

课程实验报告

课 程 名 称： 编译原理

实验项目名称： 正则运算表达式的DFA构建

专 业 班 级： 软件工程2201

姓 名： 滕明君

学 号： 202226010111

指 导 教 师： 黎文伟

完 成 时 间： 2025 年 04 月 26 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. 字符集管理 2. 有限自动机实现 3. 词法分析器 | |
| 实验环境：  Cangjie、CodeArts IDE for Cangjie | |
| 实验内容及操作步骤：  **一、**  基于上述数据结构的定义，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  (2)int union(char c1, char c2)； // 字符的并运算  (3)int union(int charSetId, char c)；// 字符集与字符之间的并运算  (4)int union(int charSetId1,int charSetId2)；//字符集与字符集的并运算  (5)int difference(int charSetId, char c)； // 字符集与字符之间的差运算  这5个函数都会创建一个新的字符集对象，返回值为字符集id。创建字符集，表现为往字符集表中添加新的行。当一个字符集包含多个段时，便会在字符集表中有多行，一行记录一段。  (1)range 函数  func range(fromChar: Rune, toChar: Rune): UInt32 {      charSetTable.append(CharSet(charSetCounter, 1, fromChar, toChar))      let result = charSetCounter      charSetCounter += 1      return result  }  设计思路：创建连续字符区间  要点：  生成全局唯一ID（charSetCounter自增）  单段存储（起始-结束为一个段）  (2)unionChars 函数  func unionChars(c1: Rune, c2: Rune): UInt32 {      let newId = charSetCounter++      charSetTable.append(CharSet(newId, 1, c1, c1))      charSetTable.append(CharSet(newId, 2, c2, c2))      return newId  }  设计思路：离散字符并集  要点：  每个字符独立成段  段ID递增管理（SegmentId=1和2）  (3)unionCharSetWithChar 函数  func unionCharSetWithChar(charSetId: UInt32, c: Rune): UInt32 {      let newId = charSetCounter++      // 复制原字符集所有段      for (seg in charSetTable) {          if (seg.indexId == charSetId) {              charSetTable.append(CharSet(newId, newSegmentId++, seg.              fromChar, seg.toChar))          }      }      // 添加新字符段      charSetTable.append(CharSet(newId, newSegmentId, c, c))      return newId  }  设计思路：字符集扩展  要点：  深拷贝原字符集段  新增单字符段  段ID连续递增  (4)unionCharSets 函数  func unionCharSets(charSetId1: UInt32, charSetId2: UInt32): UInt32 {      let newId = charSetCounter++      // 合并两个字符集的所有段      for (seg in charSetTable) {          if (seg.indexId == charSetId1 || seg.indexId == charSetId2) {              charSetTable.append(CharSet(newId, newSegmentId++, seg.              fromChar, seg.toChar))          }      }      return newId  }  设计思路：字符集合并  要点：  顺序合并所有段  保留原始段顺序  段ID重新编排  (5)difference 函数  func difference(charSetId: UInt32, c: Rune): UInt32 {      let newId = charSetCounter++      for (seg in charSetTable) {          if (seg.indexId == charSetId) {              if (c < seg.fromChar || c > seg.toChar) {                  // 直接复制无关段                  charSetTable.append(CharSet(newId, newSegmentId++, seg.                  fromChar, seg.toChar))              } else {                  // 拆分包含字符的段                  if (seg.fromChar < c) {                      charSetTable.append(CharSet(newId, newSegmentId++,                      seg.fromChar, c-1))                  }                  if (seg.toChar > c) {                      charSetTable.append(CharSet(newId, newSegmentId++, c                      +1, seg.toChar))                  }              }          }      }      return newId  }  设计思路：字符集剪裁  要点：  只处理包含目标字符的段  拆分成最多两个新区间  保留其他无关段  - 无损修改 ：通过创建新字符集（newId）实现不可变数据操作  - 类型安全 ：使用 Rune 类型处理 Unicode 字符，通过 UInt32 转换确保算术运算安全  - 全局管理 ：通过全局的 charSetTable 维护所有字符集段  二、  基于上述NFA的数据结构定义，请按照最简NFA构造法，实现如下函数：  (1)Graph \* generateBasicNFA(DriverType driverType，int driverId );  (2)Graph \* union(Graph \*pNFA1, Graph \*pNFA2)； // 并运算  (3)Graph \* product(Graph \*pNFA1, Graph \*pNFA2); // 连接运算  (4)Graph \* plusClosure(Graph \*pNFA) //正闭包运算  (5)Graph \* closure(Graph \*pNFA) // 闭包运算  (6)Graph \* zeroOrOne(Graph \*pNFA)； // 0或者1个运算。  其中第1个函数generateBasicNFA是针对一个字符或者一个字符集，创建其NFA。其NFA的基本特征是：只包含两个状态（0状态和1状态），且结束状态（即1状态）无出边。后面5个函数则都是有关NFA的组合，分别对应5种正则运算，创建一个新的NFA作为返回值。  (1) generateBasicNFA  设计思路：创建只包含两个状态（起始和接受）的基础NFA  func generateBasicNFA(driverType: DriverType, driverId: UInt32): Graph {      let g = Graph(graphIdCounter)      graphIdCounter += 1      let start = State(0, UNMATCH)      let accept = State(1, MATCH)      g.addState(start)      g.addState(accept)      g.addEdge(Edge(start.stateId, accept.stateId, driverId, driverType))      return g  }  要点：  使用全局计数器生成唯一graphId  接受状态无出边（MATCH类型）  边类型支持字符/字符集（CHAR/CHARSET）  (2) union（并运算）  设计思路：创建新初始状态，通过ε边连接两个NFA  func unionNFA(pNFA1: Graph, pNFA2: Graph): Graph {      // 处理出边/入边逻辑      // 创建新初始状态0      // 添加ε边到原NFA的初始状态（1和N+1）      // 原接受状态改为普通状态，添加ε边到新接受状态  }  状态迁移图：      ε  0 -----> 1...NFA1   | ε   └-----> N+1...NFA2  (3) product（连接运算）  核心逻辑：  func productNFA(pNFA1: Graph, pNFA2: Graph): Graph {      // 将pNFA1的接受状态与pNFA2的初始状态连接      // 复制所有状态并重新编号      // 添加ε边连接两个NFA  }  关键处理：  检查原NFA的结束状态是否有出边  使用状态重编号避免冲突  (4) closure（闭包运算）  实现方案：  func closureNFA(pNFA: Graph): Graph {      // 添加ε边：起始→接受，接受→起始      // 处理已有出边的情况      // 状态重编号保证连接正确  }  (5) zeroOrOne（0或1次运算）  代码亮点：  func zeroOrOneNFA(pNFA: Graph): Graph {      // 创建新初始状态      // 添加两条ε边：直接到接受状态 或 通过原NFA  }         ε  0 ------------> [接受]   |   ε   └-> 原NFA  func unionNFA(pNFA1: Graph, pNFA2: Graph): Graph {  ε  ↙ ↘  NFA1 NFA2  ↘ ↙  ε    详细结构：  [0]───ε───▶[1..n-1]───────────▶[maxState] (新接受状态)  │ │  │ └───▶[pNFA1内部状态迁移]  │  └───ε───▶[n..n+m-2]──────────▶[maxState] (新接受状态)  │  └───▶[pNFA2内部状态迁移]  public func unionNFA(pNFA1: Graph, pNFA2: Graph): Graph {  // ===== 前置检查阶段 =====  // 检查两个NFA的出边/入边情况（影响合并后的状态迁移）  for (e in pNFA1.pEdgeTable) {  // 检查NFA1结束状态是否有出边（影响接受状态合并）  if (e.fromState == pNFA1结束状态) hasOutEdges1 = true  // 检查是否有指向初始状态的循环边（需要状态偏移）  if (e.nextState == pNFA1初始状态) hasInEdges1 = true  }  // ===== 状态预处理阶段 =====  // 处理携带词法类别的接受状态（需要创建新接受状态）  if (hasOutEdges1 || hasCategory1) {  pNFA1结束状态.stype = UNMATCH // 原接受状态降级为普通状态  pNFA1.addState(新接受状态) // 新建独立接受状态  pNFA1.addEdge(原结束状态 → 新接受状态 via ε)  }  // ===== 状态ID偏移处理 =====  if (hasInEdges1) {  // 整体偏移状态ID（避免循环边冲突）  for (s in pNFA1状态) s.stateId += 1  for (e in pNFA1边) e.from/to += 1  pNFA1.addState(新初始状态0) // 添加偏移后的初始状态  }  // ===== 新NFA构建阶段 =====  // 合并后的NFA结构：  // [新初始状态0]  // → [pNFA1中间状态1..m-1]  // → [pNFA2中间状态m..n]  // → [最终接受状态n+1]    // 添加pNFA1中间状态（排除首尾）  for (i in 1..pNFA1.numOfStates-1) {  g.addState(状态i with原category)  }  // 添加pNFA2中间状态（偏移后）  for (i in 1..pNFA2.numOfStates-1) {  g.addState(状态i+pNFA1状态数偏移量 with原category)  }  // ===== 边重组阶段 =====  // 处理pNFA1原有边的重定向  for (e in pNFA1边) {  if (e指向原结束状态) → 重定向到新接受状态  else → 保留原连接关系  }  // 处理pNFA2边的连接  for (e in pNFA2边) {  if (来自原初始状态) → 连接到pNFA1中间状态链  else → 应用状态偏移量  }    return g  }  Case1：pNFA1有出边且pNFA2有入边  [0]───pNFA1内部边───▶[1..n-1]──ε───▶[n]───pNFA2偏移边───▶  │ ▲  └───pNFA1其他边───────────────┘  [n+1..m+n-1]───▶[m+n-1] (新接受状态)  Case2：普通串联  [0]───pNFA1内部边───▶[1..n-1]───────▶[n..n+m-2]───▶[n+m-2] (新接受状态) │ ▲  └───pNFA2偏移边────────────┘  public func productNFA(pNFA1: Graph, pNFA2: Graph): Graph {  // ===== 前置检查阶段 =====  // 检查pNFA1的接受状态是否有出边（影响连接方式）  for (e in pNFA1.pEdgeTable) {  if (e.fromState == pNFA1.pStateTable[pNFA1.numOfStates - 1].stateId) {  hasOutEdges = true  break  }  }  // 检查pNFA2的初始状态是否有入边（需要状态偏移）  for (e in pNFA2.pEdgeTable) {  if (e.nextState == pNFA2.pStateTable[0].stateId) {  hasInEdges = true  break  }  }  // ===== 分支处理阶段 =====  if (hasOutEdges && hasInEdges) {  // 情况1：两个NFA都有出/入边时的连接方式  let maxState = pNFA1.numOfStates + pNFA2.numOfStates - 1  // ... existing code ...    // 核心操作：添加ε边连接两个NFA  g.addEdge(Edge(pNFA1.numOfStates - 1, pNFA1.numOfStates , 0, NULL));  } else {  // 情况2：至少一个NFA没有出/入边时的连接方式  let maxState = pNFA1.numOfStates + pNFA2.numOfStates - 2  // ... existing code ...  }  // ===== 通用构建阶段 =====  // 构造新NFA的状态链（保留两个NFA的中间状态）  for (i in 1..(maxState)) {  g.addState(State(i, UNMATCH))  }  // 添加最终接受状态  g.addState(accept)    // ===== 边重组阶段 =====  // 移植pNFA1的原始边  for (e in pNFA1.pEdgeTable) {  g.addEdge(e)  }  // 处理pNFA2边的状态偏移  for (e in pNFA2.pEdgeTable) {  g.addEdge(Edge(e.fromState + 偏移量, e.nextState + 偏移量, e.driverId, e.etype))  }  // ... existing code ...  }    public func plusClosure(pNFA: Graph): Graph {  // ===== 状态复制阶段 =====  let maxState = pNFA.numOfStates - 1 // 原NFA结束状态ID  let g = Graph(graphIdCounter) // 创建新NFA图    // 全量复制原NFA的状态和边（深拷贝）  for (s in pNFA.pStateTable) {  g.addState(s) // 保留原状态ID和类型  }  for (e in pNFA.pEdgeTable) {  g.addEdge(e) // 保留原始转移关系  }  // ===== 循环构建阶段 =====  // 添加ε边：原接受状态 → 初始状态（形成循环）  g.addEdge(Edge(maxState, 0, 0, NULL))    return g // 返回新构造的NFA  }   1. \*\*无入边和出边\*\*：原NFA没有循环结构，直接添加ε边形成循环和新接受状态。      1. \*\*无入边但有出边\*\*：原接受状态有出边，需要创建新接受状态，并调整边连接。      1. \*\*有入边但无出边\*\*：原初始状态有入边，需要整体偏移状态ID，添加新起始状态。      1. \*\*既有入边又有出边\*\*：最复杂的情况，需要同时处理入边和出边，添加多个ε边。     public func closureNFA(pNFA: Graph): Graph {  // ===== 前置检查阶段 =====  var hasOutEdges = false // 原NFA结束状态是否有出边  var hasInEdges = false // 原NFA初始状态是否有入边    // 遍历边集检查状态依赖关系  for (e in pNFA.pEdgeTable) {  // 检查结束状态出边（可能形成循环）  if (e.fromState == pNFA结束状态ID) hasOutEdges = true  // 检查初始状态入边（需要状态偏移）  if (e.nextState == pNFA初始状态ID) hasInEdges = true  }  // ===== 分支处理阶段 =====  if (!hasInEdges && !hasOutEdges) {  // 情况1：简单循环结构  let maxState = pNFA.numOfStates - 1  // ... existing code ...  }  else if (!hasInEdges && hasOutEdges) {  // 情况2：带出边的结束状态处理  let maxState = pNFA.numOfStates  // ... existing code ...  }  else if (hasInEdges && !hasOutEdges) {  // 情况3：带入边的初始状态处理  let maxState = pNFA.numOfStates  // ... existing code ...  }  else {  // 情况4：复杂循环结构  let maxState = pNFA.numOfStates + 1  // ... existing code ...  }  }   1. 原NFA无入边/出边      1. 原NFA有出边无入边      1. 原NFA有入边无出边     4. 原NFA既有入边又有出边    public func zeroOrOneNFA(pNFA: Graph): Graph {  // ===== 前置检查阶段 =====  // 检查原NFA结束状态是否有出边（影响新接受状态位置）  for (e in pNFA.pEdgeTable) {  if (e.fromState == pNFA结束状态ID) hasOutEdges = true  }    // ===== 分支处理阶段 =====  if (!hasInEdges && !hasOutEdges) {  // 情况1：简单可选结构  g.addEdge(Edge(0, maxState, 0, NULL)) // 初始→接受  // ... existing code ...  }  else if (!hasInEdges && hasOutEdges) {  // 情况2：带出边的可选结构  g.addEdge(Edge(0, maxState, 0, NULL)) // 初始→新接受  g.addEdge(Edge(maxState-1, maxState, 0, NULL)) // 原结束→新接受  // ... existing code ...  }  else if (hasInEdges && !hasOutEdges) {  // 情况3：带入边的可选结构  g.addEdge(Edge(0, 1, 0, NULL)) // 新初始→原初始  g.addEdge(Edge(0, maxState, 0, NULL)) // 新初始→新接受  // ... existing code ...  }  else {  // 情况4：复杂可选结构  g.addEdge(Edge(0, 1, 0, NULL)) // 新初始→原初始  g.addEdge(Edge(maxState-1, maxState, 0, NULL))// 原结束→新接受  g.addEdge(Edge(0, maxState, 0, NULL)) // 新初始→新接受  // ... existing code ...  }  }  **三、**  针对上述NFA的数据结构定义，实现如下函数：  子集构造法中的3个函数：move，ε\_closure，DTran；  将NFA转化为DFA的函数：  Graph \* NFA\_to\_DFA(Graph \*pNFA)；  在这个函数的实现代码中，会创建一个DFA，作为返回值。  四、  实现了上述函数之后，请以正则表达式(a|b)\*abb来测试，检查实现代码的正确性。然后再以TINY语言的词法来验证程序代码的正确性，得出TINY语言的词法的DFA；  收获与体会：  1.理论到实践的跨越 通过实现正则表达式到NFA/DFA的转换算法，深刻理解了自动机状态转移、ε闭包、子集构造等抽象概念。特别是 `Graph.cj` 中 NFA\_to\_DFA 函数的实现，让我掌握了状态集合的幂集构造方法。  2.字符集处理能力 在 `CharSet.cj` 模块开发过程中，通过实现字符范围的并集/差集运算，掌握了连续字符段的拆分与合并策略，这对后续词法分析器的开发至关重要。  3.工程化思维培养 实验要求将零散的自动机操作（如 closureNFA 、 unionNFA 等）模块化封装，这让我学会用面向对象思想组织复杂算法，代码结构在`Graph.cj` 中体现得尤为明显。  4.调试能力提升 在实现 tokenize 函数时，通过可视化打印DFA状态表（ printTable 函数），培养了使用诊断工具追踪状态转移的能力。 | |
| 实  验成绩 |  |