湖南大學
HUNAN UNIVERSITY

课程实验报告

课　程　名　称：编译技术与应用

实验项目名称：正则运算表达式的 DFA 构建

专　业　班　级：软件 2105

姓　　名：马小梅

学　　号：202126010530

指　导　教　师：杨金民

完　成　时　间：2024 年 3 月 24 日

信息科学与工程学院

实验题目：正则运算表达式的 DFA 构建

实验目的：

（1）学习和掌握正则表达式构造 NFA 的方法

（2）学习和掌握 NFA 转换为 DFA 的方法和步骤

（3）掌握字符集的基本运算（如范围运算、连接运算、并运算、差运算等）

（4）掌握 NFA 和 DFA 的基本方法和运算

实验环境：笔记本电脑、Win11、idea Java 环境

实验内容及操作步骤：

一、定义基本数据结构

1）字符集

```
public class CharSet {
    /** 字符集id*/
    3 个用法
    private final int indexId;
    /**字符集中的段id（一个字符集可以有多个段*/
    3 个用法
    private final int segmentId;
    /** 段的起始字符from */
    3 个用法
    private final char fromChar;
    /** 段的结束字符to */
    3 个用法
    private final char toChar;

    2 个用法
    private static int indexIdNum = 0;
    2 个用法
    private static int segmentIdNum = 0;
```

2）字符集表定义

```
20 个用法
static private ArrayList<CharSet> pCharSetTable = new ArrayList<>();
```

3）NFA 或 DFA 定义

```
13 个用法
public class Graph {
        3 个用法
        static private int graphIdNum = 0;

        5 个用法
        private int graphId;
        8 个用法
        private int numOfStates; //状态序号
        6 个用法
        private State start;

        17 个用法
        private ArrayList<Edge> pEdgeTable; //边集合
        23 个用法
        private ArrayList<State> pEndStateTable; //终止状态集合
        14 个用法
        private ArrayList<State> pStateTable; //状态集合
```

4）边定义

```
public class Edge {
    5 个用法
    private int fromState;
    5 个用法
    private int nextState;
    5 个用法
    private int driverId;
    5 个用法
    private DriverType type;
```

5）状态定义

```
public class State {
    4 个用法
    private int stateId;
    4 个用法
    private StateType type;
    4 个用法
    private LexemeCategory category;

    2 个用法
    static private int stateIdNum = 0;
```

6）转换枚举类型

```
1 个用法
public enum DriverType  {
    //空字符
    4 个用法
    NULL,
    //  字符
    0 个用法
    CHAR,
    //字符集
    0 个用法
    CHARSET,
}
```

7）状态取值枚举类型

```
public enum StateType {
    // 匹配和不匹配
    3 个用法
    MATCH,
    4 个用法
    UNMATCH
}
```

8）词类别枚举类型

```
public enum LexemeCategory {
    // 空字符
    3 个用法
    EMPTY,
    // 整数常量
    0 个用法
    INTEGER_CONST,
    // 实数常量
    0 个用法
    FLOAT_CONST,
    // 科学计数法常量
    0 个用法
    SCIENTIFIC_CONST,
    // 数值运算词
    0 个用法
    NUMERIC_OPERATOR,
    // 注释
    0 个用法
    NOTE,
    // 字符串常量
    0 个用法
    STRING_CONST,
    // 空格常量
    0 个用法
    SPACE_CONST,
    // 比较运算词
    0 个用法
    COMPARE_CONST,
    // 变量词
    0 个用法
    ID,
    // 逻辑运算词
    0 个用法
    LOGIC_OPERATOR,
    // 关键字
    0 个用法
    KEYWORD
}
```

9）正则表达式定义

```
0 个用法
public class RegularExpression {
    2 个用法
    int regularId;
    1 个用法
    String name;

    /**
     * 正则运算符，共有 7 种：'=', '~', '-', '|', '.', '*', '+', '?'
     */
    1 个用法
    char operatorSymbol;
    /**
     * 左操作数
     */
    1 个用法
    int operandId1;
    /**
     * 右操作数（一元运算时为null）
     */
    1 个用法
    int operandId2;
    /**
     * 左操作数的类型
     */
    1 个用法
    OperandType type1;
    /**
     * 右操作数的类型（一元运算时为null）
     */
    1 个用法
    OperandType type2;
    /**
     * 运算结果的类型
     */
    1 个用法
    OperandType resultType;
    /**
     * 词的 category 属性值
     */
    1 个用法
    LexemeCategory category;
    /**
     * 对应的 NFA
     */
    1 个用法
    Graph pNFA;

    1 个用法
    private static int regularIdNum = 0;
```

二、针对字符集的创建，实现如下函数

1）int range (char fromChar, char toChar); // 字符的范围运算

函数作用：得到起始字符到结束字符之间的任意字符集

实现方法：新建一个字符集，直接加入字符集表即可。

实现函数：

```
/**
 * 创建一个新的字符集（初始化）
 */
8 个用法
public CharSet(char fromChar,char toChar) {
    this.indexId = indexIdNum++;
    this.segmentId = segmentIdNum++;
    this.fromChar = fromChar;
    this.toChar = toChar;
}
```

```
public static int range(char fromChar, char toChar) {
    CharSet ch = new CharSet(fromChar, toChar);
    pCharSetTable.add(ch);
    return ch.getIndexId();
}
```

2）int union(char c1, char c2); // 字符的并运算

函数作用：进行字符与字符之间的并运算

实现方法：新建一个字符集对象，判断 c1 和 c2 是否相等，不相等的话新建一个段，加入字符集表。

实现函数：

```
public static int union(char c1, char c2) {
    CharSet charSet1 = new CharSet(c1,c2);
    pCharSetTable.add(charSet1);
    //判断c1 | c2 是否是新的字符集
    if(c1 != c2) {
        CharSet charSet2 = new CharSet(c1,c2,charSet1.getIndexId());
        pCharSetTable.add(charSet2);
    }
    return charSet1.getIndexId();
}
```

3）int union(int charSetId, char c); // 字符集与字符之间的并运算

函数作用：进行字符和字符集之间的并运算

实现方法：首先在字符集表中找到 id 是 charSetId 的段。新建一个字符集，获取其 indexId，把原字符集的所有段赋值给新建字符集，再给字符新建一个段，放入字符集表中。最后返回新得到的字符集的 Id。

实现函数

```
public static int union(int charSetId,char c) {
    //判断字符集表是否已存在charsetid对应的字符集
    boolean flag = false;
    int newId = 0;
    for(CharSet charSet:pCharSetTable) {
        if(charSet.getIndexId() == charSetId) {
            if(flag) {
                //新建一个段
                CharSet charSet1 = new CharSet(charSet.getFromChar(),charSet.getToChar(),newId);
                pCharSetTable.add(charSet);
            } else {
                // 新建字符集（newId）
                CharSet charSet1 = new CharSet(charSet.getFromChar(),charSet.getToChar());
                newId = charSet1.getIndexId();
                pCharSetTable.add(charSet1);
                flag = true;
            }
        }
    }

    //新建一个段
    CharSet newSegment = new CharSet(c,c,newId);
    pCharSetTable.add(newSegment);
    return  newId;
}
```

4）int union(int charSetId1,int charSetId2)；//字符集与字符集的并运算

函数作用：字符集与字符集的并运算

实现方法:首先遍历整个字符集表,找到所有是 charSetId1 和 charSetId2 的段,直接将两个字符集的所有段加到新的字符集中,并返回相应 Id 即可。

实现函数：

```
public static int union(int charSetId1,int charSetId2) {
    boolean flag = false;
    int newId = 0;
    ArrayList<CharSet> newList = new ArrayList<>();
    //字符集1
    for (CharSet charSet: pCharSetTable) {
        if (charSet.getIndexId() == charSetId1) {
            if(flag) {
                //新建段
                CharSet newcharSet = new CharSet(charSet.getFromChar(), charSet.getToChar(),newId);
                newList.add(newcharSet);
            } else {
                //新建字符集
                CharSet newCharSet = new CharSet(charSet.getFromChar(),charSet.getToChar());
                newId = newCharSet.getIndexId();//获取新的id
                newList.add(newCharSet);
                flag = true;
            }
        }
    }
    //字符集2
    for (CharSet charSet: pCharSetTable) {
        if (charSet.getIndexId() == charSetId2) {
            //新建段
            CharSet newcharSet = new CharSet(charSet.getFromChar(), charSet.getToChar(),newId);
            newList.add(newcharSet);
        }
    }

    for (CharSet charSet:newList) {
        pCharSetTable.add(charSet);
    }
    return newId;
}
```

5）int difference(int charSetId, char c)；//字符集与字符之间的差运算
实现方法：首先在字符集表中找到所有属于该字符集的段，有四种情况。如果 c 是某一段的头，就取掉头，取余下的为一段；如果 c 在某一段的中间，就分为两段，取掉中间的 c；如果 c 是某一段的尾，就取掉尾，取余下的为一段；如果 c 不属于这一段，就不处理。最后返回新字符集 id 即可。
实现函数：

```
public static int difference(int charSetId, char c) {
    boolean flag = false;
    int newId = 0;
    for (CharSet charSet : pCharSetTable) {
        if(charSet.getIndexId() == charSetId) {
            //包含字符c的情况 分段
            if(charSet.getFromChar() < c && charSet.getToChar() > c) {
                if(flag) {
                    //新建一个段
                    CharSet newcharSet = new CharSet(charSet.getFromChar(), (char)(c-1),newId);
                    pCharSetTable.add(newcharSet);

                } else {
                    //新建字符集
                    CharSet newcharSet = new CharSet(charSet.getFromChar(), (char)(c-1));
                    newId = newcharSet.getIndexId();
                    pCharSetTable.add(newcharSet);
                    flag = true;
                }
                CharSet newcharset = new CharSet((char)(c+1),charSet.getToChar(),newId);
                pCharSetTable.add(newcharset);
            }
            else if(charSet.getFromChar() ==  c) {
                if(flag) {
                    //新建一个段
                    CharSet newcharSet = new CharSet((char)(c+1),charSet.getToChar(),newId);
                    pCharSetTable.add(newcharSet);

                } else {
                    //新建字符集
                    CharSet newcharSet = new CharSet((char)(c+1),charSet.getToChar());
                    newId = newcharSet.getIndexId();
                    pCharSetTable.add(newcharSet);
                    flag = true;
                }

            }
```

```
        else if (charSet.getToChar() == c) {
            if(flag) {
                //新建一个段
                CharSet newcharSet = new CharSet(charSet.getFromChar(),(char)(c-1),newId);
                pCharSetTable.add(newcharSet);

            } else {
                //新建字符集
                CharSet newcharSet = new CharSet(charSet.getFromChar(), (char)(c-1));
                newId = newcharSet.getIndexId();
                pCharSetTable.add(newcharSet);
                flag = true;
            }
        }
```

```
        else {
            if(flag) {
                //新建一个段
                CharSet newcharSet = new CharSet(charSet.getToChar(), charSet.getToChar(),newId);
                pCharSetTable.add(newcharSet);

            } else {
                //新建字符集
                CharSet newcharSet = new CharSet(charSet.getFromChar(), charSet.getToChar());
                newId = newcharSet.getIndexId();
                pCharSetTable.add(newcharSet);
                flag = true;
            }
        }

    }
}
return newId;
```

三、基于 NFA 的数据结构定义，按照最简 NFA 构造法，实现如下函数。

1）Graph * generateBasicNFA(DriverType driverType, int driverId);

函数作用：构造一个最简单的 NFA

实现方法：首先新建一个 graph，设置序号和状态数，然后 创建首尾状态，添加状态列表，创建连接起始状态和终止状态的边，并将边加入图的边列表。此处新增了一个 category 属性便于之后词法分析的识别。

实现函数：

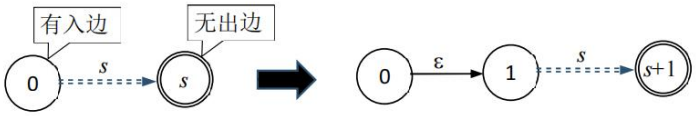
```
public static Graph generateBasicNFA(DriverType driverType,int driverId, LexemeCategory category) {
    //NFA
    Graph pNFA = new Graph();
    State pState1 = new State( stateId: 0,StateType.UNMATCH,LexemeCategory.EMPTY);
    State pState2 = new State( stateId: 1,StateType.MATCH,category);
    Edge edge = new Edge(pState1.getStateId(),pState2.getStateId(),driverId,driverType);
    //加入NFA中
    pNFA.addState(pState1);
    pNFA.addState(pState2);
    pNFA.addEdge(edge);
    return pNFA;
}
```

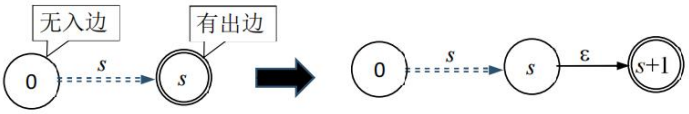
2）Graph * union(Graph *pNFA1, Graph *pNFA2); // 并运算

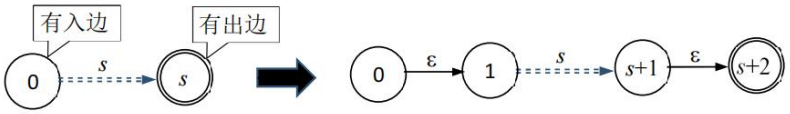
函数作用：两个 NFA 进行并运算。

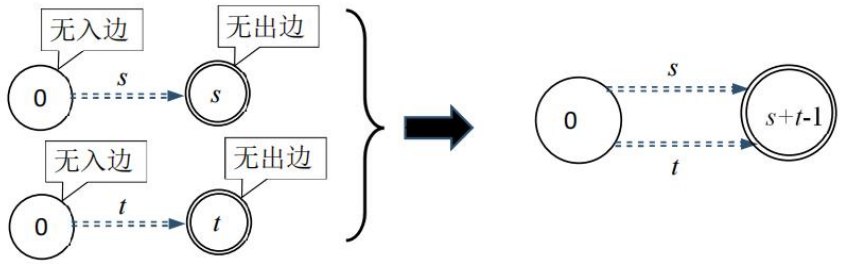
实现方法：如果起始状态有入边，就在起始状态添加一个状态，如果结束状态有出边，就在结束状态添加一个状态。创建新的图，初始化图的状态。

等价改造规则如下：









其中具体函数实现如下：

1. transForm 函数：若初始状态存在入边，则新增一个初始状态，用 ε 边连接原初始状态；若终结状态存在出边，则构造一个状态设为终结状态，所有原终结状态连接该状态。

```
0 个用法
public int transForm() {
        int transType = 0;//标志初始、结束状态是否有出入边
        // 初始状态有入边
        if (haveInsideEdge(start)) {
                // 新增一个状态
                State newStart = new State( stateId: 0, StateType.UNMATCH, LexemeCategory.EMPTY);
                addState(newStart);
                // 重新编号(序号+1)
                reNumber( index: 1);
                // 添加一条newStart到初始状态的ε边
                Edge edge = new Edge( fromState: 0,  nextState: 1, DriverType.NULL);
                pEdgeTable.add(edge);
                // 设置初始状态
                start = newStart;
                // 最低位为1表示初始状态有入边
                transType |= 1;
        }
        //结束状态有出边
        if (haveOutsideEdge(pEndStateTable)) {
                // 新增一个状态
                State newEnd = new State(pStateTable.size(), StateType.UNMATCH, LexemeCategory.EMPTY);
                addState(newEnd);
                // 添加结束状态到newEnd的ε边
                for (State state: pEndStateTable) {
                        Edge edge = new Edge(state.getStateId(), newEnd.getStateId(), DriverType.NULL);
                        pEdgeTable.add(edge);
                }
                // 清空当前结束状态
                pEdgeTable.clear();
                // 设置新的结束状态
                pEndStateTable.add(newEnd);
                // 第二低位为1表示终结状态有出边
                transType |= 2;
        }
        return transType;
}
```

haveInsideEnge()函数：判断初始状态是否有入边：

```
    public boolean haveInsideEdge(State start) {
            for (Edge edge : pEdgeTable) {
                    if (edge.getNextState() == start.getStateId()) {
                            return true;
                    }
            }
            return false;
    }
```

haveOutsideEdge 函数：判断终结状态是否有出边

```
public boolean haveOutsideEdge(ArrayList<State> stateArrayList) {
        for (Edge edge : pEdgeTable) {
                for (State state : stateArrayList) {
                        if (edge.getFromState() == state.getStateId()) {
                                return true;
                        }
                }
        }
        return false;
}
```

2. reNumber 函数：对状态和边对应的状态重新编号，确保状态有序。

```
public void reNumber(int index) {
        // 状态重新编号
        for (State state : pStateTable) {
                int stateId = state.getStateId();
                state.setStateId(stateId+ index);
        }
        // 边重新编号
        for (Edge edge : pEdgeTable) {
                int fromState = edge.getFromState();
                edge.setFromState(fromState + index);
                int nextState = edge.getNextState();
                edge.setNextState(nextState + index);
        }
}
```

3. addToTable 函数：将参数 NFA 中的所有边、状态、结束状态（重编号之后）加入到该 NFA 中。

```
0 个用法
public void addToTable(Graph g) {
        pEdgeTable.addAll(g.getpEdgeTable());
        pStateTable.addAll(g.getpStateTable());
        pEndStateTable.addAll(g.pEndStateTable);
}
```

4. mergeEnState 函数：将 pNFA1 的终结状态合并到 pNFA2 中，终结状态的序号为**最大值，即 stateNum1+stateNum2-3**

```
public void mergeEndState(Graph NFA, int endStateId) {
        ArrayList<State> newEndList = new ArrayList<>();
        for (State state: NFA.pEndStateTable) {
                for (Edge edge: NFA.getpEdgeTable()) {
                        // 找到所有到达终结状态的边
                        if (edge.getNextState() == state.getStateId()) {
                                edge.setNextState(endStateId);
                        }
                }
                pStateTable.remove(state);
                newEndList.add(state);
        }
        for (State state: newEndList) {
                pEndStateTable.remove(state);
        }
}
```

5. mergeStart 函数：将 pNFA2 的初始状态合并到 pNFA1 中，初始状态的序号为 0

```
public void mergeStart(Graph NFA) {
        State start = NFA.getStart();
        for (Edge edge: NFA.getpEdgeTable()) {
                // 找到所有从初始状态出发的边
                if (edge.getFromState() == start.getStateId()) {
                        edge.setFromState(0);
                }
        }
        pStateTable.remove(start);
}
```

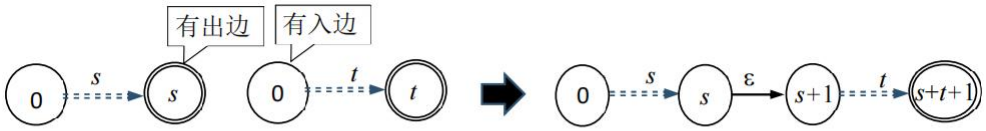
则两个 NFA 的并运算实现函数如下：对于第一个图，只改变末尾边，最终状态的下个状态和后面的状态合并。对于第二个图，改变所有边的首尾，连接到第一个图为基础的后面

```
0 个用法
public static Graph union(Graph pNFA1, Graph pNFA2) {
    Graph graph = new Graph();
    //等价改造两个NFA(根据初始状态的入边、结束状态的出边合理添加ε边)
    pNFA1.transForm();
    pNFA2.transForm();
    State start = pNFA1.getStart();//NFA1 初始状态
    graph.setStart(start);
    //状态数量
    int stateNum1 = pNFA1.getNumOfStates();
    int stateNum2 = pNFA2.getNumOfStates();
    graph.setNumOfStates(stateNum1+stateNum2-2);//合并的状态数
    //重编号NFA2
    pNFA2.reNumber( index: stateNum1-2);
    graph.addToTable(pNFA1);
    graph.addToTable(pNFA2);
    //合并终止状态
    graph.mergeEndState(pNFA1, endStateId: stateNum1+stateNum2-3);
    //合并初始状态
    graph.mergeStart(pNFA2); //pNFA1初始状态->pFA2初始状态
    return graph;
}
```

3）Graph * product(Graph *pNFA1, Graph *pNFA2); // 连接运算

函数作用：对两个 NFA 进行连接运算

实现思路：NFA 的连接运算分为两种情况，情况之一是前一个图的接收状态有出边，后一个图的初状态有入边，则需要中间添加一个空状态来防止倒灌；其余的情况则是前一个的接收状态和后一个的初状态合二为一，然后根据状态 Id 的变化添加 Id 和添加边即可。最后返回一个新建的图。改造规则如下：



（a）s 的 NFA 的结束状态 s 有出边且 t 的 NFA 的开始状态 0 有入边

（b）其他情形

removeStateById 函数：移除 pNFA 中 StateId 对应的状态

```
public void removeStateById(int stateId) {
        Iterator<State> iterator = pStateTable.iterator();
        while (iterator.hasNext()){
                State state = iterator.next();
                if (state.getStateId() == stateId) {
                        iterator.remove();
                }
        }
          for (State state : pStateTable) {
                  if (state.getStateId() == stateId) {
                          pStateTable.remove(state);
                  }
          }
        numOfStates --;
}
```

实现函数：

```
public static Graph product(Graph pNFA1, Graph pNFA2){
    Graph graph = new Graph();
    //获得PNFA1初始状态
    graph.setStart(pNFA1.getStart());
    graph.addToTable(pNFA1);
    //获取状态数量
    int stateNum1 = pNFA1.getNumOfStates();;
    int stateNum2 = pNFA2.getNumOfStates();
    //判断是否有出入边
    if(pNFA1.haveOutsideEdge(pNFA1.getpEndStateTable()) && pNFA2.haveInsideEdge(pNFA1.getStart())) {
       //NFA2重新编号
        pNFA2.reNumber(stateNum1);
        graph.setNumOfStates(stateNum1+stateNum2);
        // 添加ε边
        for(State state:pNFA1.getpEndStateTable()) {
            Edge edge = new Edge(state.getStateId(),pNFA2.getStart().getStateId(),DriverType.NULL);
            graph.addEdge(edge);
        }
    } else {
        //NFA2重新编号
        pNFA2.reNumber( index: stateNum1-1);
        //合并NF1的终止和NFA2的初始状态
        pNFA2.removeStateById( stateId: stateNum1-1);
        graph.setNumOfStates(stateNum1+stateNum2-1);
    }
    for(State state: graph.getpStateTable()) {
        state.setType(StateType.UNMATCH);
    }
    graph.setpEndStateTable(new ArrayList<>());
    //加上pNFA->pNFA2的边
    graph.addToTable(pNFA2);
    return graph;
}
```

4）Graph * plusClosure(Graph *pNFA) //正闭包运算

函数作用：实现除了 0 个以外的图重复

实现思路：因为没有 0 到结束状态的干扰，可以直接添加一条边，从接收状态到初状态，转换条件为空。

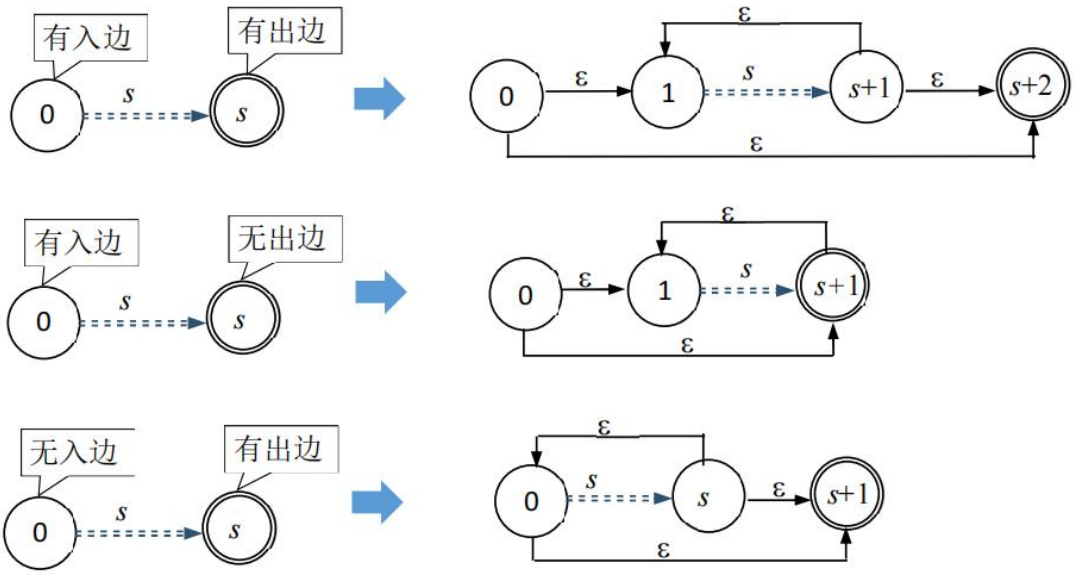
实现函数：

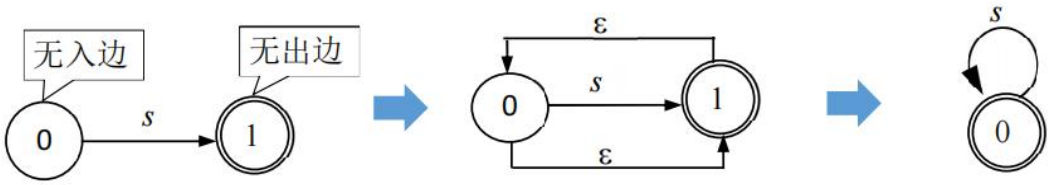
```
public static Graph plusClosure(Graph pNFA) {
    Graph graph = new Graph();
    //构建初始状态、状态数
    graph.setStart(pNFA.getStart());
    graph.setNumOfStates(pNFA.getNumOfStates());
    graph.addToTable(pNFA);
    //构建PNFA从终止状态到初始状态的边
    pNFA.addEndEdgeToOtherState(pNFA.getStart());
    return graph;

}
```

5）Graph * closure(Graph *pNFA) // 闭包运算

函数作用：包含 0 次和很多次的图的重复

实现思路：在 4 的基础上增加一个从初始状态到接收状态的边，需要考虑初状态是否有入边，接受状态是否有出边，即首先进行改造，改造规则如下：





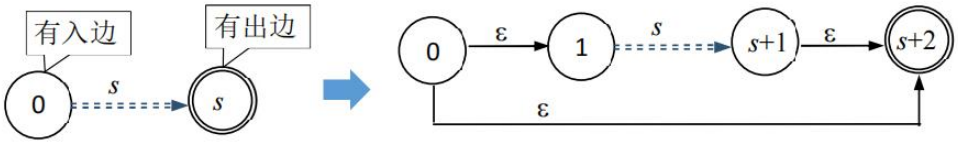
实现函数：

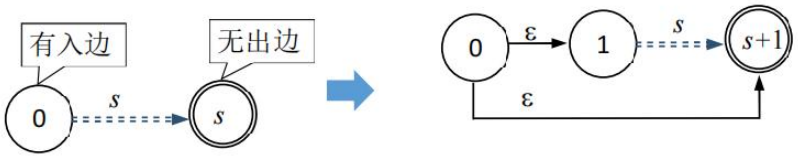
```
0个用法
public static Graph closure(Graph pNFA) {
    //判断初始、终止状态有无出入边并进行改造
  int transType = pNFA.transForm();
  if(transType == 0) {
      //没有出入边 添加终止状态到初始状态的ε边
      pNFA.mergeEndState(pNFA, endStateId: 0);
      //状态数量、匹配状态
      pNFA.setNumOfStates(pNFA.getpStateTable().size());
      for (State state:pNFA.getpStateTable()) {
          if(state.getStateId() == 0) {
              state.setType(StateType.MATCH); //匹配状态
              pNFA.addEndState(state);//终止状态
          }
      }
  } else {
      // 添加从原开始状态到原终结状态的ε边 遍历终止状态集合
      for (State state:pNFA.getpEndStateTable()) {
          Edge edge = new Edge(pNFA.getStart().getStateId(),state.getStateId(),DriverType.NULL);
          pNFA.addEdge(edge);
      }
      // 添加从开始到终结的ε边
      ArrayList<Edge> edgeList = new ArrayList<>();
      //遍历终止状态集合 添加从初始状态到终止状态的边
      for(State state: pNFA.getpEndStateTable()) {
          Edge edge = new Edge(pNFA.getStart().getStateId(),state.getStateId(),DriverType.NULL);
          // 由于涉及到更改结束状态集合的操作，因此需要先遍历再添加边
          edgeList.add(edge);
      }
      for(Edge edge:edgeList) {
          pNFA.addEdge(edge);
      }
  }
    //新建NFA
    Graph graph = new Graph(pNFA);
    return graph;
}
```

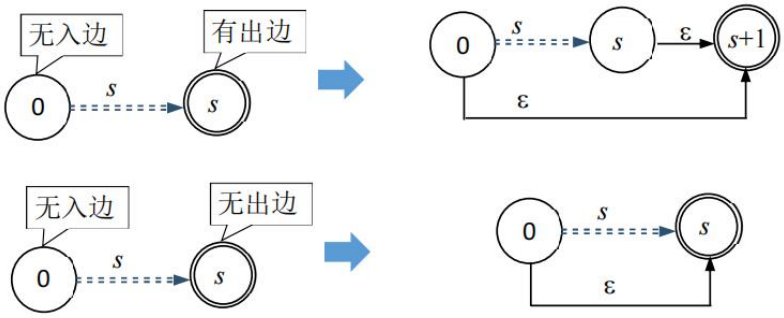
6）Graph * zeroOrOne(Graph *pNFA)； // 0 或者 1 个运算。

函数作用：进行图的一次或者 0 次运算

实现思路：在实现之前先判断有误出入边并进行等价改造，再添加一条初状态到终止状态的边。







实现函数：

```
public static Graph zeroOrOne(Graph pNFA) {
    Graph graph = new Graph();
    graph.setStart(pNFA.getStart());
    graph.addToTable(pNFA);
    //有误出入边进行等价改造
    graph.transForm();
    //状态数
    graph.setNumOfStates(pNFA.getpStateTable().size());
    // 添加从开始到终结的ε边
    for(State state:graph.getpEndStateTable()) {
        Edge edge = new Edge(graph.getStart().getStateId(),state.getStateId(),DriverType.NULL);
        graph.addEdge(edge);
    }
    return graph;
}
```

三、基于 NFA 数据结构定义，实现如下函数。

1）子集构造法

① list<int>move(Graph* pNFA, list<int> stateIdTable, int driverId)

函数作用：找到从一个状态集合通过某个转换条件可以跳转到的下一个状态集合

实现思路：遍历状态列表中的每一个状态，在图的所有边中遍历，如果出现开始状态是存在对应集合中，并且是该引导条件 Id，则将该状态 id 存入 List 中并返回。

实现函数：

```
public static ArrayList<Integer> move(Graph graph, int stateId, int driveId) {
    ArrayList<Edge> edgeList = graph.getpEdgeTable();//边的集合
    ArrayList<Integer> stateList = new ArrayList<>();//状态集合
    for(Edge edge:edgeList) {
        if(edge.getFromState() == stateId && edge.getDriverId() == driveId) {
            int nextState = edge.getNextState();//下一状态
            stateList.add(nextState);
        }
    }
    return stateList;
}
```

② list<int> ε _closure(Graph* pNFA, list<int> stateIdTable)

函数作用：得到状态集合中起始状态是该状态且驱动字符是空变换的集合

实现思路：**将参数中传入的状态集合保存下来**(resultStateList、currentStateList)，**从当前状态集合 currentStateList 开始查找到下一状态是否有 ε 转换**，如果有的存储状态的 Id 到 resultStateList 中并进行返回。**因为可能会出现连续的多个空转移，故可在最外层进行 while 循环查找，直到找全其空转换状态集合为止。**

实现函数：

```
public static ArrayList<Integer> closure(Graph graph, ArrayList<Integer> stateArrayList)  {
    ArrayList<Edge> edgeList = graph.getpEdgeTable();//边的集合
    ArrayList<Integer> currentStateList = new ArrayList<>();//当前状态集合
    ArrayList<Integer> resultStateList = new ArrayList<>();//闭包运算结果->状态集合
    ArrayList<Integer> nextStateList = new ArrayList<>();//next状态集合
    //初始化
    resultStateList = stateArrayList;
    currentStateList = stateArrayList;

    while(!currentStateList.isEmpty()) {
        for (Edge edge:edgeList) {
            //查找能进行空变迁的状态集合
            if(currentStateList.contains(edge.getFromState()) && edge.getType() == DriverType.NULL) {
                if(!nextStateList.contains(edge.getNextState())) {
                    //保存空变迁指向的next状态
                    nextStateList.add(edge.getNextState());
                }
            }
        }

        //去除重复搜索的状态
        for(Integer stateId:resultStateList) {
            if(nextStateList.contains(stateId)) {
                nextStateList.remove(stateId);
            }
        }
        //去除后为next为空则闭包搜索结束
        if(nextStateList.isEmpty()){
            break;
        } else {
            //以next为当前状态继续搜索
            currentStateList = nextStateList;
            resultStateList.addAll(nextStateList);
        }
    }
    return resultStateList;
}
}
```

③ list<int>DTran(Graph*pNFA, list<int> stateIdTable,int driverId)

函数作用：将前面两个函数功能合并计算结果状态集合

实现思路：直接调用前面实现的函数并且对空转换进行多次循环

实现函数：

```
0 个用法
public static ArrayList<Integer> dTran(Graph graph, ArrayList<Integer> currentStateArrayList, int driveId) {
    ArrayList<Integer> nextStateList = new ArrayList<>();//下一状态集合
    for(Integer stateId:currentStateArrayList) {
        //先求转移状态集合
        ArrayList<Integer> stateList = move(graph,stateId,driveId);
        nextStateList.addAll(stateList);
    }
    //再求闭包运算结果
    return closure(graph,nextStateList);
}
```

2）Graph * NFA_to_DFA(Graph *pNFA)

函数作用：将 NFA 转换为 DFA

实现思路：**计算出 NFA 初状态的空转换状态集合**并且**保存整个图的驱动 id**，然后通过此状态集合，对驱动 id 进行循环，调用 DTran 函数，得到可达的**状态集合**并存入 List 中。检查是否已存在的状态，不是就将该状态集合并且为其编号，向新建的 DFA 添加这些状态。接着通过这些状态再次对驱动 id 进行循环并且得到相应的状态集合，找到这些状态集合的对应的状态 id，最后则得到了边，并将向 DFA 中添加这些边，最后返回 DFA。

实现函数：

```
0 个用法
public static Graph NFAToDFA(Graph pNFA) {
    Graph graph = new Graph();//新建DFA图
    //首先穷举NFA开始状态集合的出边，驱动字符，以及下一状态集合
    ArrayList<Integer> currentStateList = new ArrayList<>();
    currentStateList.add(pNFA.getStart().getStateId());//当前从初始状态集合开始
    currentStateList = closure(pNFA,currentStateList); //闭包结果集合

    //存储所有状态集合
    ArrayList<ArrayList<Integer>> allStateList = new ArrayList<>();
    allStateList.add(currentStateList);//存储初始状态集合
    //存储DFA驱动边集合
    ArrayList<Edge> newEdgeList = new ArrayList<>();
    //存储DFA状态集合
    ArrayList<State> newStateList = new ArrayList<>();
    //存储DFA终止状态集合
    ArrayList<State> newEndState = new ArrayList<>();
    //DFA的初始状态
    State newStart = new State( stateId: 0,StateType.UNMATCH,LexemeCategory.EMPTY);
    newStateList.add(newStart);
    graph.setStart(newStart);

    int currentStateId = 0;//当前状态Id
    ArrayList<Edge> edgeList = pNFA.getpEdgeTable();
    while(currentStateId < allStateList.size()) {
        currentStateList = allStateList.get(currentStateId);//当前状态集合
        //存储DFA驱动Id
        ArrayList<Integer> newDriverIdList = new ArrayList<>();
        //遍历NFA图
        for(Edge edge:edgeList) {
            if(currentStateList.contains(edge.getFromState()) && edge.getType() != DriverType.NULL && !newDriverIdList.contains(edge.getDriverId())) {
                //调用DTranh函数 得出每一转换的下一状态集合
                ArrayList<Integer> nextStateList = dTran(pNFA,currentStateList,edge.getDriverId());
                newDriverIdList.add(edge.getDriverId());//添加驱动
                //判断下一状态集合与前面已知的状态集合（即开始状态集合）是否相同
                if(!allStateList.contains(nextStateList)) {
```

```
                    //不相同则分配序号
                    allStateList.add(nextStateList);
                    //添加xin边
                    Edge newEdge = new Edge(currentStateId,newStateList.size(),edge.getDriverId(),edge.getType());
                    newEdgeList.add(newEdge);
                    //添加新的状态，判断是否终止，有无出边 设置DFA状态的type
                    StateType type = StateType.UNMATCH;
                    LexemeCategory category = LexemeCategory.EMPTY;
                    //遍历NFA
                    for(State state1 : pNFA.getpStateTable()) {
                        //DFA状态集合包含NFA终止状态，则状态的type=match
                        if(nextStateList.contains(state1.getStateId()) && state1.getType() == StateType.MATCH) {
                            type = StateType.MATCH;
                        }
                        //DFA状态集合中包含NFA中属性不为空的状态，则属性也为该状态的属性
                        if(state1.getCategory() != LexemeCategory.EMPTY) {
                            category = state1.getCategory();
                        }
                    }
                    State state2 = new State(newStateList.size(),type,category);//新建DFA状态
                    //判断是否为终止状态
                    if(type == StateType.MATCH) {
                        newEndState.add(state2);
                    }
                    //DFA添加状态
                    newStateList.add(state2);
                } else {
                    //如果是已有的状态集合,得到集合，添加边
                    int stateNUm = allStateList.indexOf(nextStateList);
                    Edge newEdge = new Edge(currentStateId,stateNUm,edge.getDriverId(),edge.getType());
                    newEdgeList.add(newEdge);
                }
            }
        }
        currentStateId++;
    }
    graph.setNumOfStates(currentStateId);
    graph.setpEndStateTable(newEndState);
    graph.setpEdgeTable(newEdgeList);
    return graph;
}
```

四、请以正则表达式(a|b)*abb 来测试，检查实现代码的正确性

实现思路：依次构建正则表达式的 NFA 图，再将其转换为 DFA 图

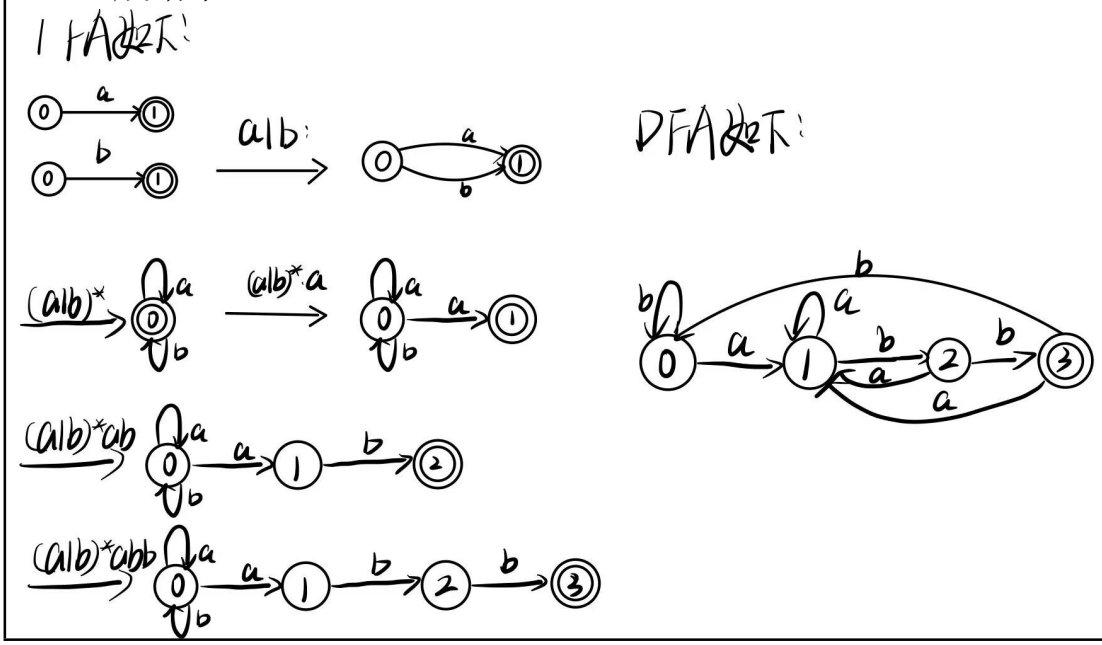
实现代码

```
public void test1() {
    //驱动字符集
    int driverIdA = Problem1.range('a','a');
    int driverIdB = Problem1.range('b','b');
    System.out.println("driverId1: "+driverIdA);
    System.out.println("driverId2: "+driverIdB);

    //构建NFA
    System.out.println("构建NFA如下：");
    //a
    Graph graphA = Problem2.generateBasicNFA(DriverType.CHAR,driverIdA,LexemeCategory.EMPTY);
    System.out.println("a");
    System.out.println(graphA);
```

```
//b
Graph graphB = Problem2.generateBasicNFA(DriverType.CHAR,driverIdB,LexemeCategory.EMPTY);
System.out.println("b");
System.out.println(graphB);
//a|b
Graph graph3 = Problem2.union(graphA,graphB);
System.out.println("a|b");
System.out.println(graph3);
//(a|b)*
Graph graph4 = Problem2.closure(graph3);
System.out.println("(a|b)*");
//(a|b)*a
graphA = Problem2.generateBasicNFA(DriverType.CHAR,driverIdA,LexemeCategory.EMPTY);
Graph graph5 = Problem2.product(graph4,graphA);
System.out.println("(a|b)*a");
System.out.println(graph5);
```

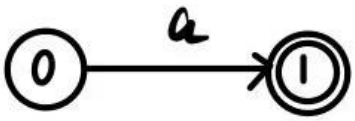
```
    //(a|b)*ab
    graphB = Problem2.generateBasicNFA(DriverType.CHAR,driverIdB,LexemeCategory.EMPTY);
    Graph graph6 = Problem2.product(graph5,graphB);
    System.out.println("(a|b)*ab");
    System.out.println(graph6);
    //(a|b)*abb
    graphB = Problem2.generateBasicNFA(DriverType.CHAR,driverIdB,LexemeCategory.EMPTY);
    Graph graph7 = Problem2.product(graph6,graphB);
    System.out.println("(a|b)*abb");
    System.out.println(graph7);

    System.out.println("构建DFA如下: ");
    Graph graphDFA = Problem3.NFAToDFA(graph7);
    System.out.println(graphDFA);


}
```

生成结果:

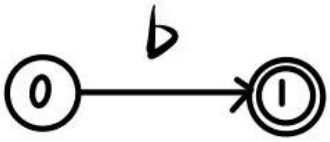


代码输出如下：

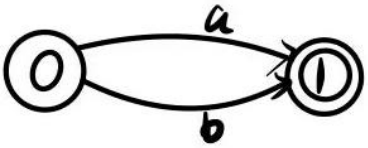
```
a
Graph{
 graphId: 0
 numOfStates: 2
 start: State{stateId=0, type=UNMATCH, category=EMPTY}
 pEndStateTable: 1
  State{stateId=1, type=MATCH, category=EMPTY}
 pEdgeTable: 1
  Edge{fromState=0, nextState=1, driverId=0, type=CHAR}
 pStateTable: 2
  State{stateId=0, type=UNMATCH, category=EMPTY}
  State{stateId=1, type=MATCH, category=EMPTY}
}
```




```
b
Graph{
 graphId: 1
 numOfStates: 2
 start: State{stateId=0, type=UNMATCH, category=EMPTY}
 pEndStateTable: 1
  State{stateId=1, type=MATCH, category=EMPTY}
 pEdgeTable: 1
  Edge{fromState=0, nextState=1, driverId=1, type=CHAR}
 pStateTable: 2
  State{stateId=0, type=UNMATCH, category=EMPTY}
  State{stateId=1, type=MATCH, category=EMPTY}
}
```

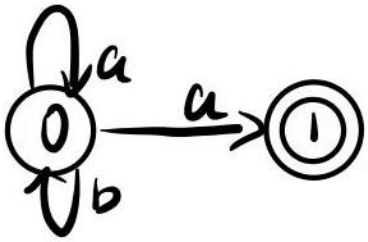



```
a|b
Graph{
 graphId: 2
 numOfStates: 2
 start: State{stateId=0, type=UNMATCH, category=EMPTY}
 pEndStateTable: 1
  State{stateId=1, type=MATCH, category=EMPTY}
 pEdgeTable: 2
  Edge{fromState=0, nextState=1, driverId=0, type=CHAR}
  Edge{fromState=0, nextState=1, driverId=1, type=CHAR}
 pStateTable: 2
  State{stateId=0, type=UNMATCH, category=EMPTY}
  State{stateId=1, type=MATCH, category=EMPTY}
}
```

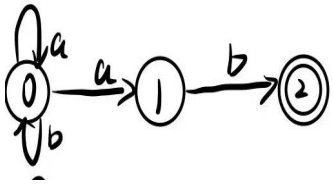



```
(a|b)*
Graph{
 graphId: 3
 numOfStates: 1
 start: State{stateId=0, type=MATCH, category=EMPTY}
 pEndStateTable: 1
  State{stateId=0, type=MATCH, category=EMPTY}
 pEdgeTable: 2
  Edge{fromState=0, nextState=0, driverId=0, type=CHAR}
  Edge{fromState=0, nextState=0, driverId=1, type=CHAR}
 pStateTable: 1
  State{stateId=0, type=MATCH, category=EMPTY}
}
```

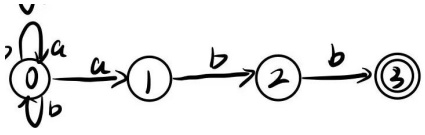
```
(a|b)*a
Graph{
 graphId: 5
 numOfStates: 2
 start: State{stateId=0, type=UNMATCH, category=EMPTY}
 pEndStateTable: 1
  State{stateId=1, type=MATCH, category=EMPTY}
 pEdgeTable: 3
  Edge{fromState=0, nextState=0, driverId=0, type=CHAR}
  Edge{fromState=0, nextState=0, driverId=1, type=CHAR}
  Edge{fromState=0, nextState=1, driverId=0, type=CHAR}
 pStateTable: 2
  State{stateId=0, type=UNMATCH, category=EMPTY}
  State{stateId=1, type=MATCH, category=EMPTY}
}
```




```
(a|b)*ab
Graph{
 graphId: 7
 numOfStates: 3
 start: State{stateId=0, type=UNMATCH, category=EMPTY}
 pEndStateTable: 1
  State{stateId=2, type=MATCH, category=EMPTY}
 pEdgeTable: 4
  Edge{fromState=0, nextState=0, driverId=0, type=CHAR}
  Edge{fromState=0, nextState=0, driverId=1, type=CHAR}
  Edge{fromState=0, nextState=1, driverId=0, type=CHAR}
  Edge{fromState=1, nextState=2, driverId=1, type=CHAR}
 pStateTable: 3
  State{stateId=0, type=UNMATCH, category=EMPTY}
  State{stateId=1, type=UNMATCH, category=EMPTY}
  State{stateId=2, type=MATCH, category=EMPTY}
}
```

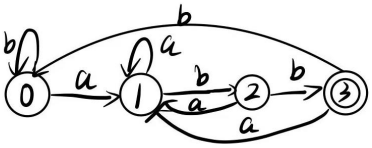



```
(a|b)*abb
Graph{
 graphId: 9
 numOfStates: 4
 start: State{stateId=0, type=UNMATCH, category=EMPTY}
 pEndStateTable: 1
  State{stateId=3, type=MATCH, category=EMPTY}
 pEdgeTable: 5
  Edge{fromState=0, nextState=0, driverId=0, type=CHAR}
  Edge{fromState=0, nextState=0, driverId=1, type=CHAR}
  Edge{fromState=0, nextState=1, driverId=0, type=CHAR}
  Edge{fromState=1, nextState=2, driverId=1, type=CHAR}
  Edge{fromState=2, nextState=3, driverId=1, type=CHAR}
 pStateTable: 4
  State{stateId=0, type=UNMATCH, category=EMPTY}
  State{stateId=1, type=UNMATCH, category=EMPTY}
  State{stateId=2, type=UNMATCH, category=EMPTY}
  State{stateId=3, type=MATCH, category=EMPTY}
}
```



```
构建DFA如下:
Graph{
 graphId: 10
 numOfStates: 4
 start: State{stateId=0, type=UNMATCH, category=EMPTY}
 pEndStateTable: 1
  State{stateId=3, type=MATCH, category=EMPTY}
 pEdgeTable: 8
  Edge{fromState=0, nextState=1, driverId=0, type=CHAR}
  Edge{fromState=0, nextState=0, driverId=1, type=CHAR}
  Edge{fromState=1, nextState=1, driverId=0, type=CHAR}
  Edge{fromState=1, nextState=2, driverId=1, type=CHAR}
  Edge{fromState=2, nextState=1, driverId=0, type=CHAR}
  Edge{fromState=2, nextState=3, driverId=1, type=CHAR}
  Edge{fromState=3, nextState=1, driverId=0, type=CHAR}
  Edge{fromState=3, nextState=0, driverId=1, type=CHAR}
 pStateTable: 4
}
```

DFA如下:



二、以 TINY 语言的词法来验证程序代码的正确性。

① TINY 语言的详细定义：

TINY 的程序结构很简单，它在语法上与 Ada 或 Pascal 的语法相似：仅是一个由分号分隔开的语句序列。另外，它既无过程也无声明。所有的变量都是整型变量，通过对其赋值可较轻易地声明变量。

它只有两个控制语句：if 语句和 repeat 语句，这两个控制语句本身也可包含语句序列。if 语句有一个可选的 else 部分且必须由关键字 end 结束。除此之外，read 语句和 write 语句完成输入/输出。在{}中可以有注释，但注释不能嵌套。

TINY 的表达式也局限于布尔表达式和整型算术表达式。布尔表达式由对两个算术表达式的比较组成，该比较使用<与=比较算符。算术表达式可以包括整型常数、变量、参数以及 4 个整型算符+、一、、/，此外还有一般的数学属性。布尔表达式可能只作为测试出现在控制语句中——而没有布尔型变量、赋值或 I / O。

② TINY 语言特点总结：

a.语句序列用分号隔开

b.所有变量都是整形变量，且不需要声明

c.只有两个控制语句，if 和 repeat

d.if 判断语句必须以 end 结束，且有可选的 else 语句

e.read 和 write 完成输入输出

f.{}表示注释，但不允许嵌套注释

g.有<和=两个比较运算符 h.有+、-、、/简单运算符

③ 关键字表：

词法单元有{ID,单词字母(小写)}(其中识别的单词字母中有可能有 TINY 保

留字 if,then,else,end,repeat,until,read,write)、{NUM,数字}、{COMMENT,注释}、{ASSIGN,特殊符号:=}、{SYMBOL,特殊符号±*/=<();}、{ERROR,错误符号}。

④ 构建如下：

1.构建字符集表

```
public void test2() {
    //构建字符集
    int charSetLetterLower = Problem1.range('a','z');   //indexId = 0 segmentId = 0
    int charSetLetterUpper = Problem1.range('A','Z');   //indexId = 1 segmentId = 1
    int charSetDigit = Problem1.range('0','9');         //indexId = 2 segmentId = 2
    //字母集
    int charSetLetter = Problem1.union(charSetLetterLower,charSetLetterUpper);  //indexId = 3, ['a','Z'],segmentId = 3

    ArrayList<Integer> charSetList = new ArrayList<>();
    for(char ch = 'a'; ch <= 'z'; ch++) {
        charSetList.add(Problem1.range(ch,ch));         //给每一个小写字母添加一个字符集，添加成段
    }
    //System.out.println(charSetList);  4-29
```

```
//定义运算符
int charSetPlus = Problem1.range('+', '+');
int charSetSub = Problem1.range('-', '-');
int charSetMul = Problem1.range('*', '*');
int charSetDiv = Problem1.range('/', '/');
int charSetEqual = Problem1.range('=', '=');
int charSetLt = Problem1.range('<', '<');
int charSetLeftBracket = Problem1.range('(', '(');
int charSetRightBracket = Problem1.range(')', ')');
int charSetSemicolon = Problem1.range(';', ';');
int charSetColon = Problem1.range(':', ':');
int charSetSpace = Problem1.range(' ', ' ');
int charSetLeftK = Problem1.range('{', '{');
int charSetRightK = Problem1.range('}', '}');
```

2、构建关键字的 NFA

1） if

```
//关键字if
Graph graphI = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(8),LexemeCategory.EMPTY);
Graph graphF = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(5),LexemeCategory.KEYWORD);
Graph graphIF = Problem2.product(graphI,graphF);
```

2） then

```
//关键字then
Graph graphT = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(19),LexemeCategory.EMPTY);
Graph graphH = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(7),LexemeCategory.EMPTY);
Graph graphE = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(4),LexemeCategory.EMPTY);
Graph graphN = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(13),LexemeCategory.KEYWORD);
Graph graphTH = Problem2.product(graphT,graphH);
Graph graphTHE = Problem2.product(graphTH,graphE);
Graph graphTHEN= Problem2.product(graphTHE,graphN);
```

3） else

```
//关键字else
graphE = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(4),LexemeCategory.EMPTY);
Graph graphL = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(11),LexemeCategory.EMPTY);
Graph graphS = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(18),LexemeCategory.EMPTY);
Graph graphE1 = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(4),LexemeCategory.KEYWORD);
Graph graphEL = Problem2.product(graphE,graphL);
Graph graphELS = Problem2.product(graphEL,graphS);
Graph graphELSE = Problem2.product(graphELS,graphE1);
```

4） end

```
//end
graphE = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(4),LexemeCategory.EMPTY);
graphN = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(13),LexemeCategory.EMPTY);
Graph graphD = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(3),LexemeCategory.KEYWORD);
Graph graphEN = Problem2.product(graphE,graphN);
Graph graphEND = Problem2.product(graphEN,graphD);
```

5） repeat

```
Graph graphR = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(17),LexemeCategory.EMPTY);
graphE = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(4),LexemeCategory.EMPTY);
Graph graphP = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(15),LexemeCategory.EMPTY);
graphE1 =Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(17),LexemeCategory.EMPTY);
Graph graphA = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(0),LexemeCategory.EMPTY);
graphT = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(19),LexemeCategory.KEYWORD);
Graph graphRE = Problem2.product(graphR,graphE);
Graph graphREP = Problem2.product(graphRE,graphP);
Graph graphREPE = Problem2.product(graphREP,graphE1);
Graph graphREPEA = Problem2.product(graphREPE,graphA);
Graph graphREPEAT = Problem2.product(graphREPEA,graphT);
```

6） until

```
Graph graphU = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(20),LexemeCategory.EMPTY);
graphN = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(13),LexemeCategory.EMPTY);
graphT = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(19),LexemeCategory.EMPTY);
graphI =Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(8),LexemeCategory.EMPTY);
graphL = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(11),LexemeCategory.KEYWORD);
Graph graphUN = Problem2.product(graphU,graphN);
Graph graphUNT = Problem2.product(graphUN,graphT);
Graph graphUNTI = Problem2.product(graphUNT,graphI);
Graph graphUNTIL = Problem2.product(graphUNTI,graphL);
```

7） read

```
graphR = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(17),LexemeCategory.EMPTY);
graphE = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(4),LexemeCategory.EMPTY);
graphA = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(0),LexemeCategory.EMPTY);
graphD = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(3),LexemeCategory.KEYWORD);
graphRE = Problem2.product(graphR,graphE);
Graph graphREA = Problem2.product(graphRE,graphA);
Graph graphREAD = Problem2.product(graphREA,graphD);
```

8） write

```
Graph graphW = Problem2.generateBasicNFA(DriverType.CHAR, charSetList.get(22), LexemeCategory.EMPTY)
graphR = Problem2.generateBasicNFA(DriverType.CHAR, charSetList.get(17), LexemeCategory.EMPTY);
graphI = Problem2.generateBasicNFA(DriverType.CHAR, charSetList.get(8), LexemeCategory.EMPTY);
graphT = Problem2.generateBasicNFA(DriverType.CHAR, charSetList.get(19), LexemeCategory.EMPTY);
graphE = Problem2.generateBasicNFA(DriverType.CHAR, charSetList.get(4), LexemeCategory.KEYWORD);
Graph graphWR = Problem2.product(graphW, graphR);
Graph graphWRI = Problem2.product(graphWR, graphI);
Graph graphWRIT = Problem2.product(graphWRI, graphT);
Graph graphWRITE = Problem2.product(graphWRIT, graphE);
```

3、构建专用符号

```
//+
Graph graphPlus = Problem2.generateBasicNFA(DriverType.CHAR,charSetPlus,LexemeCategory.NUMERIC_OPERATOR);
//-
Graph graphSub = Problem2.generateBasicNFA(DriverType.CHAR,charSetSub,LexemeCategory.NUMERIC_OPERATOR);
//*
Graph graphMul = Problem2.generateBasicNFA(DriverType.CHAR,charSetMul,LexemeCategory.NUMERIC_OPERATOR);
// /
Graph graphDiv = Problem2.generateBasicNFA(DriverType.CHAR,charSetDiv,LexemeCategory.NUMERIC_OPERATOR);
//<
Graph graphLt = Problem2.generateBasicNFA(DriverType.CHAR,charSetLt,LexemeCategory.COMPARE_CONST);
// =
Graph graphEQ = Problem2.generateBasicNFA(DriverType.CHAR,charSetEqual,LexemeCategory.LOGIC_OPERATOR);
// (
Graph graphLeftBracket = Problem2.generateBasicNFA(DriverType.CHAR,charSetLeftBracket,LexemeCategory.LOGIC_OPERATOR);
// )
Graph graphRightBracket = Problem2.generateBasicNFA(DriverType.CHAR,charSetRightBracket,LexemeCategory.LOGIC_OPERATOR);
// ;
Graph graphSemicolon = Problem2.generateBasicNFA(DriverType.CHAR,charSetSemicolon,LexemeCategory.LOGIC_OPERATOR);
// :=
Graph graphColon = Problem2.generateBasicNFA(DriverType.CHAR,charSetColon,LexemeCategory.LOGIC_OPERATOR);
// ID =letter+
Graph graphLetter = Problem2.generateBasicNFA(DriverType.CHARSET,charSetLetter,LexemeCategory.ID);
Graph graphID = Problem2.plusClosure(graphLetter);
//Num = digit+
Graph graphDigit = Problem2.generateBasicNFA(DriverType.CHARSET,charSetDigit,LexemeCategory.INTEGER_CONST);
Graph graphNum = Problem2.plusClosure(graphDigit);
// 空格
Graph graphSpace = Problem2.generateBasicNFA(DriverType.CHARSET,charSetSpace,LexemeCategory.SPACE_CONST);
//注释用{...}围起来，且不能嵌套。
Graph graphLeftK = Problem2.generateBasicNFA(DriverType.CHAR,charSetLeftK,LexemeCategory.EMPTY);
Graph graphLetter1 = Problem2.generateBasicNFA(DriverType.CHARSET,charSetLetter,LexemeCategory.EMPTY);
Graph graphContent = Problem2.closure(graphLetter1);  //闭包运算求内容
Graph graphRightK = Problem2.generateBasicNFA(DriverType.CHARSET,charSetRightK,LexemeCategory.NOTE);
Graph graphCOMMENT = Problem2.product(Problem2.product(graphLeftK,graphContent),graphRightK);
//error 错误信息
graphE = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(4),LexemeCategory.EMPTY);
graphR = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(17),LexemeCategory.EMPTY);
Graph graphR1 = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(17),LexemeCategory.EMPTY);
Graph graphO = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(14),LexemeCategory.EMPTY);
Graph graphR2 = Problem2.generateBasicNFA(DriverType.CHAR,charSetList.get(17),LexemeCategory.ERROR);
Graph graphER = Problem2.union(graphE,graphR);
Graph graphERR = Problem2.union(graphER,graphR1);
Graph graphERRO = Problem2.union(graphERR,graphO);
Graph graphERROR = Problem2.union(graphERRO,graphR2);
```

1. 构建 NFA

```
//构建NFA
Graph graph1 = Problem2.union(graphIF,graphTHEN);
Graph graph2 = Problem2.union(graph1,graphELSE);
Graph graph3 = Problem2.union(graph2,graphEND);
Graph graph4 = Problem2.union(graph3,graphREPEAT);
Graph graph5 = Problem2.union(graph4,graphUNTIL);
Graph graph6 = Problem2.union(graph5,graphREAD);
Graph graph7 = Problem2.union(graph6,graphWRITE);
Graph graph8 = Problem2.union(graph7,graphPlus);
Graph graph9 = Problem2.union(graph8,graphSub);
Graph graph10 = Problem2.union(graph9,graphMul);
Graph graph11 = Problem2.union(graph10,graphDiv);
Graph graph12 = Problem2.union(graph11,graphEQ);
Graph graph13 = Problem2.union(graph12, graphLt);
Graph graph14 = Problem2.union(graph13, graphLeftBracket);
Graph graph15 = Problem2.union(graph14, graphRightBracket);
Graph graph16 = Problem2.union(graph15, graphSemicolon);
Graph graph17 = Problem2.union(graph16, graphColon);
Graph graph18 = Problem2.union(graph17, graphID);
Graph graph19 = Problem2.union(graph18, graphNum);
Graph graph20 = Problem2.union(graph19, graphSpace);
Graph graph21 = Problem2.union(graph20,graphERROR);
Graph graphNFA = Problem2.union(graph21, graphCOMMENT);
```

2. 转换为 DFA

```
//构建DFA
Graph graphDFA = Problem3.NFAToDFA(graphNFA);
System.out.println(graphD);
```

3. 结果输出

```
Graph{
 graphId: 110
 numOfStates: 27
 start: State{stateId=0, type=UNMATCH, category=EMPTY}
 pEndStateTable: 6
  State{stateId=3, type=MATCH, category=NOTE}
  State{stateId=4, type=MATCH, category=NOTE}
  State{stateId=7, type=MATCH, category=NOTE}
  State{stateId=8, type=MATCH, category=NOTE}
  State{stateId=9, type=MATCH, category=NOTE}
  State{stateId=25, type=MATCH, category=NOTE}
```

```
 pEdgeTable: 47
  Edge{fromState=0, nextState=1, driverId=12, type=CHAR}
  Edge{fromState=0, nextState=2, driverId=23, type=CHAR}
  Edge{fromState=0, nextState=3, driverId=8, type=CHAR}
  Edge{fromState=0, nextState=4, driverId=21, type=CHAR}
  Edge{fromState=0, nextState=5, driverId=24, type=CHAR}
  Edge{fromState=0, nextState=6, driverId=26, type=CHAR}
  Edge{fromState=0, nextState=7, driverId=30, type=CHAR}
  Edge{fromState=0, nextState=7, driverId=31, type=CHAR}
  Edge{fromState=0, nextState=7, driverId=32, type=CHAR}
  Edge{fromState=0, nextState=7, driverId=33, type=CHAR}
  Edge{fromState=0, nextState=7, driverId=34, type=CHAR}
  Edge{fromState=0, nextState=7, driverId=35, type=CHAR}
  Edge{fromState=0, nextState=7, driverId=36, type=CHAR}
  Edge{fromState=0, nextState=7, driverId=37, type=CHAR}
  Edge{fromState=0, nextState=7, driverId=38, type=CHAR}
  Edge{fromState=0, nextState=7, driverId=39, type=CHAR}
  Edge{fromState=0, nextState=8, driverId=3, type=CHARSET}
  Edge{fromState=0, nextState=9, driverId=2, type=CHARSET}
  Edge{fromState=0, nextState=7, driverId=40, type=CHARSET}
  Edge{fromState=0, nextState=7, driverId=18, type=CHAR}
  Edge{fromState=0, nextState=10, driverId=41, type=CHAR}
  Edge{fromState=1, nextState=7, driverId=9, type=CHAR}
  Edge{fromState=2, nextState=11, driverId=11, type=CHAR}
  Edge{fromState=3, nextState=12, driverId=15, type=CHAR}
  Edge{fromState=3, nextState=13, driverId=17, type=CHAR}
  Edge{fromState=4, nextState=14, driverId=8, type=CHAR}
  Edge{fromState=5, nextState=15, driverId=17, type=CHAR}
  Edge{fromState=6, nextState=16, driverId=21, type=CHAR}
  Edge{fromState=10, nextState=10, driverId=3, type=CHARSET}
  Edge{fromState=10, nextState=7, driverId=42, type=CHARSET}
  Edge{fromState=11, nextState=17, driverId=8, type=CHAR}
  Edge{fromState=12, nextState=18, driverId=22, type=CHAR}
  Edge{fromState=13, nextState=7, driverId=7, type=CHAR}
  Edge{fromState=14, nextState=19, driverId=19, type=CHAR}
  Edge{fromState=14, nextState=20, driverId=4, type=CHAR}
  Edge{fromState=15, nextState=21, driverId=23, type=CHAR}
  Edge{fromState=16, nextState=22, driverId=12, type=CHAR}
  Edge{fromState=17, nextState=7, driverId=17, type=CHAR}
  Edge{fromState=18, nextState=7, driverId=8, type=CHAR}
  Edge{fromState=19, nextState=23, driverId=21, type=CHAR}
  Edge{fromState=20, nextState=7, driverId=7, type=CHAR}
  Edge{fromState=21, nextState=24, driverId=12, type=CHAR}
  Edge{fromState=22, nextState=25, driverId=23, type=CHAR}
  Edge{fromState=23, nextState=26, driverId=4, type=CHAR}
  Edge{fromState=24, nextState=7, driverId=15, type=CHAR}
  Edge{fromState=25, nextState=7, driverId=8, type=CHAR}
  Edge{fromState=26, nextState=7, driverId=23, type=CHAR}
 pStateTable: 27
}
```

收获与体会：

（1）通过本次试验对最简生成 NFA 的方法有了进一步的理解，并且在编码实现的过程中可以逐渐熟练掌握最简生成 NFA 的方法。

（2）进一步加深对 NFA 子集构造法转换为 DFA 也有了进一步的了解，在实现空转换函数和跳转函数的时候，对子集构造法有了进一步的学习和记忆。

（3）对于 NFA 和 DFA 的作用有了进一步的了解

（4）对字符集的构造有了一定的了解，进一步熟练了字符和字符，字符集和字符集，字符和字符集之间的运算。

（5）进一步加深对如何通过正则表达式构建 DFA 有更深的认知。

实验成绩	