**实 验 报 告**

|  |  |
| --- | --- |
| **课程名称：** | **编译技术** |
| **学生姓名：** |  |
| **学生学号：** |  |
| **学生专业：** | **软件工程** |
| **开课学期：** | **2024-2025第二学期** |

**软件学院**

**2025年05月**

# From Reguler Experession to NFAs

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **地 点：** | | | B7 楼 | | 333房； | | **实验台号：** | |  | |
| **实验日期与时间：** | | | 2025/5/16 | | | | **评 分：** | |  | |
| **预习检查纪录：** | | |  | | | | **实验教师：** | | 应伟勤 | |
| **电子文档存放位置：** | | |  | | | | | | | |
| **电子文档文件名：** | | |  | | | | | | | |
| **批改意见：** | | |  | | | | | | | |
| 出勤情况（20%） | 课堂态度（20%） | | 实验报告质量（60%） | | | | | |
| 任务完成情况 | | 源代码注释 | | 报告内容 | |
| □ 出勤  □ 缺勤  □ 早退  □ 请假 | □ 认真  □ 不认真 | | □ 完成  □ 未完成 | | □ 优秀  □ 良好  □ 通过  □ 未通过 | | □ 优秀  □ 良好  □ 通过  □ 未通过 | |

# 报告内容

1、实验目标

（1）使用Thompson Construction算法实现正则表达式转换为NFA

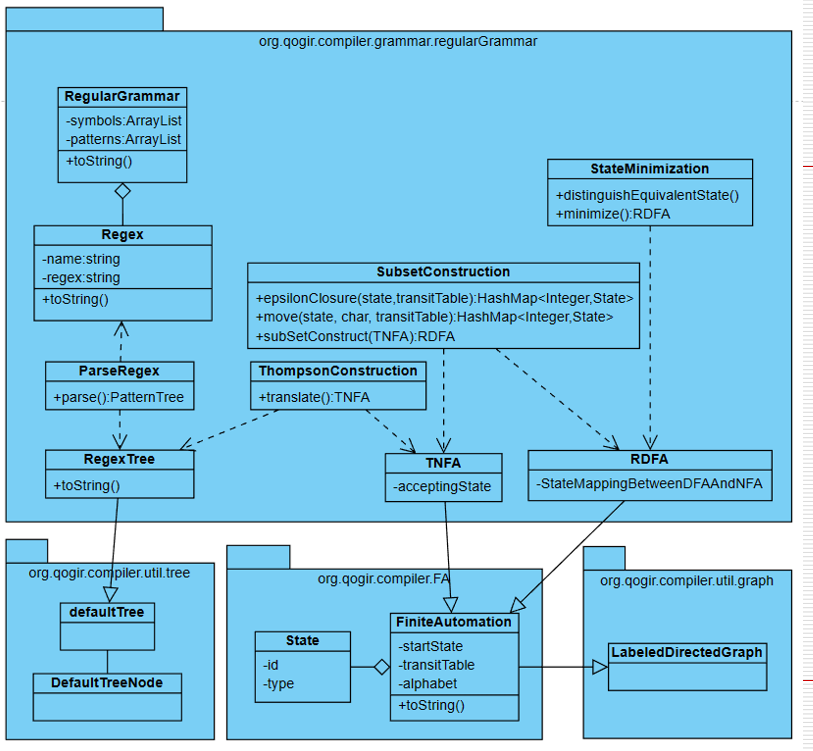
（2）巩固词法分析相关的正则文法、有穷自动机相关理论知识;

2、实验环境

IDEA，Vscode

1. 关键数据结构和核心算法

3.1 数据结构

先读入正则文法，再针对每个正则表达式：解析正则表达式为正则表达式树，利用Thompson构造法将正则表达式树按节点转换为NFA

3. 2 核心算法

使用Thompson Construction算法作为核心算法，代码详情如下：

当状态为1时

if (type == 1) { // Concatenation (-)  
 if (queue.isEmpty()) return null;  
  
 TNFA first = queue.poll(); // 第一个子 NFA  
 tnfa.setStartState(first.getStartState());  
 tnfa.getTransitTable().merge(first.getTransitTable());  
  
 first.getStartState().setType(State.*START*);  
 first.getAcceptingState().setType(State.*MIDDLE*);  
  
 TNFA prev = first;  
 while (!queue.isEmpty()) {  
 TNFA curr = queue.poll();  
 tnfa.getTransitTable().addEdge(prev.getAcceptingState(), curr.getStartState(), 'ε');  
 tnfa.getTransitTable().merge(curr.getTransitTable());  
 curr.getStartState().setType(State.*MIDDLE*);  
 curr.getAcceptingState().setType(State.*ACCEPT*);  
 prev = curr;  
 }  
 tnfa.setAcceptingState(prev.getAcceptingState());  
  
 // 设置起始状态为 START，接受状态为 ACCEPT  
 tnfa.getStartState().setType(State.*START*);  
 tnfa.getAcceptingState().setType(State.*ACCEPT*);  
}

当状态为3时

else if (type == 3) { // Kleene Star (\*), only one child  
 RegexTreeNode child = (RegexTreeNode) node.getFirstChild();  
 TNFA childNFA = translate(child);  
  
 tnfa.getTransitTable().addEdge(tnfa.getStartState(), childNFA.getStartState(), 'ε');  
 tnfa.getTransitTable().addEdge(childNFA.getAcceptingState(), tnfa.getAcceptingState(), 'ε');  
 childNFA.getTransitTable().addEdge(childNFA.getAcceptingState(), childNFA.getStartState(), 'ε');  
 tnfa.getTransitTable().addEdge(tnfa.getStartState(), tnfa.getAcceptingState(), 'ε');  
  
 tnfa.getTransitTable().merge(childNFA.getTransitTable());  
  
 // 设置子 NFA 的开始和结束状态为中间状态  
 childNFA.getStartState().setType(State.*MIDDLE*);  
 childNFA.getAcceptingState().setType(State.*MIDDLE*);  
}

3. 3 错误处理

if (type < 0 || type > 3) {  
 throw new IllegalArgumentException("未知的正则表达式节点类型: " + type);  
}

if (value == null || value.toString().isEmpty()) {  
 throw new IllegalArgumentException("基本字符节点值不能为空");  
}

RegexTreeNode child = (RegexTreeNode) node.getFirstChild();  
if (child == null) {  
 throw new IllegalArgumentException("Kleene Star (\*) 操作符必须有一个子节点");  
}  
TNFA childNFA = translate(child);  
if (childNFA == null) {  
 throw new IllegalStateException("子节点转换为 NFA 时返回了 null");  
}

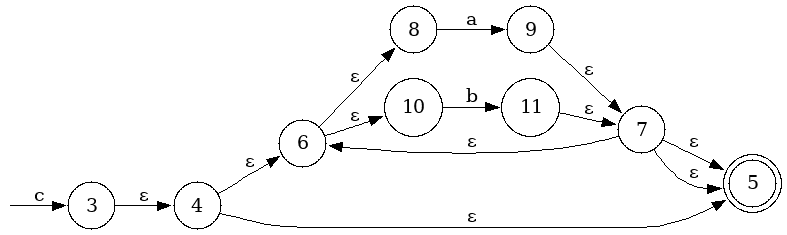
if (queue.size() < 2) {  
 throw new IllegalArgumentException("Union (|) 至少需要两个子表达式");  
}

3.4 运行结果

（1）对输入"regex1 := c(a|b)\*"

Regular Grammar  
Alphabet:[c, a, b]  
Regexes:  
[regex1 := c(a|b)\*]  
{regex1 := c(a|b)\*=The regex tree:  
(-:1)  
 firstChild:(c:0)  
 (\*:3)  
(c:0)  
(\*:3)  
 firstChild:(|:2)  
(|:2)  
 firstChild:(a:0)  
 (b:0)  
(a:0)  
(b:0)  
}  
Show the NFA:  
Only one regex in the regular grammar.c(a|b)\*  
Alphabet:[c, a, b]  
Total edges:12  
Start State:2  
the transitTable is:   
(4:1->6:1@ε)  
(4:1->5:2@ε)  
(8:1->9:1@a)  
(7:1->6:1@ε)  
(9:1->7:1@ε)  
(7:1->5:2@ε)  
(3:1->4:1@ε)  
(11:1->7:1@ε)  
(2:0->3:1@c)  
(10:1->11:1@b)  
(6:1->8:1@ε)  
(6:1->10:1@ε)

2为起始状态，5为结束状态，2接受一个c才能继续。



4、总结

（1）通过将状态分为不同类别（如 [START], [ACCEPT], [MIDDLE]），可以更清晰地表达每个状态在整个自动机中的作用：

（2）在 Thompson Construction 中，每次合并子 NFA 后都需要重置子结构的状态类型为 [MIDDLE]，以确保整体结构只有一个 [START] 和一个 [ACCEPT]。这要求我们在递归构造过程中时刻注意状态类型的更新，否则容易导致多个起点/终点，破坏自动机结构。

（3）虽然每个节点有不同的语义，但它们都实现了相同的 [RegexTreeNode]接口，并通过 [getType()] 统一调度，方便后续扩展新的操作符（如 `+`, `?`, `{n,m}`）。