编译原理实习三

——LR(1)语法分析

班级：
姓名：AnDJ
学号：

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、DFA、简化 DFA、解析表、解析过程和结果。
7） 空字这里默认为“ε”，键盘无法输入可以复制粘贴提示中给出的该字符。

四、 补充说明

1.’ε’ 字符已经不是在 ASCII 字符,UTF-8 字符集下可以正常输入输出

2.如果继续使用’ε’为空字符,建议使用 Linux 系统编译执行该.java 文件,Windows 系统下控制台我始终设置不好控制台输入输出该字符。

3.如果不想使用’ε’为空字符,可以修改源代码第 484 行:

```
Grammar grammar = new Grammar(productions,startNonTerminal,'ε');
```

将’ε’修改为合适的字符作为替换。注意设置的空字字符代替字符将不能出现在产生式的字母表内.

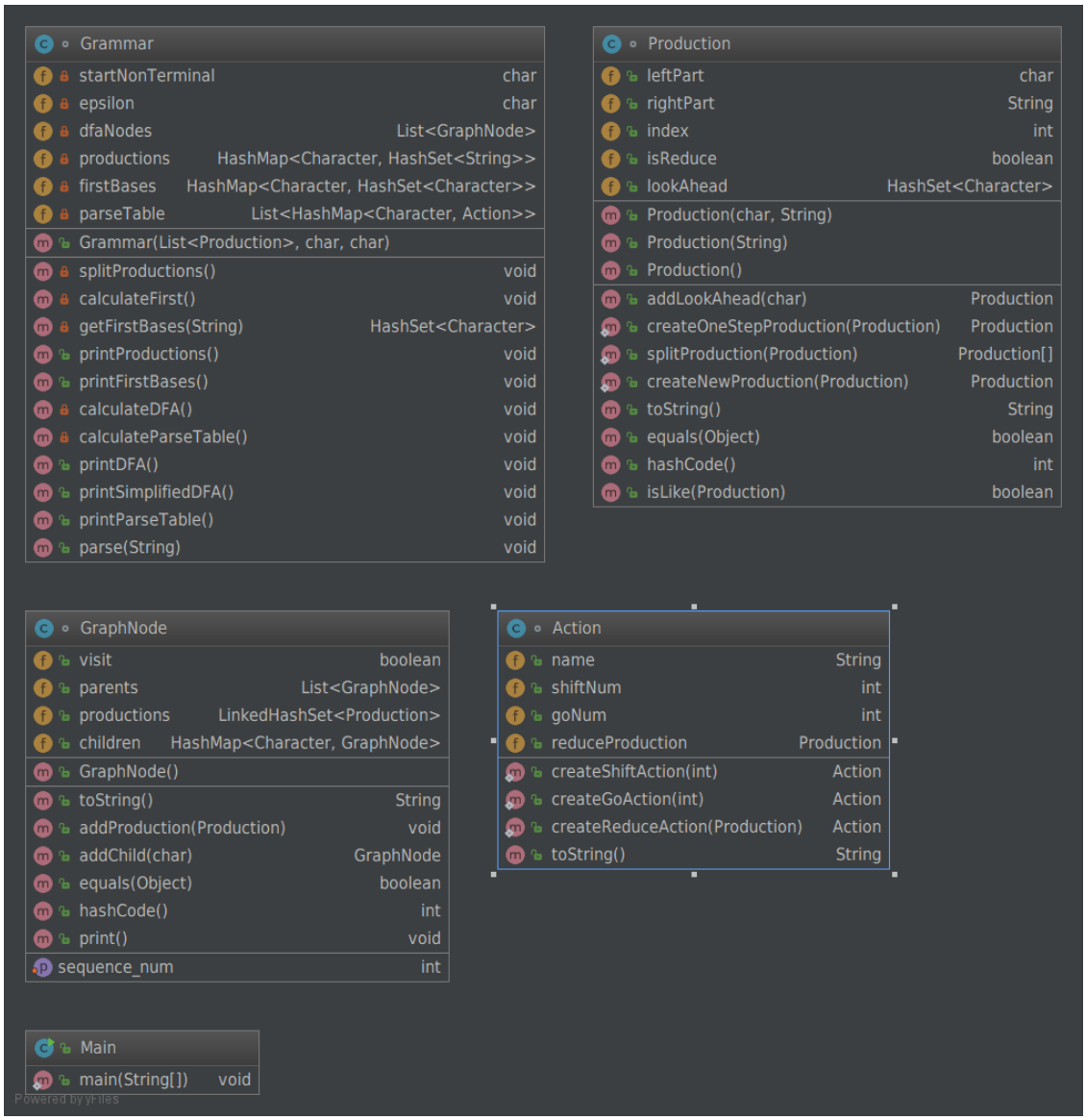
五、 运行约束

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‘-’‘>’。

六、 代码文件类图



七、 实现过程

1. 产生式的解析和存储

产生式的数据结构实现为 Production 类，根据 LR(1)语法解析过程所需要，产生式的结构需包含：

```
C-> A,C    [$]
    ^
```

即左部、右部、展望字符集、Index。关于类 Production 的属性请查看 UML 类图，leftPart（左部）、rightPart（右部）、lookAhead（展望字符集，Set 类型）、index，即字段 isReduce，isReduce 字段用于标志该产生式是否 reduce。

该类具体封装的函数有：

1） createOneStepProduction(Production)，用于构造一个参数 production 右部向右移动 index 的新的 production。如原产生式为 A->.BC，产生新的为 A->B.C。

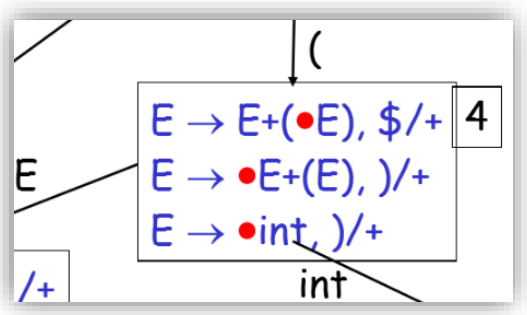
2） addLookAhead(char ahead)，添加一个展望字符。

3） hashCode()与 equals()，这里让所有产生式的 hashCode 都为 0，比较两个产生式的异同通过 equals 函数，将该类的所有字段全部参与比较。

4） isLike(Production)函数，该函数用于判断两个产生式除展望字符外是否相同，如 A->B.C [a] 和 A->B.C [b]，这样帮助后面的状态简化，即将 like 的两个产生式进行合并，如上述和合并为 A->B.C [a,b]。

2. LR(1)DFA 图数据结构

LR(1)的 DFA 节点结构如图所示：



该 DFA 节点需要包含所有的父节点(parents)、子节点及到达子节点的条件(children)、内含的所有产生式(productions)、节点序号(sequence_num)，这些字段在类 GraphNode 中均实现，GraphNode 类中实现了帮助遍历的字段 visit。该类实现的函数有：

1） addProduction(Production)用于添加产生式。

2） addChild(char)用于添加孩子节点。如果该孩子节点已存在（该节点通过指定条件已可以访问到存在的子节点）则不添加子节点，保证 DFA 的性质。

3） equals()函数判断两个 DFA 节点是否等价，这个用于帮助后面语法分析的过程中不会无限制产生图节点，一些等价的节点是可以合并的。判断两个节点是否等价通过判断它们所含的所有产生式是否相等得出。

3. LR(1)三种动作数据结构

LR(1)算法计算解析表时，针对特定的环境有三种动作可以执行：

1） Shift

2） Go

3） Reduce

类 Action 给出了针对三种动作的封装。这里没有实现太复杂的方式（其实使用枚举会更好一些），单纯通过字符串匹配区别动作，“shift”、“go”、“reduce”分别对应三种 Action，name 字段进行存储。此外，针对 shift 和 go 动作，还包含目的节点的 sequence_num；针对 reduce 动作，还包含规约的 production。（Action 似乎用多态实现更好，时间原因实现从简了）。

类中包含三种静态函数，用于创建对应的动作。

4. Grammar：初始化

Grammar 类实现了针对 LR(1)算法的封装。该类初始化必须保存一些用户输入的字段，如 epsilon(用来表示 epsilon 的字符)、startNonTerminal(起始非终结符)、productions(该语法所有产生式)等。此外还包含中间计算产物字段如 firstBases 和 parseTable、dfaNodes，后面详述。

5. Grammar：求解 First 基

这个过程和 LL(1)算法求解的过程一致，这里不再给出算法具体步骤，求解实现函数为 calculateFirst 函数，计算结果保存至 firstBases 字段，该属性为 HashMap<Character, HashSet<Character>>类型，保存单个非终结符的所有 first 基集合。

6. Grammar：计算 DFA 与简化

这个属于 LR(1)算法的核心实现，实现过程为 calculateDFA 函数。该函数的输入是语法规则，输出为 DFA，且该 DFA 保存至 dfaNodes 字段，该字段是 GraphNode 链表。

1） 辅助函数 getFirstBases(String input)，用于寻找该字符串的 first 基。比如有语法规则 S->AcDb 且 A->ab，如果 DFA 节点有 S->.AcDb 且 A->.ab 时，需要对该节点的第二条产生式重新计算展望字符，即计算 getFirstBases(cDb)，其实就是计算 A 在第一条产生式后面紧跟的 follow。算法：

```
For every character in input
    If 是 epsilon
        跳过
    If 是终结符
        加入返回的展望字符集合
        结束循环
    If  是非终结符
        将其非 epsilon 的所有 first 基加入
        If 其 first 基中不包含 epsilon
            跳出循环
```

2） 计算 DFA

计算所有 DFA 节点，我使用了队列 queue 进行辅助计算，类似于广度优先遍历图的算法，计算当前节点后，将当前节点可以到达的节点放入队列中，使得队列中始终保存的是未计算完全的 DFA 节点。

算法如下：

a.构造新起始非终结符指向起始非终结符(S->E)

b.12 以此产生式构造节点并放入队列 queue 中

c.

```
While(queue not empty)
    currentNode = queue.pop
    for every production in currentNode.productions:
        if index 到右部结尾
            continue
        if handle 是终结符
            continue
        if handle 是非终结符
            将其为左部的产生式全部加入 currentNode.productions
            重新计算其展望字符
    for every production in currentNode.productions:
        计算其走一步后的产生式
        依据此产生式构造新节点（同样条件的合并）
```

```
For node in dfaNodes:
    If node 和 currentNode 等效(like)
        Current.parents all point to node
If not find like(等效) node
    Queue.pushAll(current.children)
    dfaNodes.add(currentNode)
```

该算法的实现为 calculateDFA 函数。该函数执行完毕，dfaNodes 字段包含所有的 DFA 中的节点，且这些节点构成了完整的 DFA 图（包含序号）。

7. Grammar：计算 LR(1)解析表

该函数的实现体为 calculateParseTable()函数，过程为：

```
For every node in dfaNodes:
    For every production in dfaNodes.productions:
        If production need reduce
            Fill lookaheads' location by 'reduce'
    For every child in dfaNode.children:
        If condition(转换条件) is terminal:
            Fill condition's location by 'shift'
        Else
            Fill condition's location by 'go'
```

8. Grammar：解析字符串过程函数

该算法实现函数为 Grammar.parse(String input)函数。

这里的主要算法思想就是根据 parseTable 进行计算，reduce 时弹状态栈和符号栈，然后记得 Go Action 的执行即可。这里解释从略了，解析过程可以参考样例。

八、 测试

示例 1：

起始非终结符：E

语法规则：*E->E+(E)/i*

待匹配的字符串：i+(i)+(i)

测试结果：查看附件 *example1.pdf*

示例 2：

起始非终结符：S

语法规则：*S->[B*
A->i/[B
B->]/C
C->A]/A,C

待匹配的字符串：[i,i]

测试结果：查看附件 *example2.pdf*

九、 输出说明

1. 关于 simplified DFA,示例：

```
node 1:
    Reduce  S->E on [$]
    via + ==> 3
```

意即 node1 在$执行 S->E 的规约，通过+执行 shift 到 node3

2. 关于 parse table，示例：

   0 {E=go 1, i=shift 2}

   意即：node0 遇到 E 执行 go 1 动作，遇到 i 执行 shift 2 动作。