编译原理Lab1实验报告

余旻晨 141250177

1. 实验介绍

本次实验旨在完成编译中的词法分析部分，写一个自己的lex。在这次实验中，我完成了lex的核心部分，即用程序实现了从正则表达式到NFA，再从NFA到DFA，最后进行DFA的优化。然后再自己定义正则表达式，得到优化后的DFA后编写词法分析程序，分析测试程序得到token。

2. 方法说明

2.1 lex实现部分

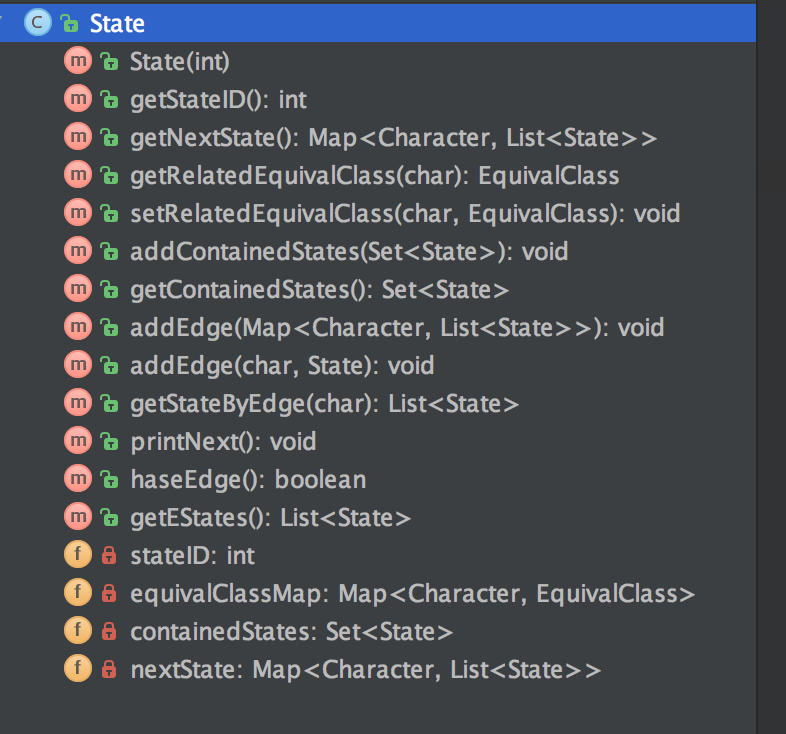
lex的实现在代码中放置于analyzer包和util包中。analyzer包中的4个类REHandler、NFAHandler、DFAHandler、DFAOptimization分别对应lex实现过程中的四个步骤。每个类中都写有main方法用于输出lex进行到当前步骤时的结果。检查时可以用于测试，结果附在测试展示中。

2.1.1正则表达式部分的实现

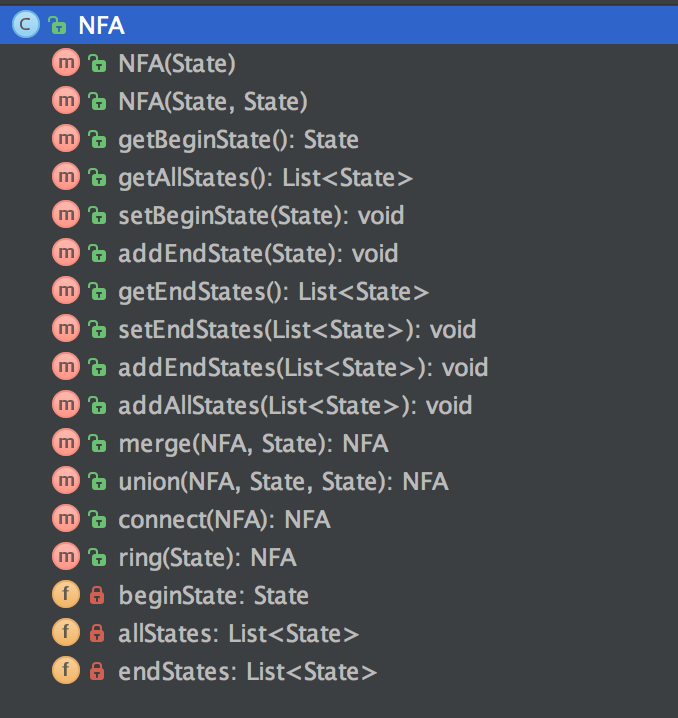
正则表达式的处理为：先从doc/RE.txt文件中一行行读取正则表达式,加入存放初始String list中，然后对list中的每一个正则表达式分别进行加“·”和中缀转后缀的计算。这部分参见java类REHandler的addDot方法和infix2suffix方法。

2.1.2 RE到NFA的转换

在util包先定义State的数据结构，每个节点对应一个id，同时包含相连节点信息nextState（containedStates用于NFA到DFA的转换，equivalenceClassMap用于DFA的优化）。



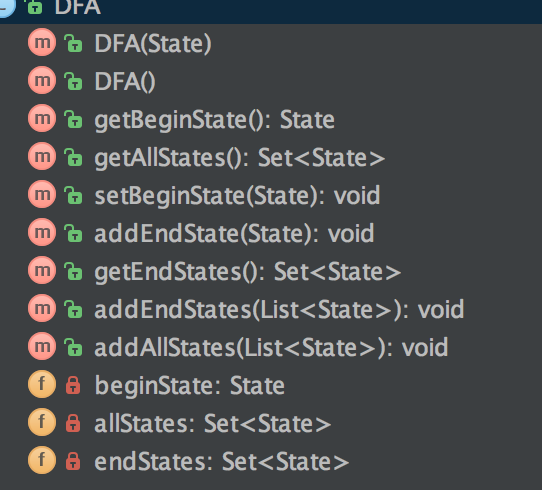
定义NFA的数据结构，NFA中记录起始点以及所有的终止点，记录所有包含的节点。同时，便于NFA之间的合并，创建union（|），connect（·），ring（\*）方法，创建merge方法合并不同RE生成的子NFA



正则表达式转化为NFA的实验思路基于汤普森算法，先根据后缀表达式形式的正则表达式建立单个字母的NFA，再根据运算符“|”，“·”，“\*”将NFA串联成单个正则表达式对应的NFA。最后将所有的NFAmerge成一个NFA。实现代码参见java类NFAHandler。

2.1.3 NFA到DFA的转换

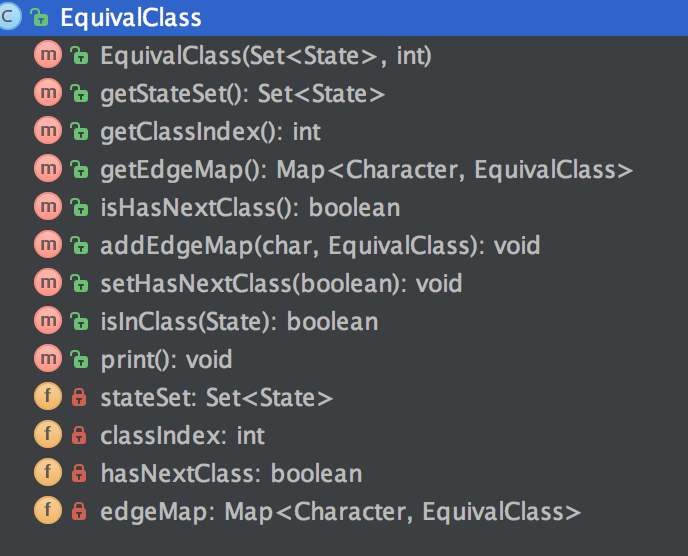
定义DFA的数据结构，DFA中记录起始点以及所有的终止点，记录所有包含的节点。



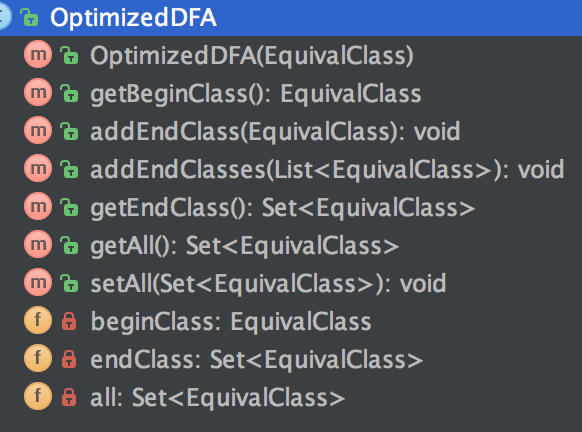
NFA到DFA的转化方法为epsilon闭包法。即先从起始点开始，找到对应的epsilon闭包。然后以该epsilon闭包为单位分别找到各边对应的节点集合的epsilon闭包，以此类推对每个闭包进行操作，注意要识别已经重复的epsilon闭包。最后生成对应的DFA。在代码中，把属于同一epsilon闭包的节点放置在新生成节点的containedState中便于比较重复生成的epsilon闭包。用栈结构存储新节点，并通过BFS对节点的边进行遍历，确认DFA中各节点的边关系。具体实现参加DFAHandler，核心方法为NFA2DFA，setContainOther和setEquals用于判断set包含、相等关系，closureRepeated用于判断闭包是否重复，getEpsilonClosure通过递归计算epsilon闭包。

2.1.4 DFA的优化

定义了EquivalClass和OptimizedDFA数据结构。EquivalClass规定了等价类的属性和方法，有标识id，包含原DFA中的节点集合，是否进行了分割hasNextClass以及相应边对应的等价类。



OptimizedDFA定义了以等价类为节点的DFA，属性和方法同DFA类似。



将DFA转化为Optmized DFA的方法为，先根据原DFA中的终止态节点将所有节点划分为两个等价类，并将是否进行分割属性置为false，加入classList等价类列表中。然后对classList进行遍历，对于每一个等价类，查找对于每一条边（a或b），是否等价类中存在节点对应的边不在同一个等价类中，判断是否对这一等价类进行划分。若需要划分，则把该等价类的hasNextClass置为true，并基于分类结果新建等价类，将新建的等价类加入classList中，继续遍历。分类的实现见classify方法。最后需要回头看看是否应为后期的等价类划分导致的开始划分的等价类不准确，来重新划分等价类。故把hasNextClass为false的等价类重新放入一个列表lastListTmp，记录这个列表的size，对这个列表重复上述步骤的计算，直到列表的size不发生变化。最后得到的结果即为成功划分后的等价类。然后基于这个等价类list生成OptimizedDFA。参见核心方法optimization。

2.2 词法分析程序部分

词法分析部分的代码放置在lex包以及util包中，main是词法分析程序的启动程序。

2.2.1词法规范定义

在本次实验中，我将词法分析的词定义了五大类：一般符号（“[”, “(”等)；运算符号（加减乘除，比较等运算）；数字（整数或浮点数）；ID（定义的变量名）；关键字（设定的关键字）。具体参见util包下的type类。第一类共有8个枚举类型，第二类有20个枚举类型；后三类各自成一个枚举类。关键字列表可参见Utility类的keyWords列表。

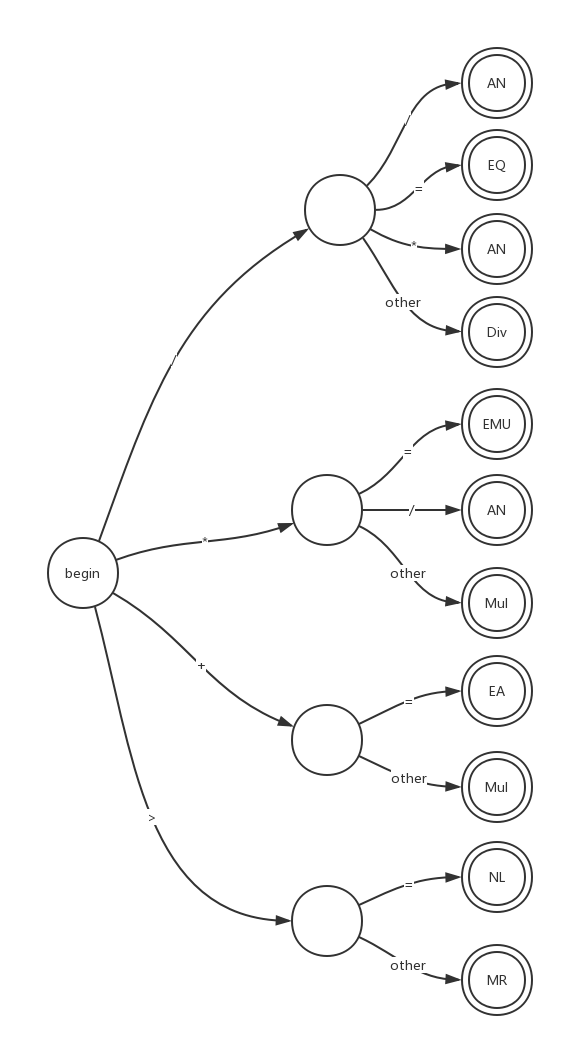
a. 对于一般符号，正则表达式对应的结果仅有1种，故在程序中直接对读取的char通过switch判断，不做专门分析。在Tpye中的对应表格如下：

|  |  |
| --- | --- |
| Type枚举 | 符号 |
| LM | } |
| RM | { |
| LB | [ |
| RB | ] |
| LC | ( |
| RC | ) |
| AN | // |
| AN | /\* |
| Dot | . |

b. 对应运算符号，在Tpye中的对应表格如下：

|  |  |
| --- | --- |
| Type枚举 | 符号 |
| Add | + |
| EA | += |
| Minus | - |
| EMI | -= |
| Mul | \* |
| EMU | \*= |
| Div | / |
| ED | /= |
| Assign | = |
| MR | > |
| NM | <= |
| LS | < |
| NL | >= |
| EQ | == |
| UE | != |
| OR | | |
| DOR | || |
| AND | & |
| DAND | && |
| NOT | ! |

对应的DFA（典型的部分，重复类似的不画在内）为：



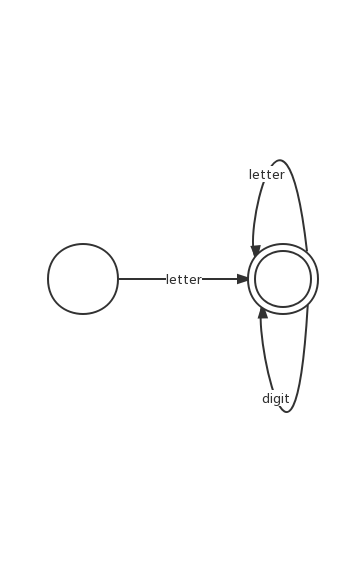
c. 对于ID，对应的正则表达式为：

digit -> ( 0|1|2|3|4|5|6|7|8|9)

letter -> [a-zA-Z]

id -> letter (letter | digit)\*

对应的DFA为：

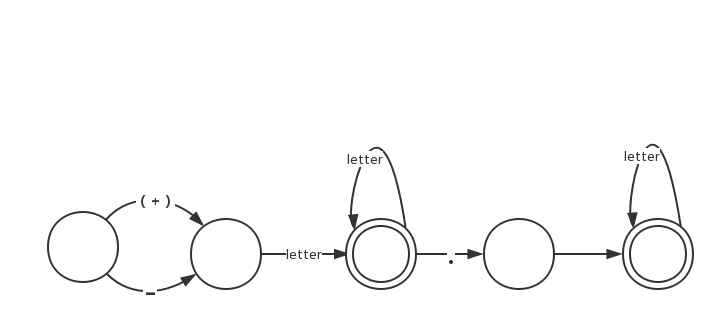


d. 对于数字，正则表达式为：(匹配整数和浮点数)

digit -> ( 0|1|2|3|4|5|6|7|8|9)

num -> digit digit\*( digit\* |. digit\*)

对应的DFA为(加号+可省略)



e. 对于关键字，枚举类型与关键字列表一一对应。

2.2.2 token定义

token定义为一个三元组，(value, type, line)，对应为该词的值，该词的类型以及该词出现的行数。参见Token类的描述。

2.2.3 词法分析程序

词法分析程序中的核心方法是handle和scan，handle的任务是从doc/input.txt中读取代码，然后以一行做为单位放在scan方法中分析，得到一个token的list，最后将token的list返回。Main中的saveTokens方法得到handle传来的token list，将其输出到控制台和制定的doc/output.txt文件中。

在scan方法中，首先对符号进行switch判断，依据DFA对可能出现的情况进行覆盖。然后分析是否为数字，若是则进入读取数字的流程。之后判断是否为字母，若是则进入判断是否为ID或者关键字的流程。

3. 测试展示

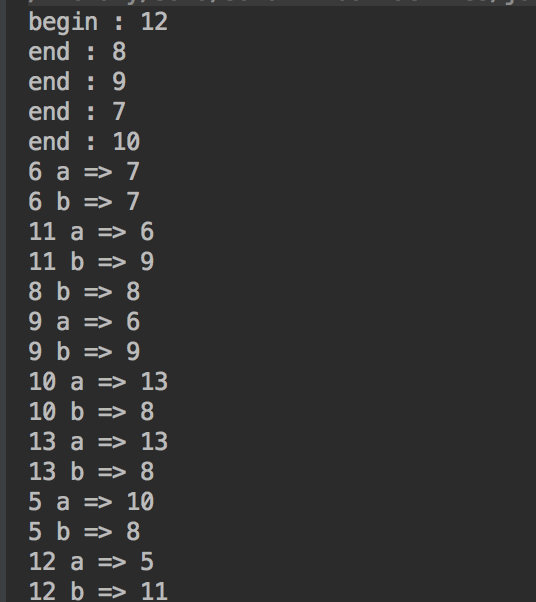
3.1 lex展示

lex的输入为正则表达式集合，测试输入如下：

a\*(a|b)b\*b

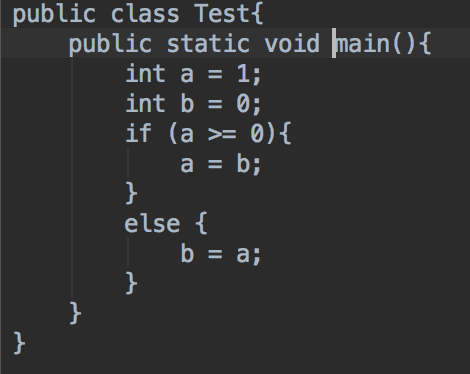
b\*a(a|b)

得到最终优化后的DFA为(编号不是按照顺序编号)

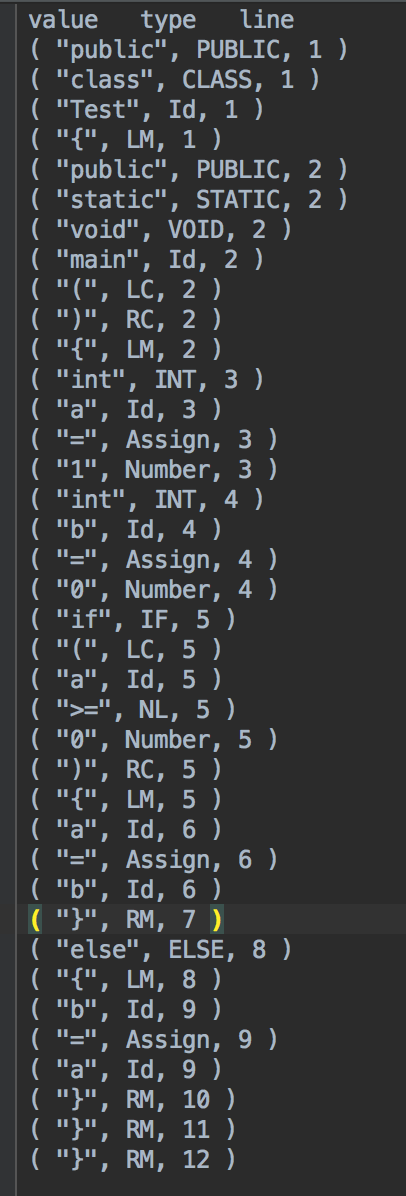


3.2 Token展示

输入的程序代码如下：



得到的输出结果为：



4. 其他

4.1 不足与改进

这次实验中，感觉最有挑战性的步骤是DFA的优化。由于之前设计State和DFA的数据结构时没有很好地考虑到优化部分，所以最后为了完成优化在原有的结构基础上糅杂了一些不适合放在一起的数据，使得代码的结构不是很清晰。以后的实验中需要注意这一点

4.2 总结

在实验中，虽然遇到了一些难题，但还是加以思考加以解决，从中也收获了解决问题之后的成就。这是一次很充实的体验。