# 2 词法分析

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆⋆ 掌握正则表达式的表示方法及其应用，包括字符表示、构造方式和常见扩展符号
- ⋆ 理解 Thompson 构造法及其实现原理，能够将正则表达式转换为 NFA
- ⋆ 理解子集构造法，能够将 NFA 转换为 DFA

## 2.1 词法声明：正则表达式

正则表达式（Regular Expression，简称 Regex）是一种形式化工具，用于描述字母表 $\Sigma$ 上的字符串集合。它由基本字符元素及一系列构造规则组成，能够精确地定义文本模式，因此在词法分析、文本搜索以及其他各种字符串匹配任务中得到广泛应用。

### 2.1.1 字符表示

单个字符元素是正则表达式的最基本组成单位，用于目标字符串中的单个符号。表 2.1 列出了常用的单个字符元素的表示方法及其对应的含义。

表 2.1: 单个字符元素正则表示方法

| 字符元素 | 表述形式 | 含义 |
|---|---|---|
| 特定字符 | $a$ | $x = a$ |
| 字符范围 | $[ab]$ | $x \in \{a, b\}$ |
| 字符范围 | $[a - z]$ | $x \in \{a, ..., z\}$ |
| 字符范围 | $[a - zA - Z]$ | $x \in \{a, ..., z, A, ..., Z\}$ |
| 通配符 | . | $x \in \Sigma$ |
| 排除特定字符 | $\hat{a}$ | $x \in \Sigma \backslash \{a\}$ |
| 空字符 | $\epsilon$ | $x \in \emptyset$ |
| 特定字符或空 | $a?$ | $x = a$ or $x = \epsilon$ |

### 2.1.2 构造方式

字符元素可以通过选择（Union）、连接（Concatenation）和闭包（Kleene Closure）三种基本方式组合，形成更复杂的正则表达式。表 2.2列出了常用的构造方法及其优先级。为了保持一般性，这些构造方式通常采用递归定义，即复杂正则表达式可以由子正则表达式逐层组合而成。

表 2.2: 正则表达式构造方法，其中 S 和 T 为子正则表达式或单个字符

| 构造方式 | 符号 | 优先级 | 示例 | 含义 |
|---|---|---|---|---|
| 选择 | $\mid$ | 1 | $S \mid T$ | $x \in \{S \cup T\}$ |
| 连接 | | 2 | $ST$ | $x \in \{st \mid s \in S, t \in T\}$ |
| 闭包 | $*$ | 3 | $S^*$ | $x \in \{s_1..s_n \mid \forall 1 \le i \le n, s_i \in S, 0 \le n < \infty\}$ |
| 正闭包（扩展） | $+$ | 3 | $S^+$ | $x \in \{s_1..s_n \mid \forall 1 \le i \le n, s_i \in S, 1 \le n < \infty\}$ |
| 区间闭包（扩展） | | 3 | $S^{\{min,max\}}$ | $x \in \{s_1..s_n \mid \forall 1 \le i \le n, s_i \in S, min \le n \le max\}$ |

在解析正则表达式时，需要遵循一定的优先级规则：闭包 > 连接 > 选择。例如，对于正则表达式“`a|bc`$^*$”，按照优先级进行解析后，其对应的字符串集合（亦称为正则集）为 `{a, bc, bcc, ...}`。需要注意的是，字符串“`abc`”并不属于该集合。在定义正则表达式时，还可以通过使用括号来改变运算的结合方式。例如，正则表达式“`(a|b)c`$^*$”所表示的正则集为 `{ac, bc, acc, bcc, ...}`。

下面我们使用正则表达式来设计一个简单计算器的词法规则。首先，计算器的输入符号来自一个有限符号集：

$$\Sigma = \{0, 1, 3, 4, 5, 6, 7, 8, 9, \ .\,, +, -, *, /, \hat{}\,, (, )\}$$

表 2.3 给出了该计算器的词法规则定义。

表 2.3: 计算器词法定义

| 词元类型 | 含义 | 定义 |
|---|---|---|
| <UNUM> | 无符号数字 | $[0\text{-}9]^+(.[0\text{-}9]^+ \mid \epsilon)$ |
| <ADD> | 加号 | $+$ |
| <SUB> | 减号 | $-$ |
| <MUL> | 乘号 | $*$ |
| <DIV> | 除号 | $/$ |
| <EXP> | 指数运算 | ˆ |
| <LPAR> | 左括号 | ( |
| <RPAR> | 右括号 | ) |

需要注意的是，当一组正则表达式规则同时使用时，可能会产生二义性。具体而言，不同词元类型的规则之间可能存在交集，从而使某些字符串同时符合多种词元类型的定义。例如，在编程语言中，标识符与许多关键字之间往往存在交集。在这种情况下，字符串“`iffy`”（本应被解析为一个标识符）可能会被错误地识别为关键字 `if` 与标识符 `fy` 的组合。

在实际的词法分析工具中，通常可以通过以下两种方式解决二义性问题：

- 通过约定规则顺序消除二义性，即优先匹配先定义的规则；
- 通过为关键字设置边界约束来排除非关键字的情况，例如规定关键字之后只能出现特定的分隔符（如空格、换行或括号等）。

此外，在使用正则表达式扫描字符串时默认采用最长匹配原则：若已匹配到某个词元且仍可继续向前匹配，则应尽可能匹配更长的字符串。例如，字符串“`iffy`”应被识别为一个标识符，而不是四个标识符。

## 2.2 词法解析：有穷自动机及其构造

本节将解决一个重要问题：给定一组由正则表达式定义的词元规则，如何自动生成相应的词元识别程序。从理论上讲，所有正则表达式都可以用确定性有穷自动机（DFA：Deterministic Finite Automaton）表示，而 DFA 又可以进一步转换为等价的程序实现。

**定义 1** (有穷自动机（FA: Finite Automaton）)**.** 有穷自动机是一个五元组：$(S, s_0, T, \Sigma, \Delta)$，其中：

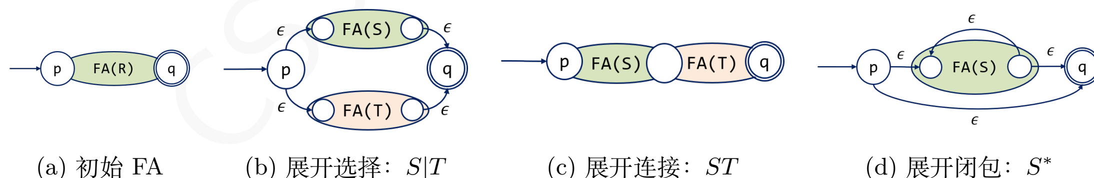
- $S$ 是有限状态集合；
- $s_0 \in S$ 是初始状态；
- $T \subseteq S$ 是结束状态集合；
- $\Sigma$ 是有限字符集合；
- $\Delta \subseteq S \times \Sigma \times S$ 是边的集合，表示状态转移关系，其中每个转移对应一个输入字符和两个状态。若对于任意状态 $s_i \in S$ 和输入字符，至多存在一个转移目标状态，则该有穷自动机称为确定性有穷自动机；否则称为非确定性有穷自动机（NFA: Non-Deterministic Finite Automaton）。

将一组正则表达式转换为 DFA 的过程大致可以分为以下三个步骤：

1) 根据正则表达式构造 NFA；
2) 将 NFA 转换为 DFA；
3) 对 DFA 进行优化。

### 2.2.1 根据 Regex 构造 NFA

本节介绍一种经典的 NFA 构造算法：McNaughton–Yamada–Thompson 构造法（简称 Thompson 构造法）[1, 3]。该方法的基本思想是：针对正则表达式中的三种基本构造方式，分别设计相应的 NFA 构造规则。具体而言，从初始 NFA 出发，按照运算符的优先顺序（逆序）递归展开正则表达式，并根据当前子表达式的构造方式选择相应的 NFA 构造规则进行组合，从而最终得到与该正则表达式等价的 NFA。




(a) 初始 FA　(b) 展开选择：$S|T$　(c) 展开连接：$ST$　(d) 展开闭包：$S^*$

图 2.1: Thompson 构造法

图 2.1 展示了初始 NFA 以及表 2.2 中三种主要正则构造方式对应的 NFA 构造方法。从图 2.1a 中的初始 NFA 出发，该 NFA 的边表示目标正则表达式 $R$。若 $R$ 的最外层构造为选择 $S|T$，则按照图 2.1b 展开；若为连接 $ST$，则根据图 2.1c 展开；若为闭包 $S^*$，则采用图 2.1d 的方式展开。

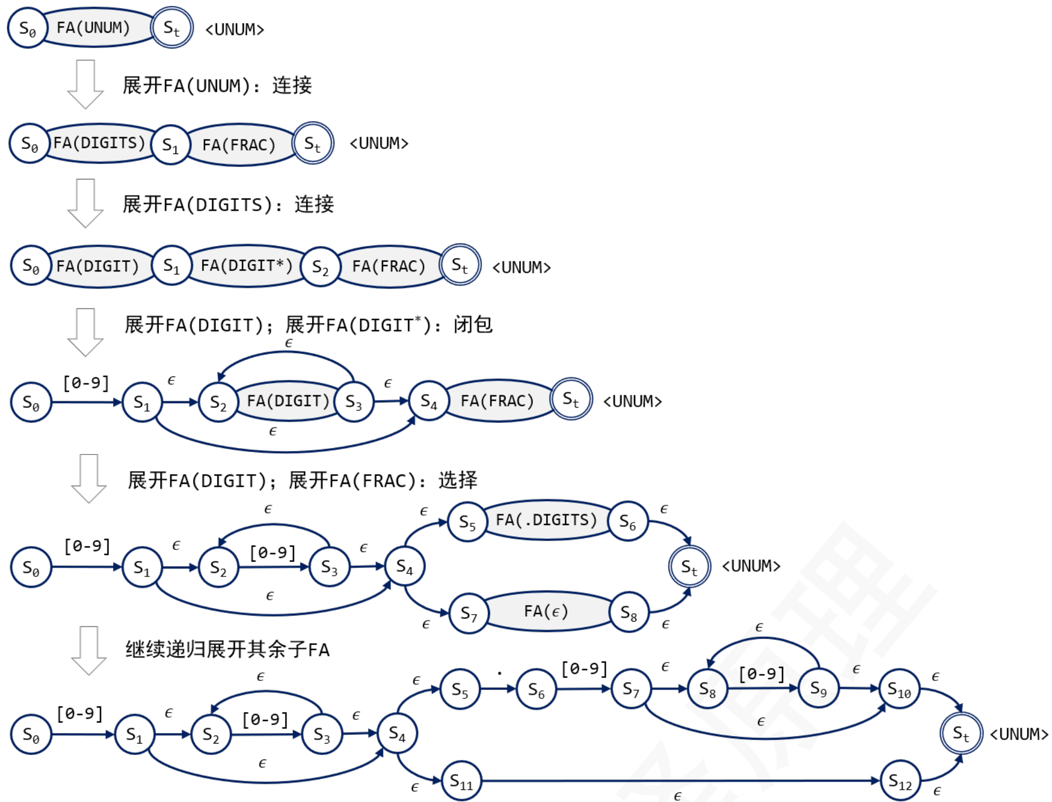



图 2.2: 应用 Thompson 构造法将词元 `<UNUM>` 对应的正则表达式转换为 NFA

图 2.2 以表 2.3 中 `<UNUM>` 词元对应的正则表达式为例，详细展示了该递归构造过程。在此之前，需要将该表达式中的正闭包改写为普通闭包形式：`[0-9][0-9]*(.[0-9][0-9]*|`$\epsilon$`)`。

在构造出每种词元类型对应的 NFA 后，可以进一步通过 $\epsilon$ 转移将它们合并为一个整体 NFA。图 2.3a 展示了表 2.3 中所有词元类型对应的 NFA。

### 2.2.2 将 NFA 转换为 DFA

所有 NFA 都可以转换为 DFA。本节将介绍一种基于子集构造法（powerset）的 DFA 构建方法。在介绍具体方法之前，我们首先定义两个基本概念：$\epsilon$ 闭包（$\epsilon$-closure）和 $\alpha$ 转移（$\alpha$-transition）。

对于 NFA 中的单个状态 $s_i$，其 $\epsilon$ 闭包指通过 $\epsilon$ 转移能够到达的所有状态的集合：

$$Cl^{\epsilon}(s_i) = \{s_j \mid (s_i, \epsilon) \to^* (s_j, \epsilon)\}$$

对于 NFA 中的状态集合 $S$，其 $\epsilon$ 闭包定义为 $S$ 中所有状态的 $\epsilon$ 闭包的并集：

$$Cl^{\epsilon}(S) = \{q' \mid \forall q \in S, (q, \epsilon) \to^* (q', \epsilon)\}$$

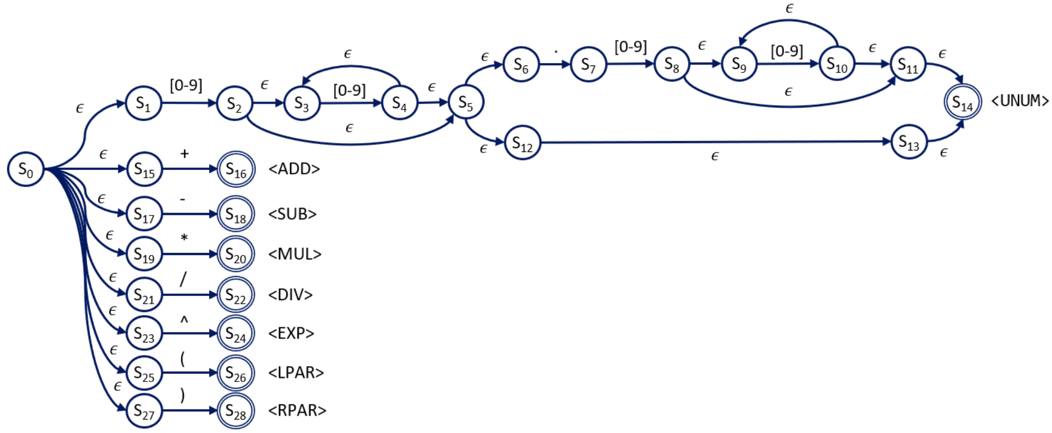
对于状态集合 $S$ 及输入字符 $\alpha$，其 $\alpha$ 转移指：集合 $S$ 在读取字符 $\alpha$ 后，所有可达状态的 $\epsilon$ 闭包的并集：

$$\Delta(S, \alpha) = Cl^{\epsilon}(\{q' \mid \forall q \in S, (q, a) \to q'\})$$

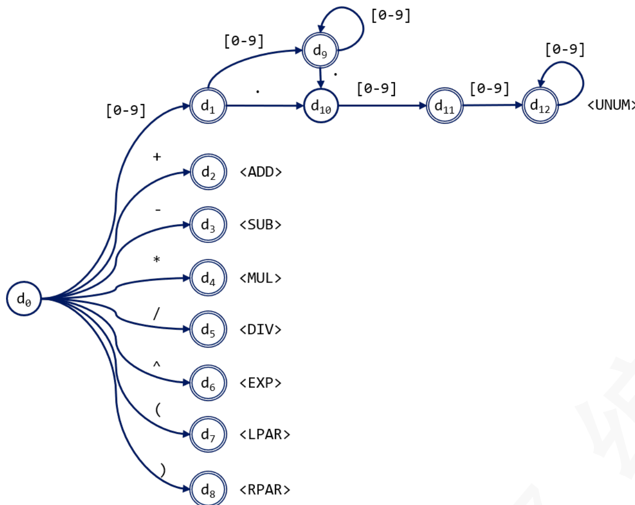
基于上述概念，我们可以定义 NFA 到 DFA 的构造方法

**定义 2** (NFA→DFA)**.** 给定一个 NFA $\{N, n_0, N_f, \Sigma, \Delta\}$，其对应的 DFA 可表示为 $\{D, d_0, D_f, \Sigma, \Theta\}$，其中：

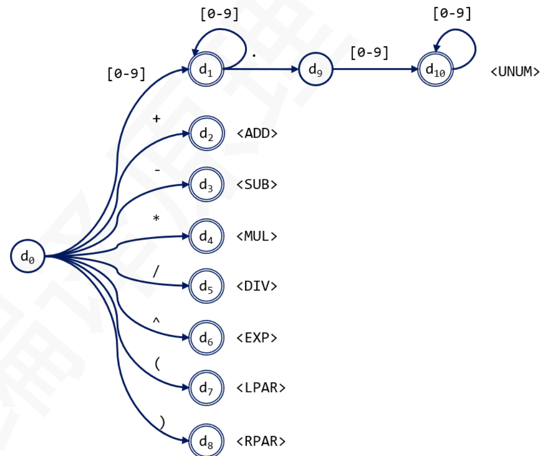
CS3001 编译原理




(a) 合并所有词元后的 NFA




(b) NFA 转换后的 DFA




(c) 优化后的 DFA

图 2.3: NFA 转换 DFA

- $D$ 为 DFA 的状态集合，每个状态 $d_i$ 是若干 NFA 状态的集合；
- $d_0$ 为 DFA 的初始状态，且 $d_0$ 为 $n_0$ 的 $\epsilon$ 闭包，即 $d_0 = Cl^{\epsilon}(n_0)$；
- $D_f$ 为 DFA 的结束状态集合，且每个结束状态 $d_f \in D_f$ 都满足 $d_f \cap N_f \neq \emptyset$；
- $\Theta$ 为 DFA 的状态转移集合，对应 NFA 中状态集合 $S$ 的 $\alpha$ 转移：$\Theta = \{(S, a, \Delta(S, \alpha)), \alpha \in \Sigma\}$。

算法 1 描述了将 NFA 转换为 DFA 的具体步骤。首先，将初始状态 $d_0$ 放入工作列表 $worklist$。然后，对于 $d_0$ 的每个输入字符 $\alpha$，计算其 $\alpha$ 转移，得到一组新的状态集合，并将这些状态加入 $worklist$。接下来，对 $worklist$ 中新加入的 DFA 状态重复上述过程，直到不再产生新的状态为止。

**算法 1** NFA 转换为 DFA

1: **procedure** NFAtoDFA($\{N, n_0, N_f, \Sigma, \Delta\}$)
2: let $d_0 = Cl^{\epsilon}(n_0)$
3: let $D = \{d_0\}$
4: let $worklist = \{d_0\}$
5: **while** $worklist \neq$ NULL **do**
6: $d = worklist$.pop()
7: **for each** $\alpha \in \Sigma$ **do**:
8: $t = \Delta(d, \alpha)$
9: **if** !D.find($t$) **then**:
10: $worklist$.add($t$)
11: $D$.add($t$)
12: **end if**
13: **end for**
14: **end while**
15: **end procedure**

应用算法 1 ，我们可以将图 2.3a 中的 NFA 转换为等价的 DFA。表 2.4 给出了具体的计算过程，最终得到的 DFA 如图 2.3b 所示。从图中可以看出，该 DFA 中仍存在一些冗余的 $\epsilon$ 转移，前后状态之间可以合并，因此该 DFA 可以进一步优化。

表 2.4: 应用子集构造法将图 2.3a 中的 NFA 转换为 DFA

| **DFA 状态** | **NFA 状态集合** | **0-9** | **.** | **+** | **-** | ***** | **/** | **ˆ** | **(** | **)** |
|---|---|---|---|---|---|---|---|---|---|---|
| $d_0$ | $\{s_0, s_1, s_{15}, s_{17}, s_{19}, s_{21}, s_{23}, s_{25}, s_{27}\}$ | $d_1$ | - | $d_2$ | $d_3$ | $d_4$ | $d_5$ | $d_6$ | $d_7$ | $d_8$ |
| $d_1$ | $\{s_2, s_3, s_5, s_6, s_{12}, s_{13}, s_{14}\}$ | $d_9$ | $d_{10}$ | - | - | - | - | - | - | - |
| $d_2$ | $\{s_{16}\}$ | - | - | - | - | - | - | - | - | - |
| $d_3$ | $\{s_{18}\}$ | - | - | - | - | - | - | - | - | - |
| $d_4$ | $\{s_{20}\}$ | - | - | - | - | - | - | - | - | - |
| $d_5$ | $\{s_{22}\}$ | - | - | - | - | - | - | - | - | - |
| $d_6$ | $\{s_{24}\}$ | - | - | - | - | - | - | - | - | - |
| $d_7$ | $\{s_{26}\}$ | - | - | - | - | - | - | - | - | - |
| $d_8$ | $\{s_{28}\}$ | - | - | - | - | - | - | - | - | - |
| $d_9$ | $\{s_3, s_4, s_5, s_6, s_{12}, s_{13}, s_{14}\}$ | $d_9$ | $d_{10}$ | - | - | - | - | - | - | - |
| $d_{10}$ | $\{s_7\}$ | $d_{11}$ | - | - | - | - | - | - | - | - |
| $d_{11}$ | $\{s_8, s_9, s_{11}, s_{14}\}$ | $d_{12}$ | - | - | - | - | - | - | - | - |
| $d_{12}$ | $\{s_9, s_{10}, s_{11}, s_{14}\}$ | $d_{12}$ | - | - | - | - | - | - | - | - |

### 2.2.3 优化 DFA

DFA 优化的核心思想是合并可以合并的状态节点。例如，对于两个状态节点 $d_i$ 和 $d_j$，如果满足以下条件，则可以将它们合并：

$$\forall \alpha \in \Sigma, \Theta(d_i, \alpha) = \Theta(d_j, \alpha)$$

基于这一思想，最经典的方法是 Hopcroft 分割算法 [4]，其核心在于利用状态集合的划分来实现 DFA 的最小化。如算法 2 所示，具体步骤如下：

1) 将 DFA 的状态集合 $D$ 初始化为两个子集：结束状态 $D_f$ 和其他状态 $D \backslash D_f$；

2) 对每个子集进行检查，判断是否需要进一步划分：即是否存在某个输入字符，使得子集内的状态转移指向不同的子集；

3) 重复上述过程，直到无法再进行划分为止，从而得到最优 DFA。

**算法 2** Hopcroft 分割算法（DFA 最小化）

```
1:  procedure OptDFA({D, d_0, D_f, Σ, Θ})
2:      let R = {D_f, D\D_f}                    ▷ 将状态集合划分为结束状态与非结束状态
3:      let S = {}                              ▷ 初始化上一轮划分集合为空
4:      while S ≠ R do
5:          S ← R                               ▷ 保存当前划分作为上一轮划分
6:          R ← {}                              ▷ 准备存放新的划分结果
7:          for each s_i ∈ S do:
8:              R ← R∪ Split(s_i)               ▷ 调用 Split 函数，对子集 s_i 根据输入字符划分状态
9:          end for
10:     end while
11: end procedure
12: procedure Split(S)
13:     for each α ∈ Σ do
14:         if α splits S into {s_1, s_2} then
15:             return {s_1, s_2}               ▷ 如果输入字符 α 能将 S 划分为两个子集，返回划分结果
16:         end if
17:     end for
18:     return {S}                              ▷ 如果没有划分，返回原集合
19: end procedure
```

图 2.3c 展示了对图 2.3b 优化后的最终 DFA。除了上述“先构造 NFA，再转换为 DFA 并优化”的方法外，还有一种直接构造最优 DFA 的方法，即 Brzozowski 算法 [2]。有兴趣的同学可以查阅相关资料，进一步了解其原理和实现。

本章介绍的正则表达式解析技术已经相当成熟，并被广泛应用于实际工程中。许多强大且易用的工具，如传统的 C 语言工具 Flex[1]、Python 的 `re` 库[2]，以及 Rust 的 pest[3] 和 lalrpop[4]，可以直接用于处理各种词法分析任务，从而大幅提升相关软件原型的开发效率。

# 参考文献


[1] Robert McNaughton, and Hisao Yamada. “Regular expressions and state graphs for automata.” *IRE Transactions on Electronic Computers*, 1960.

[2] Janusz A. Brzozowski, “Canonical regular expressions and minimal state graphs for definite events.” *In Symposium of Mathematical Theory of Automata*, 1962.

[3] Ken Thompson. “Programming techniques: Regular expression search algorithm.” *Communications of the ACM*, 1968.

[4] John E. Hopcroft, “An nlogn algorithm for minimizing the states in a finite automaton.” *The Theory of Machines and Computation*, 1970.


[1] Flex：`https://github.com/westes/flex`

[2] `re`：`https://docs.python.org/3/library/re.html`

[3] pest：`https://github.com/pest-parser/pest`

[4] lalrpop：`https://github.com/lalrpop/lalrpop`

# 练习

1) 假设某编程语言在词法分析环节需要支持日期识别，日期的格式为 `YYYY-MM-DD`，例如 `1980-01-12` 或 `0001-01-02`。回答以下两个问题：

   a) 写出正则表达式，要求合法日期格式中月份 MM 必须是 01-12，日期 DD 必须是 01-31。

   b) 改进正则表达式，使 DD 满足下列条件：

      - 如果月份为 01, 03, 05, 07, 08, 10, 12，则日期范围为 01-31；
      - 如果月份为 04, 06, 09, 11，则日期范围为 01-30；
      - 如果月份为 02，则日期范围为 01-28。

2) 选择一门你熟悉的语言（如 Markdown、LaTeX、HTML、Python 或 Golang），并按以下步骤进行词法分析：

   a) 找出该语言中的词法词元；

   b) 使用正则表达式设计词法规则；

   c) 将正则表达式转换为 NFA；

   d) 将 NFA 转换为 DFA 并进行优化。

3) 在 RESTful API 中，API 路径由字符串常量和变量拼接而成，其中变量以 : 开头。例如，以下三个 API 中的 `:id`、`:branch` 和 `:sha` 都是变量：

```
API-1: GET /projects/:id/repository/branches
API-2: GET /projects/:id/repository/branches/:branch
API-3: GET /projects/:id/repository/commits/:sha
```

   调用 API 时，变量会被替换为具体的参数值，例如下面的调用日志对应 API-1 和 API-3：

```
2021-07-04 16:43:47.193: Sending:
 'GET /projects/MyProject/repository/branches?'
2021-07-04 16:43:49.761: Sending:
 'GET /projects/MyProject/repository/commits/ed899a2f?'
```

   问：给定多个 API 定义和一组访问日志，如何识别每条日志属于哪个 API？该问题是否可以用正则表达式解决？