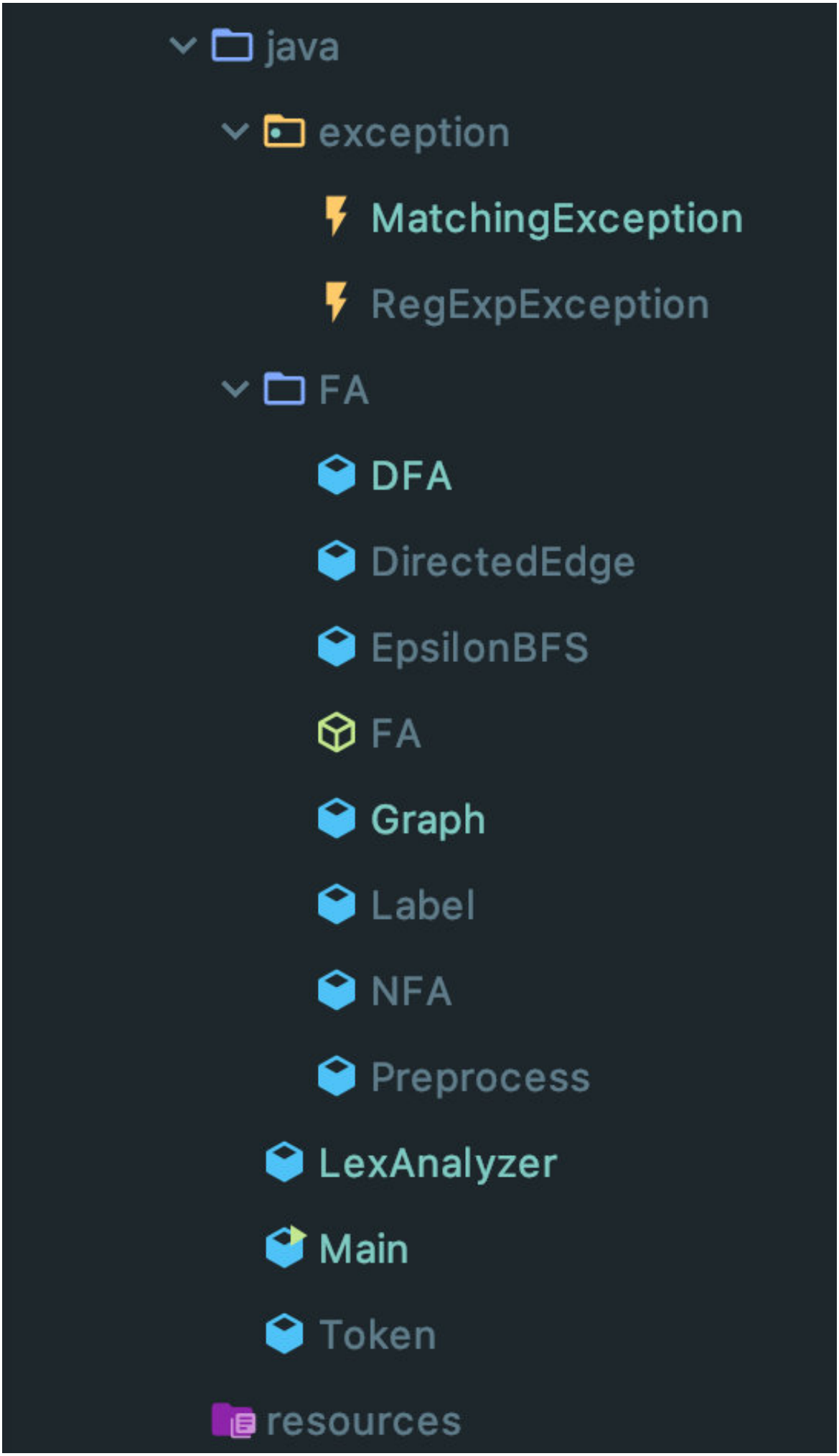
Lex实验报告

学号： 171250649 姓名： 赵文祺

截图

源文件目录截图


java
exception
MatchingException
RegExpException
FA
DFA
DirectedEdge
EpsilonBFS
FA
Graph
Label
NFA
Preprocess
LexAnalyzer
Main
Token
resources

REJava.l资源文件截图

```
delim   [\t\r\n ]
ws      {delim}+
number  {digits}(.{digits})?(e(\+|-)?{digits})?
digit   [0-9]
digits  {digit}+
primitive_type      boolean|byte|char|double|float|int|long|short
control_stmt    break|case|continue|default|do|else|for|if|instanceof|return|switch|while
modifier        abstract|class|extends|final|implements|interface|native|new|static|strictfp|synchronized|transient|volatile
key_words     assert|catch|const|enum|finally|goto|import|package|private|protected|public|super|this|throw|throws|try|void
reserved_words    false|true|null|{key_words}
id      {letter_}({letter_}|{digit})*
letter_     [A-Za-z_]
arithmetic_op       ~|-|\+\+|--|\+|-|\*|/|%|=|\+=|-=|\*=|/=|%=|&=|^=|\|=|<<=|>>=|<<|>>|>>>|&|\||^
bool_op         \?|\\|:|==|!=|>|<|>=|<=|&&|!|\|\|
punctuation     \(|\)|\{|\}|\[|\];|"|'|,|.
```

输入文件/流内容截图

```
public class Main {
    public static void main(String[] args) {
        double a = 1 + 2.0 + 2e-10;
    }
}
```

输出token的截图

```
Token{pattern='key_words', lexeme='public', id=0}
Token{pattern='modifier', lexeme='class', id=0}
Token{pattern='id', lexeme='Main', id=0}
Token{pattern='punctuation', lexeme='{', id=0}
Token{pattern='key_words', lexeme='public', id=1}
Token{pattern='modifier', lexeme='static', id=1}
Token{pattern='key_words', lexeme='void', id=2}
Token{pattern='id', lexeme='main', id=1}
Token{pattern='punctuation', lexeme='(', id=1}
Token{pattern='id', lexeme='String', id=2}
Token{pattern='punctuation', lexeme='[', id=2}
Token{pattern='punctuation', lexeme=']', id=3}
Token{pattern='id', lexeme='args', id=3}
Token{pattern='punctuation', lexeme=')', id=4}
Token{pattern='punctuation', lexeme='{', id=5}
Token{pattern='primitive_type', lexeme='double', id=0}
Token{pattern='id', lexeme='a', id=4}
Token{pattern='arithmetic_op', lexeme='=', id=0}
Token{pattern='number', lexeme='1', id=0}
Token{pattern='arithmetic_op', lexeme='+', id=1}
Token{pattern='number', lexeme='2.0', id=1}
Token{pattern='arithmetic_op', lexeme='+', id=2}
Token{pattern='number', lexeme='2e-10', id=2}
Token{pattern='punctuation', lexeme=';', id=6}
Token{pattern='punctuation', lexeme='}', id=7}
Token{pattern='punctuation', lexeme='}', id=8}
```

1.实现词法分析需要的具体步骤

1. 从REJava.I中读取正则表达式
2. 规范化正则表达式

3. 将正则表达式转为后缀的形式
4. 从后缀的正则表达式构造NFA
5. 从NFA构建对应的DFA
6. 最小化DFA
7. 模拟DFA运行进行字符串匹配
8. 匹配成功返回Token，匹配失败提供报错

2.重要的转化步骤的实现

2.1 规范化正则表达式

2.1.1 消除 {}，该正则表达式的定义依赖于另一个正则表达式

比如：

```
number   {digits}(\.{digits})?(e(\+|-)?{digits})?
digit    [0-9]
digits   {digit}+
```

这时我选择递归地将 {} 中的内容替换为不含大括号的正则表达式，若出现一个正则表达式中出现自己饮用自己的情况，则抛出栈溢出异常 `throw new StackOverflowError("Recursive definition" + pattern + "can not exist!");`。

2.1.2 消除 []

将 `[A-Z]` 转为 `(A|b|c.....Y|Z)`，依靠A到Z在ascii码中按顺序排列的性质，只需要遍历添加即可

2.1.3 添加必要的连接符

- 我选择中文的书名号 `《` 作为连接
- 为了在适当的位置添加连接符，我将需要用到的操作符分为两类

所有操作符: `Arrays.asList('|', '?', '+', '*', '^', ')');`

二元操作符: `Collections.singletonList('|')`

- 对正则表达式进行遍历，判断在当前字符之后是否应该添加连接符 `《`

同时对当前字符和下一个字符进行判断，如果当前字符是二元操作符，或者下一个字符是所有操作符，都不应该在当前字符之后添加连接符 `《`

2.1.4 转义字符

在以上的讨论中，如果出现转义字符 `\`，则需要特殊处理。

- 所有的操作符前有转义字符则将其看成一个普通字符
- 转义字符和其下一个字符之间不应出现连接符 `《`
- 双转义字符 `\\` 来匹配单个反斜杠也被考虑在内

2.2 将正则表达式转化为后缀形式

这里我定义了一个新的类Label来对每个字符进行包装，使得每个单字符和转义字符都被包装在一个Label中。使两者能够被统一处理

```
public class Label {
    char c;
    boolean isEscape;
}
```

借鉴数据结构中将计算式转为后缀形式的思想，定义各个操作符的优先级。优先级越高越先与操作数结合

```
/** Operators precedence map. */
public static final Map<Character, Integer> precedenceMap = new HashMap<>();
static {
    precedenceMap.put('(', 1);
    precedenceMap.put('|', 2);
    precedenceMap.put('《', 3); // explicit concatenation operator
    precedenceMap.put('?', 4);
    precedenceMap.put('*', 4);
    precedenceMap.put('+', 4);
}
```

使用一个Label的栈

- 如果遇到操作数，我们就直接将其加入 `List<Label>` 中。
- 如果遇到操作符，则我们将其放入到栈中，遇到左括号时我们也将其放入栈中。
- 如果遇到一个右括号，则将栈元素弹出，将弹出的操作符加入 `List<Label>` 中直到遇到左括号为止。左括号只弹出并不加入 `List<Label>` 。
- 如果遇到任何其他的操作符，从栈中弹出元素直到遇到发现更低优先级的元素(或者栈为空)为止。弹出完这些元素后，才将遇到的操作符压入到栈中。有一点需要注意，只有在遇到") "的情况下我们才弹出" ("，其他情况我们都不会弹出" ("。
- 如果我们读到了输入的末尾，则将栈中所有元素依次弹出。

2.3 从后缀的正则表达式构造NFA

NFA的底层是一个带边有向图，我根据Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne，对其中的带权重的有向图进行改造，每条边上包含一个Label表示这条边的字符，如果是 ϵ 边则将这个Label设为null。并向外部提供addEdge()的接口

先根据TOMPSON算法定义了构造NFA的三个基本操作，

详见NFA.java中的`NFA kleene(), NFA concat(NFA another), NFA union(NFA another)` 方法

- 连接

- 或
- * 零次或多次

并在此基础上定义了两个拓展操作

- + 一次或多次
- ？ 零次或一次

基本的思想是，创建一个新的NFA，然后将原本NFA中的有向边拷贝到新的NFA中。最后添加必要的 ϵ 边。需要注意的是对于节点的编号，在所有的NFA中，都将源节点的编号设置为0，而将终结节点的编号设置为总节点数减一。

使用了一个Map来建立终结节点与模式名称之间的关联，为后续合并成统一的NFA做准备。

2.4 从NFA构建DFA

在构建的过程中，使用了两个Map，分别用于记录NFA中集合与DFA中状态编号的映射，以及DFA中的邻接表。并使用一个队列来记录待处理的集合。

```
// map between old graph set to new graph id
Map<Set<Integer>, Integer> stateIdMap = new HashMap<>();
// record edges in new graph
Map<Integer, Set<DirectedEdge>> adjMap = new HashMap<>();

Queue<Set<Integer>> queue = new ArrayDeque<>();
```

对于待处理的每个集合

- 先求它的 ϵ 闭包
- 然后在符号集合上进行迭代
    - 对于每个转换符号构建能够到达的子集
    - 在这个子集上求 ϵ 闭包
    - 如果之前没有遇到过这个集合，则放入队列中等待处理
    - 将这个转换符作为一条新的边记录在邻接表中
- 根据邻接表来构建新的DFA
- 设置DFA中终止状态所对应的模式名

详见DFA.java中的 `static DFA builder(NFA nfa)` 静态方法

2.5 最小化DFA

进行最小化DFA状态时，我选择了[HopCroft's algorithm]，并针对多个不同的终止节点的情况进行改进。

```
P := {F, Q \ F};
W := {F, Q \ F};
```

```
while (W is not empty) do
     choose and remove a set A from W
     for each c in Σ do
          let X be the set of states for which a transition on c leads to a
state in A
          for each set Y in P for which X ∩ Y is nonempty and Y \ X is
nonempty do
               replace Y in P by the two sets X ∩ Y and Y \ X
               if Y is in W
                    replace Y in W by the same two sets
               else
                    if |X ∩ Y| <= |Y \ X|
                         add X ∩ Y to W
                    else
                         add Y \ X to W
          end;
     end;
end;
```

初始化时，标准的HopCroft算法将其划分为终止状态和非终止状态两类，但在DFA中可能出现不同模式的终止状态，这些不同模式的装置状态应当被初始化为不同的集合。

- 根据HopCroft算法得到无法继续划分的集合
- 在每个集合中选出一个代表
- 根据原DFA中的邻接关系来构建新DFA中的邻接关系
- 设置DFA中终止状态所对应的模式名

详见DFA.java中的 `NFA minimize()` 方法

2.6 模拟DFA运行

在输入的字符串上迭代，对当前状态下的转换边进行检查，如果有匹配的字符，则转换当前状态。最后检查当前状态是否在终止状态中，如果在终止状态则输出相应的模式名，否则输出空字符串 `""`

```
for (char c: charArr) {
    boolean isMatch = false;
    for (DirectedEdge e: graph.outEdges(currentState)) {
        if (e.label().isMatch(c)) {
            currentState = e.to();
            isMatch = true;
            break;
        }
    }
    if (!isMatch) return "";
}
if (!finalStateMap.containsKey(currentState)) return "";
return finalStateMap.get(currentState);
```

2.7 匹配成功返回Token，匹配失败提供报错

使用两个指针来获取整个文本中需要识别的字符串。保持头指针不动，将尾指针不断后移，截取字符串进行DFA匹配，并记录上一次匹配成功的位置。这样使我们可以选取到能够匹配的最长字符串。

使用一个Token类来记录每个Token的信息，每个token是一个（模式名、词素、id）的三元组。我使用了一个静态的Map来记录每个模式出现的次数，这样实现了同一模式的不同id。

3 Error handling

3.1 从正则表达式构建NFA时

用户书写的正则表达式可能是不符合规范的，这通常会造成构建NFA中从栈中获取下一个NFA时爆出 `NullPointerException` 空指针。我手动获取了可能产生的空指针异常，并包装成 `RegExpException` ，记录下出错的正则表达式和模式名。下方举例中，用户输入的long和short中出现了两个或操作符。

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_192.jdk/Contents/Home/bin/java ...

exception.RegExpException: Parsing failed, please check your
Pattern: primitive_type
RegExp: boolean|byte|char|double|float|int|long||short



    at LexAnalyzer.initNFA(LexAnalyzer.java:136)
    at LexAnalyzer.initFA(LexAnalyzer.java:68)
    at LexAnalyzer.<init>(LexAnalyzer.java:46)
    at LexAnalyzerTest.parseFile(LexAnalyzerTest.java:27) <19 internal calls>
```

3.2 匹配用户输入程序时

保持头指针不动，不断移动尾指针来试图对两个指针之间的字符串进行匹配，**如果尾指针移动到末尾仍没有成功匹配**，则记录下头指针所在的整行，抛出MatchingException，提供用户友好的报错信息。

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_192.jdk/Contents/Home/bin/java ...
Exception in thread "main" exception.MatchingException: Regular expressions fail to match at line: 4
        String s = `hello`;
                   ^
    at LexAnalyzer.analyze(LexAnalyzer.java:178)
    at Main.main(Main.java:15)

Process finished with exit code 1
```

4 遇到的问题和解决方法

4.1 正则表达式预处理

在处理正则表达式时，经常会出现许多情况，难以一次性全部考虑到，通常会出现改了一个bug引发更多bug的情况。

- 建立Label类来封装转义字符和下一个字符，简化操作流程
- 采用**测试驱动**的开发方法，在初步建立好接口之后先写测试用例。选出代表性的用例划分等价类，并在遇到新的bug时将其加入测试用例中。使用maven进行项目管理，一共2000+行左右的源码中，大概700行都是测试用例。

4.2 最小化DFA

标准的HopCroft算法将其划分为终止状态和非终止状态两类，但在DFA中可能出现**不同模式的终止状态**，这些不同模式的装置状态应当被初始化为多个不同的集合。

4.3 模拟DFA运行

最开始我尝试将通配符加入到支持的集合中。

在NFA阶段，通配符是能够被很好地支持的，因为NFA存在 ϵ 边的性质，本身就需要拥有同时“探索”多条转换边的能力。

但在DFA阶段，就出现了问题。因为理论上说，DFA中某一状态在匹配某一个字符时，要么匹配不上，要么有且只有一条转换边能够与该字符匹配。而通配符的出现可能会造成某个字符能够匹配多条边，这就与DFA的定义相违背。

查阅了资料后，似乎也没有特别好的解决方法，除非像模拟NFA那样去模拟DFA，让DFA也能够同时“探索”多条边。思考之后，我认为词法分析器或许可以不使用通配符，也能够进行词法分析。

因此我选择不支持通配符，来保持DFA的性质。

5 感受和实验评价

在参考HopCroft算法时，我开始采用的wiki中文上的伪代码并进行改进，但是化简后的DFA始终不能通过所有的测试用例。后来查阅了资料，自己理解之后才发现**中文版的算法写错了**（被坑死/////）。所以说，不能一味迷信wiki和stackoverflow。实践出真知，只有自己实现过一次，将理论变成算法，算法变成代码，踩过一个个坑，才能有自己的理解。

Chinese

```
P := {F, Q \ F};
W := {F};
while (W is not empty) do
     choose and remove a set A from W
     for each c in Σ do
          let X be the set of states for which a transition on c leads to a state in A
          for each set Y in P for which X ∩ Y is nonempty and Y \ X is nonempty do
               replace Y in P by the two sets X ∩ Y and Y \ X
               if Y is in W
                    replace Y in W by the same two sets
               else
                    if |X ∩ Y| <= |Y \ X|
                         add X ∩ Y to W
                    else
                         add Y \ X to W
          end;
     end;
end;
```

English

```
P := {F, Q \ F};
W := {F, Q \ F};
while (W is not empty) do
     choose and remove a set A from W
     for each c in Σ do
          let X be the set of states for which a transition on c leads to a state in A
          for each set Y in P for which X ∩ Y is nonempty and Y \ X is nonempty do
               replace Y in P by the two sets X ∩ Y and Y \ X
               if Y is in W
                    replace Y in W by the same two sets
               else
                    if |X ∩ Y| <= |Y \ X|
                         add X ∩ Y to W
                    else
                         add Y \ X to W
          end;
     end;
end;
```