编译原理实习一

——词法分析

班级：
姓名：AnDJ
学号：

一、 提交文件

Lexer.java

二、 编译及运行环境

JDK 1.8.0

三、 执行方式

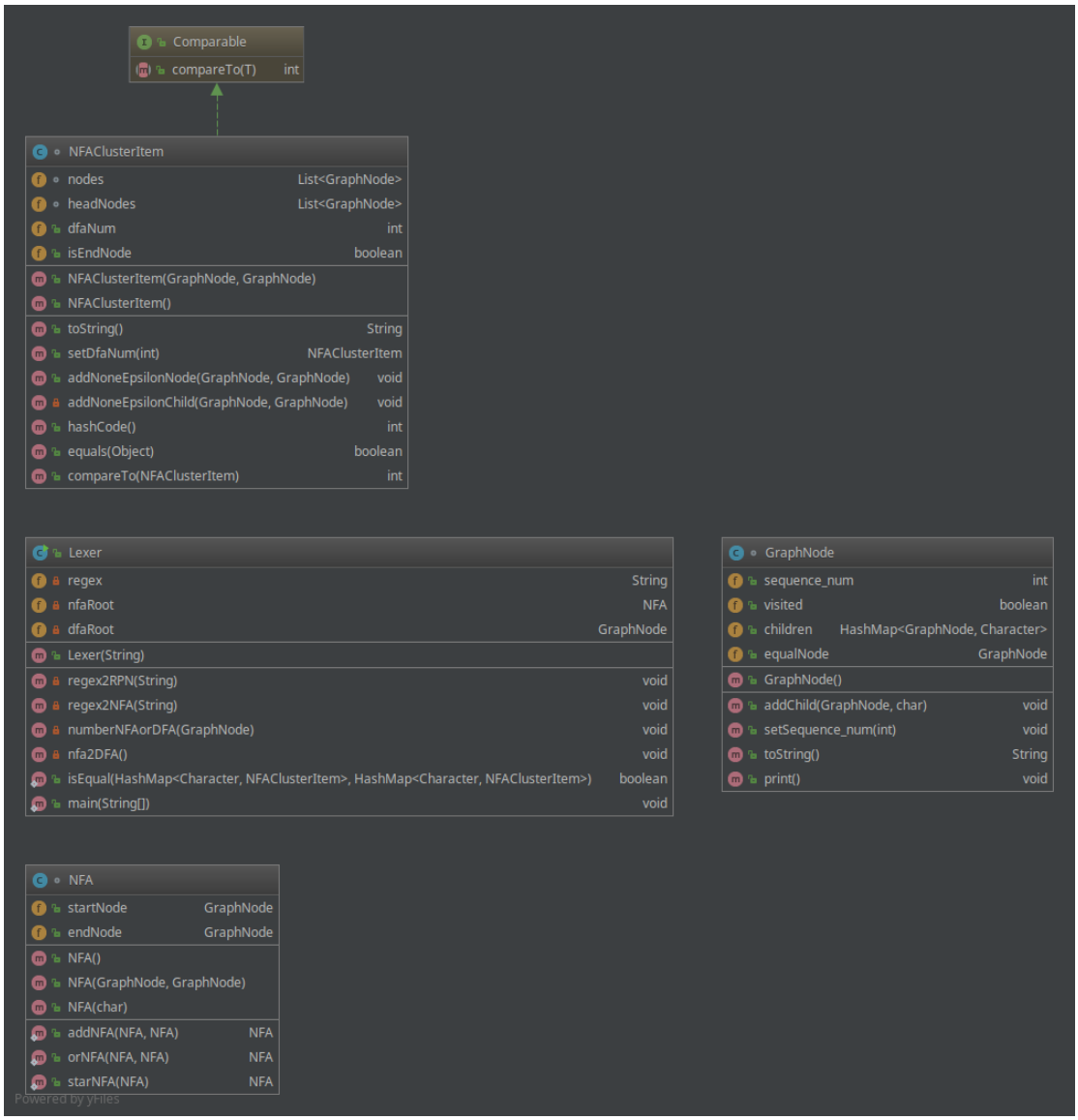
1） javac Lexer.java
2） java Lexer
3） 输入正则表达式
4） 命令行回车得出运行结果

四、 运行约束

正则表达式约束如下：

1） 正则表达式包含 ASCII 所有可见字符，“+”、“|”、“*”、“(”、“)”除外。
2） “+”代表字串连接，如 A+B 代表字符串“AB”。
3） “|”代表或连接，如 A|B 代表集合{“A”，“B”}中的任一字符串。
4） “*”代表任意匹配，如 A* 代表集合{“”、“A”、“AA”、“AAA”.....}中任一字符串。
5） “(”和“)”包含的部分为一个整体，用于提高计算优先级。

五、 代码文件类图



六、 实现过程

1. 正则表达式（regular expression）转逆波兰式（RPN）

   正则表达式转 RPN 的目的是为了消除括号，方便后面的 Regex 向 NFA 转换。在正则表达式中，运算符号“+”、“|”、“*”的优先级为“*”>“+”>“|”。因此，采用一般的中缀转后缀表达式的方式转换即可，即字符栈和运算符栈协作。策略如下：

   1) 读取字符为非运算符，压入字符栈中。
   2) 读取字符为运算符“+”、“|”、“*”，比较其与运算符栈顶元素的优先级。如果栈顶元素的优先级大则运算符弹栈并将弹出符号压入字符栈，然后继续按照这种策略继续检查至直到栈空或栈顶元素优先级小于该读取字符；如果栈顶元素的优先级小于该读取字符，则将其压入运算符栈。
   3) 读取字符为“(”，将其压入运算符栈中。
   4) 读取字符为“)”，则循环弹出运算符栈并将弹出元素压入字符栈中，直到运算符栈顶元素为“(”并将其弹出。

   这样实现了正则表达式 Regex 向 RPN 转换的过程。对应实现为 Lexer.regex2RPN(String)函数。

2. NFA 和 DFA 的存储。

   NFA 和 DFA 是基于图的，因此需要构造合适的图节点来构建图。GraphNode 类实现了针对状态机图的存储，包含如下 4 个字段：

   sequence_num（特定图中的次序）、

   visited（用于遍历图）、

   children（HashMap<GraphNode,Character>类型，由于构造 NFA 时节点反而是唯一的（针对 NFA 的合并过程，一个节点到另一个节点的直接转换过程是唯一的），因此存储格式为 Key=GraphNode，Value=char 的哈希表）、

   equalNode（该节点的等效节点，用于 DFA 的简化过程）。

   包含如下函数：

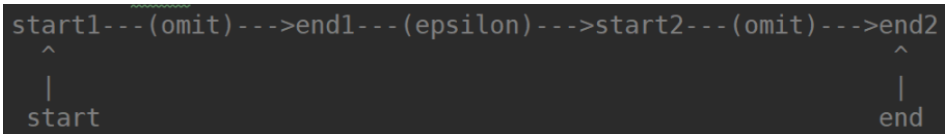
   adddChild 添加子节点，

   setSquence_num 设置针对于指定图的节点序号等。

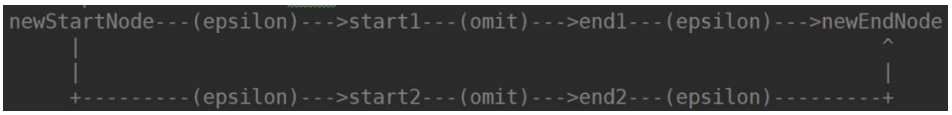
3. NFA 的单独存放。

   NFA 与 DFA 略有不同，因为需要针对 NFA 有额外的构造和合并操作，因此封装了类 NFA。该类实现了构建不同 NFA 的构造函数，注释部分给出。同时，给出了针对 NFA 操作封装的函数，代码注释部分同样有详解。下面给出图示：

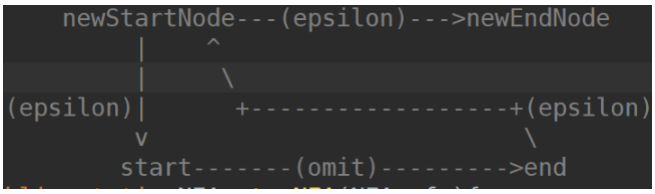
   1） add 操作



   2） or 操作



   3） star 操作



   详细参数解释请查看类 NFA 的代码注释部分。

   这些函数用来针对单个或两个 NFA 进行对应操作，帮助实现正则表达式到 NFA 的转换过程。

   NFA 包含 StartNode 和 endNode 字段，用来存储该 NFA 的起始节点和终止节点。

4. 正则表达式转换为 NFA。

   有了上面函数的封装，这里的操作变得很容易。使用栈管理 NFA 的节点。

   1） 当读取到“+”时，将栈顶两个 NFA 进行 add 操作，并将结果压入栈。

   2） 当读取到“*”时，将栈顶 NFA 进行 star 操作。

   3） 当读取到“|”时，将栈顶两个 NFA 进行 or 操作，并将结果压入栈。

   4） 当读取到其他字符时，构造 start—(condition)-→endNFA 并压入栈。

   相应实现过程为 Lexer 类的 regex2NFA 函数。这样构建出来的是一个没有编号(non-numbered)的 NFA。下面的过程解决编号的问题。

5. 针对 NFA 进行编号。

   这个过程也是非常简单的,针对从首节点为NFA.startNode的图进行广度优先遍历，使用队列进行辅助遍历，同时借用 visited 字段帮助，同时利用指示变量进行累加计数，帮助第一次遍历图时进行节点编号。第二遍遍历的过程可以将 NFA 的节点输出，输出包含该节点 ID、该节点到每个子节点的 condition 和对应子节点的 ID。这里的算法即广度优先遍历。相应功能实现为 Lexer.numberNFAorDFA 函数。该函数后期可复用于 DFA 的编号及输出。

6. 构建 DFA—>DFA 表 1

   该表的形状如下：

	a	b	c
0	121	3213	321
1,2,3	21	321	432
2,3,4	21	321	43
…	…	…	…

   这里我将如(1,2,3)这样的单位抽象为 NFACluster，即 NFA 状态簇，并且该簇是与 non-simplified DFA status（未简化 DFA 状态）是一一对应的。

   1） NFACluster 封装

   字段 1：

   NFACluster 需要包含所有 cluster 内的节点，这里使用 List<GraphNode>来存储，存储字段未 nodes。

   字段 2：

   因为后面要比较进行 cluster 之间的比较和相等性判断的关系，且簇内点的构建方式是,将某些状态及该状态的 epsilon 条件能到达的所有状态装入 cluster 内，因此比较两个 cluster 是否是同一状态，只需要比较前者即可。这里使用 headNode:List<GraphNode>存储前者。

   字段 3：

   Cluster 与 non-simplified DFA 状态一一对应，因此有唯一的 dfa 编号，及 dfaNum。

   字段 4：

   该 cluster 对应的 DFA 状态需要判断是否为终态，方便后面进行简化。因此需要 boolean 类型的 isEndNode 字段。

   函数：

   添加某 NFA 状态及该状态经 epsilon 能达到的其他所有状态，这个我使用了递归实现，调用函数为 addNoneEpsilonNode，辅助递归函数为 addNoneEpsilonChild 函数。

   函数 2：

   Equals 函数。用于比较两个 cluster 是否为同一 cluster。方法即上面所说的比较 headNode。

   函数 3：

   compareTo 函数，implement 接口 comparable 接口的该函数，实现了针对两个 cluster 对应的 DFA 状态进行比较的方法，通过比较两个 cluster 的 dfaNum，这个是为了方便下面进行针对 NFA->DFA 表构建采取的一个“迫不得已”的办法。

   2） 构建 NFA->DFA 表

基于上面的分析，该表的格式如下：

	condition1	condition2
clusterItem1(1*,2,3*,5)	clusterItem2(3*,5)	clusterItem3(4*,2)
clusterItem2(3*,5)	clusterItem2(3*,5)	clusterItem3(4*,2)

每一行使用(NFAClusterItem,HashMap(condition,NFAClusterItem))的键值对存储，单行每一列使用(condition,NFAClusterItem)的键值对进行存储，该键值对存在于单行的 HashMap 中。所有行的键值对也使用 hashMap 存储，变量名为 nfaTable(存在于 lexer.nfa2DFA( )函数中)。

7. 构建 NFA->DFA 表 2

对应函数实现为 lexer.nfa2DFA 函数。

1） 过程起始条件：将 NFA 0 状态及其通过 epsilon 到达的所有状态构建 cluster 并放入表（nfaTable）中一行，作为起始行。

2） 将 nfaTable（该变量即为存储总表的 hashmap）的键值进行按 dfaNum 升序排序，这个过程是一个迫不得已的过程，因为 Java 底层实现 HashMap 的方式是红黑树，因此每当插入键值对时，之前存储的“线性”顺序，即表中从上而下的顺序就会被打乱，因此需要根据 dfaNum 进行排序，保证表内每一行都不会被遗漏。

3） 遍历目前需要检查的单行 clusterItem 包含的所有状态，检查这些状态的所有达到条件非 epsilon 的子状态，即针对上图表第一行，就是检查 1,2,3,5 各自的所有到达条件非 epsilon 子状态。

达到以上条件的子状态 node 即可将<condition，clusterItem(node)>放入该行的 hashMap 中，当然这个键值对的放入是需要策略的：

node 和 node 的非 epsilon 到达的子状态都要放进去，且如果该行的 hashMap 存在该 condition 键值的键值对，那么把新构造的键值对并入进去，而不是替换进去。

例如该行的 key 值 currentCluster(2,3,4)对应的 hashMap 在计算’2’时存储了<condition1,(1,2,3)>的键值对，而现在 currentCluster 计算’3’时产生了新的<condition1,(4,5,6)>键值对，那么并入后改 currentCluster(2,3,4)对应的键值对为<condition1,(1,2,3,4,5,6)>。

4） 针对该单行添加的键值对<condition1,cluster1>,<condition2,cluster2>,…进行检查，如果总表中某一行的 cluster 与键值对中的 clusterX 相同（即包含的 nodes 一致），那么总表将不会产生新的行；否则总表加入新的 clusterX 建立新的行。

5） goto 执行 2），直到遍历完表的所有行。需要知道的是，该表的行数在过程中是动态增加的，但是不是无限制增加的，因为不会无限制地产生新的 cluster。

6） 这个过程中注意给 cluster 进行编号的问题，只要总表产生新的行，那么就会给新行编号。

以上过程结束给出的表中，每一行的 key 代表的 DFA 状态即未经简化的 DFA 中的状态，且状态之间的转换关系已经给出。

8. 简化 DFA

这里我没有采用课上讲到的树的形式进行简化，我的策略是直接遍历。首先创建判断两个 clusterItem 是否等效的函数：isEqual，通过检查两个行的 hashmap 来进行比较，如果这一行的 hashMap 完全一致的话，那么这两行等效，这两行的 key 值（即对应的 DFA 状态）也等效，即可化简。第一遍遍历找到每个节点的等效节点并存储下来，第二遍使用等效节点代替该节点进行构造 DFA，该 DFA 存储至 dfaRoot 变量中。

七、 测试

1. 输入：1+(0*+1)*|0+(1*+0)*
   输出：

```
1.---------------regex to RPN--------------------

 10*1+*+01*0+*+|

2.-----------------NFA--------------------------

0: ( ,1) ( ,2)
1: (0,3)
2: (1,4)
3: ( ,5)
4: ( ,6)
5: ( ,7) ( ,8)
6: ( ,9) ( ,10)
7: ( ,11)
8: ( ,12) ( ,13)
9: ( ,14) ( ,15)
10: ( ,11)
11:
12: ( ,16)
13: (1,17)
14: ( ,18)
15: (0,19)
16: (0,20)
17: ( ,8)
18: (1,21)
19: ( ,9)
20: ( ,5)
21: ( ,6)

3.------------the dfa to nfa table-------------

0: [0, 1, 2] false
{0=[3, 5, 7, 11, 8, 12, 16, 13], 1=[4, 6, 9, 14, 18, 15, 10, 11]}
```

1: [3, 5, 7, 11, 8, 12, 16, 13] true
{0=[20, 5, 7, 11, 8, 12, 16, 13], 1=[17, 8, 12, 16, 13]}

2: [4, 6, 9, 14, 18, 15, 10, 11] true
{0=[19, 9, 14, 18, 15], 1=[21, 6, 9, 14, 18, 15, 10, 11]}

3: [20, 5, 7, 11, 8, 12, 16, 13] true
{0=[20, 5, 7, 11, 8, 12, 16, 13], 1=[17, 8, 12, 16, 13]}

4: [17, 8, 12, 16, 13] false
{0=[20, 5, 7, 11, 8, 12, 16, 13], 1=[17, 8, 12, 16, 13]}

5: [19, 9, 14, 18, 15] false
{0=[19, 9, 14, 18, 15], 1=[21, 6, 9, 14, 18, 15, 10, 11]}

6: [21, 6, 9, 14, 18, 15, 10, 11] true
{0=[19, 9, 14, 18, 15], 1=[21, 6, 9, 14, 18, 15, 10, 11]}

4.------------------DFA-------------------------

0: (1,1) (0,2)
1: (0,3) (1,1)
2: (0,2) (1,4)
3: (0,3) (1,1)
4: (0,2) (1,4)

2. 输入：a+(a|b)*+a

1.---------------regex to RPN-------------------

aab|*a++

2.------------------NFA-------------------------

0: (a,1)
1: (,2)
2: (,3) (,4)
3: (,5)
4: (,6) (,7)
5: (a,8)
6: (a,9)
7: (b,10)
8:
9: (,11)
10: (,11)

11: (,2)

3.------------the dfa to nfa table-------------

0: [0] false
{a=[1, 2, 3, 5, 4, 6, 7]}

1: [1, 2, 3, 5, 4, 6, 7] false
{a=[8, 9, 11, 2, 3, 5, 4, 6, 7], b=[10, 11, 2, 3, 5, 4, 6, 7]}

2: [8, 9, 11, 2, 3, 5, 4, 6, 7] true
{a=[8, 9, 11, 2, 3, 5, 4, 6, 7], b=[10, 11, 2, 3, 5, 4, 6, 7]}

3: [10, 11, 2, 3, 5, 4, 6, 7] false
{a=[8, 9, 11, 2, 3, 5, 4, 6, 7], b=[10, 11, 2, 3, 5, 4, 6, 7]}

4.-----------------DFA-------------------------

0: (a,1)
1: (a,2) (b,1)
2: (a,2) (b,1)

3. 输入：(a|b)*+a+(a|b)+(a|b)
输出：

1.--------------regex to RPN-------------------

ab|*aab|ab|+++

2.-----------------NFA-------------------------

0: (,1) (,2)
1: (,3) (,4)
2: (,5)
3: (a,6)
4: (b,7)
5: (a,8)
6: (,9)
7: (,9)
8: (,10)
9: (,0)
10: (,11) (,12)
11: (a,13)
12: (b,14)
13: (,15)

14: (,15)
15: (,16)
16: (,17) (,18)
17: (a,19)
18: (b,20)
19: (,21)
20: (,21)
21:

3.------------the dfa to nfa table-------------

0: [0, 1, 3, 4, 2, 5] false
{a=[6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12], b=[7, 9, 0, 1, 3, 4, 2, 5]}

1: [6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12] false
{a=[6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12, 13, 15, 16, 17, 18], b=[7, 9, 0, 1, 3, 4, 2, 5, 14, 15, 16, 17, 18]}

2: [7, 9, 0, 1, 3, 4, 2, 5] false
{a=[6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12], b=[7, 9, 0, 1, 3, 4, 2, 5]}

3: [6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12, 13, 15, 16, 17, 18] false
{a=[6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12, 13, 15, 16, 17, 18, 19, 21], b=[7, 9, 0, 1, 3, 4, 2, 5, 14, 15, 16, 17, 18, 20, 21]}

4: [7, 9, 0, 1, 3, 4, 2, 5, 14, 15, 16, 17, 18] false
{a=[6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12, 19, 21], b=[7, 9, 0, 1, 3, 4, 2, 5, 20, 21]}

5: [6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12, 13, 15, 16, 17, 18, 19, 21] true
{a=[6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12, 13, 15, 16, 17, 18, 19, 21], b=[7, 9, 0, 1, 3, 4, 2, 5, 14, 15, 16, 17, 18, 20, 21]}

6: [7, 9, 0, 1, 3, 4, 2, 5, 14, 15, 16, 17, 18, 20, 21] true
{a=[6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12, 19, 21], b=[7, 9, 0, 1, 3, 4, 2, 5, 20, 21]}

7: [6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12, 19, 21] true
{a=[6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12, 13, 15, 16, 17, 18], b=[7, 9, 0, 1, 3, 4, 2, 5, 14, 15, 16, 17, 18]}

8: [7, 9, 0, 1, 3, 4, 2, 5, 20, 21] true
{a=[6, 9, 0, 1, 3, 4, 2, 5, 8, 10, 11, 12], b=[7, 9, 0, 1, 3, 4, 2, 5]}

4.----------DFA----------

0: (b,0) (a,1)
1: (b,2) (a,3)
2: (a,4) (b,5)
3: (a,6) (b,7)
4: (b,2) (a,3)
5: (b,0) (a,1)
6: (a,6) (b,7)
7: (a,4) (b,5)

4. 输入：a*+b+a*+b+a*+b+a*
输出：

1.----------regex to RPN----------

a*ba*ba*ba*++++++

2.----------NFA----------

0: (,1) (,2)
1: (a,3)
2: (,4)
3: (,0)
4: (b,5)
5: (,6)
6: (,7) (,8)
7: (,9)
8: (a,10)
9: (b,11)
10: (,6)
11: (,12)
12: (,13) (,14)
13: (,15)
14: (a,16)
15: (b,17)
16: (,12)
17: (,18)
18: (,19) (,20)
19:
20: (a,21)

```
21: ( ,18)

3.------------the dfa to nfa table-------------

0: [0, 1, 2, 4] false
{a=[3, 0, 1, 2, 4], b=[5, 6, 7, 9, 8]}

1: [3, 0, 1, 2, 4] false
{a=[3, 0, 1, 2, 4], b=[5, 6, 7, 9, 8]}

2: [5, 6, 7, 9, 8] false
{a=[10, 6, 7, 9, 8], b=[11, 12, 13, 15, 14]}

3: [10, 6, 7, 9, 8] false
{a=[10, 6, 7, 9, 8], b=[11, 12, 13, 15, 14]}

4: [11, 12, 13, 15, 14] false
{a=[16, 12, 13, 15, 14], b=[17, 18, 19, 20]}

5: [16, 12, 13, 15, 14] false
{a=[16, 12, 13, 15, 14], b=[17, 18, 19, 20]}

6: [17, 18, 19, 20] true
{a=[21, 18, 19, 20]}

7: [21, 18, 19, 20] true
{a=[21, 18, 19, 20]}


4.------------------DFA-------------------------

0: (b,1) (a,0)
1: (a,1) (b,2)
2: (b,3) (a,2)
3: (a,3)
```