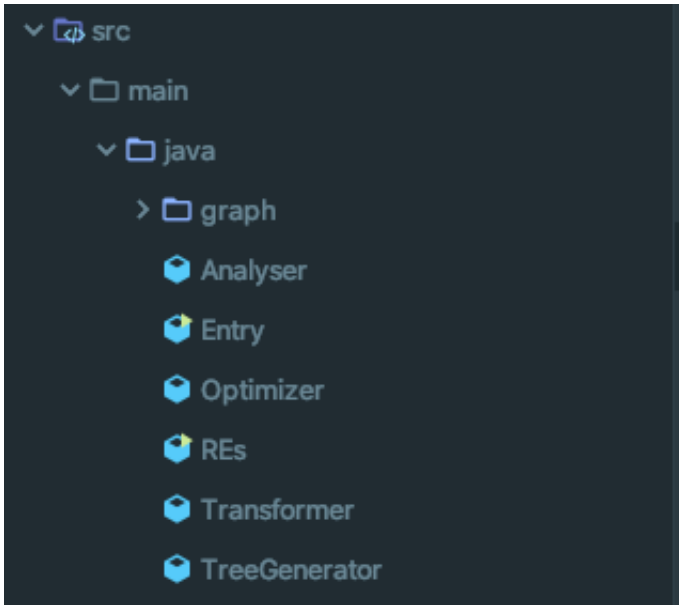
词法分析器 实验报告

学号 171250660

陆放明

0. 截图

0.1 源文件目录截图



说明：

- `Graph` 包中的 `enclosure` 类负责将多个 `FANode` 进行 ϵ 闭包的整合，能够根据当前闭包，根据某一条出边 *edge* 来获取下一个闭包的内容，从而帮助我们进行转换表 (*transition table*) 的构建
- `Entry` 作为**函数主入口**，主要进行 *RE* 文件的读取，组织$RE => NFA => DFA => DFA^O$的构建
- `REs` 主要存储基本的正则表达式信息，作为其他模块的**共享数据中心**
- `Transformer` 主要负责核心部分的 $RE => NFA$, $NFA => DFA$, NFA 优化。具体算法逻辑请见下文
- `TreeGenerator` 辅助 `Transformer` 模块，负责最终 `Enclosure` & `FANode` 的图构建 (*Thompson* 算法)，图的 `DFS , BFS` 遍历算法实现
- `Analyser` 是在 `transfomer` 获取得到优化 DFA 之后，根据 DFA 来进行我们最终的词法分析，也是我们的词法分析器的执行模块。它将会读取指定的 `input.txt` 文本文件，基于之前给定的正则 REs , 来进行词法分析，输出最终的 `tokens`
- `optimizer` 进行 *DFA* 优化的工作

0.2 输入文件格式截图

> $REs.txt$ 文件内容

```
letter->([a-z]|[A-Z]|_).([0-9]|[a-z]|[A-Z])*
number->([0-9])*
operator->+|-|/|%|<|=|>|&
separator->,|;|{|}
```

> 输入文件内容

```
3 + 4 ;
int a = 3 ;
var a3 = 4 ;

let t = 3 + 2 ;
let num = 4 & 5 ;
if a < 3 return 4 ;
illegalVariable 3a4
legalVariable a34 _a3 aabc ;
```

> 输出 *token* 内容

```
<number,3>
<operator,+>
<number,4>
<separator,;>
<letter,int>
<letter,a>
<operator,=>
<number,3>
<separator,;>
<reservedWord,var>
<letter,a3>
<operator,=>
<number,4>
<separator,;>
<reservedWord,let>
<letter,t>
<operator,=>
<number,3>
<operator,+>
<number,2>
<separator,;>
<reservedWord,let>
<letter,num>
<operator,=>
<number,4>
<operator,&>
<number,5>
<separator,;>
<reservedWord,if>
<letter,a>
<operator,<>
<number,3>
<reservedWord,return>
<number,4>
<separator,;>
<letter,illegalVariable>
Syntax error on the content < 3a >
<letter,legalVariable>
<letter,a34>
<letter,_a3>
<letter,aabc>
<separator,;>
```

1. 词法分析步骤简介

1.1 正则表达式解析

主要进行 `\w`，`[0-9]`，`[a-zA-Z]` 这类正则表达式的解析，由于后续步骤中需要进行 **中缀转后缀** 的表达式转换，需要将这些抽象表达式均通过 `|` 进行连接，同时在 **连接** 操作的时候添加 `.` 来进行分隔

1.2 RE => NFA

1. 先把中缀形式的正则表达式转换为后缀表达式
2. 根据获取得到的后缀表达式和课程介绍的 *Thompson* 算法，构建出一个 NFA 的拓扑图

1.3 NFA=> DFA

根据 NFA，进行迭代式的 ϵ 闭包求解，逐步构建出 $TransitionTable$，直到我们的闭包集合不再变化

1.4 DFA优化

由 DFA 来进行进一步的优化，使用二叉树来进行逐步的分解判定，最终得到的新的转换表就是我们所希望的优化 DFA^0

1.5 文本词法分析

这一步就是我们真正做解析文本内容的时候了，根据之前已经优化过后的 DFA 转换表，来逐个字符的进行 $leftmost$ 文本解析

2. 实现步骤具体分析

2.1 正则表达式解析

> 这里的正则表达式解析相对来说比较基础，只能够支持 `|`， `*`， `()` 这些基本的正则表达式解析。

在当我们从文本文件读入正则表达式之后，我们首先把正则表达式进行初步的解析。这一步做的主要工作就是——添加 `.` 连接符和 `[0-9]`，`[a-z]` 这类抽象 RE 的具体化。这时候我们已经能够得到一个相对容易进行后缀表达式的中缀表示。

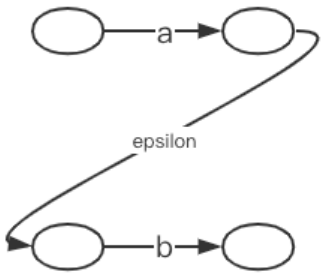
接下来进行 *Infix=>postfix* 的转化，具体逻辑在 `TreeGenerator.infix2PostFix(String infix)` 函数中，它的逻辑和普通的表达式转后缀完全一致，不需要过多赘述。

2.2 RE=> NFA

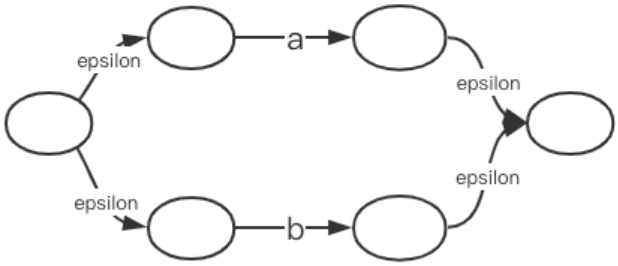
根据 *Thompson* 算法，我们针对已经获得的后缀表达式进行 **图转化**，具体转化形式如下。代码逻辑在

`TreeGenerator.singleNode`，`TreeGenerator.connectNode`，`TreeGenerator.loopNode` 分别进行实现

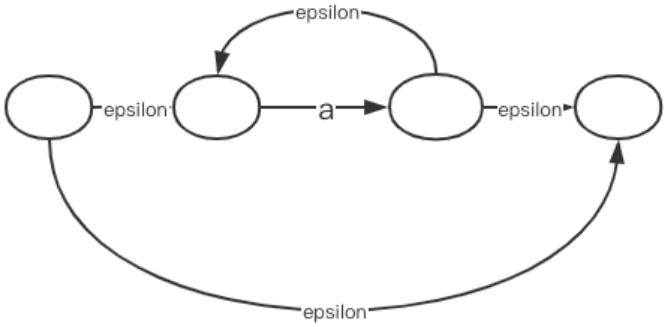
a·b



a | b



a*



对于每一条 *RE*，我们都可以获取到一个子图 $FATGraph_i$，而最终我们期望使用一个 NFA 来进行表示，所以额外添加一个 S 起始节点，使用 ϵ 边逐个连接各个子图，构成最终的完整的 NFA

2.3 NFA=>DFA

下一步所要做的就是把我们的 NFA，通过求 ϵ 闭包和子集构造法，来得到最终的 *Transition table* 。

这里主要使用到了如下几个方法

- 根据给定的一系列 `FATNode` 集合，来构造以它为 *enclosure core* 的 ϵ 闭包

  ```
  public Enclosure(Set<FANode> nodes)
  ```

- 通过当前闭包 *Enclosure* ，和给定的某一条出边 *edge* ，来求出所有该闭包内拥有该 *edge* 的 $FATNode$ 集合

  ```
  public Set<FANode> nextSet(String edge)
  ```

- 判定两个闭包是否相同的重写函数

> `public boolean equals(Object obj)`

通过如上封装的函数内容，我们可以很容易的按照如下算法进行 $NFA => DFA$ 转换。

具体算法如下：

> 1. 以初始节点为 *enclosure core* 构造第一个闭包 I_0
> 2. 根据当前闭包，对于每一条可能拥有的出边，计算出所有对应的 `nextSet` , 对他们分别求闭包
>   1. 如果新的闭包 I_i 在已经出现过的闭包当中，那么不进行操作
>   2. 如果新的闭包 I_i 并没有出现在已经出现过的闭包中，那么在闭包列表中加入这一项，等待后续的操作
> 3. 当前闭包所有的出边分析完毕之后，来到下一条闭包内容，重复 1 ，2 步骤，直到闭包列表的长度不再发生变化

经过如上的步骤产生的转换表放置在类 `Transformer` 中的 `public static List<Map<String, Enclosure>> trasitionTable` 当中。

根据它的定义形式，可以很容易看出，它是一个常见的哈希表结构：

- 纵向的 `enclosure` 列表作为哈希表的**纵向链表结构**
- 每一个链表都对应了转换表中的**一个行**，为了查询方便起见，不使用哈希表中的链表，而是直接使用另一个 `HashMap` 来进行存储。

  > 那么当我们在之后想要求出：某一个闭包 I_i 在某一个出边 $edge$ 情况下，进行的状态转换就可通过
  >
  > `transitionTable[i].get(edge)` 来快速获取

2.4 文本文件输入的解析

有了前面几个步骤的铺垫，这一步就非常简单了。我们不需要进行硬编码来做 *FA* 的状态转换 ，而是以 *Table Driven* 的方式来依据之前获得的转换表进行状态转换即可。当某一个转换的单元为空时即进行相对应的错误处理。

具体的逻辑都在 `Analyser.java` 模块当中，大体的思路为

> 1. 逐个字符，自左向右扫描文本文件
> 2. 对于非空格字符，进入转换表中进行查表，查询下一个状态内容
>   1. 如果下一个状态存在，即表示字符合法，可以进行下一个字符的扫描。直到遇到 *EOF* 或下一个空格字符
>   2. 如果下一个状态不存在，表示当前词法错误，进行相对应的错误提示

3. 错误处理

输入来源只有两个：`正则表达式的文本文件`和`输入流文件`。那么在这里就根据这两者进行错误的检测。

3.1 不合法的正则表达式

当一个正则表达式不满足操作数和操作符的数量关系或不满足结合律与操作目数量的时候，会导致中缀转后缀过程中出栈时栈为空的情况发生。此时将会输出 `invalid RE expression`

3.2 无法识别的文本内容

举例说明

```
//REs.txt
letter->[a-zA-Z]([0-9]|[a-zA-Z])*
number->([0-9])*

//input.txt
int 3a = 4 ;
```

在这个情况下，我们所定义的 `letter` 无法匹配 `3a`，将会输出 `Syntax error on the content < 3a >`，表示我们无法识别 `3a`。注意：这里并不会采取贪心策略告知整个 `token`，而是将错误发生的位置显示出来。

4. 问题与解决方法

4.1 建模问题

一个比较显著的问题就是：如何把 `FA` 进行建模？这个直接关系到了最终的图算法的执行过程和 DFA 优化过程。这里我定义了两种数据结构：

- `FANode`：表示 NFA 的每一个节点，具体定义如下

```
class FANode{
    public boolean  isEnd;         //指示是否是自动机结尾
    public String   identifier;    //指示node节点的标识符
    public Map<String, List<FANode>>  edgeFANodeMap;//
}
```

这里较为复杂的是 `edgeFANodeMap` 这一个属性，它表示的 *map* 关系 <edge> => <Node list>，指的是从节点 *node* 的出边 `edge`，到边的终点的列表这一个映射。

它包含了如下关系：

- 一个 *node* 可以有多个出边——使用 *map* 来进行描述；
- 一个 *node* 的某一条出边有不止一个终点—— *map* 的 *key* 值使用列表

此外，我们还需要对 ϵ 闭包进行建模，其具体定义如下

```
class Enclosure{
    public Set<FANode> enclosure; //闭包内包含的所有 FANode
    public int identifier;        //闭包唯一标识
}
```

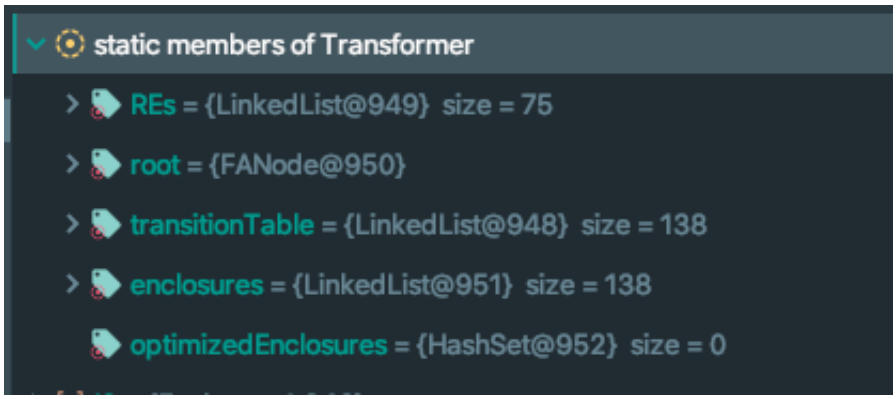
4.2 运行速度问题

当我的 *REs* 如下定义的时候

```
letter->([a-z]|[A-Z]|_).([0-9]|[a-z]|[A-Z])*
number->([0-9])*
operator->+|-|/|%|<|=|>|&
separator->,|;|{|}
```

其运行速度可以说是特别地慢，解析简短的10行代码也要接近十秒。

我特别地去查看了一下中间转换的过程中的节点个数，下图是在 *NFA=>DFA* 之后的运行结果图：



可以看到，转换表 `tansitionTable` 的条目达到了 **138** 条之多，那么 *NFA* 的数量就想必更加庞大，而我在填表的后期，使用的是 **DFS** 算法，其运行速度慢也就不足为奇。至于其优化方案，初步设想就是是否能够利用 《龙书》上的 $RE => DFA$ 直接进行计算，减少前期因为深度遍历造成的浪费。

5. 实验评价和感觉

此次实验的重点在于对原来的算法的理解的基础上，进行建模工作。而其中一些常用的栈操作、DFS、BFS算法也是一个很好的巩固机会，能够进一步加深对于课程内容的理解。而且在此基础上，Lab2中的YACC也能够借鉴我这一次的数据结构内容，从而也算是突破了不少的难关。