编译原理 (南京大学软件学院) **Spring 2024**

第 3 节: 词法分析

Lecturer: 魏恒峰 *Scribes:* 张枕玮

§1 词法分析

1.1. 语言. 语言是字符串的集合.

定义 1.1 (字母表)**.** 字母表 Σ 是一个有限的符号集合.

现在, 符号没有任何的含义. 符号是什么意思是语义干的事情.

定义 1.2 (字母表)**.** 字母表 Σ 上的串 (s) 是由 Σ 中符号构成的一个有穷序列.

其中特殊的串是空串, $|\epsilon| = 0$

定义 1.3 (串上的连接运算)**.** 例如 $x = \text{dog}, y = \text{house} \quad xy = \text{doghouse}$, $s\epsilon = \epsilon S = s$

定义 1.4 (串上的指数运算)**.**

$$s^0 \triangleq \epsilon$$
$$s^i \triangleq ss^{i-1}, i > 0$$

定义 1.5. 语言是给定字母表 Σ 上一个任意的可数的串集合。

注:

1. 注意 $\emptyset$ 和 $\{\epsilon\}$. 后者是只有空串的语言.
2. 例如 ws : {blank, tab, newline }
3. 这就可以通过集合操作构造新的语言.

定义 1.6. 我们可以通过如下的方式构造新语言.

运算	定义和表示
L 和M 的并	$L \cup M := \{s \mid s$ 属于L 或者s 属于$M\}$
L 和M 的连接	$LM := \{st \mid s$ 属于L 且t 属于$M\}$
L 的 Kleene 闭包	$L^* := \cup_{i=0}^{\infty} L^i$
L 的正闭包	$L^+ := \cup_{i=1}^{\infty} L^i$

注: L^* (L^+) 允许我们构造无穷集合

例子 1.1. 假设 $L = \{A, B, \ldots, Z, a, b, \ldots, z\}, D = \{0, 1, \ldots, 9\}$. 那么

- $L \cup D =$ 所有的大小写字母和所有的一位数码构成的集合.
- $|LD| = 52 \times 10 = 520$.
- $L^4 =$ 所有长度为 4 的由字母构成的语言的集合.

- L^*= 所有字母构成的字符串 (包含空串) 构成的集合.
- D^+ = 所有数字构成的字符串 (不包含空串) 构成的集合.
- $L(L \cup D)^*$ = 以字母开头, 跟上 0 个或者若干个字母或者数字的字符串构成的集合. (也就是类似于 id).

1.2. 正则表达式. 注意区分语法和语义. 例如正则表达式中, 每个正则表达式 r 对应一个正则语言 $L(r)$. 正则表达式是语法, 正则语言是语义.

定义 1.7 (正则表达式)**.** 给定字母表 Σ, Σ 上的正则表达式由且仅由以下规则定义:

1. ϵ 是正则表达式;
2. $\forall a \in \Sigma, a$ 是正则表达式;
3. 如果 r 是正则表达式, 则 (r) 是正则表达式;
4. 如果 r 与 s 是正则表达式, 则 $r \mid s, rs, r^*$ 也是正则表达式.

运算优先级: ()$\succ$ $*$ $\succ$ 连接 $\succ|$

例如

$$(a)\,|((b)^*(c)) \equiv a|\,b^*c.$$

每个正则表达式 r 对应一个正则语言 $L(r)$, 这代表了它的语义.

定义 1.8 (正则表达式对应的正则语言)**.**

$$\begin{aligned} L(\epsilon) &= \{\epsilon\} \\ L(a) &= \{a\}, \forall a \in \Sigma \\ L((r)) &= L(r) \\ L(r \mid s) = L(r) \cup L(s) \quad L(rs) &= L(r)L(s) \quad L\left(r^*\right) = (L(r))^* \end{aligned}$$

例子 1.2. 如果 $\Sigma = \{a, b\}$, $L(a \mid b) = \{a, b\}$. 那么

- $L\left(a^*\right) = \{\epsilon, a, aa, aaa, \cdots, \}$.
- $L\left((a \mid b)^*\right)$= 由 a 和 b 构成的任意长度的字符串 (包括空串).
- $L\left(a \mid a^*b\right) = \{a, b, ab, aab, aaab, \cdots\}$.

实际生活中, 我们的正则表达式的列表如下:

表达式	匹配	例子
c	单个非运算符字符 c	a
$\backslash c$	字符 c 的字面值	$\backslash *$
$"s$	串 s 的字面值	$"**"$
.	除换行符以外的任何字符 (看环境, ANTLR4 中可以匹配换行符)	$a.*b$
^	一行的开始	$\hat{}abc$
$	行的结尾	$abc$$
$[s]$	字符串 s 中的任何一个字符	[abc]
$[\hat{}s]$	不在串 s 中的任何一个字符	$\hat{}[abc]$
r^*	和 r 匹配的零个或多个串连接成的串	$a*$
$r+$	和 r 匹配的一个或多个串连接成的串	$a+$
$r?$	零个或一个 r	a ?
$r\{m,n\}$	最少 m 个, 最多 n 个 r 的重复出现	$a\{1,5\}$
r_1r_2	r_1 后加上 r_2	ab
$r_1 \mid r_2$	r_1 或 r_2	$a \mid b$
(r)	与 r 相同	$(a \mid b)$
r_1/r_2	后面跟有 r_2 时的 r_1	$abc/123$

有一些简单记录方法 (在 Vim 中)

- $\backslash w$ 表示所有大小写字母, 数字, 以及下划线
- $\backslash W$ 表示除去所有大小写字母, 数字, 以及下划线
- $\backslash d$ 表示所有数码
- $\backslash D$ 表示除去所有数码

例子 1.3.

$$\left(0 \mid \left(1\,(01^*0)^*\,1\right)\right)^*$$

表示的二进制的 3 的倍数. 可以从`regex.com`中观察一些例子.

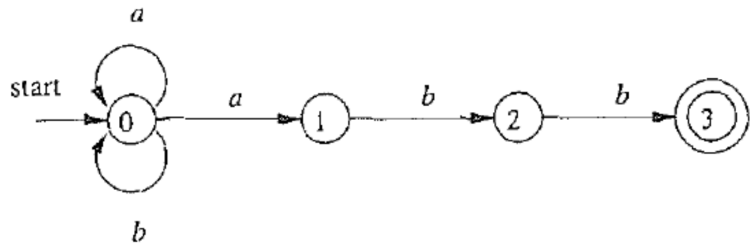
接下来介绍自动机.

定义 1.9 (NFA)**.** 非确定性有穷自动机 $\mathcal{A}$ 是一个五元组 $\mathcal{A} = (\Sigma, S, s_0, \delta, F)$:

1. 字母表 $\Sigma(\epsilon \notin \Sigma)$;
2. 有穷的状态集合 S;
3. (唯一) 的初始状态 $s_0 \in S$;
4. 状态转移函数 δ.
$$\delta : S \times (\Sigma \cup \{\epsilon\}) \to 2^S$$
5. 接受状态集合 $F \subseteq S$

注:

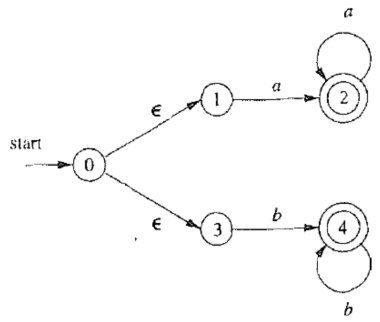
1. 非确定性指的是其 “出路” 可能不唯一. 此外, 可以接受空串然后进行转移 (称为 ϵ 转移).
2. 所有没有对应出边的字符默认指向“空状态”$\emptyset$. 此状态无论接受什么字符都到自身. 一旦进入将无法出去.



下面来看 NFA 的语义. (非确定性) 有穷自动机是一类极其简单的计算装置, 它可以识别 (接受/拒绝) Σ 上的字符串.

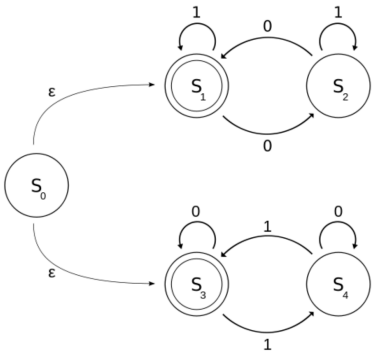
定义 1.10 (接受 (Accept))**.** (非确定性) 有穷自动机 $\mathcal{A}$ 接受字符串 x, 当且仅当存在一条从开始状态 s_0 到某个接受状态 $f \in F$ 、标号为 x 的路径.

例子 1.4. 考虑下面的自动机:



对于上面的一个状态图, $L(\mathcal{A}) = L\left((aa^* \mid bb^*)\right)$.

例子 1.5. 考虑下面的自动机:



实际上 $L(\mathcal{A}) = \{$ 包含偶数个 1 或偶数个 0 的 01 串 $\}$.

因此, $\mathcal{A}$ 定义了一种语言 $L(\mathcal{A})$: 它能接受的所有字符串构成的集合.

关于自动机 $\mathcal{A}$ 的两个基本问题:

1. Membership 问题: 给定字符串 $x, x \in L(\mathcal{A})$?
2. $L(\mathcal{A})$ 究竟是什么?

定义 1.11 (DFA)**.** 确定性有穷自动机 $\mathcal{A}$ 是一个五元组 $\mathcal{A} = (\Sigma, S, s_0, \delta, F)$:

- 字母表 $\Sigma(\epsilon \notin \Sigma)$
- 有穷的状态集合 S
- 唯一的初始状态 $s_0 \in S$
- 状态转移函数 δ

$$\delta : S \times \Sigma \to S$$

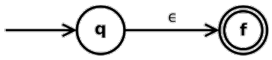
- 接受状态集合 $F \subseteq S$

上述约定 (所有没有对应出边的字符默认指向一个“死状态”) 同样适用于这里.

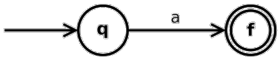
NFA 简洁易于理解, 便于描述语言 $L(\mathcal{A})$ DFA 易于判断 $x \in L(\mathcal{A})$, 适合产生词法分析器. 下面我们来走 RE $\Longrightarrow$ NFA $\Longrightarrow$ DFA $\Longrightarrow$ 词法分析器的流程.

a) 正则表达式到 NFA 要求 $L(N(r)) = L(r)$. 可以使用 Tompson 构造法. 使用四种基础的归纳.

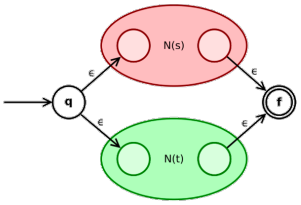
ϵ 是正则表达式。



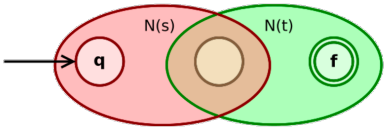
$a \in \Sigma$ 是正则表达式。



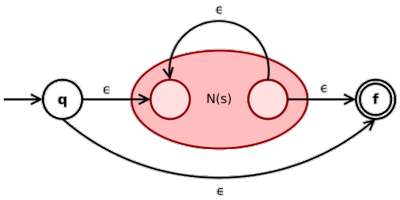
如果 s, t 是正则表达式, 则 $s|t$ 是正则表达式。



如果 s, t 是正则表达式, 则 st 是正则表达式。



如果 s 是正则表达式, 则 s^* 是正则表达式。

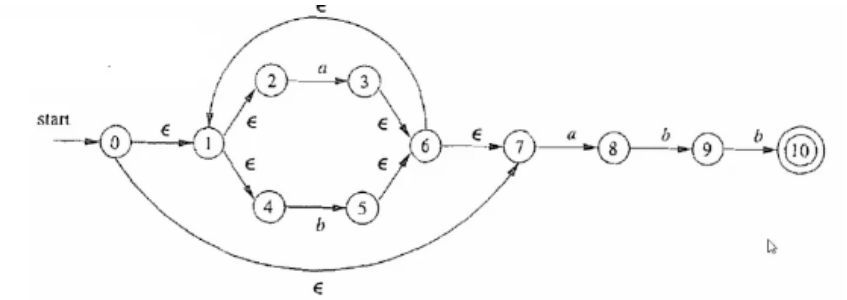


$N(r)$ 的性质以及 Thompson 构造法复杂度分析

1. $N(r)$ 的开始状态与接受状态均唯一。
2. 开始状态没有入边, 接受状态没有出边。
3. $N(r)$ 的状态数 $|S| \leq 2 \times |r|$ 。
4. 每个状态最多有两个 ϵ - 入边与两个 ϵ-出边。
5. $\forall a \in \Sigma$, 每个状态最多有一个 a-入边与一个 a-出边。($|r|$: r 中运算符与运算分量的总和)

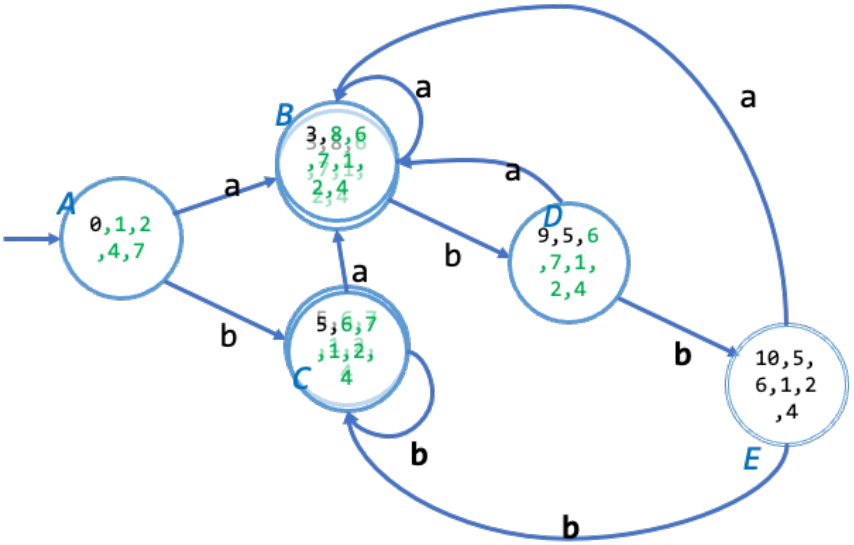
b) 从 NFA 到 DFA 的转换 主要思想: 使用 DFA 模拟 NFA – 子集构造法

假设我们想要把下图的 NFA(用正则语言表示就是 $L(\mathcal{A}) = L\left((a \mid b)^* abb\right)$) 转换为 DFA, 我们可以



- 找到 NFA 的初始状态. 也就是从起始点开始, 仅通过 ϵ 转移可以到达的所有的状态的集合. 如 0, 1, 2, 4, 7 号.
- 从这个当前的 "等价状态集合" 每个元素出发, 每次消耗字母表中的一个相同字符, 每一个状态往前走一步, 看一看到达哪些状态. 然后进行 ϵ 转移. 然后把等价的状态集合合并.
- 直到不会再出现新的状态了.

上图经过转换的等价的 DFA 是



用表格表示就是

NFA 状态	DFA 状态	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,4,5,6,7,10\}$	E	B	C

回顾 NFA 的接受是存在一条路径可以接受, 那么根据状态的等价性, 等价的状态都是可以接受的. 因此状态 E 是接受状态.

将刚刚的内容形式化:

定义 1.12 (ϵ 闭包)**.** 从状态 s 开始, 只通过 ϵ-转移可达的状态集合

$$\epsilon\text{-closure}\ (s) = \left\{ t \in S_N \mid s \xrightarrow{\epsilon^*} t \right\}$$

并且我们可以构建一个状态集合的 ϵ 闭包, 就是看每一个元素的 ϵ 闭包之并. 也就是

ϵ-closure $(T) = \bigcup_{s \in T} \epsilon$-closure (s)

定义 1.13 (move 描述)**.** 那么当前在的状态往下读入一个字符, 比如 a, 定义做一次 move, 定义为

$$\text{move}(T, a) = \bigcup_{s \in T} \delta(s, a)$$

那么子集构造法对于一个 NFA 转移为 DFA 的原理就是假设有 NFA 和 DFA, 分别叫做 N 和 D,

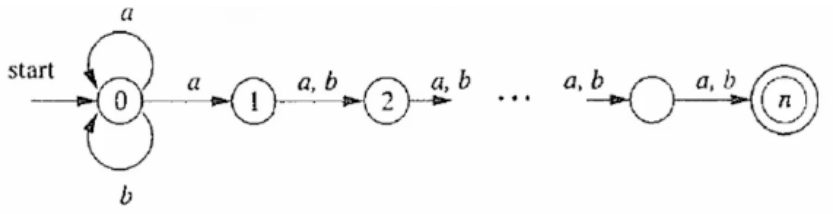
$$N : (\Sigma_N, S_N, n_0, \delta_N, F_N)$$
$$D : (\Sigma_D, S_D, d_0, \delta_D, F_D)$$

我们构造的 D 满足

1. 字母表: $\Sigma_D = \Sigma_N$
2. 状态集合: $S_D \subseteq 2^{S_N} \quad (\forall s_D \in S_D : s_D \subseteq S_N)$
3. 初始状态: $d_0 = \epsilon$-closure (n_0)
4. 转移函数: $\forall a \in \Sigma_D : \delta_D(s_D, a) = \epsilon$-closure $(\text{move}(s_D, a))$
5. 接受状态集: $F_{\mathcal{D}} = \{ s_D \in S_{\mathcal{D}} \mid \exists f \in F_N . f \in s_D \}$

下面分析构造复杂度. 假设 $|S_N| = n$, 构造出来的 DFA 的状态数不会超过 2^n. 这从状态集合的定义可以立即得出. 也就是 $|S_D| = \Theta(2^n)$. 实际上, 遗憾的事情是, 构造出的下界也是 2^n: 存在一个用例, 它恰好生成一个 2^n 个状态的 DFA. 也就是 $|S_D| = O(2^n) \cap \Omega(2^n)$. 实际上, 可以证明, 对于任何算法, 最坏情况下, $|S_D| = \Omega(2^n)$. 这说明这个复杂度是问题本身导致的, 而不是取决于发明算法人的聪明程度.

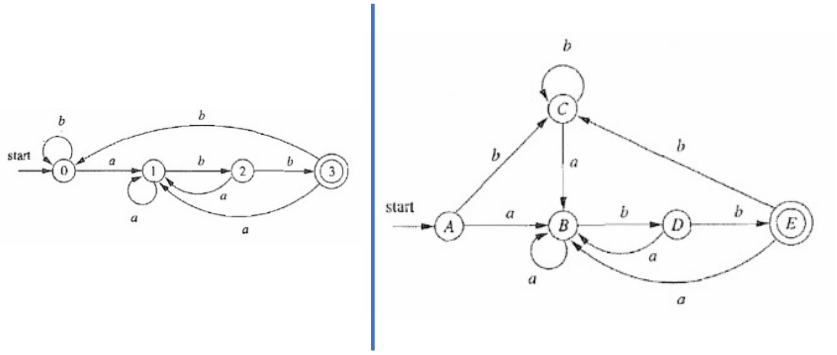
下面给出这个用例: "长度为 $m \geq n$ 个字符的 a, b 串, 且倒数第 n 个字符是 a". 其中 n 是给定的整数. 这用正则语言写作 $L_n = (a \mid b)^* a (a \mid b)^{n-1}$. 用 NFA 表示就是如下图.



其对应的 DFA 是

下面回顾闭包的概念. 我们通常会说一个 "f-closure (T)", 这表示我们在初始的集合 T 上对于每一个元素重复作用函数 f, 并且不断地把新得到的元素放到 T 中. 直到继续应用 f 之后没有新的元素加入集合为止. 换句话说, 也就是构造一系列的序列, $T \Longrightarrow f(T) \Longrightarrow f(f(T)) \Longrightarrow f(f(f(T))) \Longrightarrow \ldots$, 直到找到 x 使得 $f(x) = x$ (x 称为 f 的不动点) 的时候结束.

c) DFA 最小化 比如语言 $L(\mathcal{A}) = L\left((a \mid b)^* abb\right)$, 下图的左边和右边同样识别了我们的语言, 但是左边的更简洁. 我们希望让形如右图的 DFA 简化为左图的. 这个过程叫做 DFA 的最小化.

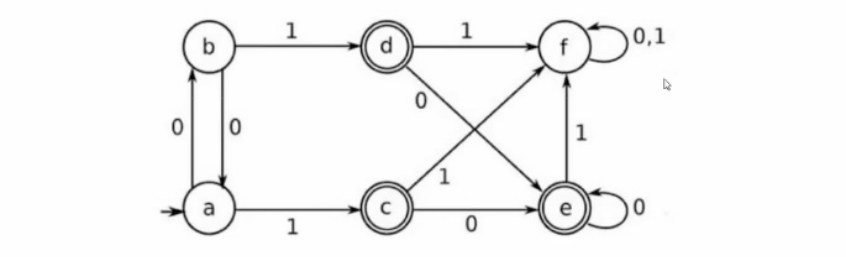


其核心思想是 "等价的状态可以合并". 但是定义等价状态绝非易事. 首先的尝试是如果状态 s 和 t 等价, 那么

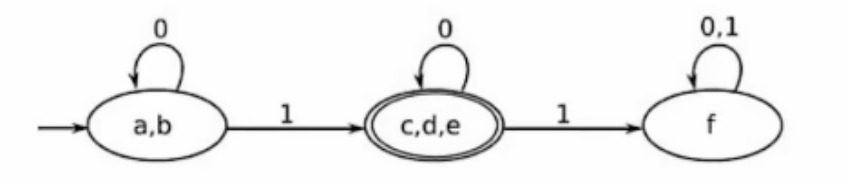
$$s \sim t \Longleftrightarrow \forall a \in \Sigma. \left(\left(s \xrightarrow{a} s'\right) \wedge \left(t \xrightarrow{a} t'\right)\right) \Longrightarrow (s' = t')$$

但是这个定义行不通. 我们考察上图右侧的例子. 可以发现 $A \sim C \sim E$. (通过 a 都可以到达状态 B, 通过 b 都可以到达 C) 但是他们显然不等价! 因为 A, C 是非结束状态, E 是结束状态. 我们把不等价的状态认做了等价的状态.

下面的例子展示了我们这样的定义同样可以把等价的状态认做不等价的状态. 如下图



例如 $a \nsim b$, 通过相同的字符到达的状态都不是同一个状态. 但是这两个状态是等价的. 如下图所示.



这个定义需要错的地方在于: 经过同一个字符, 我们应当允许到达不同的状态, 但是到达的这个不同的状态要等价. 也就是原来的定义太 “短视” 了. 它只考虑了一步的情况, 忽略了很多步最后等价的情况. 因而, 只要取一个反就好了.

定义 1.14. DFA 中的两个状态不等价, 定义如下:

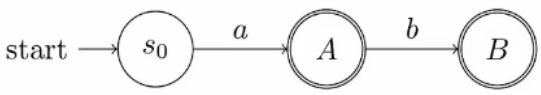
$$s \nsim t \iff \exists a \in \Sigma. \left(s \xrightarrow{a} s'\right) \wedge \left(t \xrightarrow{a} t'\right) \wedge (s' \nsim t')$$

也就是可以像这样定义两个状态的等价:

$$s \sim t \iff \forall a \in \Sigma. \left(\left(s \xrightarrow{a} s'\right) \wedge \left(t \xrightarrow{a} t'\right)\right) \implies (s' \sim t')$$

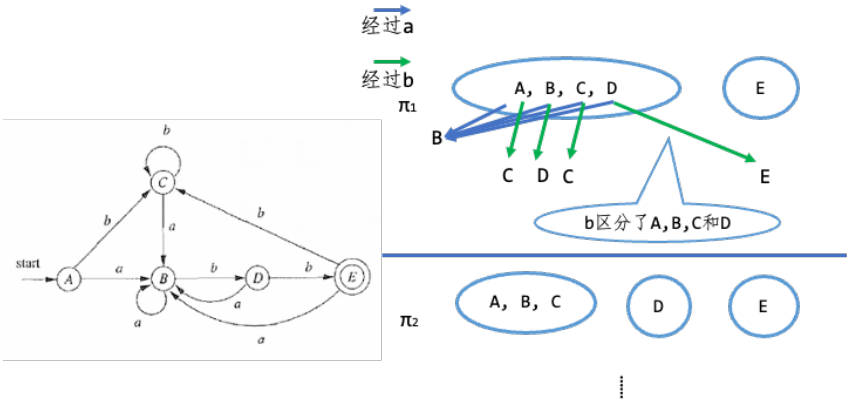
也就是存在一个字符 a. 可以 “区分” 两个状态, 使他们在相同的字符的驱使下到达了不同的状态, 也就是 s' 和 t' 不等价. 这样的定义就要求我们, 不断合并等价的状态, 直到无法合并为止. 但是, 这是一个递归定义, 从哪里开始呢?

首先, 所有接受状态都是等价的吗? 如下图的自动机, A 可以接受的是 a, B 可以接受的是 ab, 他们显然不等价. 因为如果我们可以把他俩合并的话, 整个自动机就剩下 2 个状态了, 自然不可能接受长度为 2 的字符串了.



我们的基础情况就出现问题了. 既然正面很难入手, 考虑问题的反面. 我们来划分状态, 而非合并. 首先假设所有状态都是等价的, 然后根据状态不等价的定义进行 “分裂”. 这样, 我们首先知道所有的接受状态和非接受状态肯定不等价, 于是我们现在的划分是 $\Pi = \{F, S\backslash F\}$. 这也就是我们递归的基础. 然后, 根据不等价的定义 $s \nsim t \iff \exists a \in \Sigma. \left(s \xrightarrow{a} s'\right) \wedge \left(t \xrightarrow{a} t'\right) \wedge (s' \nsim t')$ 继续分裂.

用下图的例子, 我们首先将 A, B, C, D 归为一类, E 分为一类. 接下来对于每一类, 寻找一类里面是不是存在 “这个类里面的元素受相同的字符的驱使, 随后却到达不同的状态” 的情况. 如果是这样, 我们就做分裂. 然后继续这样做. 下图展示了第一次操作.



最后的迭代过程就是

$$\Pi_0 = \{\{A, B, C, D\}, \{E\}\}$$
$$\Pi_1 = \{\{A, B, C\}, \{D\}, \{E\}\}$$
$$\Pi_2 = \{\{A, C\}, \{B\}, \{D\}, \{E\}\}$$
$$\Pi_3 = \Pi_2 = \Pi_{\text{final}}$$

实际上这就是寻找闭包的过程. 因此原来最少可以有 4 个状态, 将等价的状态合并即可. 我们在合并的时候, 会不会让它成了一个 NFA? 简而言之, 实际上是不会的. 如果有的话, 说明分裂不彻底, 还可以再分裂.

初始状态、接受状态是哪些? 原先的初始状态就是初始状态, 原来的结束状态就是结束状态.

注:

1. 注意考虑把 “死状态” 单独分为一类.
2. 这是 DFA 最小化算法. 不适用于 NFA 最小化; NFA 最小化问题是 PSPACE-complete 的 (复杂度很高).

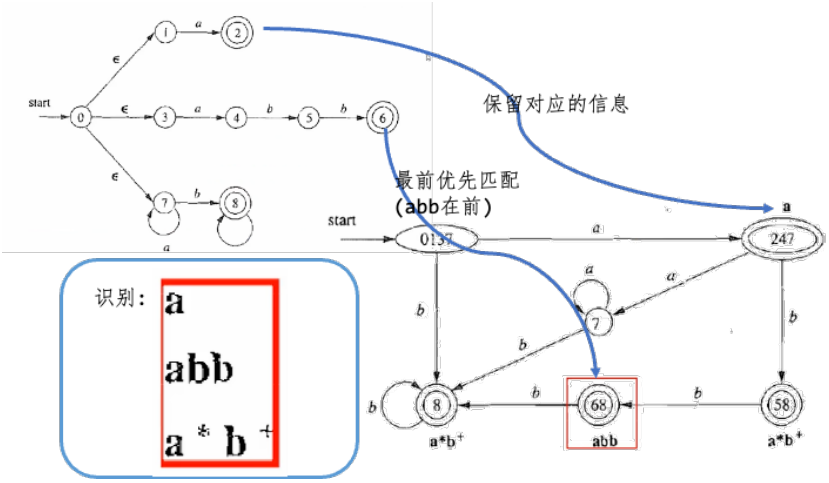
补充资料: 如何分析 DFA 最小化算法的复杂度? 为什么 DFA 最小化算法是正确的? 最小化 DFA 是唯一的吗?

定义 1.15 (可区分的 (Distinguishable); 等价的 (Equivalent))**.** 如果存在某个能区分状态 s 与 t 的字符串, 则称 s 与 t 是可区分的; 否则, 称 s 与 t 是等价的.

定义 1.16 (字符串 x 区分了 s 和 t)**.** 如果分别从 s 与 t 出发, 沿着标号为 x 的路径到达的两个状态中只有一个是接受状态, 则称 x 区分了状态 s 与 t .

然后我们得到了最小的 DFA 就可以写词法分析器了.

d) 构造词法分析器 记得我们匹配的原则: 最前优先匹配: abb (比如关键字), 以及最长优先匹配: $aabbb$. 比如我们要构造下图的词法分析器, 就要保留各个 NFA 的接受状态信息 (识别出来了个什么), 并采用最前优先匹配 (两个接受状态合并的时候, 接受状态信息要是靠前的那一个).

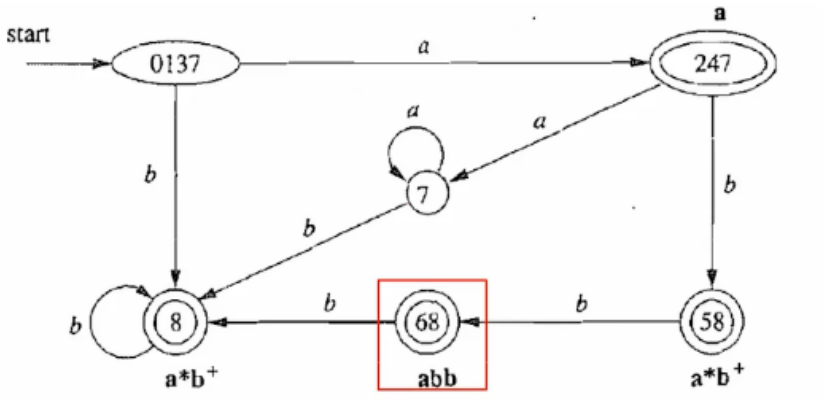


另外要注意, 需要消除"死状态", 避免词法分析器徒劳消耗输入流. 这就要保证 $\forall a \in \Sigma \cdot \delta(\emptyset, a) = \emptyset$.

做了这样的准备, 我们就可以模拟运行该 DFA, 直到无法继续为止 (输入结束或状态无转移). 假设此时状态为 s.

1. 若 s 为接受状态, 则识别成功;
2. 否则, 回溯 (包括状态与输入流) 至最近一次经过的接受状态, 识别成功;
3. 若没有经过任何接受状态, 则报错 (忽略第一个字符), 这个字符程序不被允许.

例子 1.6. 对于下图的 DFA:



- $x = a$, 输入结束; 接受; 识别出 a
- $x = abba$ 状态无转移; 回溯成功; 识别出 abb;
- $x = aaaa$ 输入结束; 回溯成功; 识别出 a
- $x = cabb$ 状态无转移; 回溯失败; 报错 c

如果希望对于 DFA 最小化, 需要做一些小改动. 初始划分需要考虑不同的词法单元. 一开始我们只是分为了接受状态, 拒绝状态, (以及死状态); 初始划分需要考虑不同的词法单元. 也就是上述例子中一开始我们应该这样分组: $\Pi_0 = \{\{0137, 7\}, \{247\}, \{8, 58\}, \{68\}, \{\emptyset\}\}$.

§2 拓展阅读: Kleene 算法

Kleene 算法主要做的是给一个 DFA 转换为 NFA. 其使用的是迭代的算法. 也就是, 一步一步地构造 NFA. 在其中的某一步, 假设有 $M = (Q, \Sigma, \delta, q_0, F)$, 这时候状态为 $Q = \{q_0, \ldots, q_n\}$. 那么算法会计算把状态机从 q_i 转移到 q_j, 而不经过编号高于 k 的节点的字符串的集合. (但是起始节点和终止节点可以高于 k). 计算的每一步都可以用正则表达式表达, 当算法对 $k = -1, 0, 1, ..., n$ 计算后, 就可以转换为正则表达式. 具体就是 $R^n_{01}|\ldots|R^n_{0f}$ 表示了对应的 NFA.

1. 初始状态, 对于 $k = -1$, 有

$$R^{-1}_{ij} = a_1|\ldots|a_m \quad \text{where } i \neq j, q_j \in \delta(q_i, a_1), \ldots, q_j \in \delta(q_i, a_m)$$
$$R^{-1}_{ii} = a_1|\ldots|a_m \mid \varepsilon \quad \text{where } q_i \in \delta(q_i, a_1), \ldots, q_i \in \delta(q_i, a_m)$$

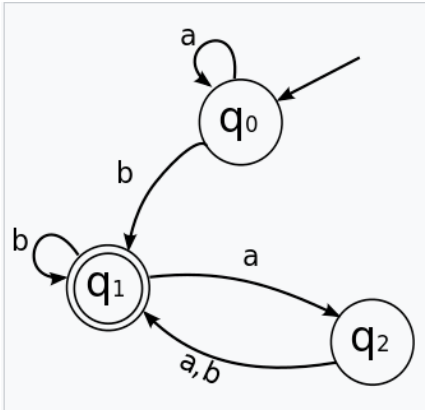
   也就是说 R_{ij} 会提到所有的字符, 在相等的时候, 我们也会提到空字符.

2. 然后每一个接下来的一步, 都会由

$$R^k_{ij} = R^{k-1}_{ik}\left(R^{k-1}_{kk}\right)^\star R^{k-1}_{kj} \mid R^{k-1}_{ij}$$

   给出. 也就是: 当状态 k 被移除时, 描述从状态 $i > k$ 到状态 $j > k$ 的路径所标记的单词的正则表达式 R^{k-1}_{ij} 被重写为 R^k_{ij}, 以便考虑通过被"消除"的状态 k 的可能性.

例子 2.1. 例如将下面的转换为 NFA:



第-1 步 (初始状态):

$$
\begin{aligned}
R_{00}^{-1} &= a \mid \varepsilon \\
R_{01}^{-1} &= b \\
R_{02}^{-1} &= \varnothing \\
R_{10}^{-1} &= \varnothing \\
R_{11}^{-1} &= b \mid \varepsilon \\
R_{12}^{-1} &= a \\
R_{20}^{-1} &= \varnothing \\
R_{21}^{-1} &= a \mid b \\
R_{22}^{-1} &= \varepsilon
\end{aligned}
$$

第 0 步:

$$
\begin{aligned}
R_{00}^{0} &= R_{00}^{-1}\left(R_{00}^{-1}\right)^{\star} R_{00}^{-1} \left| R_{00}^{-1} = (a \mid \varepsilon)(a \mid \varepsilon)^{\star}(a \mid \varepsilon) \right| a \mid \varepsilon = a^{*} \\
R_{01}^{0} &= R_{00}^{-1}\left(R_{00}^{-1}\right)^{\star} R_{01}^{-1} \left| R_{01}^{-1} = (a \mid \varepsilon)(a \mid \varepsilon)^{\star} b \right| b = a^{\star} b \\
R_{02}^{0} &= R_{00}^{-1}\left(R_{00}^{-1}\right)^{\star} R_{02}^{-1} \left| R_{02}^{-1} = (a \mid \varepsilon)(a \mid \varepsilon)^{*} \varnothing \right| \varnothing = \varnothing \\
R_{10}^{0} &= R_{10}^{-1}\left(R_{00}^{-1}\right)^{\star} R_{00}^{-1} \left| R_{10}^{-1} = \varnothing \quad (a \mid \varepsilon)^{*}(a \mid \varepsilon) \right| \varnothing = \varnothing \\
R_{11}^{0} &= R_{10}^{-1}\left(R_{00}^{-1}\right)^{\star} R_{01}^{-1} \left| R_{11}^{-1} = \varnothing \quad (a \mid \varepsilon)^{\star} b \right| b | \varepsilon = b | \varepsilon \\
R_{12}^{0} &= R_{10}^{-1}\left(R_{00}^{-1}\right)^{\star} R_{02}^{-1} \left| R_{12}^{-1} = \varnothing \quad (a \mid \varepsilon)^{*} \varnothing \right| a = a \\
R_{20}^{0} &= R_{20}^{-1}\left(R_{00}^{-1}\right)^{\star} R_{00}^{-1} \left| R_{20}^{-1} = \varnothing \quad (a \mid \varepsilon)^{*}(a \mid \varepsilon) \right| \varnothing = \varnothing \\
R_{21}^{0} &= R_{20}^{-1}\left(R_{00}^{-1}\right)^{\star} R_{01}^{-1} \left| R_{21}^{-1} = \varnothing \quad (a \mid \varepsilon)^{\star} b \right| a | b = a | b \\
R_{22}^{0} &= R_{20}^{-1}\left(R_{00}^{-1}\right)^{\star} R_{02}^{-1} \left| R_{22}^{-1} = \varnothing \quad (a \mid \varepsilon)^{*} \varnothing \right| \varepsilon = \varepsilon
\end{aligned}
$$

第 1 步:

$$
\begin{aligned}
R_{00}^{1} &= R_{01}^{0}\left(R_{11}^{0}\right)^{*} R_{10}^{0} \left| R_{00}^{0} = a^{*} b \quad (b \mid \varepsilon)^{*} \varnothing \right| a^{*} = a^{*} \\
R_{01}^{1} &= R_{01}^{0}\left(R_{11}^{0}\right)^{\star} R_{11}^{0} \left| R_{01}^{0} = a^{*} b \quad (b \mid \varepsilon)^{*}(b \mid \varepsilon) \right| a^{*} b = a^{*} b^{*} b \\
R_{02}^{1} &= R_{01}^{0}\left(R_{11}^{0}\right)^{*} R_{12}^{0} \left| R_{02}^{0} = a^{*} b \quad (b \mid \varepsilon)^{*} a \right| \varnothing = a^{*} b^{*} b a \\
R_{10}^{1} &= R_{11}^{0}\left(R_{11}^{0}\right)^{\star} R_{10}^{0} \left| R_{10}^{0} = (b \mid \varepsilon)(b \mid \varepsilon)^{*} \varnothing \right| \varnothing = \varnothing \\
R_{11}^{1} &= R_{11}^{0}\left(R_{11}^{0}\right)^{*} R_{11}^{0} \left| R_{11}^{0} = (b \mid \varepsilon)(b \mid \varepsilon)^{*}(b \mid \varepsilon) \right| b \mid \varepsilon = b^{*} \\
R_{12}^{1} &= R_{11}^{0}\left(R_{11}^{0}\right)^{\star} R_{12}^{0} \left| R_{12}^{0} = (b \mid \varepsilon)(b \mid \varepsilon)^{\star} a \right| a = b^{\star} a \\
R_{20}^{1} &= R_{21}^{0}\left(R_{11}^{0}\right)^{*} R_{10}^{0} \left| R_{20}^{0} = (a \mid b)(b \mid \varepsilon)^{*} \varnothing \right| \varnothing = \varnothing \\
R_{21}^{1} &= R_{21}^{0}\left(R_{11}^{0}\right)^{*} R_{11}^{0} \left| R_{21}^{0} = (a \mid b)(b \mid \varepsilon)^{*}(b \mid \varepsilon) \right| a \mid b = (a \mid b) b^{*} \\
R_{22}^{1} &= R_{21}^{0}\left(R_{11}^{0}\right)^{\star} R_{12}^{0} \left| R_{22}^{0} = (a \mid b)(b \mid \varepsilon)^{*} a \right| \varepsilon = (a \mid b) b^{*} a \mid \varepsilon
\end{aligned}
$$

第 2 步:

$$R_{00}^2 = R_{02}^1 \left(R_{22}^1\right)^\star R_{20}^1 \left| R_{00}^1 \right. = a^*b^*ba \quad ((a \mid b)b^*a \mid \varepsilon)^\star \varnothing \quad \left| a^* \right. = a^*$$

$$R_{01}^2 = R_{02}^1 \left(R_{22}^1\right)^\star R_{21}^1 \left| R_{01}^1 \right. = a^*b^*ba \quad ((a \mid b)b^\star a \mid \varepsilon)^\star (a \mid b)b^\star \quad \left| a^\star b^\star b \right. = a^*b(a(a \mid b) \mid b)^\star$$

$$R_{02}^2 = R_{02}^1 \left(R_{22}^1\right)^\star R_{22}^1 \left| R_{02}^1 \right. = a^\star b^\star ba \quad ((a \mid b)b^\star a \mid \varepsilon)^\star ((a \mid b)b^\star a \mid \varepsilon) \left| a^\star b^\star ba \right. = a^*b^\star b \left(a(a \mid b)b^\star\right)^\star a$$

$$R_{10}^2 = R_{12}^1 \left(R_{22}^1\right)^\star R_{20}^1 \left| R_{10}^1 \right. = b^\star a \quad ((a \mid b)b^\star a \mid \varepsilon)^\star \varnothing \quad \left| \varnothing \right.$$

$$R_{11}^2 = R_{12}^1 \left(R_{22}^1\right)^\star R_{21}^1 \mid R_{11}^1 = b^\star a \quad ((a \mid b)b^\star a \mid \varepsilon)^\star (a \mid b)b^\star = (a(a \mid b) \mid b)^*$$

$$R_{12}^2 = R_{12}^1 \left(R_{22}^1\right)^\star R_{22}^1 \left| R_{12}^1 \right. = b^\star a \quad ((a \mid b)b^\star a \mid \varepsilon)^\star ((a \mid b)b^\star a \mid \varepsilon) \left| b^\star a \right. = (a(a \mid b) \mid b)^\star a$$

$$R_{20}^2 = R_{22}^1 \left(R_{22}^1\right)^\star R_{20}^1 \mid R_{20}^1 = ((a \mid b)b^\star a \mid \varepsilon) ((a \mid b)b^\star a \mid \varepsilon)^\star \varnothing \quad \varnothing \quad \varnothing$$

$$R_{21}^2 = R_{22}^1 \left(R_{22}^1\right)^\star R_{21}^1 \left| R_{21}^1 \right. = ((a \mid b)b^*a \mid \varepsilon) ((a \mid b)b^*a \mid \varepsilon)^\star (a \mid b)b^* \quad \left| (a \mid b)b^* \right. = (a \mid b)(a(a \mid b) \mid b)^*$$

$$R_{22}^2 = R_{22}^1 \left(R_{22}^1\right)^\star R_{22}^1 \left| R_{22}^1 \right. = ((a \mid b)b^\star a \mid \varepsilon) ((a \mid b)b^\star a \mid \varepsilon)^\star ((a \mid b)b^\star a \mid \varepsilon) \left| (a \mid b)b^\star a \mid \varepsilon \right. = ((a \mid b)b^\star a)^\star$$

§3 设计模式: Visitors 和 Listener

(1) Listener/Observer 模式我们要感知某事件, 并在事件发生时通知依赖对象做出相应的动作. 在 ANTLR4 中可以通过继承 Enterxxx 或者 Exitxxx 代表了进入某个节点以及离开了某个节点. 在 BaseListener 可以查看可以重写哪些方法.

(2) 需要对一个包含若干类型的元素的数据结构进行遍历. 例如 List<Element>, 对于不同类型的 Element, 需要执行不同的操作. 这时候可以定义一个 accept 接口, 然后对于每一个类型者单独定制他们自己的 accept 操作, 这样依赖在外部代码遍历的时候可以直接调用 accept 方法.