编译原理第一次实验报告

概述

这是编译原理的第一次实验报告

基本部分是第一部分:基于C语言子集的词法分析程序。这部分选取了C语言的一个子集给出了词法分析程序，对给定的一段**C**语言代码**(**元素均在此子集中**)**，输出是该段代码的**token**序列，以便后面的文法分析使用。

扩展部分是第二部分:LEX的部分实现。这部分对给定的一个正则表达式，输出了其**DFA**的转化表，由于时间所限，没有实现词法分析程序的生成程序，并且NFA到DFA部分仅支持a，b两个字母构成的字母表。

实验环境

实验环境是**Windows10**系统，使用的语言是**Java**语言

第一部分: 基于C语言子集的词法分析程序

1. 假设与依赖

> 假设了实验使用的C语言子集
>
> 两个输入之间均用1个空格隔开

Token	包含的输入
关键字	break case char const continue default do double else enum float for goto if int long main short signed sizoef static switch unsigned void
操作符	( ) [ ] . ! + ++ - -- * / % < <= > >= = == != & && || ,
分隔符	; { }
注释符	// /* */ “ ” ‘ ’
忽略的字符	\n \t blank
整数	所有整数(仅限正数)
浮点数	所有浮点数(仅限正数)
标识符	由数字,大小写英文字母和和下划线组成的并且由字母或下划线开头的任意字符串(不包含关键字)

2. 思路与方法

> 本部分叙述了实验的正则表达式构造和主要思路

2.1 正则表达式的构造

分成两个部分

第一部分是有限长度，有限数目的**token**

Token	正则表达式
关键字	break|case|char|const|continue|default|do|double|else|enum|float|for|goto|if|int|long|main|short|signed|sizoef|s
操作符	()|[] .|!|+|++|-|--|*|/|%|<|<=|>|>=|=|==|!=|&|&&||| ,
分隔符	;|{|}|"|'||'
注释符	// | /* | */

第二个部分是无限长度，无限数目的**token**

Token	正则表达式
整数	*digit*(*digit*)
浮点数	*digit* (*digit*)* . *digit* (*digit*)*
标识符	*letter*(*letter*|*digit*)*

其中*digit*->0|1|2|3|4|5|6|7|8|9，letter->a|b|c…|z|A|B|C…|Z|_

2.2 思路叙述

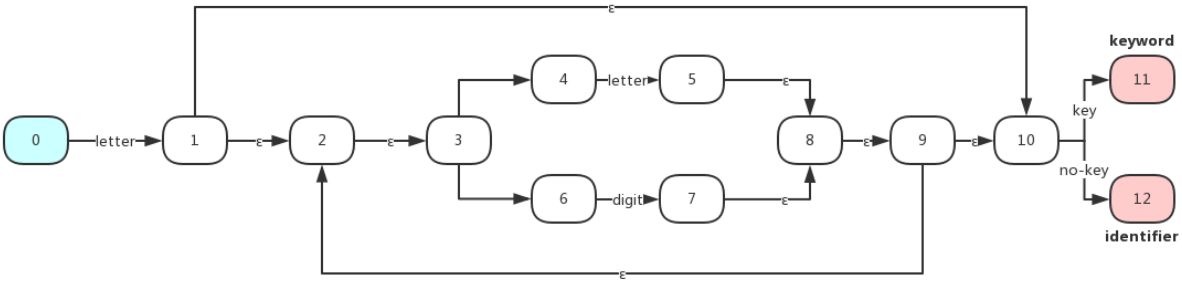
思路如下:

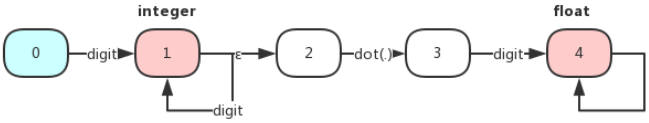
- 初始化常量
- 将文本的C程序代码读取成为字节数组
- 从左至右扫描一遍数组，利用**DFA**确定其归属的Token种类，并与原来的词法元素合并成为一个token，写入输出字符串列表中
- 将输出字符串列表中的字符串按照从前往后的顺序输出到文本

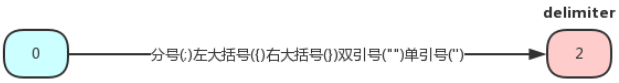
3. 自动状态机**DFA**构造描述

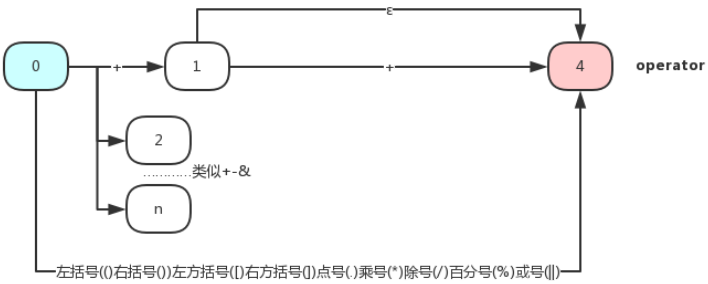
> 本部分详细叙述了从Regular Expression到DFA的过程

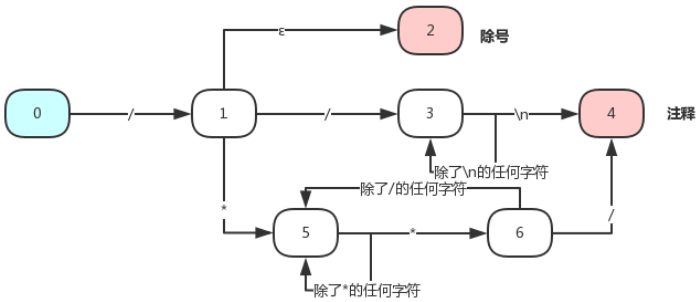
3.1 RE->NFA


keyword
identifier
letter
digit
key
no-key
ε

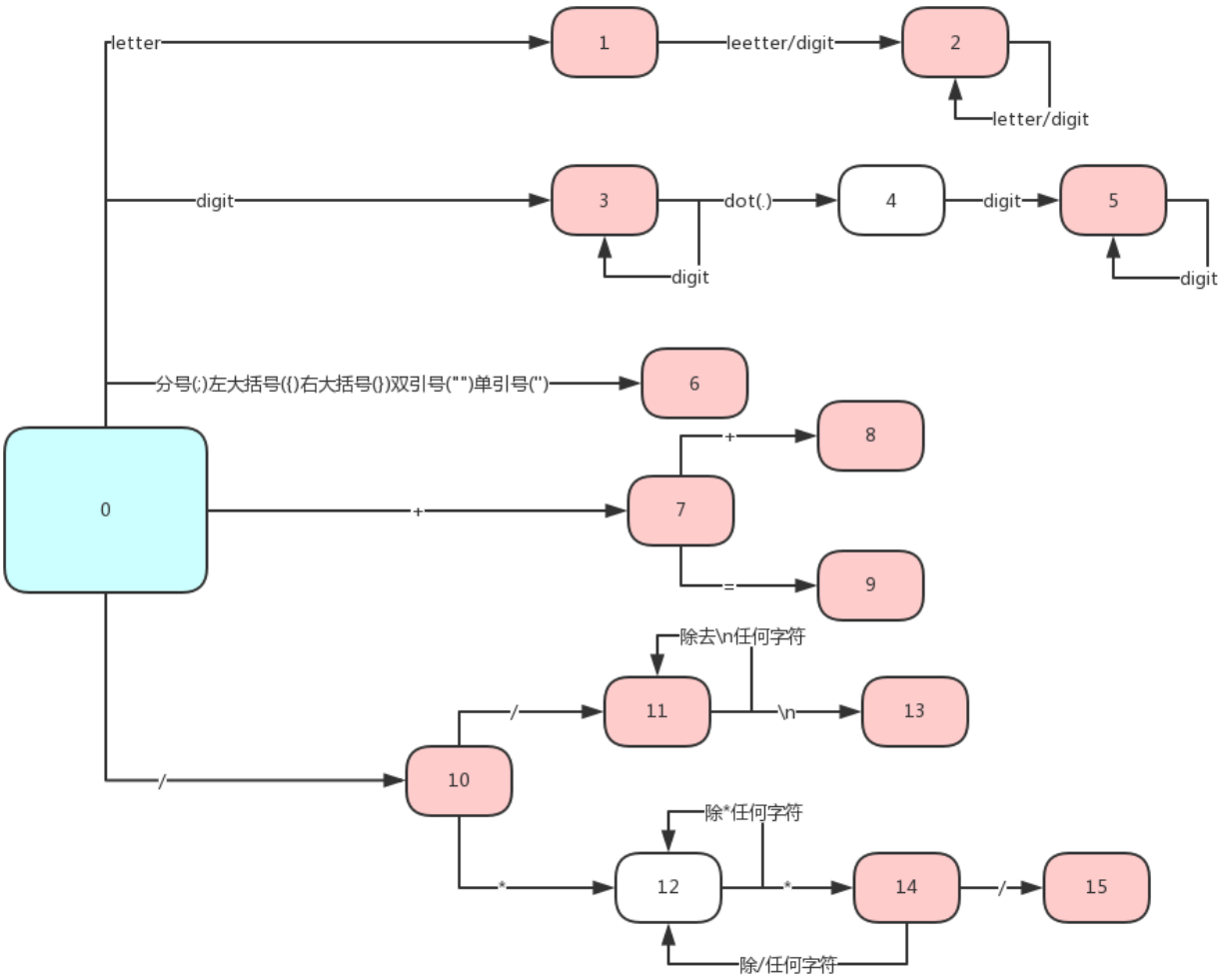

integer
float
digit
dot(.)


delimiter
分号(;)左大括号({)右大括号(})双引号("")单引号('')


operator
类似+-&
左括号(()右括号())左方括号([)右方括号(])点号(.)乘号(*)除号(/)百分号(%)或号(||)


除号
注释
\n
除了\n的任何字符
除了/的任何字符
除了*的任何字符

3.2 NFA->DFA



4. 数据结构描述

4.1 状态枚举

```
enum STATE{
    DONE,//正常情况
    INVAR,//标识符或者关键字
    INADD,INMINUS,//加减符号
    INLESS,INMORE,INEQUAL,//小于大于等于
    INEXCLAMATORY,//惊叹号
    INAND,INOR,//且 或
    INSOLIDUS,//斜线
    INDIGIT,//数字
    INDECIMALS,//小数
    ANNOTATION_ONE_LINE,ANNOTATION_MULTI_LINE,ANNOTATION_MULTI_LINE_ASTERISK,//各种注释
    SINGLE_QUOTE_MARK,DOUBLE_QUOTE_MARK//单引号双引号
}
```

其中省略了很多符号例如`{ } %`d的状态，因为这些字符一旦被扫描到那么直接结束判定为某字符不用状态转换

4.2 种类枚举以及map

```
enum TOKEN{
        KEYWORDS,//关键字
        IDENTIFIER,//标识符
        OPERATOR,//操作符
        DELIMITER,//分隔符
        INT,//整数
        DOUBLE,//浮点数
        ANNOTATION//注释符
    }
private static void initMap() {
        map.put(TOKEN.KEYWORDS, "关键字");
        map.put(TOKEN.IDENTIFIER, "标识符");
        map.put(TOKEN.OPERATOR, "操作符");
        map.put(TOKEN.DELIMITER, "分隔符");
        map.put(TOKEN.INT, "整数");
        map.put(TOKEN.DOUBLE, "浮点数");
        map.put(TOKEN.ANNOTATION, "注释符");
    }
```

4.2 关键词数组

```
static String KEYWORDS[] = {"break", "case", "char", "const",
            "continue", "default", "do", "double", "else", "enum", "float", "for", "goto", "if",
            "long","main","short","signed","sizoef","static","switch","unsigned","void"};
```

5. 核心算法描述

核心算法扫描输入字符串，通过状态转换确定token种类，对应代码中的`void AnalyzeStart() throws Exception`方法

1. 读取下一个字节
2. 建立读入缓存，这个缓存中记录现在读入的一个词法单元
3. 如果是结束字节那么结束循环跳到6如果不是继续
4. 判断状态，如果是done状态跳到1否则继续
5. switch状态，例如如果是var，那么读取字符一直到不是letter为止，然后判断是不是关键字，之后清空读入缓存，向输出数组中写入内容，状态调整为done，循环至1
6. 结束

6. 测试情况

6.1 测试用例输入

```
void test() {
        int[] a;
        a[0]=(int)1.5;
        int b=2/1;
        //注释
        a++;
        if(a!=b)
            System.out.println("woshi zifuchuan");
        System.out.println(obj[1]);
    }


    void main{
        /**
         * 这里是注释
         */
         test();
    }
```

6.2 token序列输出

```
关键字      void
标识符      test
操作符      (
操作符      )
左大括号        {
标识符      int
操作符      [
操作符      ]
标识符      a
分号        ;
标识符      a
操作符      [
整数        0
操作符      ]
操作符      =
操作符      (
标识符      int
操作符      )
浮点数      1.5
分号        ;
标识符      int
标识符      b
操作符      =
整数        2
操作符      /
整数        1
分号        ;
注释符      //
注释内容        注释
标识符      a
操作符      ++
分号        ;
关键字      if
操作符      (
标识符      a
操作符      !=
标识符      b
操作符      )
标识符      System
句号        .
标识符      out
句号        .
标识符      println
操作符      (
双引号      "
字符串      woshi zifuchuan
双引号      "
操作符      )
分号        ;
标识符      System
句号        .
标识符      out
句号        .
```

```
标识符      println
操作符      (
标识符      obj
操作符      [
整数       1
操作符      ]
操作符      )
分号       ;
右大括号          }
关键字      void
关键字      main
左大括号          {
注释符      /*
注释内容          *
      * 这里是注释

注释符      */
标识符      test
操作符      (
操作符      )
分号       ;
右大括号          }
```

第二部分:LEX部分实现

> 这一部分分成两个子部分
>
> 第一个子部分将任意正则表达式转化成为**NFA**，并返回该**NFA**的转换表；
>
> 第二个子部分将**NFA**转化成为**DFA**，仅支持字母表为**a b**的情形，并返回**DFA**的转换表；

1. 假设与依赖

- 假定给出的RE包括且仅包括a b两个非终结符**(**字符表超出**a b**不能得到正确的**DFA**，但是**NFA**没问题**)**
- 正则表达式符号只有左右括号( )，选择符号|和闭包符号* ，连接符号.必须省略
- 符号优先级是：括号>闭包>连接>选择，例如(a|bc)*abb的计算顺序是(((a|(bc)) *a)b)b

2. 数据结构描述

2.1 Node类

```
public class Node {
    private int id;//id自增长
}
```

2.2 Edge类

```
public class Edge {
    private Node begin;
    private Node end;
    private String label;//转换的标示，取值是字符表中的字符和“epsilon”
}
```

2.3 Graph类

```
public class Graph {
    private List<Edge> edges;
    private Node start;
    private Node end;
    //闭包
    public void closure();
    //连接
    public void join(Graph other);
    //选择
    public void union(Graph other);
}
```

3. 思路与方法

3.1 RE->NFA部分

维护两个栈：符号栈和Graph栈。符号栈仅包括左括号**"(",**连接符号**".",**选择符号**"|"**3种符号,原因下面会详述；**Graph**栈中保存中还没有计算的NFA图

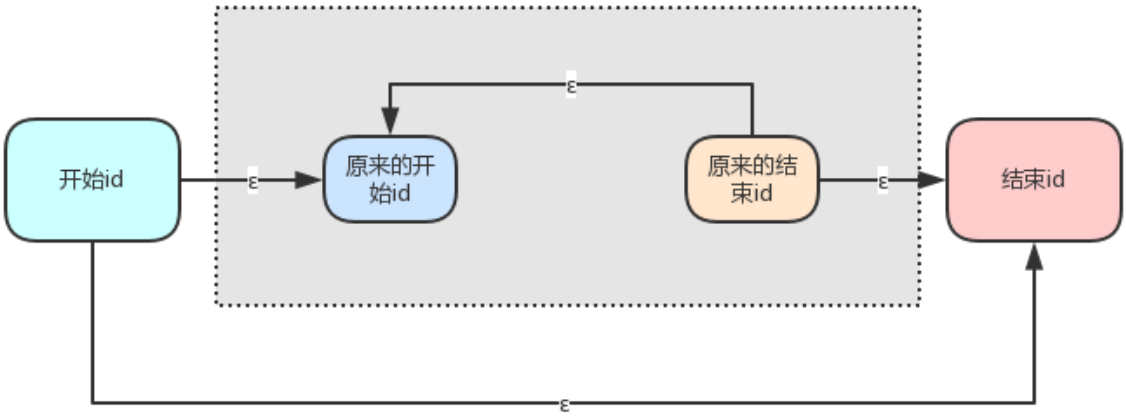
具体的主要算法执行流程：

1. 遍历输入的正则表达式，这里正则表达式的保存在REString变量中，可以通过下标访问
2. 遇到字母表中字符的时候，将这个字符转化成最简单的Graph压入栈中，之后如果它不是最后一个字符，那么检查它的下一个字符，如果还是字符或者左括号"("，就说明正则表达式省略了连接符号，这时候要向符号栈中压入连接符"."。
3. 遇到非字母表中字符时，需要分一下四种运算符的情况
    1. 如果是运算符“)”，即右括号，此符号属于运算级最高的符号了，所以它不用压入符号栈中，而是要在符号栈中弹出所有符号运算，直到遇到“)”匹配，运算过程中根据符号栈中弹出的符号计算
    2. 如果是运算符“(”，即左括号，此符号只是用来和右括号结合的，所以直接将该运算符压入符号栈中即可
    3. 如果是运算符“* ”，即闭包符号，这个在正则表达式中运算级最高，直接进行计算，因此不用压入符号栈中。运算后检查其后跟随的元素，如果是转移符号或者左括号，则必须要向符号栈中添加连接符号。
    4. 如果是运算符“|”，即选择符号，由于此符号的优先级没有连接符号高，所以此时应该弹出符号栈中优先级高于它的符号，但是“(”不参与弹出，所以这里只是弹出连接符号和自身“|”符号运算，然后将该符号压入符号栈等候计算。如下图
4. 正则表达式遍历完毕之后，需要弹出所有的符号栈进行计算，最后NFA栈中的唯一NFA就是所求的NFA。

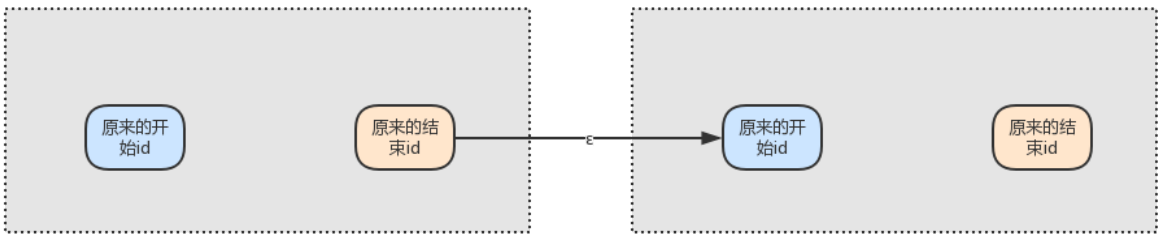
各个计算的具体算法：

1. 简单字符转换为Graph，代码`new Graph(char c)`。新建一张图，添加开始和结束节点，添加一条边，如下图。

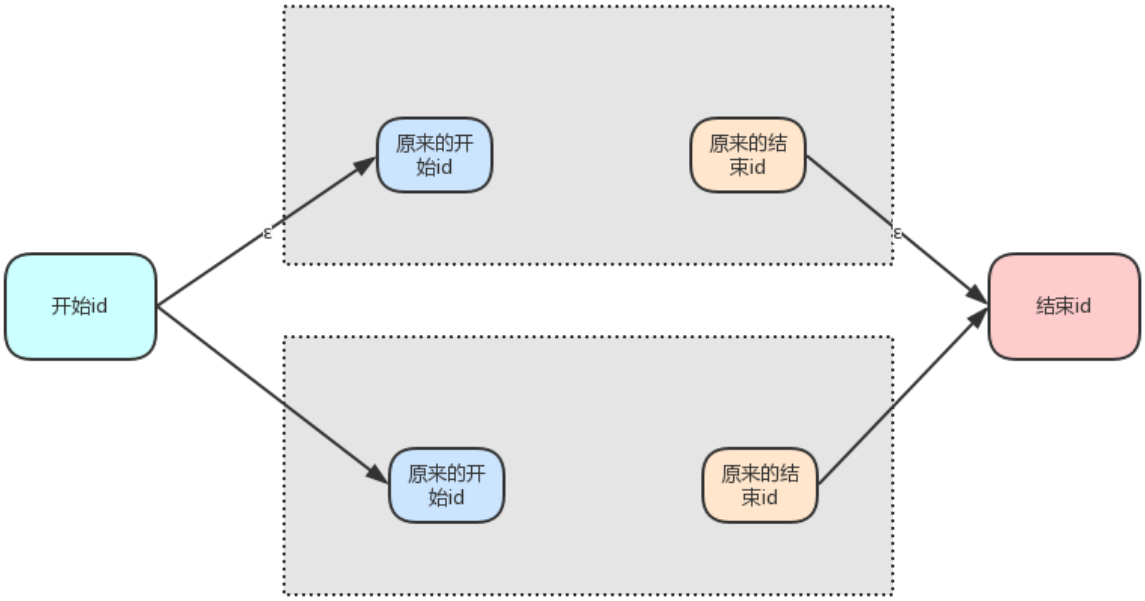
2. 闭包运算，代码`graph.closure()`。graph重新分配两个节点，作为新的NFA的start和end，然后新的start连接弹出NFA的start，弹出NFA的end连接新的end，然后添加一条新的start到新的end的一条空边和一条旧的end到旧的start的一条空边，将新的NFA压入NFA栈中。如下图



3. 连接运算，代码`graph.join(Graph other)`。将`graph`的end连接至`other`的start，然后更新新的NFA的start和end，压入NFA栈中。如下图



4. 选择运算，代码`graph.union(Graph other)`。Graph重新分配两个节点，作为新的NFA的start和end，然后新的start分别连接弹出的两个NFA的start，弹出的两个NFA的end分别连接新的end即构成新的NFA，压入NFA栈中。如下图



3.1 NFA->DFA部分

本部分仅支持字母表是a b的情况。维护三个队列(集合)，分别是`nodeList`,`nodeListA`,`nodeListB`用于模拟*DTtan*表中的第一列，第二列和第三列。再维护一个栈`stack`，用于运算过程中的结束符号。

具体的主要算法及流程：

1. 求出NFA的起点node的ε闭包并将其压入stack栈中
2. 当`stack`栈不为空的时候，循环做以下操作:
    1. 弹出`stack`栈顶数组，把它加入`nodeList`中，作为DFA的一个状态点。
    2. 计算出该状态点的move(a)和move(b)，并加入`nodeListA`和`nodeListB`中
    3. 如果move(a)和move(b)不在`stack`中也不在`nodeList`中，将其压入`stack`中，说明即将有一个新的DFA状态点
3. `stack`为空的时候说明所有DFA状态点和它们的a转换b转换都已经求出，并且对应的排列在`nodeList`，`nodeListA`，`nodeListB`中。
4. 下面把这三个队列转换成Graph结构，先重置Node的编号让它从0开始编号
    1. 对`nodeList`做遍历操作，为每个元素创建节点Node，并找出起始Node(第一个)和结束Node(包含NFA的结束Node编号的那个)
    2. 再做一次循环。把`nodeList`中的节点，`nodeListA`中的节点和a作为`start,end,label`创建边;把`nodeList`中的节点，`nodeListB`中的节点和b作为`start,end,label`创建边;
    3. 以上的起始节点，结束节点，边的集合就完成了Graph的架构
5. DFA的Graph结构创建完成

重要算法：

```
    //仅仅计算一次的epsilon闭包
    private List<Integer> epsilonClosureSimple(int nodeId)
    //循环调用只计算一次的方法直到前后两次长度相同，获得整个epsilon闭包，算法效率很有瑕疵不过懒得改
    private List<Integer> epsilonClosure(int nodeId)
    //挨个调用各个数字的闭包算法再合并，效率很差
    private List<Integer> epsilonClosure(List<Integer> nodeIdList);
    //已经计算了move(A,a)的epsilon闭包，我觉得这个算法要爆炸了
    private List<Integer> move(List<Integer> nodeIdList, char c)
```

4. 测试情况

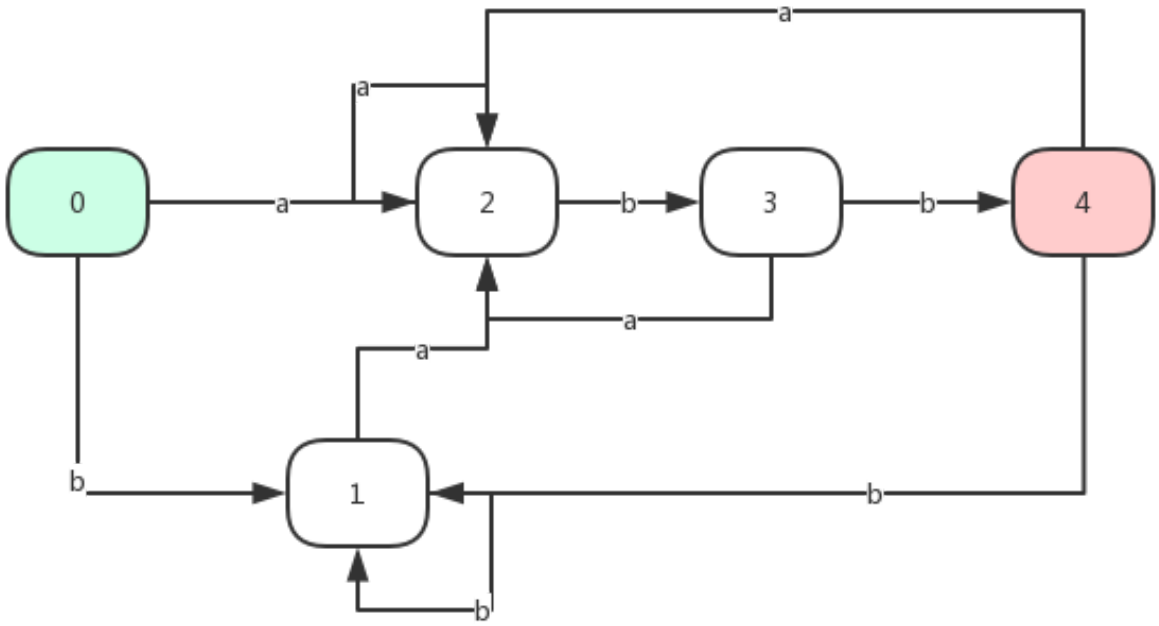
4.1 测试用例输入

```
(a|b)*abb
```

4.2 DFA结果输出

```
Start=0  End=4
Edge [begin=0, end=2, label=a]
Edge [begin=0, end=1, label=b]
Edge [begin=1, end=2, label=a]
Edge [begin=1, end=1, label=b]
Edge [begin=2, end=2, label=a]
Edge [begin=2, end=3, label=b]
Edge [begin=3, end=2, label=a]
Edge [begin=3, end=4, label=b]
Edge [begin=4, end=2, label=a]
Edge [begin=4, end=1, label=b]
```

根据这个结构画出的图如下:



困难与解决

1. 第一部分基本没遇到很多困难
2. 第二部分的困难集中在NFA转DFA这一步，完全自己动手创造数据结构，虽然算法是书上的(epsilon闭包算法)，但是它对应的数据结构还是很抽象。这个问题的解决是我简化了问题，把字母表简化为a b两个，之后维护了三个队列对应*Dtran*的三列。这样的话向表中添加元素就变得可控制了

不足与收获

不足：

1. LEX仅仅做到DFA一步
2. LEX中NFA转DFA一步仅仅支持ab字母表
3. C语言分析器中选取的子集太小了

收获：

1. 加深了对LEX的理解
2. 各个步骤的算法更加清晰了

总结

总的来讲这次实验还算成功……以后希望能把不足中的遗憾完善一下把。收(ha)获(ha)还(zhong)是(yu)蛮(zuo)大(wan)的(le)~

参考文献

编译原理 龙书第二版 *Alfred V.Aho*著

*weixliu*的博客*(*仅参考算法，代码都没看……*)*