



# 编译原理

Compilers

极夜酱

# 目录

<b>1</b>	<b>有限状态自动机</b>	<b>1</b>
1.1	字母表 . . . . .	1
1.2	语言 . . . . .	4
1.3	DFA . . . . .	6
1.4	NFA . . . . .	9
1.5	正则表达式 . . . . .	11
<b>2</b>	<b>上下文无关语言</b>	<b>14</b>
2.1	. . . . .	14

# Chapter 1 有限状态自动机

## 1.1 字母表

### 1.1.1 字母表 (Alphabet)

字母表是一个非空的有限集合，一般用  $\Sigma$  表示，集合中的元素被称为符号/字符 (symbol)。

例如：

- $\Sigma = \{0, 1\}$ ：二进制数集合。
- $\Sigma = \{a, b, \dots, z\}$ ：小写字母集合。
- $\Sigma = \{ (, ), [, ], \{, \} \}$ ：括号集合。

### 1.1.2 串 (String)

串是一个由字母表中的字符组成的有限序列。

例如：

- 0011 和 11 是  $\Sigma = \{0, 1\}$  上的串。
- abc 和 bbb 是  $\Sigma = \{a, b, \dots, z\}$  上的串。
- $()()$  和  $(())$  是  $\Sigma = \{ (, ), [, ], \{, \} \}$  上的串。

#### 空串

空串使用  $\epsilon$  表示。

## 串的长度

- $|0010| = 4$
- $|aa| = 2$
- $|\epsilon| = 0$

## 前缀 (prefix)

- aa 是 aaabc 的前缀
- aaab 是 aaabc 的前缀
- aaabc 是 aaabc 的前缀

## 后缀 (suffix)

- bc 是 aaabc 的后缀
- abc 是 aaabc 的后缀
- aaabc 是 aaabc 的后缀

## 子串 (substring)

- ab 是 aaabc 的子串
- aaa 是 aaabc 的子串
- aaabc 是 aaabc 的子串

## 连接 (concatenation)

当  $\omega = abd$ ,  $\alpha = ce$ , 那么  $\omega\alpha = abdce$ 。

### 指数 (exponentiation)

当  $\omega = abd$ , 那么  $\omega^3 = abdabdabd$ ,  $\omega^0 = \epsilon$ 。

### 反转 (reversal)

当  $\omega = abd$ , 那么  $\omega^R = dba$ 。

### 1.1.3 克林闭包 (Kleene Closure)

$\Sigma^k$  用于表示所有在字母表  $\Sigma$  上的长度为  $k$  的串的集合。

例如,  $\Sigma = \{a, b\}$ , 那么  $\Sigma^2 = \{ab, ba, aa, bb\}$ ,  $\Sigma^0 = \{\epsilon\}$ 。

克林闭包  $\Sigma^*$  用于表示所有在字母表  $\Sigma$  上能够组成的串的集合。

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots = \bigcup_{k \geq 0} \Sigma^k \quad (1.1)$$

正闭包  $\Sigma^+$  则是在  $\Sigma^*$  中除了空串以外的所有串的集合。

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots = \bigcup_{k > 0} \Sigma^k \quad (1.2)$$

## 1.2 语言

### 1.2.1 语言 (Language)

语言是一个字母表中所构成串的集合。

例如,  $\Sigma = \{a, b, c, \dots, z\}$ , 那么所有英语单词所构成的集合  $L$  就是字母表  $\Sigma$  上的语言。

假设  $A = \{good, bad\}$  和  $B = \{boy, girl\}$  是两个语言, 语言之间可以进行以下操作。

#### 并集 (union)

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\} \quad (1.3)$$

$$A \cup B = \{good, bad, boy, girl\}$$

#### 连接 (concatenation)

$$A \circ B = \{xy \mid x \in A \text{ or } y \in B\} \quad (1.4)$$

$$A \circ B = \{goodboy, goodgirl, badboy, badgirl\}$$

#### 闭包

$$A^* = \{x_1, x_2, \dots, x_k \mid k \geq 0 \text{ and each } x_i \in A\} \quad (1.5)$$

$$A^* = \{\epsilon, good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, goodgoodbad, \dots\}$$

语法和语言与自动机理论密切相关, 它们是很多软件实现的基础, 例如编译器/解释器、文本编辑器、文本搜索、系统验证等。

在自动机理论中, 要处理的问题就是判断一个给定的串是否属于某个语言。

例如:

- $0^*10^*$ : 只包含一个 1 的串的集合。
- $\Sigma^*1\Sigma^*$ : 至少有一个 1 的串的集合。
- $\Sigma^*001\Sigma^*$ : 包含子串 001 的串的集合。
- $(\Sigma\Sigma)^*$ : 长度为偶数的串的集合。
- $(\Sigma\Sigma\Sigma)^*$ : 长度为 3 的倍数的串的集合。

## 1.3 DFA

### 1.3.1 DFA (Deterministic Finite Automaton)

有限状态机 (FSM, Finite State Machine) 用于决定程序当前状态和状态间的切换，状态机最终只能指向一个结果。

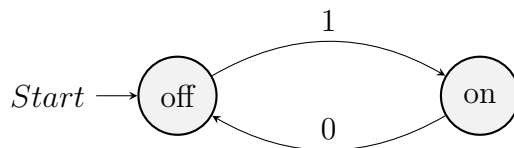
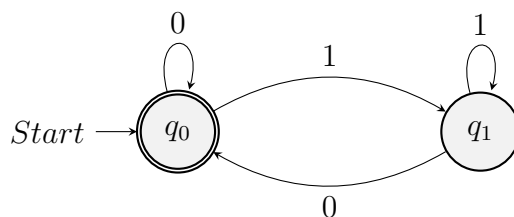


图 1.1: 有限状态机

确定性有限状态自动机 DFA 使用一个五元组  $(Q, \Sigma, \delta, q_0, F)$  表示，其中

- $Q$ : 状态的集合
- $\Sigma$ : 字母表
- $\delta$ : 状态转移函数 (transition function)
- $q_0$ : 初始状态
- $F$ : 终结状态集合

例如 DFA 可以用来识别空串或者以 0 结尾的串：



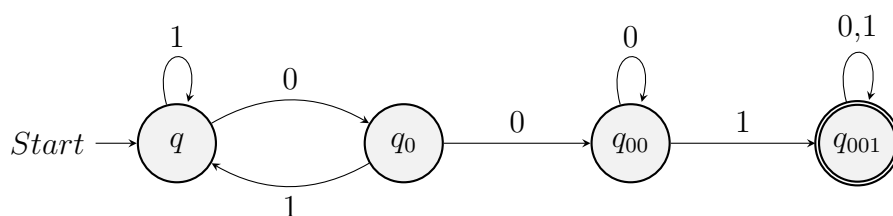
其中  $Q = \{q_0, q_1\}$ ,  $\Sigma = \{0, 1\}$ ,  $q_0$  为初始状态,  $F = \{q_0\}$ ,  $\delta$  为

状态	输入	
	0	1
$q_0$	$q_0$	$q_1$
$q_1$	$q_0$	$q_1$



能够被有限自动机接受的语言被称为正则语言 (regular language)。

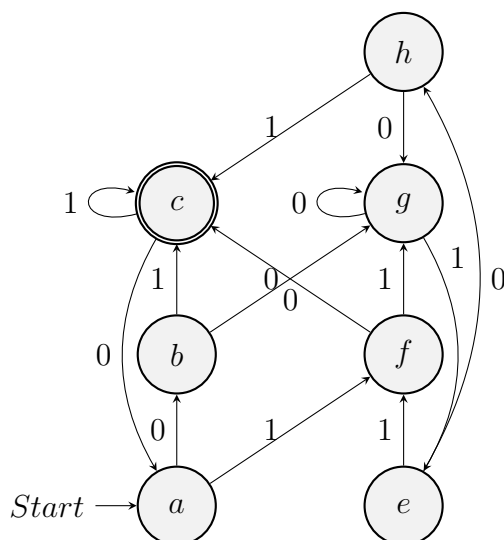
例如，构建一个能够识别所有包含子串 001 的串的 DFA：



### 1.3.2 最小化 DFA

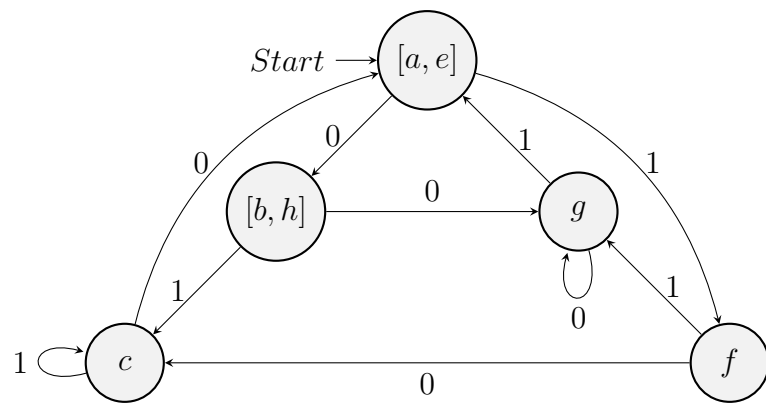
有限状态机的最小化，即将一个有限状态机转换为一个更小的有限状态机，使得状态的数目最少。

对于两个状态，如果它们之间的转移函数相同，则这两个状态可以合并为一个状态。



在这个 DFA 中，状态  $b$  和  $h$  是等价的，当接收 0 时都转移到状态  $g$ ，当接收 1 时都转移到状态  $c$ 。同时状态  $a$  和  $e$  也是等价的，状态  $a$  接收 0 转移到状态  $b$ ，状态  $e$  接收 0 转移到状态  $h$ ，状态  $a$  和  $e$  接收 1 时都转移到状态  $f$ 。

因此，状态  $b$  和  $h$  以及状态  $a$  和  $e$  可以进行合并。

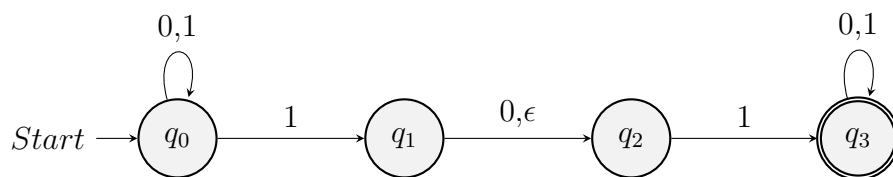


## 1.4 NFA

### 1.4.1 NFA (Non-deterministic Finite Automaton)

在 DFA 中，每个状态的下一个状态都是唯一确定的，但是非确定性有限状态自动机 NFA 可能会存在多个下一状态。

例如在这个 NFA 中，状态  $q_0$  存在两个接收 1 的箭头，而状态  $q_1$  没有接收 1 的箭头。

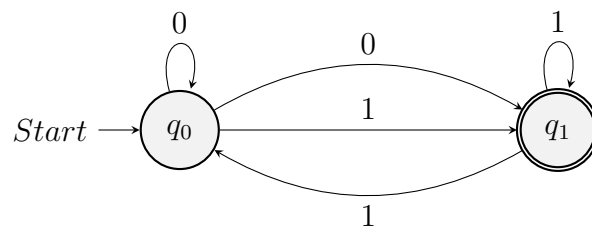


因此，在 NFA 中，每个状态允许对相同输入存在 0 个、1 个或多个转移的状态。如果存在一条能够到达终结状态的路径，那么就称当前的输入是被 NFA 接受的。

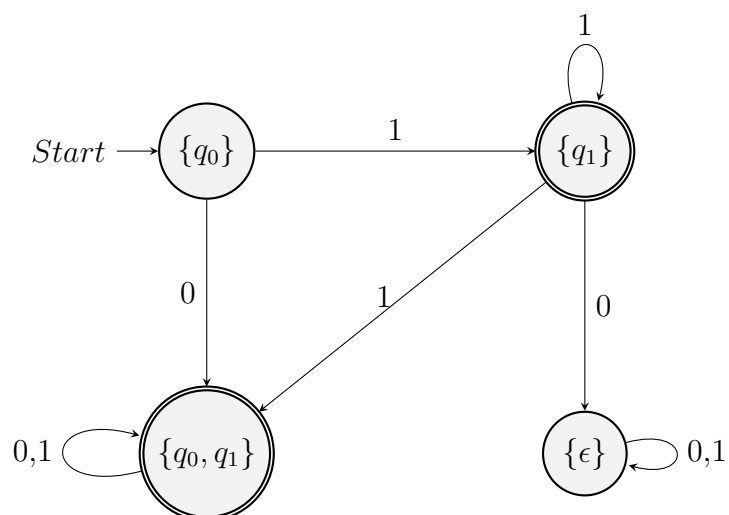
### 1.4.2 DFA 与 NFA 的转换

NFA 并不比 DFA 更加强大，理论证明 NFA 与 DFA 是等价的。

例如将一个 NFA 转换为 DFA：



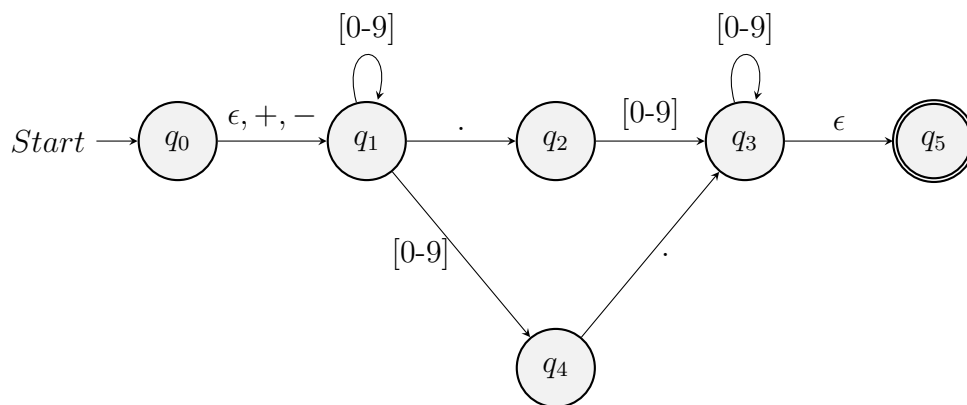
构建一个与 NFA 等价的 DFA，只需将 NFA 中转换到的状态集合作为 DFA 中的一个状态即可。



### 1.4.3 $\epsilon$ -NFA

$\epsilon$ -NFA 允许不消耗输入字符在状态之间转移。

例如以下  $\epsilon$ -NFA 能够接受小数，如 +3.14、-0.12、.71、2. 等。



## 1.5 正则表达式

### 1.5.1 编译器 (Compiler)

编译器是一种特殊的程序，可以将一种编程语言的源代码翻译成机器码、字节码或另一种编程语言。

编译器包含以下阶段：

1. 词法分析器 (lexical analyzer)
2. 语法分析器 (syntax analyzer)
3. 语义分析器 (semantic analyzer)
4. 中间代码生成器 (intermediate code generator)
5. 代码优化器 (code optimizer)
6. 代码生成器 (code generator)

### 1.5.2 词法分析

词法分析是编译器的第一步，它的主要任务是读取源代码，并生成能够被解析器 (parser) 进行语法分析的 tokens 和语法树 (syntax tree)。

例如 `time = hour * 60 + minute`，经过词法分析后，将会得到：

- `id(time)`
- `assignment(=)`
- `id(hour)`
- `op(*)`
- `num(60)`
- `op(+)`

- `id(minute)`

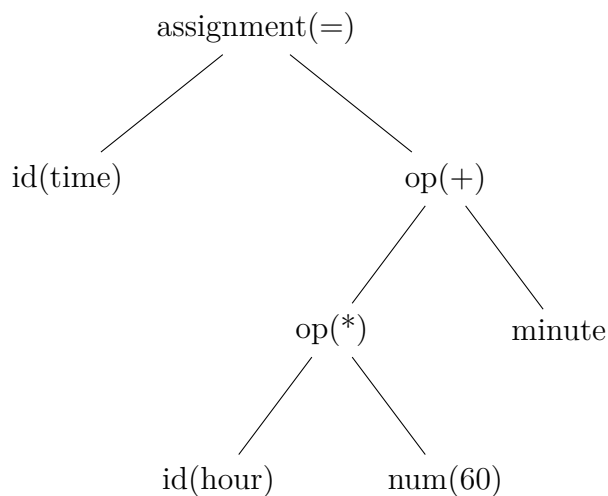


图 1.2: 语法树

### 1.5.3 正则表达式 (Regex, Regular Expression)

正则表达式描述了字符串匹配的模式 (pattern)，可以用来检查一个串是否包含某个子串、替换子串、或提取符合条件的子串。像 `grep`、`vi`、`python`、`lex` 等工具都支持正则表达式的使用。

例如用于匹配一个合法的变量名的正则表达式为 `[a-zA-Z_][a-zA-Z0-9_]*`。即变量名只能由字母或下划线开头，后面可以是任意多个字母、数字或下划线。

正则表达式支持以下操作：

- 连接： $ab$  或  $a \cdot b$
- 选择： $a \mid b$
- 克林闭包： $a^* = \{\epsilon, a, aa, aaa, \dots\}$
- 匹配至少 1 次： $a^+ = aa^*$
- 匹配 0 次或 1 次： $a? = a \mid \epsilon$
- 匹配任意字符： $.$

- 补集:  $(a \mid b)$

其中克林闭包运算的优先级最高，其次是连接，最后是选择。

例如  $(a \mid b)^*aa(a \mid b)^*$  用于匹配包含连续的 a 的串， $b^*(abb^*)(a \mid \epsilon)$  用于匹配没有连续的 a 的串。

然而  $a^n b^n$  ( $n \geq 0$ ) 却不是正则语言，因为它无法用有限个状态来验证 a 和 b 的出现次数是相等的。

## Chapter 2 上下文无关语言

### 2.1

#### 2.1.1