2 词法分析

徐辉，xuh@fudan.edu.cn

本章学习目标包括：

- 掌握正则表达式
- 掌握 Thompson 构造法
- 掌握子集构造法

2.1 词法声明：正则表达式

正则表达式（Regular Expression，简写为 Regex）定义了字母表 Σ 上的字符串集合，由基本的字符元素和构造方式组成。

2.1.1 字符表示

表 2.1定义了单个字符元素的主要表述形式及其含义。

表 2.1: 单个字符元素表示方法。

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	ϵ	$x \in \emptyset$
特定字符或空	$a?$	$x = a$ or $x = \epsilon$

2.1.2 构造方式

单个字符之间组合方式包括选择、连接、和闭包三种基本形式。为了不失一般性，一般可以采用表 2.2中的构造方式递归定义正则表达式。

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	示例	含义
选择	|	$S \| T$	$x \in \{S \cup T\}$
连接		ST	$x \in \{st \mid s \in S, t \in T\}$
闭包	$*$	S^*	$x \in \{s_0..s_n \mid \forall 0 \le i \le n, s_i \in S, 0 \le n < \infty\}$
正闭包（扩展表示）	$+$	S^+	$x \in \{s_0..s_n \mid \forall 0 \le i \le n, s_i \in S, 1 \le n < \infty\}$
区间闭包（扩展表示）		$S^{\{min,max\}}$	$x \in \{s_0..s_n \mid \forall 0 \le i \le n, s_i \in S, min \le n \le max\}$

在解析采用上述构造方式定义的正则表达式时应遵循一定的优先级顺序，即闭包 > 连接 > 选择。以正则表达式 a|bc* 的解析为例，其对应的字符串集合（正则集）为 {a,bc,bcc,...}；字符串 abc 不在该集合中。另外，在定义正则表达式时可以使用括号，如 (a|b)c* 对应的正则集是 {ac,bc,acc,bcc,...}。

下面我们使用正则表达式设计计算器的词法规则。首先，计算器的输入符号是一个有限集：

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

表 2.3定义了计算器中的词法。

表 2.3: 计算器词法定义。

标签类型	**含义**	**定义**
<UNUM>	无符号数字	[0-9]*(.[0-9]*$\mid\epsilon$)
<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	)

值得注意的是，一组正则表达式定义可能出现二义性问题，即多种标签类型的规则存在交集的情况，或一个字符串同时符合两种标签类型规则。另外，使用正则表达式扫描字符串，如果已经匹配到特定标签，且可继续向前移进时，一般应采用最长匹配原则。如字符串 “123” 应被识别为一个标签 <UNUM(123)>，而非三个标签 <UNUM(1)><UNUM(2)><UNUM(3)>。

2.2 词法解析：有穷自动机及其构造

本节解决一个重要问题：给定任意一组由正则表达式定义的标签规则，如何自动生成该规则对应的标签识别程序。理论上，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；DFA 可进一步转化为等价的程序。

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

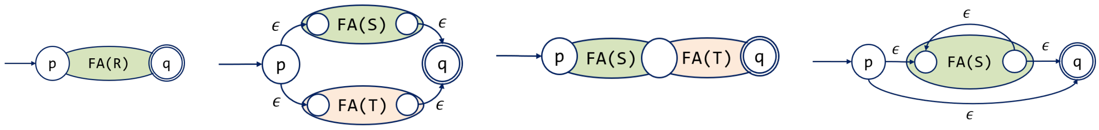
- S 表示有限状态集合；
- $s_0 \in S$ 是初始状态；
- $T \subseteq S$ 是结束状态集合；
- Σ 表示有限字符集合；
- $\Delta \subseteq S \times \Sigma \times S$ 为边的集合，表示输入特定字符后有穷自动机的状态转移关系；如果对于所有状态 $\forall s_i \in S$ 和所有字符，至多存在一个转移目标状态，则该有穷自动机为确定性有穷自动机，否则是非确定性有穷自动机（Non-Deterministic Finite Automaton）。

将一组正则表达式转化为 DFA 的过程可大致分为三步：

1) 为 Regex 构造 NFA
2) 将 NFA 转化为 DFA
3) 优化 DFA

2.2.1 为 Regex 构造 NFA

本节介绍一种经典的 NFA 构造算法：Thompson 构造法 (McNaughton–Yamada–Thompson [1, 2])。其基本思想是为正则表达式有限的三种构造方式分别设计相应的 NFA 构造方法，从初始 NFA 开始根据运算次序（逆序）对正则表达式递归展开，并根据当前正则表达式的构造方式选取相应的 NFA 构造方法。



(a) 初始 FA　(b) 选择展开 $S|T$　(c) 连接展开 ST　(d) 闭包展开 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 开始，该初始 NFA 的边为目标正则表达式 R。如果 R 的最后一层构造方式为选择，则按照图 2.1b的方式将边 S 展开；如果是连接，则根据图 2.1c的方式展开；如果是闭包，则采用图 2.1d的方式展开。图 2.2以表 2.3中`<UNUM>`为例详细阐述了该递归构造过程。

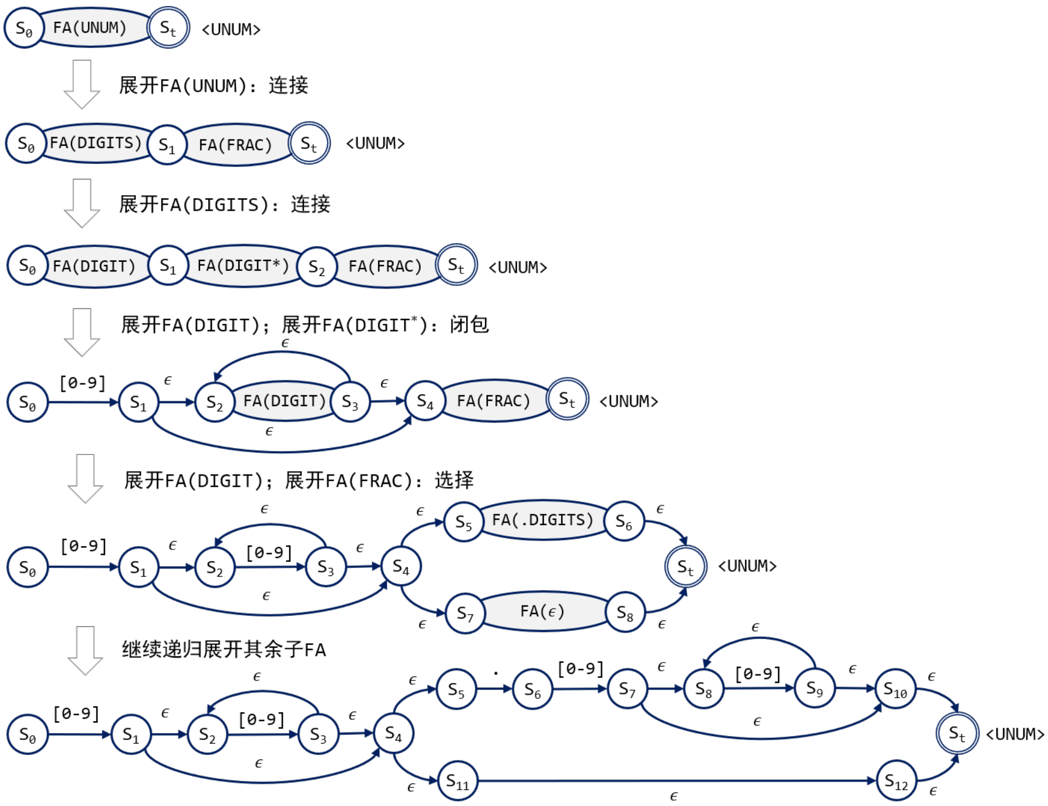
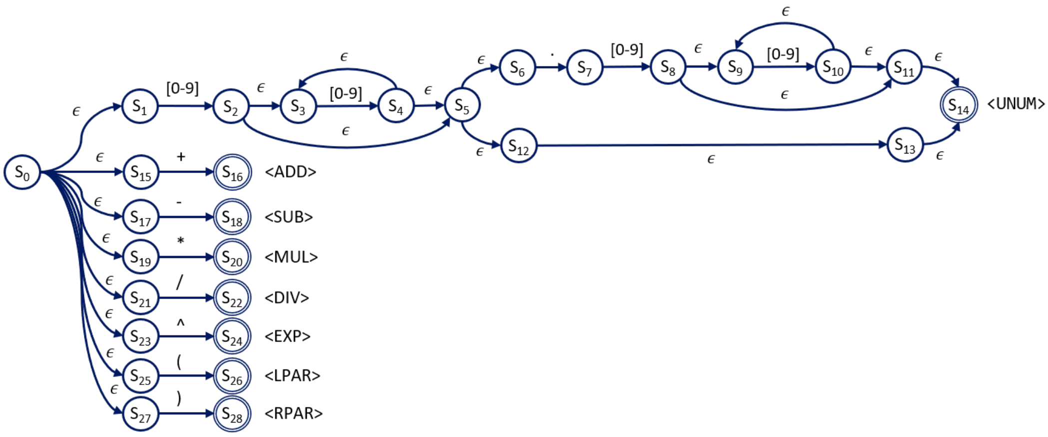


图 2.2: 应用 Thompson 构造法将 `<UNUM>` 的正则表达式转化为 NFA。其中，`DIGIT` 为 `[0-9]`，`DIGITS` 为 `DIGIT`*，`FRAC` 为`.DIGITS|`ϵ。

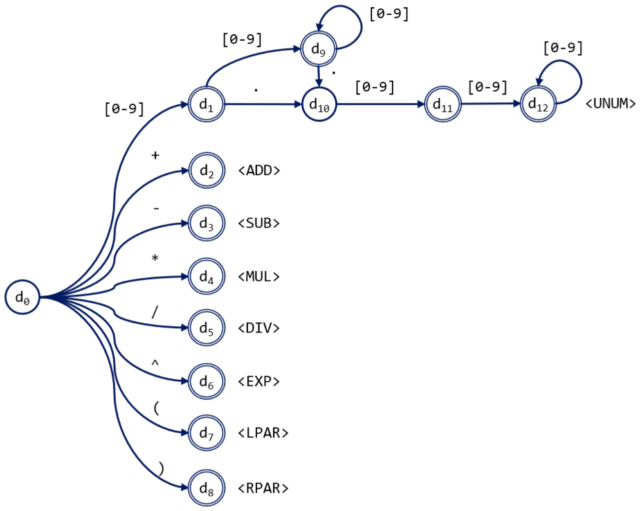
有了每种标签类型对应的 NFA，我们可以将所有标签类型对应的 NFA 通过 ϵ 转移合并为一个大的 NFA。图 2.3a展示了表 2.3中的所有标签类型对应的 NFA。

2.2.2 将 NFA 转化为 DFA

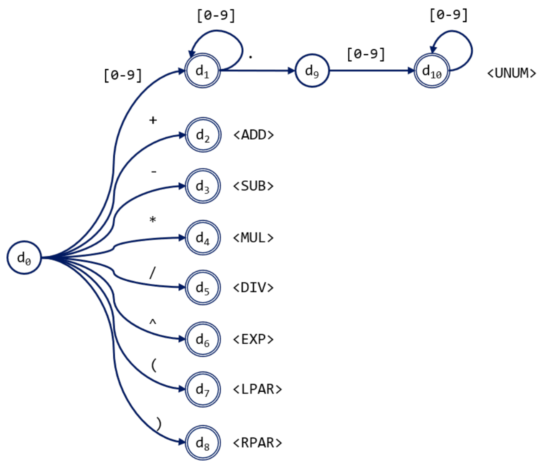
所有的 NFA 都可以转化为 DFA。本节介绍一种基于子集构造法（powerset）的 DFA 构建思路。在介绍该方法之前，我们先定义两个基本概念 ϵ 闭包（closure）和 α 转移（transition）。



(a) 合并所有标签 NFA 后的 NFA



(b) NFA 转化后的 DFA



(c) 优化后的 DFA

图 2.3: NFA 转化 DFA

对于 NFA 上的单个状态 s_i 来说，其 ϵ 闭包指的是 s_i 的 ϵ 转移的状态集合。

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

对于 NFA 上的状态集合 S 来说，其 ϵ 闭包指的是 S 中的所有状态的 ϵ 闭包的集合。

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

对于 NFA 上的状态集合 S 来说，其 α 转移指的是 S 读取字符 α 后所有状态的 ϵ 闭包的集合。

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

基于上述工具，我们可以定义 NFA 等价 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：

- D 表示 DFA 中的状态集合，其中的每一个状态 d_i 是 NFA 中若干状态的集合；
- d_0 为 DFA 的初始状态，且 d_0 是 n_0 的 ϵ 闭包，即 $d_0 = Cl^{\epsilon}(n_0)$；
- D_f 为 DFA 的结束状态集合，且对于每一个结束状态 $d_f \in D_f$ 来说，$d_f \cap N_f \neq \emptyset$；

- Θ 为边的集合，对应原 NFA 上状态集合 S 的 α 转移 $\Theta = \{(S, a, \Delta(S, \alpha)), \alpha \in \Sigma\}$。

算法 1描述了如何将 NFA 转化为 DFA 的具体思路。另初始 $worklist$ 为 d_0，分析 d_0 对于每个字符的 α 转移，得到一组新状态集合并加入到 $worklist$。对 $worklist$ 中新加入的 DFA 状态重复上述过程，迭代到不再有新的状态产生为止。

算法 1 NFA 转化为 DFA

```
procedure NFATODFA({N, n0, Nf, Σ, Δ})
    let d0 = Cl^ϵ(n0)
    let D = {d0}
    let worklist = {d0}
    while worklist ≠ NULL do
        d = worklist.pop()
        for each α ∈ Σ do:
            t = Δ(d, α)
            if !D.find(t) then:
                worklist.add(t)
                D.add(t)
            end if
        end for
    end while
end procedure
```

应用算法 1便可将图 2.3a中的 NFA 转化为等价的 DFA。表 2.4阐述了具体的计算过程，最终得到的 DFA 如图 2.3b所示。可以看出该 DFA 有一些冗余的 ϵ 转移，这些 ϵ 转移前后的状态是可以合并的，可以进一步对其进行优化。

表 2.4: 应用子集构造法将图 2.3a中的 NFA 转化为 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 优化的核心思路是找出可以合并的 DFA 节点。对于两个同类型（初始状态、中间状态、或标签类型相同的结束状态）节点 d_i 和 d_j，如果满足下列条件则可以合并：

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

下面介绍一种基于上述思路的算法实现，但采用了基于分割的实现方式，即 Hopcroft 分割算法 [3]。如算法 2所示，该方法将 DFA 的状态集合 D 初始化为两个子集：结束状态 D_f 和其它状态 $D\backslash D_f$，之后依次检查每个子集是否需要被继续分割；重复上述过程直至不能再分割为止，即找到了最优的 DFA。图 2.3c展示了对图 2.3b优化后最终得到的 DFA。

算法 2 Hopcroft 分割算法

procedure OptDFA($\{D, d_0, D_f, \Sigma, \Theta\}$)
    let $R = \{D_f,\ D\backslash D_f\}$
    let $S = \{\}$
    while $S \neq$ R **do**
        $S = R$
        $R = \{\}$
        for each $s_i \in S$ **do**:
            $R = R \cup$ Split(s_i)
        end for
    end while
end procedure
procedure Split(S)
    for each $\alpha \in \Sigma$ **do**:
        if α splits S into $\{s_1, s_2\}$ **then**:
            Return $\{s_1, s_2\}$
        end if
    end for
end procedure

除了上述方法以外，还有一种直接构造最优 DFA 的方法，称为 Brzozowski 算法 [4]，有兴趣的同学可以自己查找资料进行了解。

练习

1. RESTful API 由字符串常量和变量拼接而成。其中，变量以 “:” 开头，如下面的三个 API 中的`:id`、`:branch`和`:sha`均为变量。

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

变量在实际 API 调用时会被替换为实际的值，如下列三条 API 调用日志分别对应 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?'
```

问题：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?

2. 选择一门你熟悉的语言（如 Markdown、Latex、HTML、C/C++、Java）进行词法分析:
1）用正则表达式设计该语言的词法规则；
2）将正则表达式转化为 NFA；
3）将 NFA 转化为 DFA 并优化。

Bibliography

[1] Robert McNaughton, and Hisao Yamada. "Regular expressions and state graphs for automata." IRE Transactions on Electronic Computers 1 (1960): 39-47.

[2] Ken Thompson. "Programming techniques: Regular expression search algorithm." Communications of the ACM 11, no. 6 (1968): 419-422.

[3] John E. Hopcroft, "An nlogn algorithm for minimizing the states in a finite automaton." The Theory of Machines and Computation (1970): 189496.

[4] Janusz A. Brzozowski, "Canonical regular expressions and minimal state graphs for definite events." In Proc. Symposium of Mathematical Theory of Automata, pp. 529-561. 1962.