

2 词法分析

徐辉, xuh@fudan.edu.cn

本章学习目标:

- 掌握正则表达式。
- 掌握 $\text{regex} \Rightarrow \text{NFA}$ 转化算法。
- 掌握 $\text{NFA} \Rightarrow \text{DFA}$ 转化算法。

2.1 正则表达式

正则表达式定义了字母表 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

单个字符元素可以采用如下 (表 2.1) 的表述形式。

表 2.1: 单个字符元素表示方法。

字符元素	示例	含义
特定字符	a	$x = a$
排除特定字符	\hat{a}	$x \in \Sigma / \{a\}$
字符范围	$[ab]$	$x \in \{a, b\}$
字符范围	$[a - z]$	$x \in \{a, \dots, z\}$
字符范围	$[a - zA - Z]$	$x \in \{a, \dots, z, A, \dots, Z\}$
通配符	$.$	$x \in \Sigma$
空字符	ϵ	$x \in \emptyset$
特定字符或空	$a?$	$x = a \text{ or } x \in \emptyset$

2.1.2 构造方式

正则表达式的构造方式可以采用如下 (表 2.2) 递归形式定义。

表 2.2: 正则表达式的字符模式构造方法; 其中 S 和 T 为子正则表达式。

构造方式	符号	示例	含义	说明
选择	$ $	$S T$	匹配 S 或 T	
连接		ST	连续匹配 S 和 T	
闭包	$*$	S^*	匹配若干个 (≥ 0) 连续 S	
* 正闭包	$+$	S^+	匹配若干个 (≥ 1) 连续 S	非必要

其中优先级顺序: 闭包 $>$ 连接 $>$ 选择, 如 $a|bc^*$ 对应的正则集是 a, bc, bcc, \dots 。正则表达式可以使用括号, 如 $(a|b)c^*$ 对应的正则集是 ac, bc, acc, bcc, \dots 。

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：

$$\Sigma = \{0, \dots, 9, (,), +, -, *, /, \}$$

表 2.3 定义了计算器中的词法。

表 2.3: 计算器程序词法定义。

标签	含义	定义	说明
<DIGIT>	临时变量	$[0 - 9]$	非必要，为了 <UNUM> 方式一
<DIGITS>	临时变量	$\langle \text{DIGIT} \rangle \langle \text{DIGIT} \rangle^*$	非必要，为了 <UNUM> 方式一
<FRAC>	临时变量	$\langle \text{DIGITS} \rangle \epsilon$	非必要，为了 <UNUM> 方式一
<UNUM>	无符号数字	$\langle \text{DIGITS} \rangle \langle \text{FRAC} \rangle$ $[0 - 9]^+ ([0 - 9]^+ \epsilon)$	方式一 方式二
<ADD>	加号	+	
<SUB>	减号	-	
<MUL>	乘号	*	
<DIV>	除号	/	
<EXP>	指数运算	^	
<LPAR>	左括号	(
<RPAR>	右括号)	

在使用正则表达式进行匹配时，如同时满足多条规则或模式，一般采用最长匹配原则。

2.2 正则表达式 \Rightarrow DFA

本节解决一个重要问题：给定一组正则表达式，如何自动生成正则表达式匹配程序。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). 有穷自动机是一个五元组： $(Q, q_0, F, \Sigma, \Delta)$ ：

- Q 表示有限状态集合；
- $q_0 \in Q$ 是初始状态；
- $F \subseteq Q$ 是结束状态集合；
- Σ 表示有限字符集合；
- $\Delta \subseteq Q \times \Sigma \times Q$ 为边的集合，表示输入特定字符后 FA 的状态转移；如果对于所有状态 $\forall q_i \in Q$ 和所有字符，至多存在一个转移目标状态，则该 FA 为确定性有穷自动机 DFA，否则是非确定性有穷自动机 NFA。

正则表达式转化 DFA 的过程可分为三步：

- 1) 正则表达式 \Rightarrow NFA
- 2) NFA \Rightarrow DFA
- 3) DFA 优化

2.2.1 正则表达式 \Rightarrow NFA

Thompson 构造法 (McNaughton–Yamada–Thompson) [1, 2] 是一种经典 NFA 转化算法。其基本思想是由于所有正则表达式都可以递归展开为子正则表达式，如果可以为所有的组合模式设计相应的 NFA 构造方式，则可从初始 NFA 开始递归构造该正则表达式的 NFA。

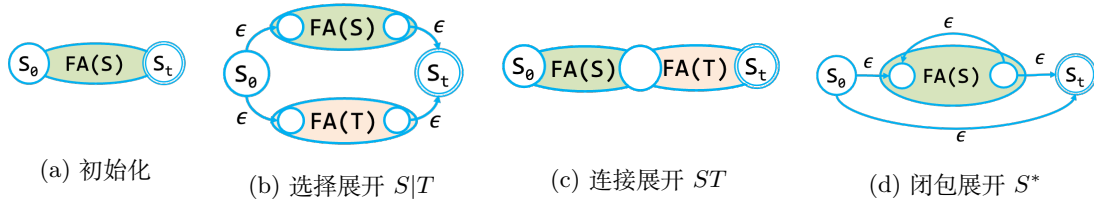


图 2.1: Thompson 构造法

图 2.1展示了表 2.2中的正则表达式构造方法对应的 NFA 构造方法。从图 2.1a开始，如果遇到选择操作，则采用图 2.1b的方式展开；如果遇到连接操作，则采用图 2.1c的方式展开；如果遇到闭包操作，则采用图 2.1d的方式展开。

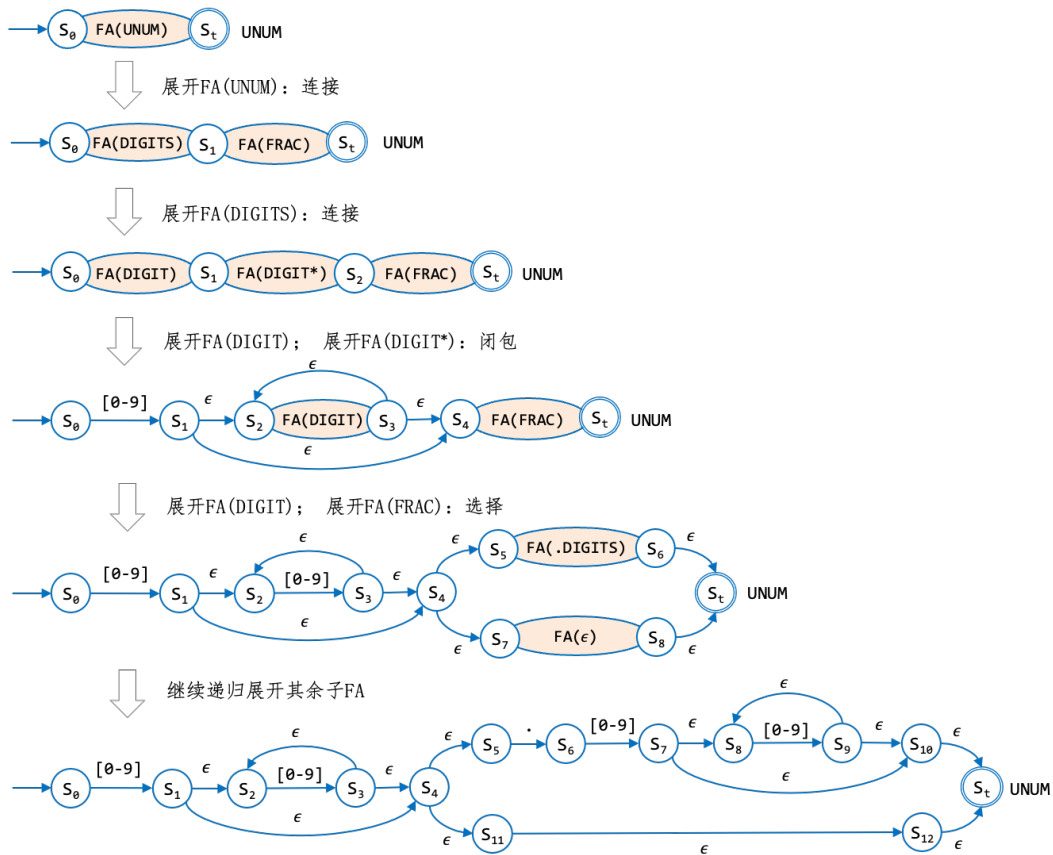
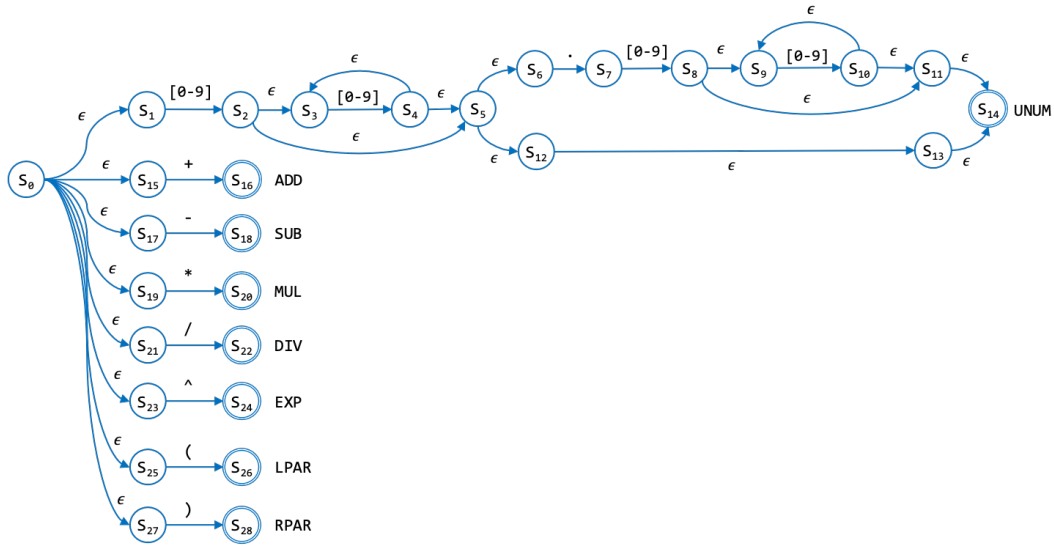


图 2.2: 应用 Thompson 构造法将 $\langle \text{UNUM} \rangle$ 的正则表达式表示为 NFA

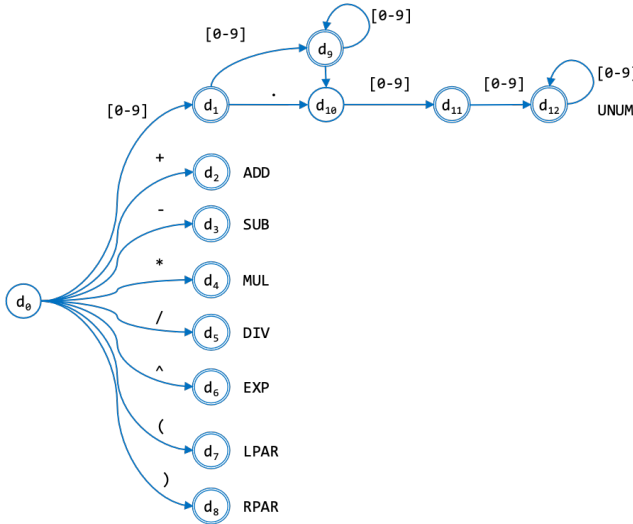
图 2.2展示了如何递归构造词法表 2.3中 $\langle \text{UNUM} \rangle$ 标签对应的 NFA。有了每个标签的 NFA，接下来我们可以将所有表 2.3中词法标签对应的 NFA 通过 ϵ 合并为一个大的 NFA (图 2.3a)。

2.2.2 NFA \Rightarrow DFA

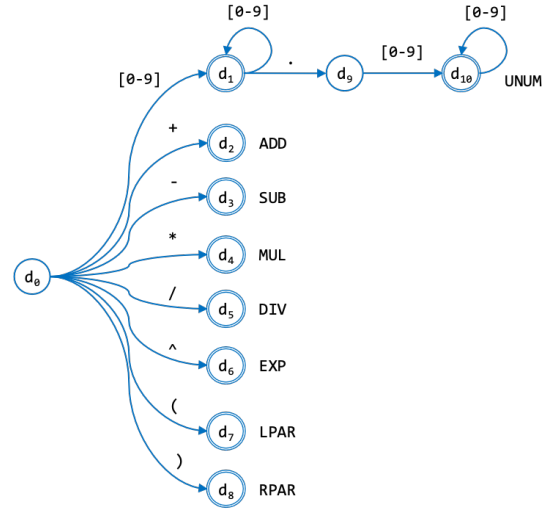
所有的 NFA 都可以转化为 DFA。本节介绍一种基于子集构造法 (powerset) 的 DFA 构建思路。在介绍该方法之前，我们先定义两个基本概念 ϵ 闭包 (closure) 和 α 转移 (transition)。



(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。

$$Cl^\epsilon(s_i) = \cup\{s_j : (s_i, \epsilon) \rightarrow^* (s_j, \epsilon)\}$$

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。

$$Cl^\epsilon(S) = \cup_{q \in S} \{q : (q, \epsilon) \rightarrow^* (q', \epsilon)\}$$

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。

$$\Delta(S, \alpha) = Cl^\epsilon(\cup_{q \in S} \{q' : (q, \alpha) \rightarrow q'\})$$

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 $d_0 := Cl^\epsilon(n_0)$;
- D_f 为 DFA 的结束状态, 且 $D_f := \{S \subseteq N, S \cap N_f \neq \emptyset\}$;
- 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 描述了将 NFA 转化为 DFA 的详细过程, 通过迭代方式从 d_0 开始, 分析其对于每个字符 $\alpha \in \Sigma$ 的 α 转移, 将得到的新状态加入 *worklist*, 直到不在有新的状态生成为止。表 2.4 描述了使用算法 1 将图 2.3a 中的 NFA 转化为等价 DFA 的过程。

算法 1 NFA 转化为 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  $worklist \neq \text{NULL}$  do
6:      $worklist.\text{pickremove}(d)$ 
7:     for each  $\alpha \in \Sigma$  do:
8:        $t = \Delta(d, \alpha)$ 
9:       if !D.find( $t$ ) then:
10:         $worklist.\text{add}(t)$ 
11:         $D.\text{add}(t)$ 
12:       end if
13:     end for
14:   end while
15: end procedure

```

表 2.4: 图 2.3a 中的 NFA 转化为等价 DFA 的过程分解

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

转化后的 DFA 如图 2.3b 所示。可以看出该 DFA 有很多冗余的 ϵ 转移，这些 ϵ 转移前后的状态是可以合并的，我们可以对其进行优化。

2.2.3 DFA 优化

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：

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

Hopcroft 分割算法 [3] 是基于上述思想的一种算法，但采用了基于分割的方法。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：接受状态 D_f 和普通状态 $D \setminus D_f$ ，然后依次检查每一个子集是否需要继续分割；之后在每一个新产生的子集上重复上述过程直至不能再分割即找到了最优的 NFA。图 2.3c 展示了优化后的 DFA。

算法 2 Hopcroft 分割算法

```

1: procedure HOPCROFTOPT( $\{D, d_0, D_f, \Sigma, \Theta\}$ )
2:   let  $R = \{D_f, D \setminus D_f\}$ 
3:   let  $S = \{\}$ 
4:   while  $S \neq R$  do
5:      $S = R$ 
6:      $R = \{\}$ 
7:     for each  $s_i \in S$  do:
8:        $R = R \cup \text{Split}(s_i)$ 
9:     end for
10:  end while
11: end procedure
12: procedure SPLIT( $S$ )
13:   for each  $\alpha \in \Sigma$  do:
14:     if  $\alpha$  splits  $S$  into  $\{s_1, s_2\}$  then:
15:       Return  $\{s_1, s_2\}$ 
16:     end if
17:   end for
18: end procedure

```

除了上述方法以外，还有一种直接构造 DFA 的方法，称为 Brzozowski 算法 [4]，有兴趣的同学可以自己学习。

2.3 练习

选择一门你熟悉的语言（如 C/C++、Java、Python）进行词法分析：1) 用正则表达式设计其词法规则；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 $n \log n$ algorithm for minimizing the states in a finite automaton." The Theory of Machines and Computation (1970): 189-496.
- [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.