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

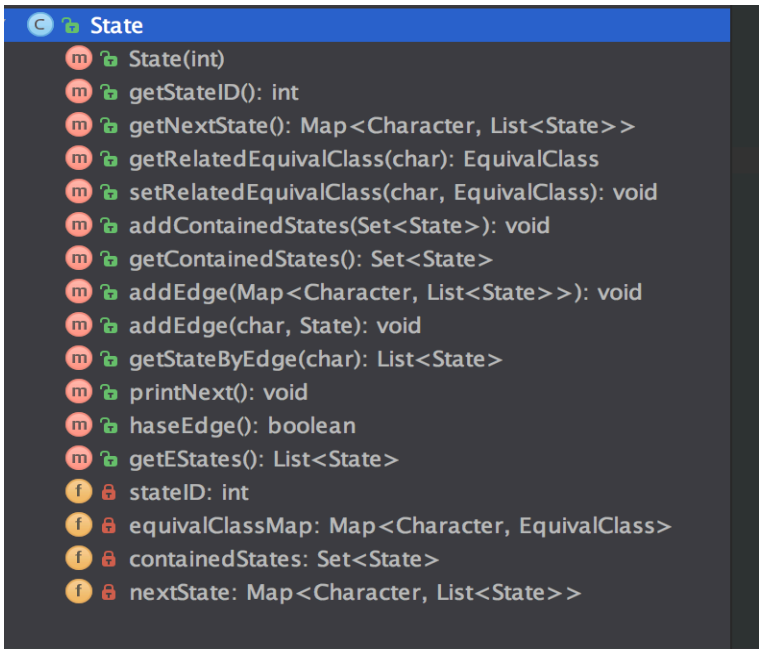
lex 的实现在代码中放置于 analyzer 包和 util 包中。analyzer 包中的 4 个类 REHandler、NFAHandler、DFAHandler、DFAOptimization 分别对应 lex 实现过程中的四个步骤。每个类中都写有 main 方法用于输出 lex 进行到当前步骤时的结果。检查时可以用于测试，结果附在测试展示中。

2.1.1 正则表达式部分的实现

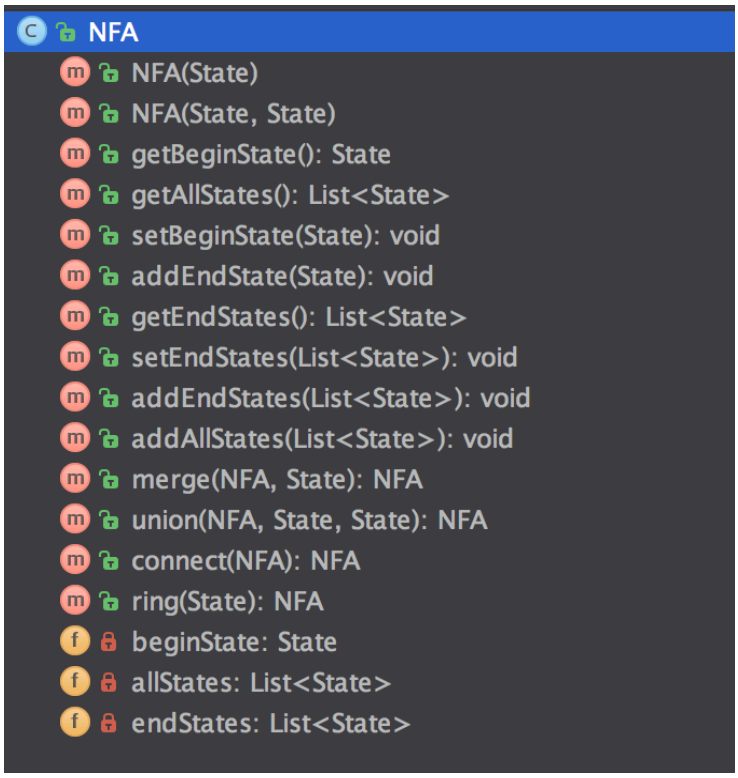
正则表达式的处理为：先从 doc/RE.txt 文件中一行行读取正则表达式，加入存放初始 String list 中，然后对 list 中的每一个正则表达式分别进行加“·”和中缀转后缀的计算。这部分参见 java 类 REHandler 的 addDot 方法和 infix2suffix 方法。

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）。



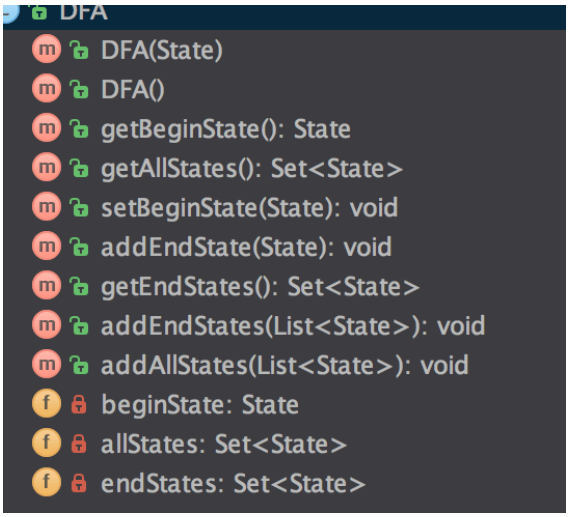
定义 NFA 的数据结构，NFA 中记录起始点以及所有的终止点，记录所有包含的节点。同时，便于 NFA 之间的合并，创建 union（|），connect（·），ring（*）方法，创建 merge 方法合并不同 RE 生成的子 NFA



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

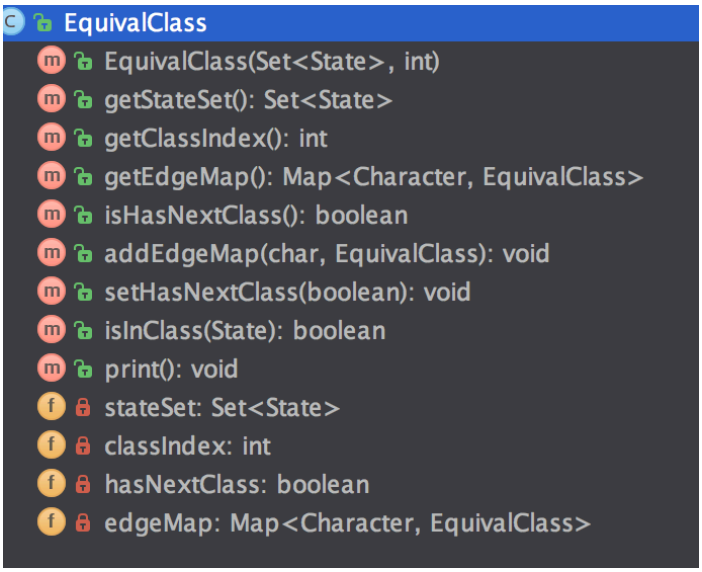
定义 DFA 的数据结构，DFA 中记录起始点以及所有的终止点，记录所有包含的节点。



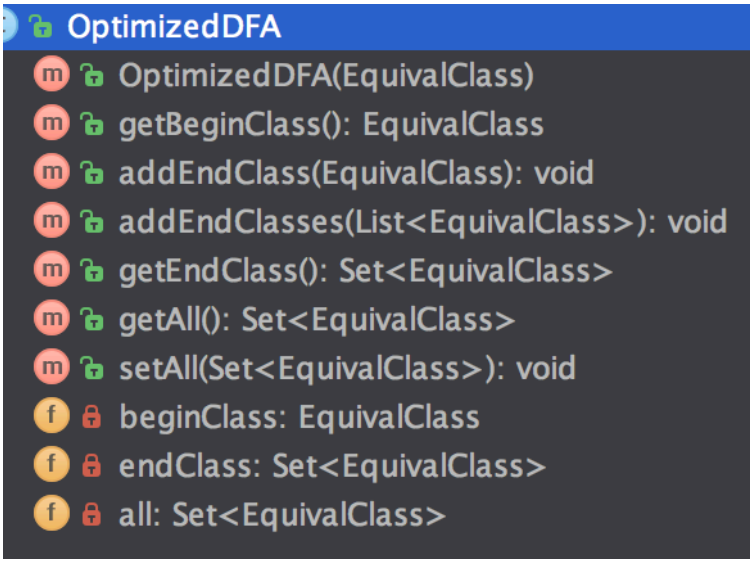
NFA 到 DFA 的转化方法为 epsilon 闭包法。即先从起始点开始，找到对应的 epsilon 闭包。然后以该 epsilon 闭包为单位分别找到各边对应的节点集合的 epsilon 闭包，以此类推对每个闭包进行操作，注意要识别已经重复的 epsilon 闭包。最后生成对应的 DFA。在代码中，把属于同一 epsilon 闭包的节点放置在新生成节点的 containedState 中便于比较重复生成的 epsilon 闭包。用栈结构存储新节点，并通过 BFS 对节点的边进行遍历，确认 DFA 中各节点的边关系。具体实现参加 DFAHandler，核心方法为 NFA2DFA，setContainOther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

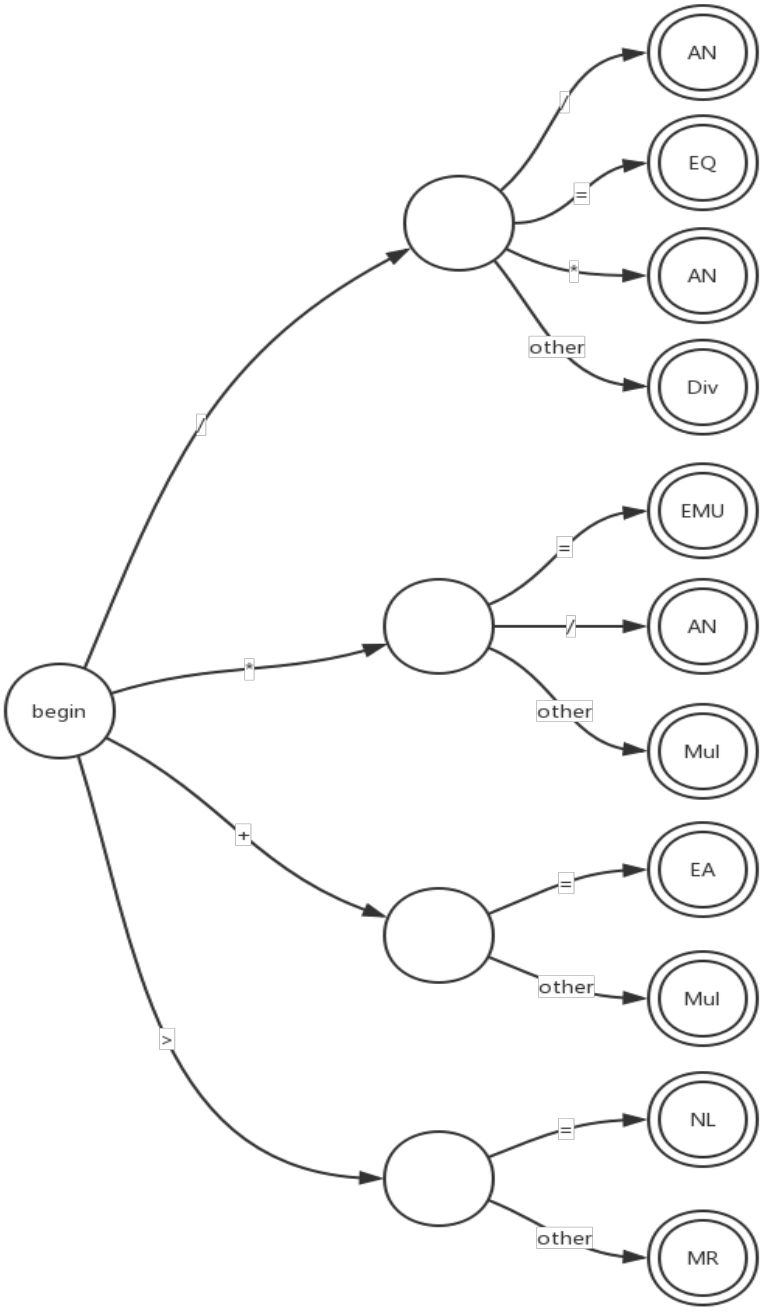
Type 枚举	符号
LM	}
RM	{
LB	[
RB	]
LC	(
RC	)
AN	//
AN	/*
Dot	.

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：



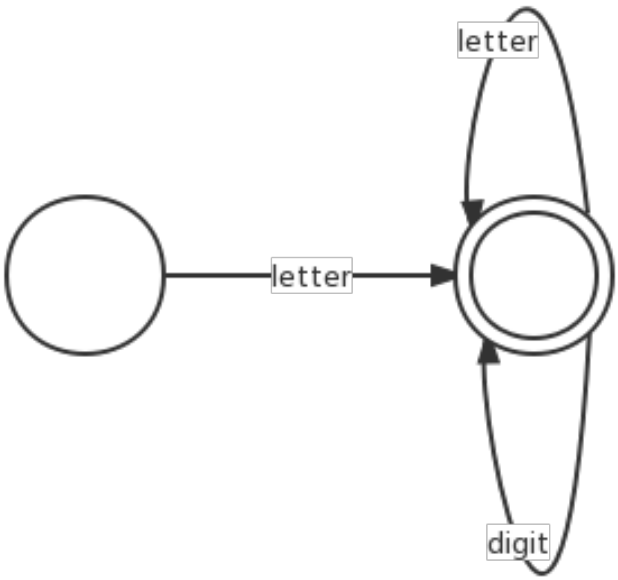
c. 对于 ID，对应的正则表达式为：

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

letter -> [a-zA-Z]

id -> letter (letter | digit)*

对应的 DFA 为：

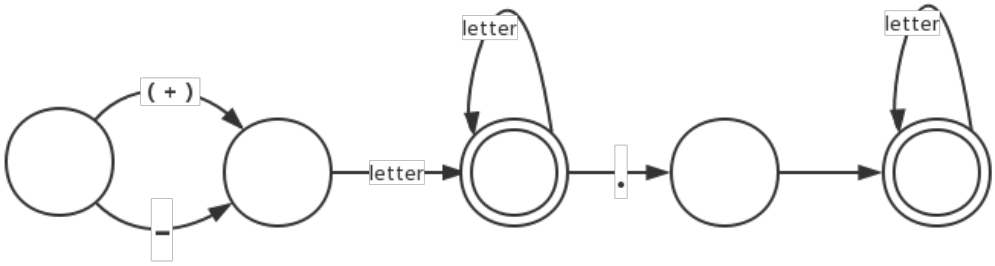


d. 对于数字，正则表达式为：(匹配整数和浮点数)

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

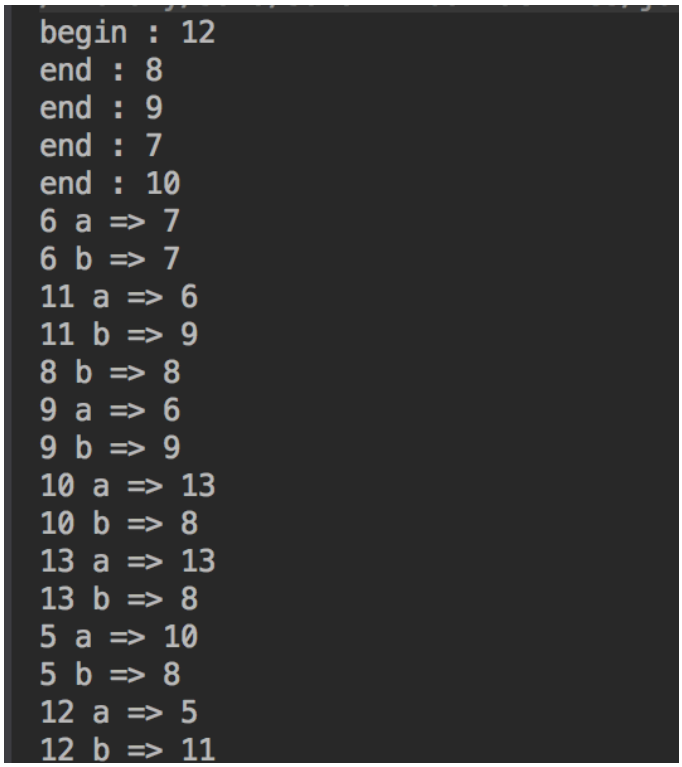
3.1 lex 展示

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：

```
public class Test{
    public static void main(){
        int a = 1;
        int b = 0;
        if (a >= 0){
            a = b;
        }
        else {
            b = a;
        }
    }
}
```

得到的输出结果为：

```
value   type   line
( "public", PUBLIC, 1 )
( "class", CLASS, 1 )
( "Test", Id, 1 )
( "{", LM, 1 )
( "public", PUBLIC, 2 )
( "static", STATIC, 2 )
( "void", VOID, 2 )
( "main", Id, 2 )
( "(", LC, 2 )
( ")", RC, 2 )
( "{", LM, 2 )
( "int", INT, 3 )
( "a", Id, 3 )
( "=", Assign, 3 )
( "1", Number, 3 )
( "int", INT, 4 )
( "b", Id, 4 )
( "=", Assign, 4 )
( "0", Number, 4 )
( "if", IF, 5 )
( "(", LC, 5 )
( "a", Id, 5 )
( ">=", NL, 5 )
( "0", Number, 5 )
( ")", RC, 5 )
( "{", LM, 5 )
( "a", Id, 6 )
( "=", Assign, 6 )
( "b", Id, 6 )
( "}", RM, 7 )
( "else", ELSE, 8 )
( "{", LM, 8 )
( "b", Id, 9 )
( "=", Assign, 9 )
( "a", Id, 9 )
( "}", RM, 10 )
( "}", RM, 11 )
( "}", RM, 12 )
```

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。