编译原理课程实验报告

- 词法分析

姓名：焦紫顺

学号：121250063

2014/11/4

目录

1. 目标

本实验是以正则表达式和有限状态机为理论基础，实现编译器编译过程前端的第一阶段，通过正则表达式自定义的计算机语言的词法单元，根据定义的词法规则对由字符序列组成文本进行分析，最后从字符流产出能够识别为某个词法单元的 Token 序列。此过程称为词法分析。本实验不但需要能够从随机文本识别出定义的 Token，更重要的是能够实现从基本正则表达式通过计算机自动构造出能够识别文本中的单词的机制。

2. 内容与功能描述

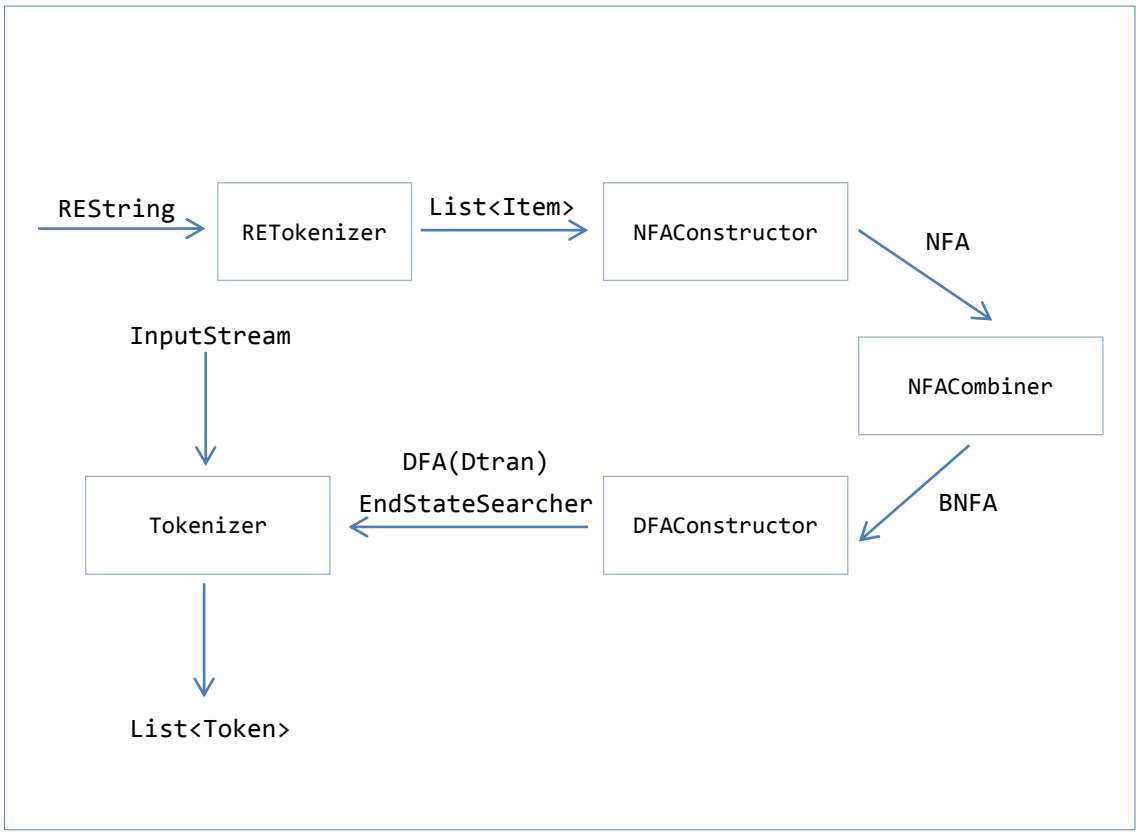
本实验工具的实现语言为 Java (SE 1.7)，目录结构和内容简述如下表。

目录	描述
Document	文档目录
\编译原理实验报告.docx	本文档
Input	输入目录
\re.txt	正则表达式定义文件
\text.txt	待分析文本文件
Lex	程序项目目录
\src	源代码目录
\basic	基本数据结构基类
\lex	主程序
\structrue	核心类和算法
\debug	输出调试辅助类
\iohelper	文件输入类

实现功能：

根据完全可自定义的正则表达式对文本进行词法分析，得到 Token 序列。

+ 提供优先级设置解决正则冲突。
+ 可简化的正则表达式描述。
+ 正则表达式重复定义错误提示。
+ 正则表达式语法错误提示，包括括号匹配、操作数个数不匹配等。
+ 对由已定义的正则表达式不能识别的词素给予提示。



3. 原理与设计

设计程序时采用了“自顶向下，逐步求精”的经典程序设计思想，将词法分析过程依次划分为正则表达式词法分析、语法分析与 NFA 构造、NFA 合并、DFA 转化、文本分析五个步骤，逐渐将正则表达式符号转化为能够分析词素的转换表和终态表，通过这种划分降低问题的复杂性，依次解决每个子任务的途径实现了词法分析。

原理上能够构造出任何符合基本正则表达式的 DFA。

正则表达式词法分析：正则表达式本身也是一种语言，因此有必要进行简单的词法分析，从中得出属于正则表达式运算符号和纯字母字符，此过程通过转义字符分辨具有歧义的字符，如‘*’，‘(’等，转化为 Item 序列。

NFA 构造：将 Item 序列转换为便于处理的后缀表达形式，然后通过 Thompson 算法根据运算符迭代地构造 NFA，每一个正则表达式对应一个 NFA。

NFA 合并：在这里，和正则运算不同，需要将所有正则表达式对应的 NFA 合并构成一个大 NFA，为 DFA 转化做准备，合为一个才能更好的进行词法分析。

DFA 转化：运用子集构造法将 NFA 转化为 DFA，这和手工运算类似。构造出的 DFA 本身就具有转换表的功能，同时需要从之前的 NFA 终态对应的词法单元得出 DFA 的终态表，便于确认词法单元。

文本分析：最后一步，读入输入字符序列，根据 DFA 和终态表分析出词法单元。

4. 约束与假设

原理上能够构造出任何符合基本正则表达式的 DFA。因此此程序不拘泥于任何限定的正则表达式。正则表达式支持(,), *, | 运算。支持转义字符。实际上，由于 ReComputer 类封装了正则运算的 NFA 构造规则，因此只要给出一个新的运算符号和相对应的 NFA 构造运算算法，即可支持该正则符号。为简单起见，测试正则表达式采用 C 语言的部分基本词法规则，可通过修改 re.txt 中的正则表达式而改变。

5. 数据结构设计

本实验的实现涉及到多种数据结构。所有的详细数据操作实现请参看源代码。

5.1 自动机

为了要抽象描述 FA 的图形结构，可以首先建立图的抽象数据类型。DirectGraph 是有向图数据结构的实现类，由于 FA 涉及频繁的增删边点的操作，因此必须采用邻接表的实现。

```
class DirectGraph<V,E> {
    LinkedList<Vertex<V> VertexList; // 顶点链表
}
class Vertex<V> {
    V v;                              // V 是参数化类型，便于复用和扩展
    LinkedList<Edge> outEdgeList; // 出边链表
    LinkedList<Edge> inEdgeList;  // 入边链表
}
class Edge<E> {
    E e;                             // 参数化类型
    Vertex source;                // 边起点
    Vertex destination;            // 边终点
}
```

详细数据操作实现参看源代码。

自然，NFA 是有向图的扩充，继承有向图类的属性和方法，描述 FA 就非常方便。NFA 类需要增加起点 start 属性作为开始态，end 属性作为结束态，为了更好地确认边点，继承时将 V 类型参数实例化为 Integer 并确保每个生成的顶点有唯一的编号，E 参数化为 Character。

DFA 与 NFA 不同的是，它可以有多于一个的终态，分别对应于不同的 Token，因此继承 NFA 类，对关于终点的操作进行重载即可。

5.2 运算栈

从正则表达式构造 NFA 时，需要建立操作数栈。遇到操作数时（普通字符）将之构建成单个顶点的 NFA 并压入栈中，若遇到运算符，则根据运算符的目数从栈中弹出操作数（NFA），如四则运算一样进行运算，这里的运算是 NFA 的自底向上的构造，其过程本质上和四则运算没有本质区别，最后留在栈中的 NFA 即为运算结果，也即最后的 NFA，若期间出现栈空异常，说明表达式中不符合操作符的所需操作数，作出警告信息。

5.3 NFA 状态集合

在构建 DFA 时，利用的子集构造法中需要状态集的模拟，可利用 Java 中的集合类方便的比较两个集合是否相等。

```
class Dstates {
    // NFA 状态列表，每个状态包含求闭包运算后的顶点，数组下标即为此状态集的记号
    private ArrayList<Set<Vertex<Integer>>> statesSet;
    // 为比较状态是否相等，需要提出顶点中的 Integer 数据作为集合，利用 equals 判断集合相等
    private ArrayList<Set<Integer>> idSetList;
    // 模拟子集构造法对状态的标记，快速取出未标记的状态
    private Map<Integer, Boolean> tagMap
}
```

详细数据操作实现参看源代码。

5.4 字符缓冲区

由于词法分析需要提取出相应的单词字符，需要标记单词开始的标志，逐个分析字符直至结束位置，然后将开始位置和结束位置的文本提取出来。为了简化操作，一开始直接将字符存至 StringBuffer 变量里,每走一步便将字符加到字符串的尾部直到推导出 Token 的结束位置，回滚最后一个字符，此时 StringBuffer 的字符串即符合词法的单词。由于需要回滚，对输入流设立简单的缓冲区。

```
public class InputHandler {
    private InputStream ins;        // 输入流
    private StringBuffer temp;     // 寄存字符串
    private static int LAST = 0;   // 输入流上一个读取字符指针
    private static int NOW = 1;    // 输入流现在读取字符指针
    private static int NEXT = 2;    // 输入流下一个读取字符指针
```

```
    private int ch[];              // 字符缓冲区
    private int ptr = NOW;      // 词法分析器读取的字符指向的指针，回滚时只需
```
让 ptr 减一，下次在读取时将指针复位至 NOW

详细数据操作实现参看源代码。

6. 算法设计

6.1 正则表达式中序转后序

需要为正则表达式定义优先级，其中*运算优先级最高，而括号对能够改变运算顺序。转后序需要维护一个栈，同时定义栈优先级(isp)和符号优先级(icp)。读入正则表达式序列，若是操作数直接接入表达式中，若是操作符，则比较 icp 和 isp 的大小。算法如下。

```
//比较 icp 和 isp 的大小。
if(operatorStack.isp() < operator.icp()) {
        operatorStack.push(operator); // 将高优先级运算符压栈
}
else if(operatorStack.isp() > operator.icp()) {
        postfixTokens.add(new Item(operatorStack.pop()));
        itemStack.push(item);    //将栈中高优先级运算符取出接到表达式后
}
else {
        operatorStack.pop();          // 括号对直接出栈
}
```

6.2 子集构造法

子集构造法算法实现上与教材相似，需要 NFA 实现三个基本闭包操作，NFA 实现该操作的接口。

```
public interface SubsetConstructionOperations {
    /**ε-closure(s)*/
    public Set<Vertex<Integer>> e_closure(Vertex<Integer> s);
    /**ε-closure(T)*/
    public Set<Vertex<Integer>> e_closure(Set<Vertex<Integer>> T);
    /**move(T,a)*/
    public abstract Set<Vertex<Integer>> move(Set<Vertex<Integer>> T,
```

```
            Character a);
}
```

```

```

闭包算法实际上需要运用 DFS 遍历的非递归算法，一开始将需要计算的所有状态压入栈中，每弹出一个顶点，若存在从该顶点出发到一个标号为 ε（程序中为 null）到达的状态，只要该状态未在结果集合中就加入集合并压栈，直至栈空为止。

```
Vertex<Integer> from;
Vertex<Integer> to;
while(! vertexStack.isEmpty()) {
    from = vertexStack.pop();
    Set<Vertex<Integer>> states = this.e_move(from);
    Iterator<Vertex<Integer>> stateItr = states.iterator();
    while(stateItr.hasNext()) {
        to = stateItr.next();
        if(! closureSet.contains(to)) {
            closureSet.add(to);
            vertexStack.push(to);
        }
    }
}
```

6.3 DFA 模拟

模拟转换表只需 DFA 实现 Dtran 接口操作即可。实际实现中，DFA 需要存放指向初始态的指针。每找到一个字符，该指针就指向由该字符边出发到达的状态。与 DFA 存放的终点列表判断是否已达终态，联合终态表找到相应的 Token。

```
/**转换表抽象接口*/
public interface Dtran {
    /**开始搜索/复位*/
    public void searchStart();
    /**是否已在终态*/
    public boolean reachEnd();
    /**根据终态表得出 Token*/
    public String getTokenType(EndStatePatternSearcher esps);
    /** 是否能够根据字符 c 找到下一状态*/
    public boolean findNext(Character c);
```

```
}
```

7. 运行用例

请查看 Input 目录并运行程序。

8. 问题与发现

自定义的正则表达式之间可能存在交集，比如关键字的正则表达式和普通的标识符正则表达式，会导致在建立 DFA 终态时出现问题，即某些状态可能对应不同的词法单元。这里的解决方案是为正则表达式建立优先级，高优先级的表达式在某些冲突状态会覆盖原来，这在程序运行时会有报告。