



编译原理

词法分析

学生姓名： 杨杰

班 号： 111172

学 号： 20171002157

指导教师： 龚君芳

中国地质大学信息工程学院

2019 年 11 月 13 日

一、实验环境及相关配置:

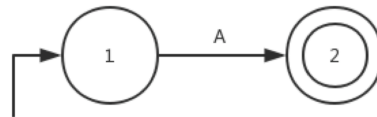
操作系统: macOS Catalina 10.15.1

集成开发环境: Xcode 11.2.1 (11B500)

开发语言: Swift 5.1

二、类属性和操作的定义:

Class Node: 状态所接收的输入。例如正则式“A”, 起始状态接收输入“A”转移至结束状态。“A”即为该Node对象。



主要的属性和操作包括:

属性 key (character): 该状态转移接收的输入值;

属性 isBegin (bool): 标记是否是开始状态进行的接收;

属性 isEnd (bool): 标记是否为达到结束结束进行的接收;

属性 list ([node]): 当前接收达到的状态, 该状态可接收的输入;

属性 isManaged (bool): 标记进行空输入的移除时对Node是否经过了处理;

属性 isFinded (bool): 在去空节点时候寻找非空节点时是否被访问过;

属性 stateNum ([string]): 对每个状态进行的编号

操作: init(): 构造函数, 初始化;

操作 copyNode(newNode: Node): 对象拷贝, 将newNode复制给本Node对象。

Class MyLexicalAnlysis: 词法分析类, 输入正则式, 获得DFA表; 输入字符串, 判断是否符合该正则表达式。

主要的属性和操作包括:

属性 regex (string): 正则表达式, 需要处理的字符串;

属性 isWrong (bool): 检测正则表达式是否输入有误;

属性 charStuck ([[node, node]]): 在将正则式转换为 ϵ -NFA时使用的存放最初输入和最后输入的输入值栈;

属性 markStuck ([[character, int]]): 在将正则式转换为 ϵ -NFA时使用的存放符号(包括“(”、“)”、“|”、“*”)和该符号所在正则式字符串的位置的符号栈;

属性 beginNodeNFA (node): 存放NFA的最初的输入;

属性 beginNodeDFA (node): 存放DFA的最初的输入;

属性 DFATable ([[String, [[Character, String]]]]): 存放最后获取的DFA表;

属性 stateEnd (string): DFA表中是结束状态的编号。

操作 init()、setRegex(str: String)、init(str: String): 构造函数, 初始化。

操作 LexicalDeal(): 处理正则式字符串获得 ϵ -NFA;

操作 LeftBracketDeal()、RightBracketDeal(i: Int)、OrDeal(i: Int)、

AnyDeal(i: Int), 在操作LexicalDeal()对不同符号进行的处理。

操作 BlankNodeDeal(): 去除无用的空输入, 将 ϵ -NFA转换为NFA;

操作 NFAtoDFA(): 将NFA转换为DFA;

操作 CreateDFATable(): 根据属性, 获得DFA表;

操作 StringLexical(str: String)->Bool: 输入string, 判断其是否属于该正则式。

三、实验流程:

(一) 正则表达式转换为 ϵ -NFA:

遍历正则式字符串的每一个字符:

当遇到输入字符集元素 Node 时, 给该 Node 加个空 Node 头 aNode 和空 Node 尾 bNode, 将该[aNode, bNode]放入 charStack;

当遇到左括号时, 将[“(”, 当前 charStack 的元素数量]放入 markStack 中;

当遇到右括号时, 循环的将左括号之后入 charStack 栈的所有[Node, Node]链接为一个新的[Node, Node], 放入 charStack;

当遇到“|”时, 将需要进行“|”操作左边的所有[Node, Node]合并为一个新的[Node, Node];

当遇到“*”时, 将 charStack 中需要进行“*”操作的[Node, Node], 添加一个新 Node 空头 aNode 和新 Node 空尾 bNode, 将 bNode 的 list 中添加 aNode, 时期可以循环, 将得到的[aNode, bNode]放入 charStack。

重复上述所有操作, 直到读取字符串所有的字符时结束。

合并所有在 charStack 中的[Node, Node]为一个[Node, Node]。该 Node 头即为 beginNodeNFA。

(二) ϵ -NFA 转换为 NFA:

DFA 中不可能存在 ϵ 边, 需要消除 ϵ 边。递归的将 beginNodeNFA 链表中所有无用的 key 值为“ ”的 Node 删除。

从头节点开始, 定义一个新的 list[], 遍历每一个旧 list[] 中的元素 Node, 将其旧 list[] 中, 值不为“ ”的加入新的 list[] 中, 遍历完成就用新的 list[] 覆盖旧的 list[], 删除 ϵ 边和一些因为删除 ϵ 边而不可能到达的状态。同时, 操作过的 Node 改变其 isManaged 的值, 防止发生死循环现象。

同时为每个状态标记序号, 存放于 Node 可达到编号 stateNum 数组中, 以便进一步处理。

(三) NFA 转换为 DFA:

NFA 和 DFA 最大区别就是从一个状态读入一个字符, DFA 得到一个状态, NFA 得到一个状态的集合。

操作类似手绘表格的形式。从开始状态开始, 将其放入 QueueState, 计算该状态经过所有不同输入种类得到的其他状态, 放入 SetState 中, 如果 SetState 中某元素属于 QueueState, 则说明已经指向了一个已知的 DFA 状态, 否则将该 SetState 该元素放入 QueueState 中, 继续进行处理。

该转换包括两层循环: 一是遍历接收输入字符的并集, 另一个是计算对于每一个接收的字符遍历所有输出的变计算 DFA 状态所包含的 NFA 的集合。

最后将得到的结果存入 DFATable ([(String, [(Character, String)])]) 中, 其中, 第一个 String 表示当前状态集合, Character 是该状态接收到的输入, 第二个 String 是当前状态接收该输入达到的状态。

同时为 Node 中 isEnd 参数为 true 的状态标记为结束状态。

同时为读入 String, 按照链表路线判断其是否属于该正则式

(四) DFA 结果展示:

使用 SwiftUI 制作 ios 界面, 打包为一个词法分析小工具。

四、 测试截图：



五、 附录

主要代码见附件*.swift 文件。

项目请访问网页: <https://github.com/LLLLLayer/Compilation-principle-lexical-analysis>