1.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：

int range (char fromChar, char toChar)； // 字符的范围运算

int union(char c1, char c2)； // 字符的并运算

int union(int charSetId, char c)；// 字符集与字符之间的并运算

int union(int charSetId1,int charSetId2)；//字符集与字符集的并运算

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。

func difference(charSetId: UInt32, c: Rune): UInt32 {

let newId = charSetCounter

charSetCounter = charSetCounter + 1

var newSegmentId: UInt32 = 1

for (i in 0..charSetCounter-1) {

if (charSetTable[Int64(i)].indexId == charSetId) {

if (c < charSetTable[Int64(i)].fromChar || c > charSetTable[Int64(i)].toChar) {

// 待删除字符不在此段，直接复制

charSetTable.append(CharSet(newId, newSegmentId, charSetTable[Int64(i)].fromChar, charSetTable[Int64(i)].toChar))

newSegmentId = newSegmentId + 1

} else {

// 如果待删除字符在段中，则拆分为最多两个段

if (charSetTable[Int64(i)].fromChar < c) {

// 左侧部分：[seg.fromChar, c-1]

charSetTable.append(CharSet(newId, newSegmentId, charSetTable[Int64(i)].fromChar, Rune(UInt32(c) - 1)))

newSegmentId = newSegmentId + 1

}

if (charSetTable[Int64(i)].toChar > c) {

// 右侧部分：[c+1, seg.toChar]

charSetTable.append(CharSet(newId, newSegmentId, Rune(UInt32(c) + 1), charSetTable[Int64(i)].toChar))

newSegmentId = newSegmentId + 1

}

}

}

}

return newId

}

- 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

2.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：

Graph \* generateBasicNFA(DriverType driverType，int driverId );

Graph \* union(Graph \*pNFA1, Graph \*pNFA2)； // 并运算

Graph \* product(Graph \*pNFA1, Graph \*pNFA2); // 连接运算

Graph \* plusClosure(Graph \*pNFA) //正闭包运算

Graph \* closure(Graph \*pNFA) // 闭包运算

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。

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

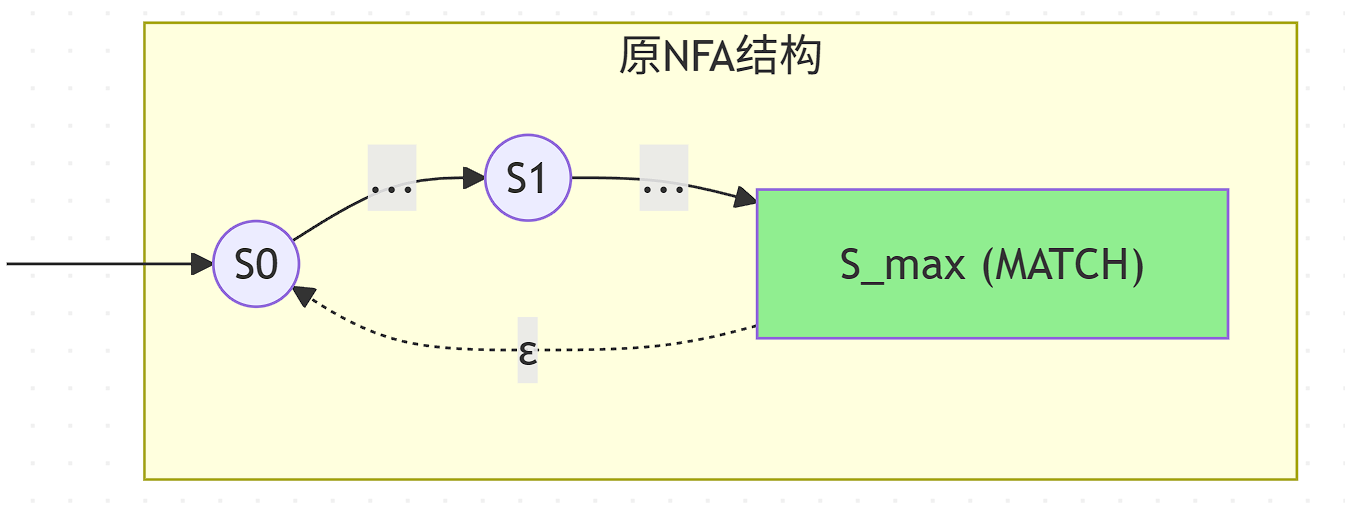
for (e in pNFA2.pEdgeTable) {

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 =====

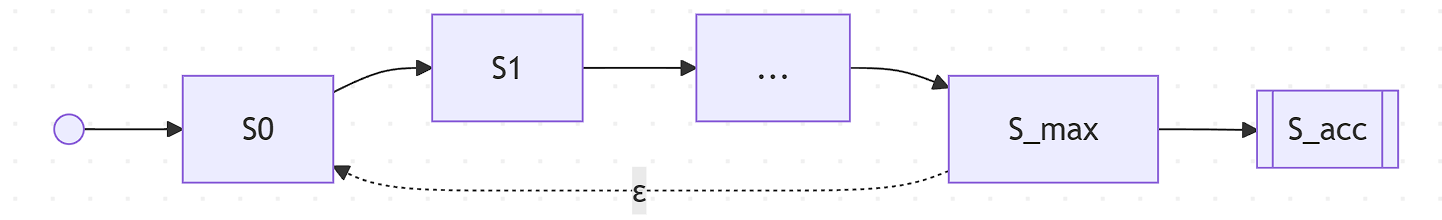
// 添加ε边：原接受状态 → 初始状态（形成循环）

g.addEdge(Edge(maxState, 0, 0, NULL))

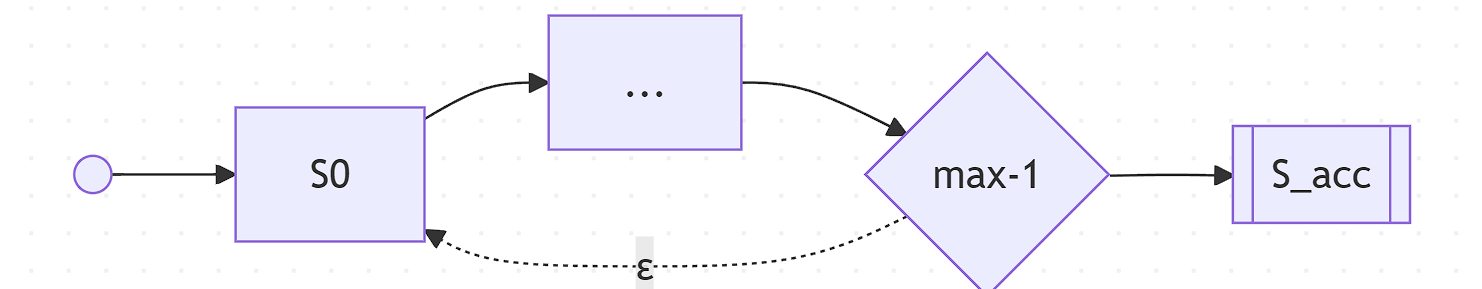
return g // 返回新构造的NFA

}

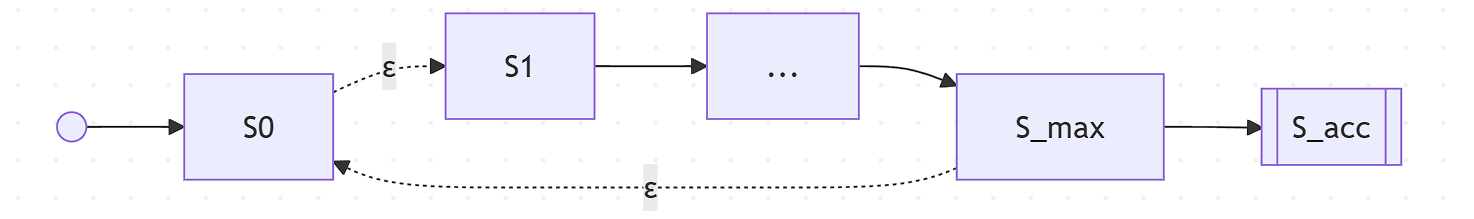
1. \*\*无入边和出边\*\*：原NFA没有循环结构，直接添加ε边形成循环和新接受状态。



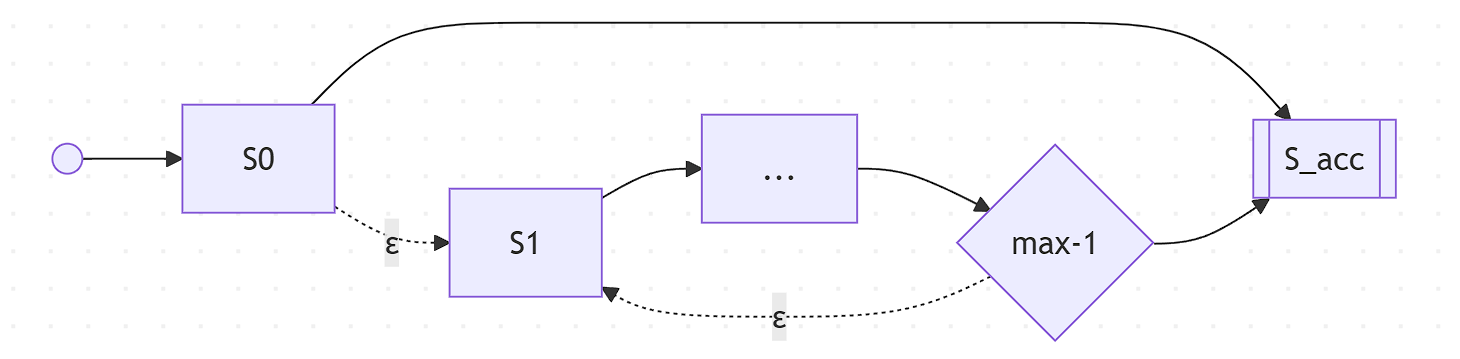
1. \*\*无入边但有出边\*\*：原接受状态有出边，需要创建新接受状态，并调整边连接。



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

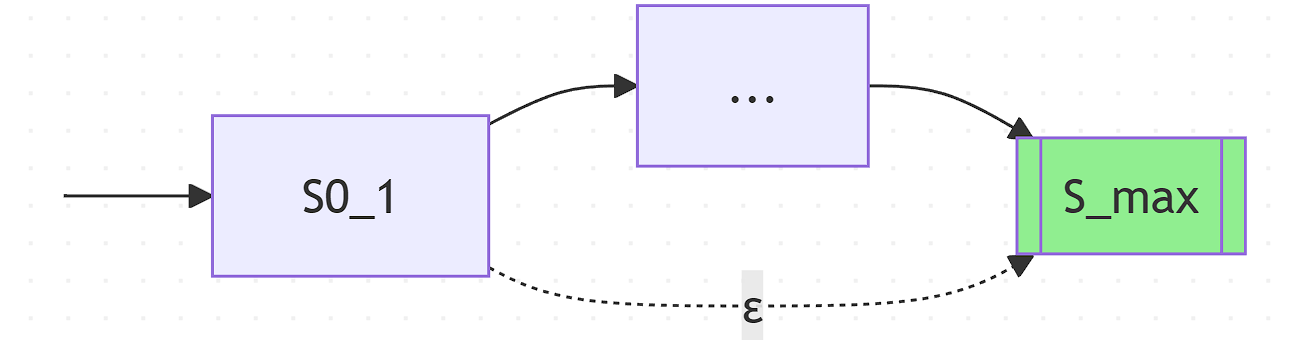
let maxState = pNFA.numOfStates + 1

// ... existing code ...

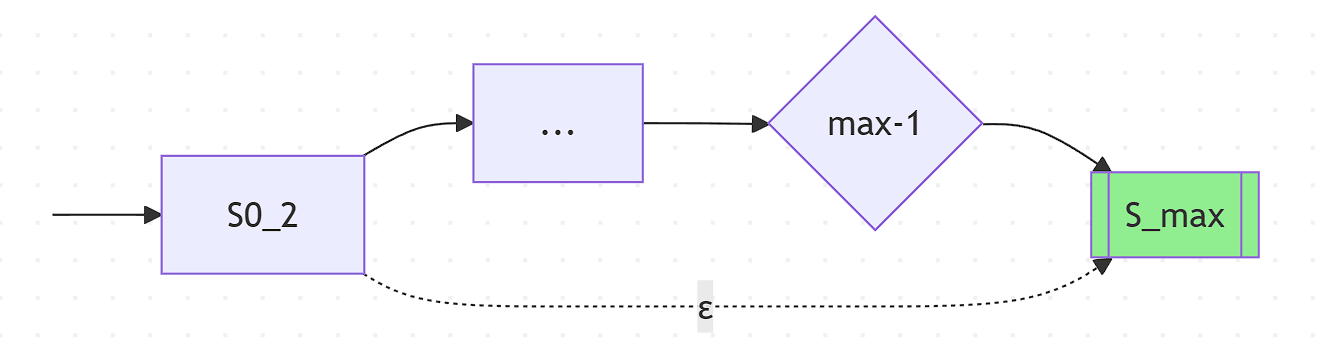
}

}

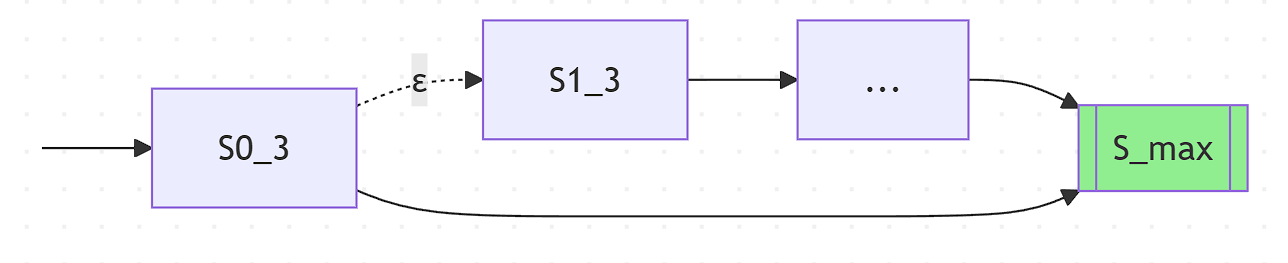
1. 原NFA无入边/出边



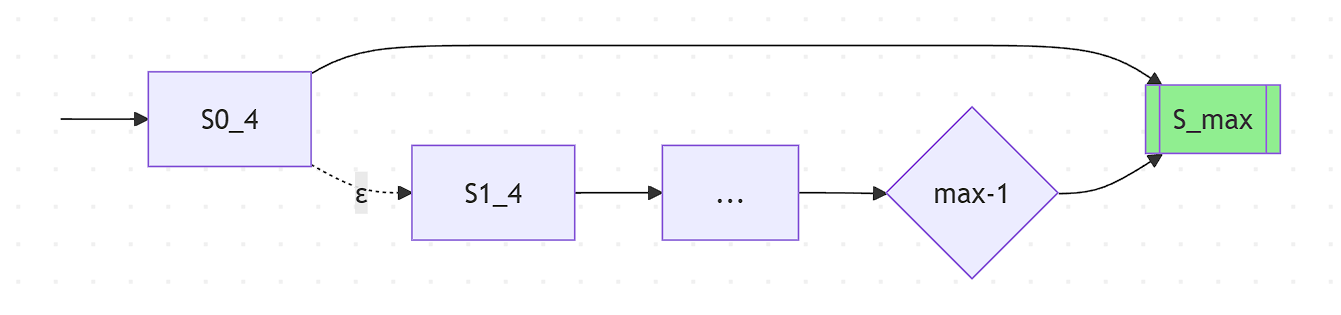
1. 原NFA有出边无入边



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 ...

}

}