# Xlex生成器

BY TANGENTA

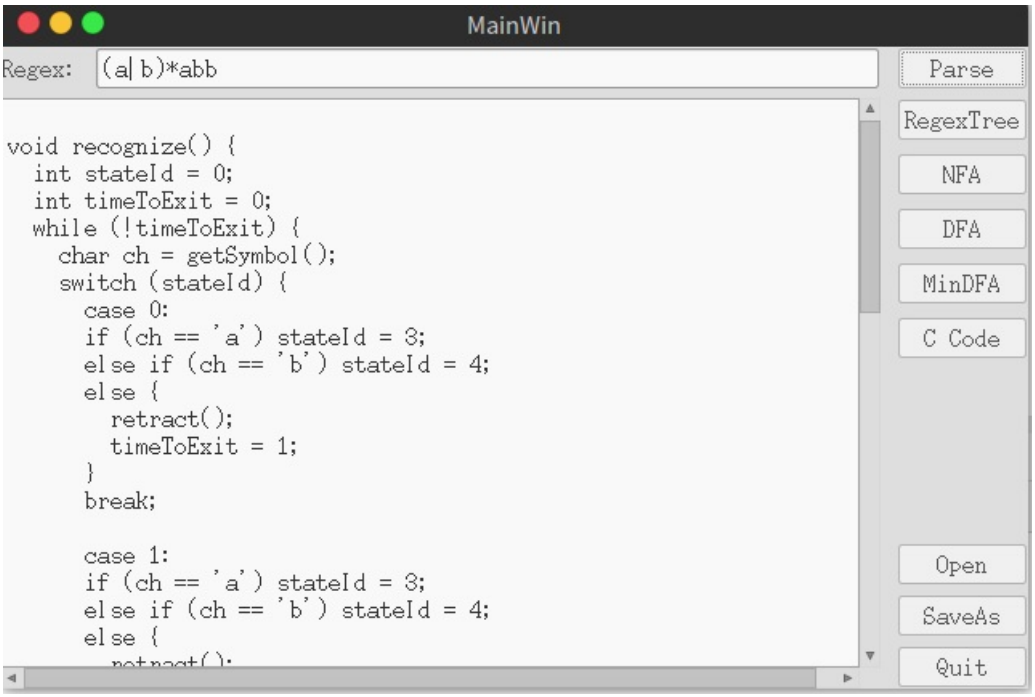
*2018,11,27*

## 1 概述

Xlex是将"基本正则表达式"自动转换为"能够识别该正则表达式的C程序代码"的工具，使用java和scala语言混合开发而成。 其中java负责实现图形界面、用户交互，scala负责实现主体程序内部逻辑。

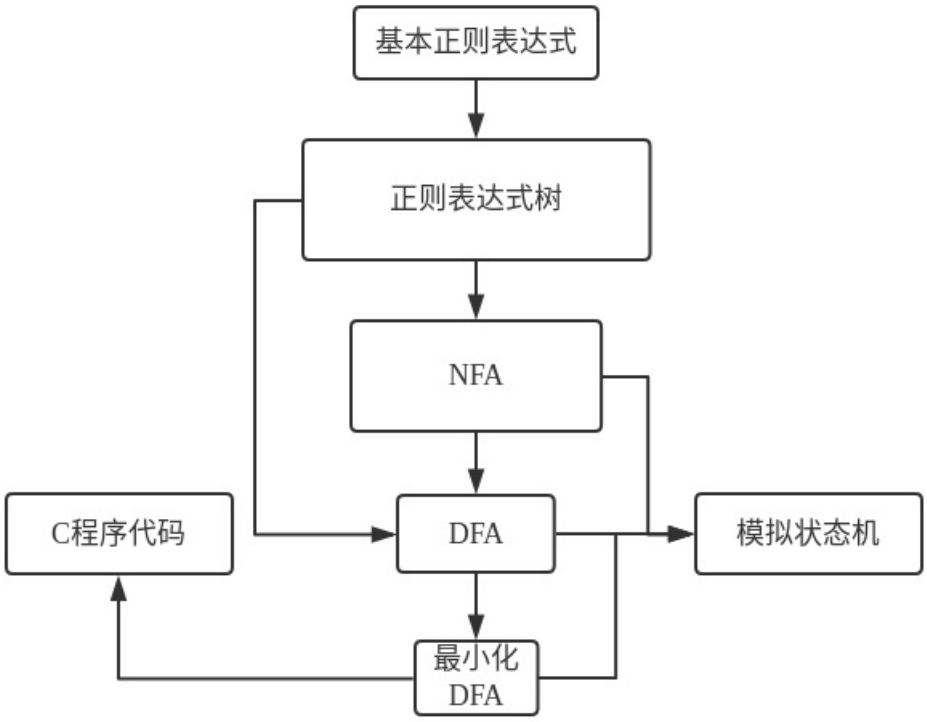
Xlex运行效果如下：


输入基本正则表达式，点击Parse后，可以选择右边五个按钮查看对应的内容，结果将显示在中间的文本编辑框中。右下方提供了打开文件和保存编辑框内容的功能。




## 2 程序分析

程序主要由正则表达式解析器、 NFA转换模块、 DFA转换模块、 C程序转换器以及各个结构的查看器组成。从整体上看，本程序实现的数据之间的转换如下图所示：




下面按照上图的数据流动逐个介绍每个转换过程。

## 2.1 基本正则表达式 -> 正则表达式树

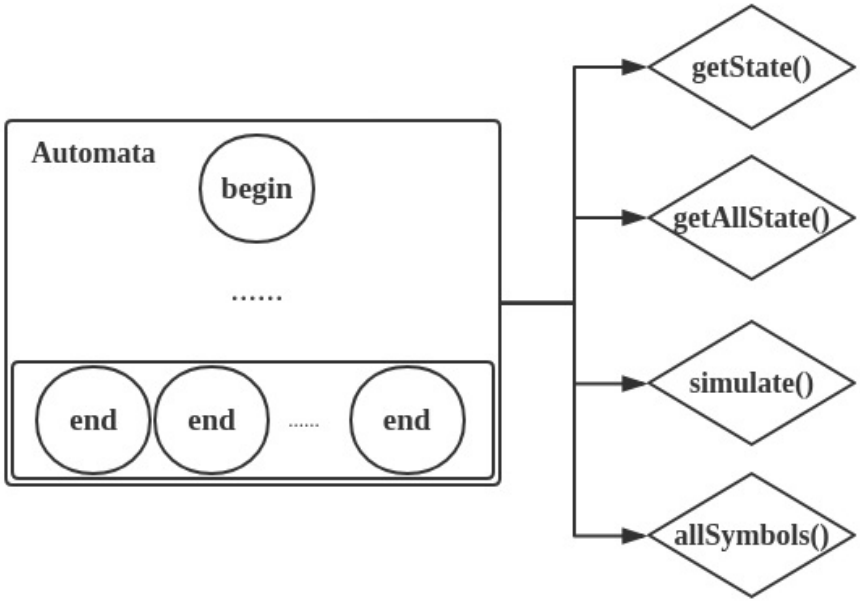
在regexParser包中提供了三种解析正则表达式并建立语法树的方法，分别是：

- 栈方法(位于RegexStackParser)，建立栈内栈外操作符优先级表和两个栈，从左向右逐个扫描字符串。
- 递归切割字符串方法(位于RegexRecurParser)，该方法每次找到某个操作符，分割为两个字符串，递归处理。和自顶向下的解析方法不同，从左到右的优先顺序必须用特殊方法处理。
- 组合子方法(位于RegexCombParser)，利用了scala外部的解析库。

这三个Parser都继承了同一个包中的RegexParser特质(相当于java的接口interface)，抽象出接口的目的是方便随时切换内部实现。由于正则表达式的连接运算符(Union)没有对应的符号，在用栈方法和切割字符串方法的时候需要进行预处理，例如添加'@'符号用于表示连接。

## 2.2 正则表达式树 -> NFA

automata包中的Automata抽象基类规定了有限自动状态机NFA和DFA的结构。




有限自动状态机由一组状态组成，其中包含一个开始状态，一组结束状态，以及另外四个接口。要创建这样的对象，必须使用其子类的工厂方法——继承自AutomataFactory的创建方法fromRegexTree()。

本质上来说，NFA和DFA都是图，这里构成图的基本要素包括顶点(表示状态)和边。状态则由一个类State表示，定义为id编号和边Edge的集合。边用类Edge表示，定义为一个字符和指向的下一个状态的id编号。因此，类NFA和类DFA的工作就是做好对这些State和Edge的管理。

```
case class State(id: Int, edges: Set[Edge])
case class Edge(symbol: Char, next: Int)
```

类NFA的伴生对象NFA实现了AutomataFactory的fromRegexTree()，用于构造NFA。代码如下：

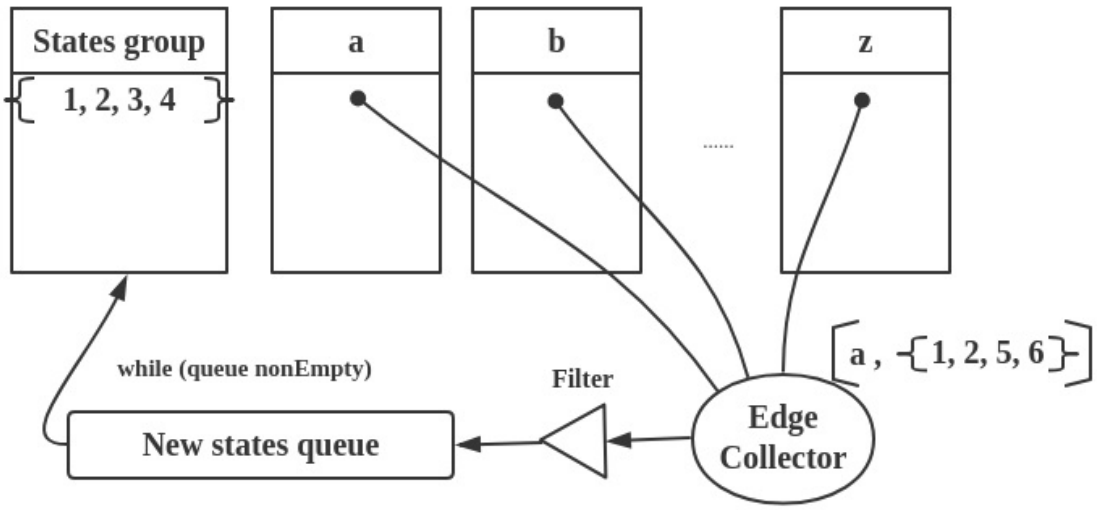
```
object NFA extends AutomataFactory {
  val epsilon = 'ε'
  import scala.collection.{mutable => m}
  private case class MState(id: Int, edges: m.Set[Edge] = m.Set()) {
```

```
    def toState: State = State(id, edges.toSet)
  }
  override def fromRegexTree(regexTree: RegexTree): NFA = {
    var id = 0
    var buffer = ArrayBuffer[MState]()

    def newState(): MState = {
      val newState = MState(id)
      buffer += newState
      id += 1
      newState
    }

    def connect(from: MState, to: MState, symbol: Char): Unit = {
      from.edges += Edge(symbol, to.id)
    }

    def helper(regexTree: RegexTree): (MState, MState) = { // return (beginState,
endState)
      regexTree match {
        case Leaf(symbol) =>
          val (b, e) = (newState(), newState())
          connect(b, e, symbol)
          (b, e)
        case UnaryNode(_, child) =>
          val (ob, oe) = helper(child)
          connect(oe, ob, epsilon)
          val (nb, ne) = (newState(), newState())
          connect(nb, ob, epsilon)
          connect(nb, ne, epsilon)
          connect(oe, ne, epsilon)
          (nb, ne)
        case BinNode(operator, left, right) =>
          import regexParser.Operator._
          operator match {
            case Concat =>
              val (b, leftEnd) = helper(left)
              val (rightBegin, e) = helper(right)
              connect(leftEnd, rightBegin, epsilon)
              (b, e)
            case Union =>
              val (lb, le) = helper(left)
              val (rb, re) = helper(right)
              val (nb, ne) = (newState(), newState())
              connect(nb, lb, epsilon)
              connect(nb, rb, epsilon)
              connect(le, ne, epsilon)
              connect(re, ne, epsilon)
              (nb, ne)
          }
      }
    }
    val (begin, end) = helper(regexTree)
    new NFA(begin.toState, Set(end.toState), buffer.map(_.toState).toArray)
  }
}
```

NFA的构造借助了辅助函数helper。helper是接受一个树节点的递归方法，功能是返回局部结构的初始状态和接受状态。遍历正则语法树的时候，按照Tompson方法，分别对不同的节点创建、连接状态，最终完成NFA图的构造。(由于scala默认的数据结构都是不可变的，例如前面的State和Edge，而不可变数据结构在"修改"尾节点的时候存在极大的不便性，因此这里引入的MState，是可变的数据结构，是过渡到State的中间状态。MState完成构造后再转为State，即可获得不可变数据结构的优越性。)

## 2.3 NFA -> DFA

NFA转换为DFA的过程被定义在了伴生对象(相当于java的单例对象)的fromNFA函数中。该函数接受一个NFA类，返回DFA类。使用的方法是子集构造(subset construction)。




主要思路是，把一组NFA的状态看作是单个DFA状态，用NState表示，是Set[Int]的别名。NEdge是由一个符号和NState组成的元组，表示边。 维护一个待处理队列pendingState和一组映射bufferMap，每次处理一个NState，方式是探索并收集每个符号对应的NFA状态，加入到bufferMap中。 每当"发现"一组新的NFA状态时，判断是否在bufferMap中，如果不在，则加入待处理队列中。代码如下：

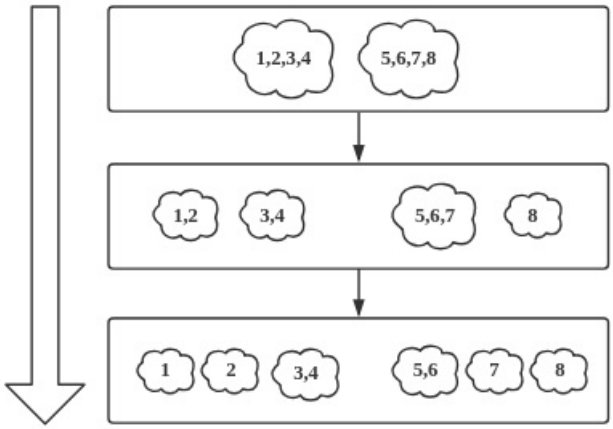
```
// method: subset construction
def fromNFA(nfa: NFA): DFA = {
  import scala.collection.{mutable => m}

  type NState = Set[Int]
  type NEdge = (Char, NState)

  val initialStates = nfa.epsilonTransition(nfa.begin.id)

  val bufferMap: m.Map[NState , Set[NEdge]] = m.Map()
  val pendingStates: m.Queue[NState] = m.Queue(initialStates)

  while (pendingStates.nonEmpty) {
    val dealing = pendingStates.dequeue()

    val edgeCollecter = (nfa.allSymbols-NFA.epsilon) map { symbol =>
      (symbol, nfa.move(dealing, symbol))
    } filter {_._2.nonEmpty}   // throw empty-state away

    // put the dealing stateGroup into bufferMap
    bufferMap += (dealing -> edgeCollecter)

    // update pendingStates
    edgeCollecter.foreach { case (_, newStates) =>
        if (!bufferMap.contains(newStates))
          pendingStates enqueue newStates
    }
  }

  // represent each stateGroup with a index
  val stateIndexMap = bufferMap.keys.zipWithIndex.toMap
  val resultTable = bufferMap.values.zipWithIndex.toArray map {
    elem =>
      State(elem._2, elem._1.map(iedge =>
        Edge(iedge._1, stateIndexMap(iedge._2))
      ))
  }
  val endingStates = (stateIndexMap filter {elem =>
```

```
      elem._1 & nfa.end.map(_.id) nonEmpty}).values.toSet

    new DFA(resultTable(stateIndexMap(initialStates)), endingStates map resultTable.apply,
  resultTable)
  }
```

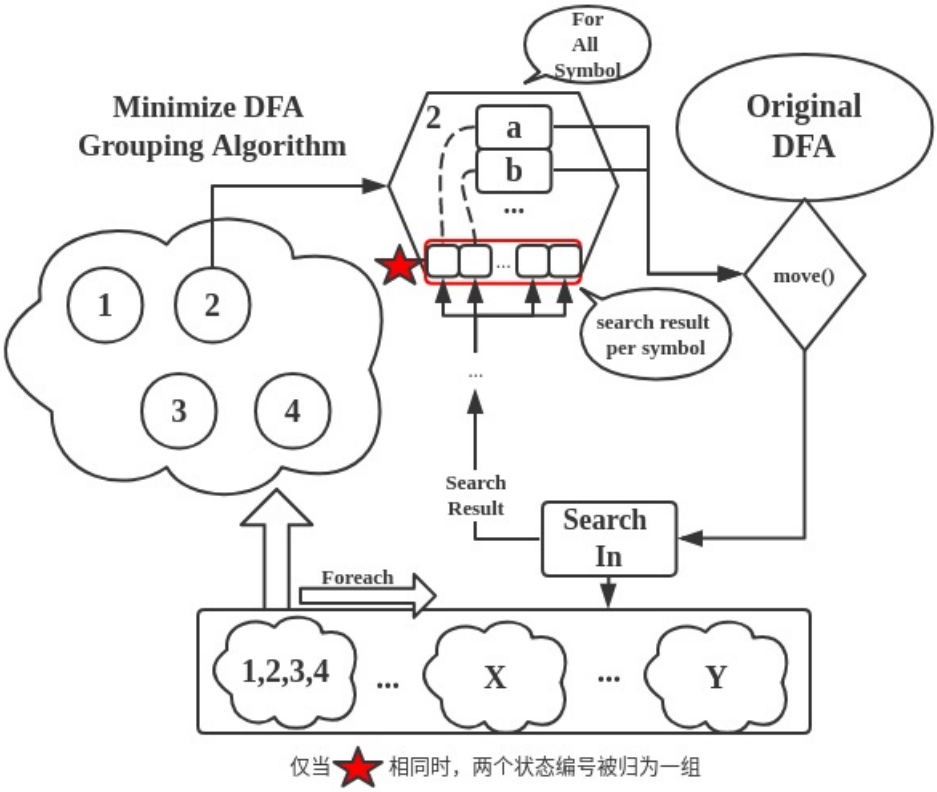
后半部分用来计算构造DFA所需要的数据——开始状态，结束状态以及保存所有状态的容器。

## 2.4 DFA -> 最小化DFA

最小化DFA使用了分组的方法，从最开始的两组——接受状态和非接受状态出发，每次迭代对里面各组进行细化，最终得到的每一组对应一个新的DFA状态，它们构成的DFA是具有最小状态数的DFA，称为最小化DFA。




分组的依据是，对于组内每一个id对应的状态，所有边各自指向下一个状态所属的组号构成的集合。迭代的终止条件是不能再分组，即大小不再变化。



关键代码如下：

```
type StatesG = Set[Int]

val endStatesId = dfa.end map {_.id}
val  initialGroups  =  List(endStatesId,  (0  until  dfa.getAllState.size).toSet  &~
endStatesId).filter(_.nonEmpty)
def helper(groups: List[StatesG]): List[StatesG] = {
  val result = groups.foldRight(List[StatesG]()) {(singleGroup, acc) => // List[StatesG]
    if (singleGroup.size == 1) singleGroup :: acc   // ignore single-state group
    else {
     // guarantee arbitrary 2 states in a group share the same transitions to some group
      val groupMap = singleGroup groupBy { singleStateID => // Set[Int]
        dfa.allSymbols map { symbol =>
          dfa.move(singleStateID, symbol) map { intElem =>
            (symbol, groups.find(_.contains(intElem)))
          }
        }
      }
      groupMap.values.toList ++ acc
    }
  }
  if (result.size == groups.size) groups    // nothing can be split
  else helper(result)
}
val finalGroups = helper(initialGroups)
```

其中定义的helper函数接受一组状态集合，返回完全分好组的状态集合。 foldRight相当于创建了一个循环，对状态组中的每一个状态集合进行操作，完成后聚集成新的一组状态。 对每个状态集合(singleGroup)需要进行的操作是如下：

1. 如果该状态集合大小为1,即无需再分，拼接到聚集列表上。
2. 否则按照如下规则分组：
    a. 对组内的每个状态进行所有符号的遍历，
    b. 计算出该状态输入每个符号所能到达的另一个状态的编号，
    c. 并在原分组中找到对应分组，将其id记录下来，连同符号组成元组，保存到List中，
    d. 同一分组的任意两个状态， 如果第c步计算出来的List不同， 则切分为不同的组。

## 2.5 最小化DFA -> C代码

一般C语言模拟DFA，识别正则表达式时采用的方法是while-switch-state，即引入一个类型为整数的状态变量state，用while循环包裹switch(state)-case结构，每一种case对应该状态的符号处理。一个能够识别"ab"例子如下：

```
void recognize() {
  int stateId = 1;
  int timeToExit = 0;
  while (!timeToExit) {
    char ch = getSymbol();
    switch (stateId) {
      case 0:
      if (ch == 'b') stateId = 2;
      else {
        retract();
        timeToExit = 1;
      }
      break;

      case 1:
      if (ch == 'a') stateId = 0;
      else {
```

```
          retract();
          timeToExit = 1;
        }
        break;

        case 2: retract(); timeToExit = 1; break;
      }
    }
  }
```

要验证是否识别成功，只需检验字符读取进度是否被getSymbol函数推进到理想位置即可。

具体利用程序自动转换为C代码的方法是：观察上述结构可以提取出共有的不变部分和可变部分，例如整体的while-switch结构，关键字都是共有的，case的个数，状态变量的值则是可变的。因此把不变部分写成模板，把可变的部分提取出来作为参数，对号入座。

由于模板的制作是通过在scala代码中用字符串手写C代码实现的，展示可能影响观感，此处略过不表。具体的代码可以在xlex包中的CppConverter查看。

## 2.6 * 正则表达式树 -> DFA

从正则表达式树也可以不经过NFA直接转换为DFA。

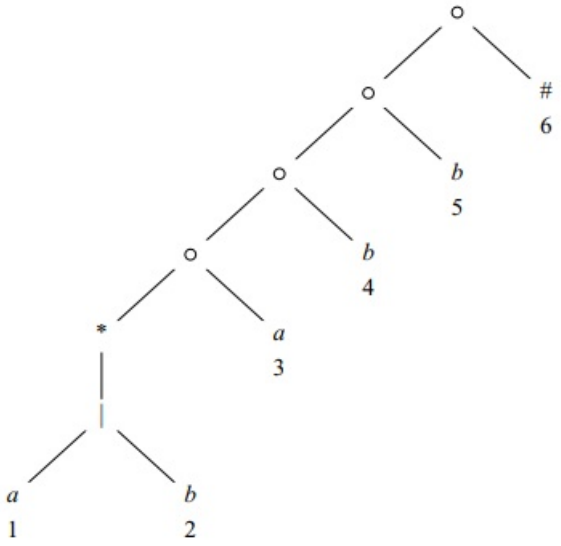



Figure 3.56: Syntax tree for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$

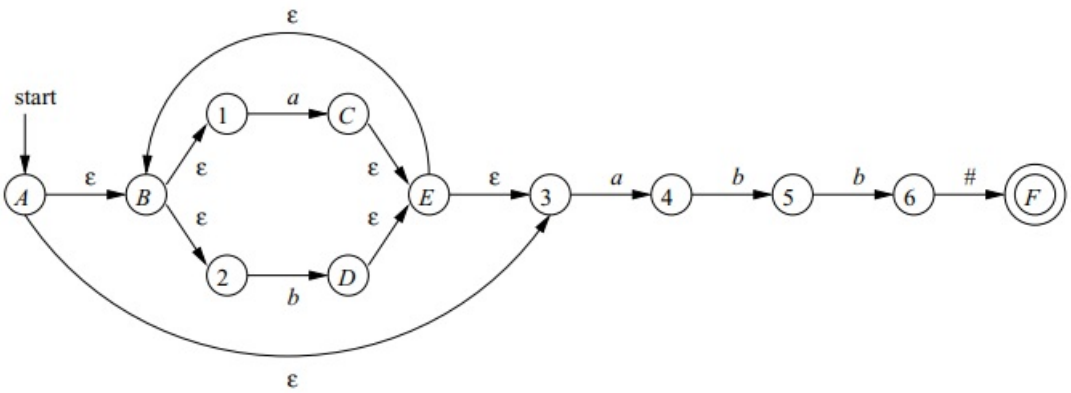



Figure 3.57: NFA constructed by Algorithm 3.23 for $(\mathbf{a}|\mathbf{b})^*\mathbf{abb}\#$

观察NFA，只有那些出边符号不为"空"的状态，才最有可能在DFA中得以留存（DFA只有两处与NFA不同，其中之一是没有符号为空的边）。 用数字标识出来，发现它们和正则语法树的叶节点一一对应，叶节点的符号表示该状态的出边符号。

如果可以知道这些状态[1]通过什么符号，能够到达哪些状态，那么DFA就能够被构建出来了。

下面引入FollowPos表的概念，解决上述问题。

FollowPos(p)表示p状态的"下一个"状态集，这里的"下一个"状态指可能出现的所有状态。

| POSITION $n$ | $followpos(n)$ |
|---|---|
| 1 | $\{1,2,3\}$ |
| 2 | $\{1,2,3\}$ |
| 3 | $\{4\}$ |
| 4 | $\{5\}$ |
| 5 | $\{6\}$ |
| 6 | $\emptyset$ |

Figure 3.60: The function *followpos*

如果有了这张表，计算DFA的过程如下：

1. 找到初始状态集，记为states，它对应即将构建的DFA的初始状态。
2. states里面的每个状态对应不同的出边符号a，以a为依据分组，得到statesGroup，它是一个分组的集合。
3. 将statesGroup中每个分组的每个状态用FollowPos表转换，合并（做并集运算），得到一组新的状态集合，其中每个状态集合对应新DFA的一条边。
4. 和子集构造类似，检查这些集合是否已经出现过，如果已出现过则忽略;如果未出现过，则把它作为新的DFA状态加入到DFA中， 并转到第2步处理它（作为states）。

那如何计算FollowPos表？ 具体方法是， 对正则树所有的节点计算几个值： firstPos，lastPos，nullable。 分别表示该节点对应的正则结构中第一个、最后一个可能出现的状态，以及是否可以为空。 每个节点的followPos可以通过维护一个followPos表，并遍历语法树计算而得。遍历时对每个节点做如下操作：

- 如果遇到的节点是闭包节点，则向它的每一个lastPos的followPos集合中添加firstPos。
- 如果遇到的节点是连接节点，则向左子节点的lastPos的followPos集合中添加右子节点的firstPos。
- 如果遇到的节点是其他类型的节点， 则不作任何操作。

为了方便得出每个节点的firstPos，lastPos和nullable，实现的时候利用了原有语法树的结构，加入这三个域后重新建立新的语法树，建立的同时计算三个域的值。

1. 这些状态被称为重要态(important state)

可以看到，从新构建的语法树根获得firstPos的状态集合，就是DFA的初始状态。

构建DFA部分的代码如下所示：

```
// construct DFA
type Pos = Int
type DState = Set[Pos]
type DEdge = (Char, DState)
def exploreEdgesOf(state: DState): Set[DEdge] = {
  state.groupBy(symbols(_)).collect { case (ch, set) if ch != '#' =>
    ch -> set.flatMap {followPosTable(_)}
  }.toSet
}
def collectFrom(state: DState, result: Map[DState, Set[DEdge]]): Map[DState, Set[DEdge]]
= {
  val edges = exploreEdgesOf(state)
  val nextStates = edges.map(_._2)
  val updatedResult = result.updated(state, edges)
  val notIncludedStates = nextStates &~ result.keys.toSet
  if (notIncludedStates.nonEmpty) {
    notIncludedStates.foldLeft(updatedResult) {(acc, elem) => collectFrom(elem, acc)}
  } else updatedResult
}
val dStateTable = collectFrom(finalTree.firstPos, Map())
```

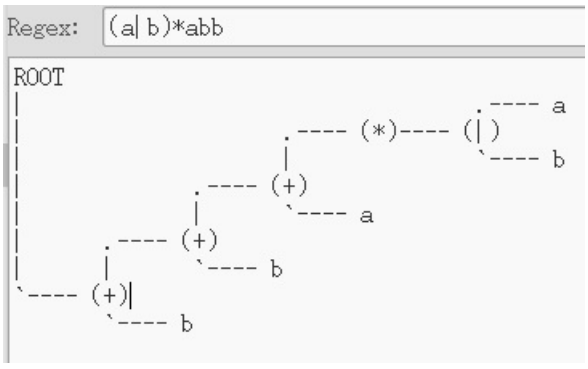
和NFA转DFA的子集构造类似，新DFA映射全部构建完成后，最后将集合替换为单个编号，计算DFA构造方法所需要的值，完成DFA的构造。

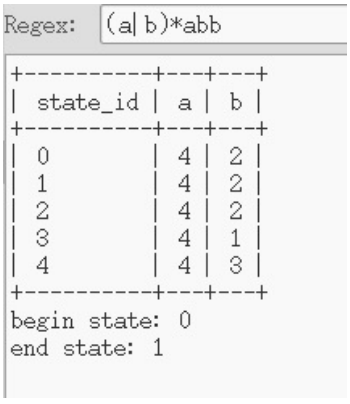
## 2.7 * NFA、 DFA模拟自动状态机

有限自动状态机可以判定一组输入，能否让其进入接受状态。 本程序NFA和DFA都继承了Automata中的simulate方法，用于模拟自动状态机的工作过程。 原理是利用已构造好的State和Edge，每处理一个符号就进行一次状态转移，当全部字符处理完毕后，查看当前状态(集)是否为(包含)接受态。

## 2.8 * 正则语法树和NFA、 DFA的图形化显示

本程序没有自动生成图片的功能，而使用了字符界面的模拟图像化方法。例如：




具体的实现可以在viewer包中的HorizontalTreeViewer和AutomataViewer类找到。