+int)$           Terminal
[$, X, ), X, Y, t, n]nt+int)$            Terminal
[$, X, ), X, Y, t]  t+int)$             Terminal
[$, X, ), X, Y]     +int)$              ε
[$, X, ), X]        +int)$              +E
[$, X, ), E, +]     +int)$              Terminal
[$, X, ), E]        int)$               intYX
[$, X, ), X, Y, t, n, i]int)$               Terminal
[$, X, ), X, Y, t, n]nt)$                Terminal
[$, X, ), X, Y, t]  t)$                 Terminal
[$, X, ), X, Y]     )$                  ε
[$, X, ), X]        )$                  ε
[$, X, )]           )$                  Terminal
[$, X]              $                   ε
[$]                 $                   Terminal
                    result:             Accept
```

2. 用例 2 测试过程

```
1.INPUT THE START NON_TERMINAL:
>S
2.INPUT THE GRAMMAR:(copy ε from there)
>S->%aT|U!
>T->aS|baT|ε
>U->#aTU|ε
>end
-----------------------------------all productions
S:
    [%aT, U!]
T:
    [ε, aS, baT]
U:
    [ε, #aTU]
-------------------------------------------first
S:
    [!, #, %]
T:
    [ε, a, b]
U:
    [ε, #]
-----------------------------------------follow
```

S:

[!, #, $]

T:

[!, #, $]

U:

[!]

----------parse table

S:

{!=U!, #=U!, $=U!, %=%aT}

T:

{!=ε, a=aS, b=baT, #=ε, $=ε}

U:

{!=ε, #=#aTU}

3.INPUT THE SENTENCE:

>#abaa%aba!

----------parse

STACK	INPUT	ACTION
[$, S]	#abaa%aba!$	U!
[$, !, U]	#abaa%aba!$	#aTU
[$, !, U, T, a, #]	#abaa%aba!$	Terminal
[$, !, U, T, a]	abaa%aba!$	Terminal
[$, !, U, T]	baa%aba!$	baT
[$, !, U, T, a, b]	baa%aba!$	Terminal
[$, !, U, T, a]	aa%aba!$	Terminal
[$, !, U, T]	a%aba!$	aS
[$, !, U, S, a]	a%aba!$	Terminal
[$, !, U, S]	%aba!$	%aT
[$, !, U, T, a, %]	%aba!$	Terminal
[$, !, U, T, a]	aba!$	Terminal
[$, !, U, T]	ba!$	baT
[$, !, U, T, a, b]	ba!$	Terminal
[$, !, U, T, a]	a!$	Terminal
[$, !, U, T]	!$	ε
[$, !, U]	!$	ε
[$, !]	!$	Terminal
[$]	$	Terminal
	result:	Accept