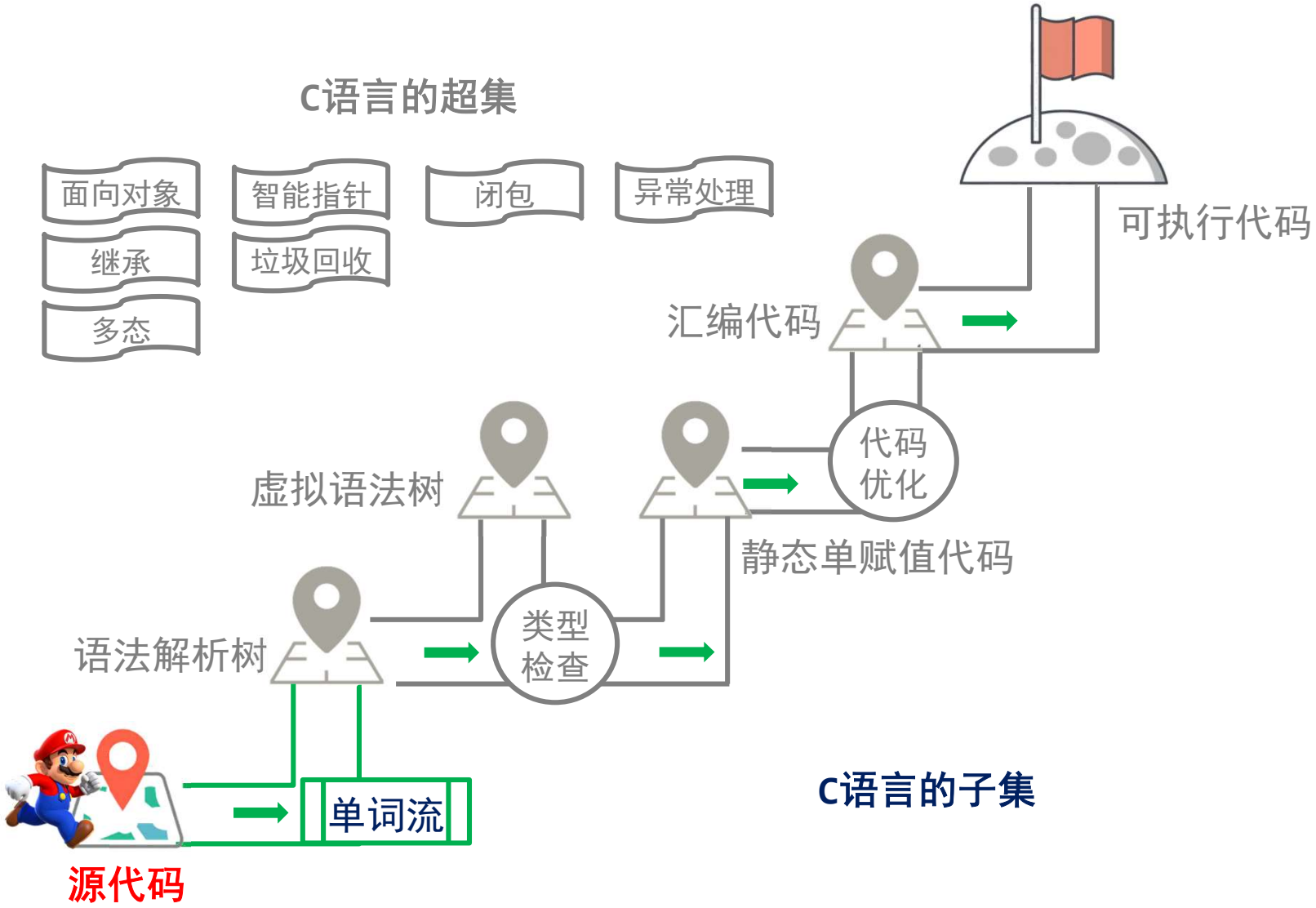
Lecture 2

词法分析

徐 辉

xuh@fudan.edu.cn

学习地图

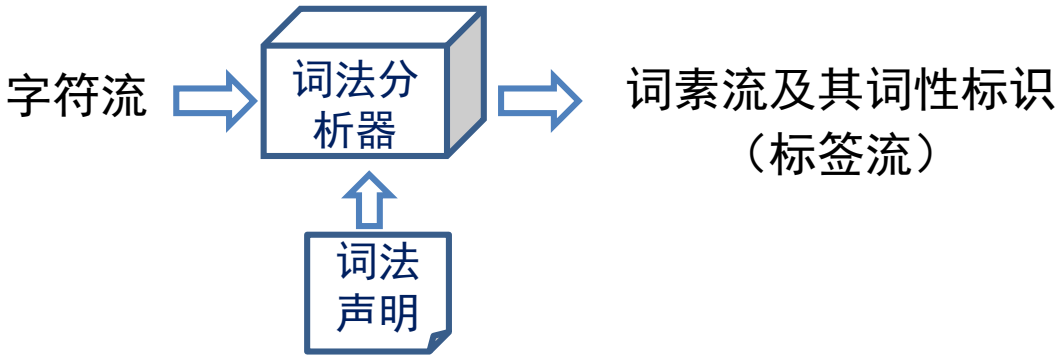

C语言的超集
面向对象
智能指针
闭包
异常处理
继承
垃圾回收
多态
可执行代码
汇编代码
代码
优化
虚拟语法树
静态单赋值代码
类型
检查
语法解析树
单词流
源代码
C语言的子集

大纲

- 一、基本概念
- 二、词法声明方法
  - 正则表达式
- 三、词法分析方法
  - 有穷自动机构造
  - 有穷自动机优化
- 四、词法分析工具
- 五、应用扩展

问题定义

- 词法声明（Lexer specification）定义了：
  - 什么对词法分析器（Lexer）有效的输入（valid inputs），
  - 及其关联标签类型（token types）。



基本概念

- Token（标签）：由标签类型和属性组成的二元组；
- Pattern（模式）：关于token组成的可能模式的描述；
- Lexeme（词素）：符合某token模式的字符串实例。

标签	模式	词素举例
ID	字母开头，字母和数字组成	pi
CMP	<，>，<=，>=，==，!=	<=
BINOP	+,-,*,/	+
NUM	任意数据常量	3.1415926
LITERAL	“引号除外的任意字符串”	“compiler”
IF	字符i和f组成	if
ELSE	字符e、l、s和e组成	else
COMMA	,	,
SEMICOLON	;	;
LPAREN	(	(
RPAREN	)	)

词素示例

- 根据词法描述，分析下列代码包含哪些词素及其标签类型？

```
printf(“total = %d\n”, score);
```

词素：	标签类型：
printf	<ID>
(	<LPAREN>
“total = %d\n”	<LITERAL>
,	<COMMA>
score	<ID>
)	<RPAREN>
;	<SEMICOLON>

标签属性

- 属性一般与编译器具体实现有关
- 通过属性关联到标签具体的内容

```
printf(“total = %d\n”, score);
```

词素:	标签+属性:
printf	<ID, 指向字符表中printf的指针>
(	<LPAREN>
“total = %d\n”	<LITERAL, 字符串内容>
,	<COMMA>
score	<ID, 指向字符表中score的指针>
)	<RPAREN>
;	<SEMICOLON>

Symbol	Type	Scope
printf	function, (const char* int,...) -> int	global
score	int	local

通常定义哪些标签类型？

- 保留字：
  - 控制流：if-else、while、for、switch-case
  - 类型：int、float、struct
  - 其它：typedef
- 运算符/操作符：
  - 二元运算符（Binary Operator）：+、-、*、/
  - 一元运算符（Unary Operator）：++、--
  - 比较运算符（Comparison Op.）：>、<、>=、<=、==、!=
  - 逻辑运算符（Logical Operator）：&&、||
  - 位运算符（Bitwise Operator）：~、&、|、<<、>>
  - 赋值操作符：=
- 标识符：变量名
- 常量：
  - 数字常量
  - 字符串常量
- 分隔符：括号、大括号、逗号、分号、问号

练习

- 下列代码包含哪些标签？

```
float square(x){
   float x;
   return (x <= -10.0 || x >= 10.0) ? 100 : x*x;
}
```

```
<FLOAT> <ID, square> <LPAREN> <ID, x> <RPAREN > <LBRACE>
<FLOAT> <ID, x>
<RETURN> <LPAREN> <ID, x> <CMP, less or equal> <NUM, -10.0>
         <LOGOP, or> <ID, x> <CMP, grater or eq> <NUM, 10.0> <RPAREN>
         <QMAKR> <NUM, 100> <COLON> <ID, x> <BINOP, times> <ID, x>
<RBRACE >
```

如何编写词法分析程序？

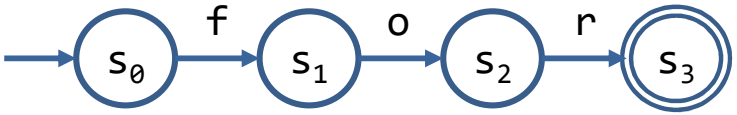
- 识别标签FOR的代码？

```
ch = nextchar();
if (ch == ‘f’) {
   ch = nextchar();
   if (ch == ‘o’) {
      ch = nextchar();
      if (ch == ‘r’) {
         printf(“success”);
      }
      else printf(“fail”);
   }
   else printf(“fail”);
}
else printf(“fail”);
```

- 识别标签WHILE的代码？
- 识别标签FOR和WHILE的代码？
- 识别复杂类型（如标识符）的代码？

基于表查找的词法分析方法

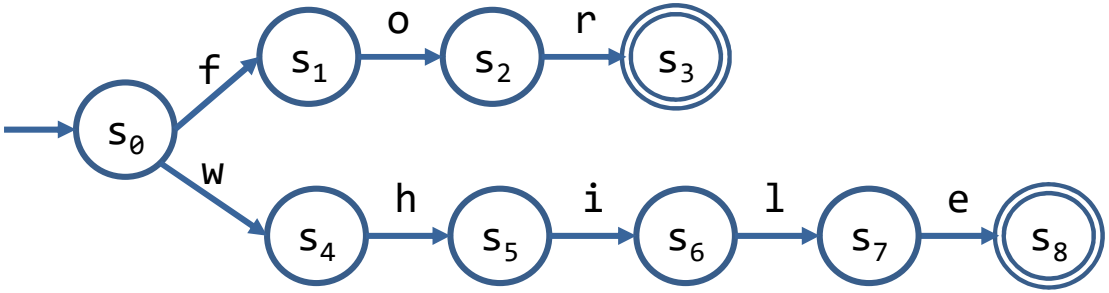
- 使用有穷自动机表示标签识别程序
- 使用表格记录有穷自动机的状态转移关系



识别标签FOR的有穷自动机

δ	f	o	r	other
s_0	s_1	s_{rej}	s_{rej}	s_{rej}
s_1	s_{rej}	s_2	s_{rej}	s_{rej}
s_2	s_{rej}	s_{rej}	s_3	s_{rej}
s_3	s_{rej}	s_{rej}	s_{rej}	s_{rej}

识别标签FOR的状态转移表



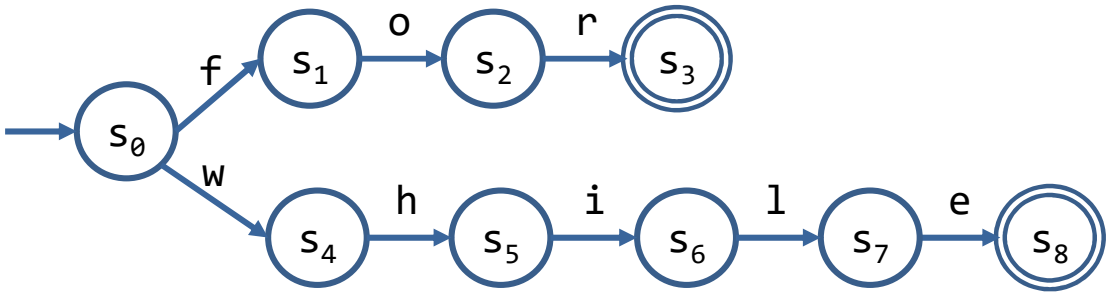
识别标签FOR和WHILE的有穷自动机

你可以自己写出识别标签FOR和WHILE的状态转移表吗？

有穷（状态）自动机：Finite Automaton

- 假设∑表示特定的有穷字符集合（如字母$a-z$、数字$0-9$、括号、运算符），则字符集∑上的有穷自动机由以下几部分组成：
  - 若干（有限）个状态 S；
  - 连接状态的边$\Delta \subseteq S \times \Sigma \times S$；
    - 每条边使用Σ中的字符标记，表示状态之间的转移条件；
  - 一个初始状态$s_0 \in S$；
  - 若干个接受状态$S_{acc} \subseteq S$。

FA示例



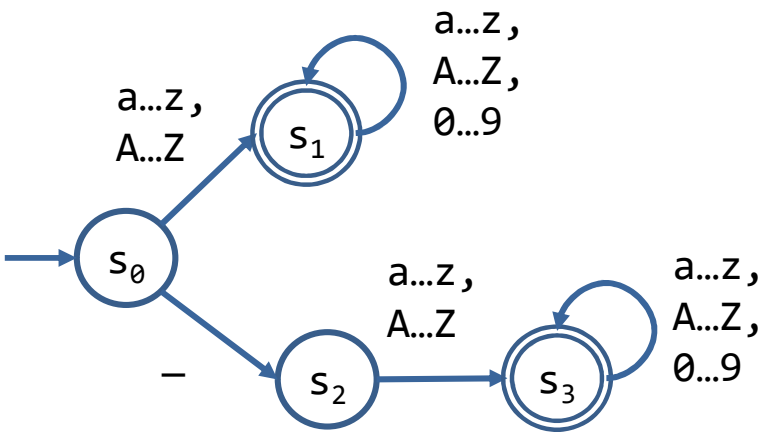
- 识别标签FOR和WHILE的FA：
  - 字符集：$\sum = \{e, f, h, i, l, o, r, w\}$ 或 $\sum = \{a, \ldots, z\}$
  - 状态集：$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$
  - 初始状态：$s_0 = \{s_0\}$
  - 接受状态：$S_{acc} = \{s_3, s_8\}$
  - 状态转移关系：$\Delta = \left\{ \begin{matrix} s_0 \xrightarrow{f} s_1, s_1 \xrightarrow{o} s_2, s_2 \xrightarrow{r} s_3, s_0 \xrightarrow{w} s_4 \\ s_4 \xrightarrow{h} s_5, s_5 \xrightarrow{i} s_6, s_6 \xrightarrow{l} s_7, s_7 \xrightarrow{e} s_8 \end{matrix} \right\}$

FA接受字符串的条件

- 假设$\sum^*$是所有由属于$\sum$的元素组成的有限长度的序列的集合，如a+5=x+y，并且$\sum^*$包含空字符串ϵ，则FA接受字符串$w = x_1 x_2 \ldots x_k \in \Sigma^*$的充分必要条件是：
  - 存在序列$s_{t_0} s_{t_1} \ldots s_{t_n} \in S$，其中$s_{t_0}$是初始状态，$s_{t_n} \in S_{acc}$；
  - 并且$\forall s_{t_{i-1}}, x_i, s_{t_i}, (s_{t_{i-1}}, x_i, s_{t_i}) \in \Delta$；
  - 即$\delta\left(\ldots \delta\left(\delta\left(s_{t_0}, x_1\right), x_2\right) \ldots, x_n\right) \in S_{acc}$。
- 如在某一状态无匹配的状态转移规则，则转移至拒绝状态s_{rej}，并将该字符串消耗至结束或遇到分隔符。

一个标签类型对应多种词素的情况

- 如何定义标识符？
  - 字母、数字和下划线组合
  - 字母或下划线开头
  - 如果下划线开头，其后一个字符应为字母



识别标识符的FA

δ	_	a…z, A…Z	0…9	其它
s_0	s_2	s_1	s_e	s_e
s_1	s_1	s_1	s_1	s_e
s_2	s_e	s_3	s_e	s_e
s_3	s_3	s_3	s_3	s_e

状态转移表

练习

- 构造识别无符号实数的FA和状态转移表？
  - 支持浮点数和整数，如0.1、1
  - 支持科学计数法表示，如1e2，2.1e-3
    - 不支持指数浮点数，如2.1e-3.1
- 编译器的词法分析功能可以识别有符号的实数吗？
  - 如何区分负号和减号？

LLVM的例子

```
int main(){
  int a=-1;
  float b = 1.2e-2;
  float r = a + b;
  printf("result: %f\n",r);
}
```

```
#:clang -fsyntax-only -Xclang -dump-tokens token.c
```

```
int 'int'        [StartOfLine]  Loc=<token.c:2:1>
identifier 'main'        [LeadingSpace] Loc=<token.c:2:5>
l_paren '('              Loc=<token.c:2:9>
r_paren ')'              Loc=<token.c:2:10>
l_brace '{'              Loc=<token.c:2:11>
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<token.c:3:3>
identifier 'a'   [LeadingSpace] Loc=<token.c:3:7>
equal '='                Loc=<token.c:3:8>
minus '-'                Loc=<token.c:3:9>
numeric_constant '1'             Loc=<token.c:3:10>
semi ';'                 Loc=<token.c:3:11>
float 'float'    [StartOfLine] [LeadingSpace]   Loc=<token.c:4:3>
identifier 'b'   [LeadingSpace] Loc=<token.c:4:9>
equal '='        [LeadingSpace] Loc=<token.c:4:11>
numeric_constant '1.2e-2'        [LeadingSpace] Loc=<token.c:4:13>
semi ';'                 Loc=<token.c:4:19>
float 'float'    [StartOfLine] [LeadingSpace]   Loc=<token.c:5:3>
identifier 'r'   [LeadingSpace] Loc=<token.c:5:9>
equal '='        [LeadingSpace] Loc=<token.c:5:11>
identifier 'a'   [LeadingSpace] Loc=<token.c:5:13>
plus '+'         [LeadingSpace] Loc=<token.c:5:15>
identifier 'b'   [LeadingSpace] Loc=<token.c:5:17>
semi ';'                 Loc=<token.c:5:18>
identifier 'printf'      [StartOfLine] [LeadingSpace]   Loc=<token.c:6:3>
l_paren '('              Loc=<token.c:6:9>
string_literal '"result: %f\n"'          Loc=<token.c:6:10>
comma ','                Loc=<token.c:6:24>
identifier 'r'           Loc=<token.c:6:25>
r_paren ')'              Loc=<token.c:6:26>
semi ';'                 Loc=<token.c:6:27>
r_brace '}'      [StartOfLine]  Loc=<token.c:7:1>
eof ''           Loc=<token.c:7:2>
```

二、词法声明方法

正则表达式（Regular Expression）

- 定义了字母表Σ上的字符串的集合，其字符元素的表述方式包括：
  - a：含义为$\{x|x = a\}$
  - $[a-z]$：含义为$\{x|x = a\ or\ \ldots or\ x = z\}$
    - $[a-zA-Z]$：含义为$\{x|x = a\ or\ \ldots or\ x = z\ or\ \ldots or\ x = Z\}$
  - ϵ：空
  - ^a：含义为$\{x|x! = a\ and\ x \in \Sigma\}$
  - $a?$：含义为$\{x|x = a\ or\ x = \epsilon\}$
- 字符元素间以及正则表达式之间的组合方法包括：
  - 选择（union）：$R|S$ ，含义为$\{x|x \in R\ or\ x \in S\}$
  - 连接（concatenation）：RS，含义为$\{xy|x \in R\ and\ x \in S\}$
  - 闭包（closure）：R^*，含义为$\bigcup_{i=0}^{\infty} R^i$，科林（Kleene）闭包
    - 正闭包：R^+,含义为$\bigcup_{i=1}^{\infty} R^i$
    - 有限闭包：为$\bigcup_{i=1}^{n} R^i$

示例

- 正则表达式是一种（表达能力有限的）语言描述方法，可用正则表达式描述的语言称为正则语言。
- 假设$\Sigma = \{a, b\}$，则
  - $a|b$表示的语言为：{a,b}（称为正则集）
  - $(a|b)(a|b)$表示的语言为：{aa,ab,bb,ba}
  - a^*表示的语言为：$\{\epsilon$,a,aa,aaa,…}
  - $(a|b)^*$表示的语言为：$\{\epsilon$,a,b,aa,ab,ba,…}
  - $a|a^*b$表示的语言为：{a,aab,aaab,…}
- 如果两个正则表达式的正则集相等，则这两个正则表达式等价，如：
  - $a|b = b|a$
  - $(a|b)^* = (a^*|b^*)^*$

基本运算法则

- 优先级顺序：
  - 闭包（$*$）优先级最高
  - 连接符其次
  - 选择符（$|$） 最低
- 运算法则：
  - 选择符满足交换律（commutative）：$r|s = s|r$,
  - 选择符满足结合律（associative）：$r|(s|t) = (r|s)|t$
  - 连接符满足结合律（associative）：$r(st) = (rs)t$
  - 连接符满足分配律（distributive）：$r(s|t) = rs|rt$
  - 闭包满足幂等率（idempotent）：$r^* = r^{**}$

练习

- 分析下列正则表达式是否等价？
  - $a^*(a|b)^*a$
  - $\left((\epsilon|a)b^*\right)^*$
  - $b^*(abb^*)^*(a|\epsilon)$

  - $\{a, aa, ba, aaa, aba, baa, bba, \ldots\}$
  - $\{\epsilon, a, aa, ab, ba, bb, \ldots\}$
  - $\{\epsilon, a, ab, ba, bb, \ldots\}$

使用正则表达式声明词法

```
        IF  ≡  if
      ELSE  ≡  else
       FOR  ≡  for
     WHILE  ≡  while
    LPAREN  ≡  (
    RPAREN  ≡  )
IDENFIFIER  ≡  [a-z]([a-z]|[0-9])*
      UINT  ≡  [0-9][0-9]*
     UREAL  ≡  ([0-9][0-9]*.[0-9]*)|(.[0-9][0-9]*)
```

利用中间变量简化词法声明

```
    LETTER  ≡  [a-z]
     DIGIT  ≡  [0-9]
IDENFIFIER  ≡  LETTER (LETTER|DIGIT)*
      UINT  ≡  DIGIT DIGIT *
     UREAL  ≡  (DIGIT DIGIT*. DIGIT*)|(.DIGIT DIGIT*)
```

练习

- 定义无符号数的正则表达式。
  - 支持浮点数和整数，如0.1、123
  - 支持科学计数法表示，如123e2，2.1e-3
    - 不支持指数浮点数，如2.1e-3.1

$$
\begin{aligned}
\text{DIGIT} &\equiv [0\text{-}9] \\
\text{DIGITS} &\equiv \text{DIGIT DIGIT}^* \\
\text{FRACTION} &\equiv .\text{DIGITS} \mid \epsilon \\
\text{EXPONENT} &\equiv (\text{e}(+\mid -\mid \epsilon)\text{DIGITS}) \mid \epsilon \\
\text{UNUM} &\equiv \text{DIGITS FRACTION EXPONENT}
\end{aligned}
$$

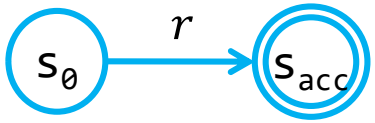
练习

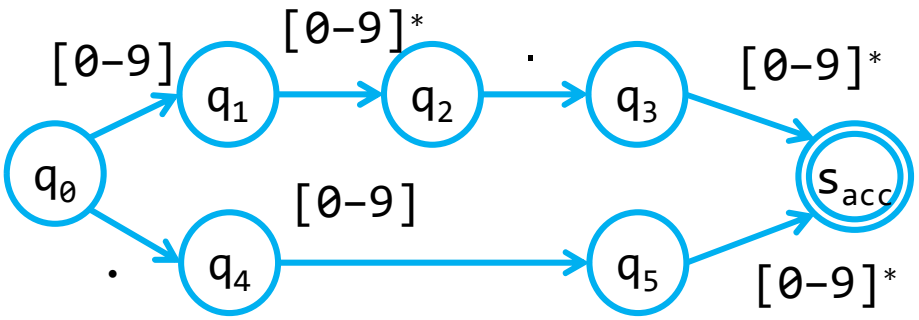
- 有些语言是大小写不敏感的，如SQL的select可写为
  - select
  - SELECT
  - sEleCt
- 如何定义其正则表达式？

三、词法分析方法

如何将正则表达式转换为FA?

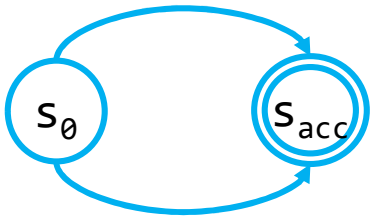
- 正则表达式：([0-9][0-9]*.[0-9]*)|(.[0-9][0-9]*)





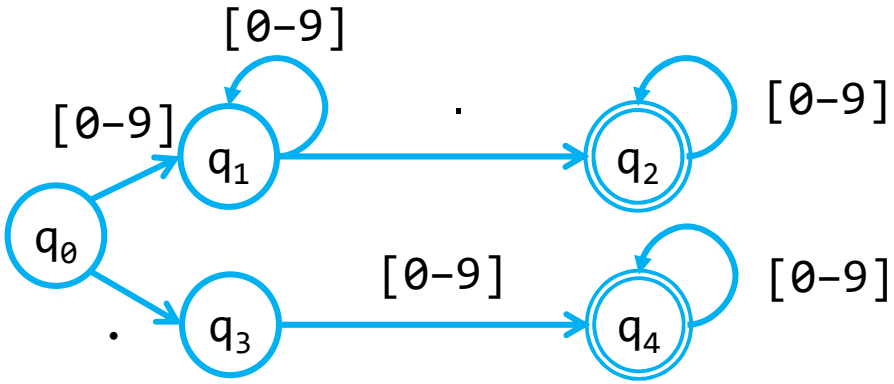
第二步

[0-9][0-9]*.[0-9]*



.[0-9][0-9]*

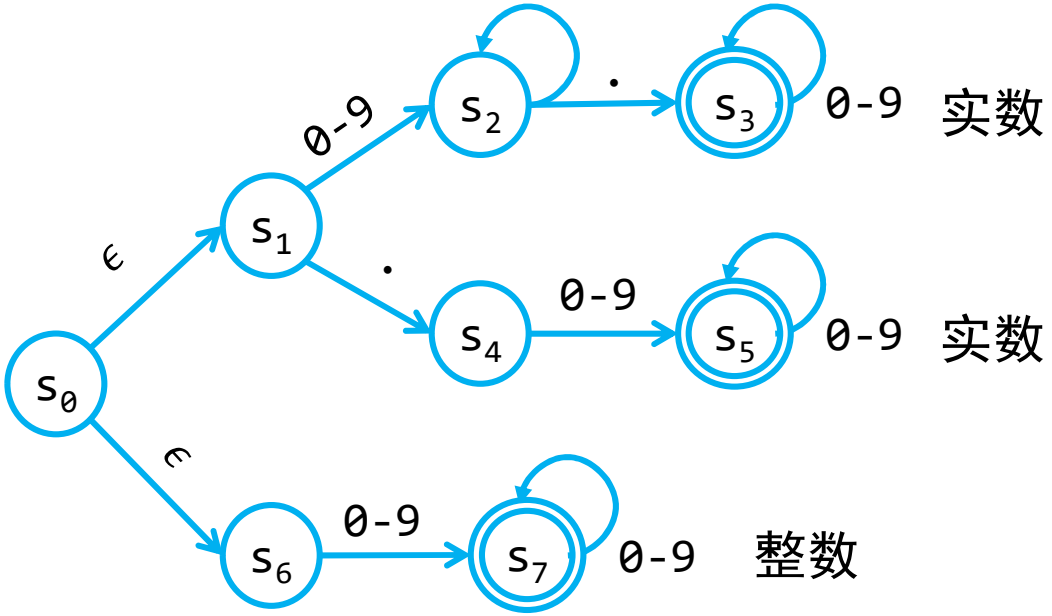
第一步



第三步

如何使用一个FA表示多个正则表达式？

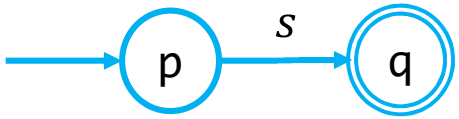
- 确定型有限自动机（Deterministic Finite Automaton）：对于FA的任意一个状态和输入字符，最多只有一条状态转移边。
- 非确定型有限自动机（Nondeterministic Finite Automaton）：对于FA的任意一个状态和输入，可能存在多条状态转移边。
- 使用ϵ转移将多个正则表达式的FA合并为一个NFA

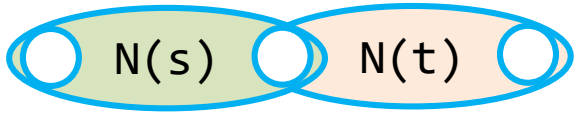


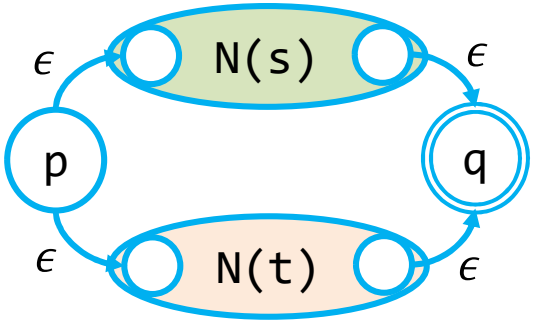
表示实数和整数的FA

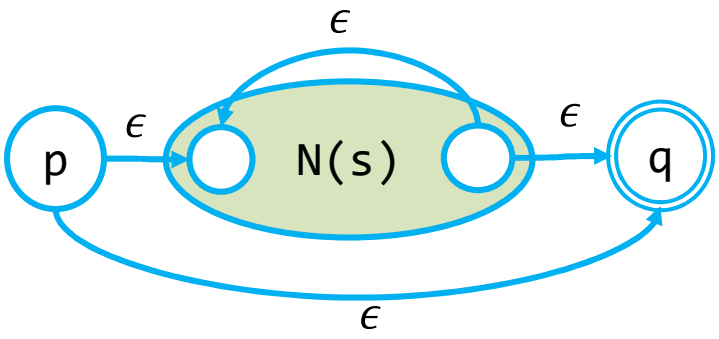
Thompson构造法：McNaughton-Yamada-Thompson

- 将正则表达式递归展开为子表达式（只有一个符号）；
  - 语法解析树
- 构造子表达式的NFA；
- 根据关系对表达式的NFA进行合并
  - 选择：$s|t$
  - 连接：st
  - 闭包：s^*



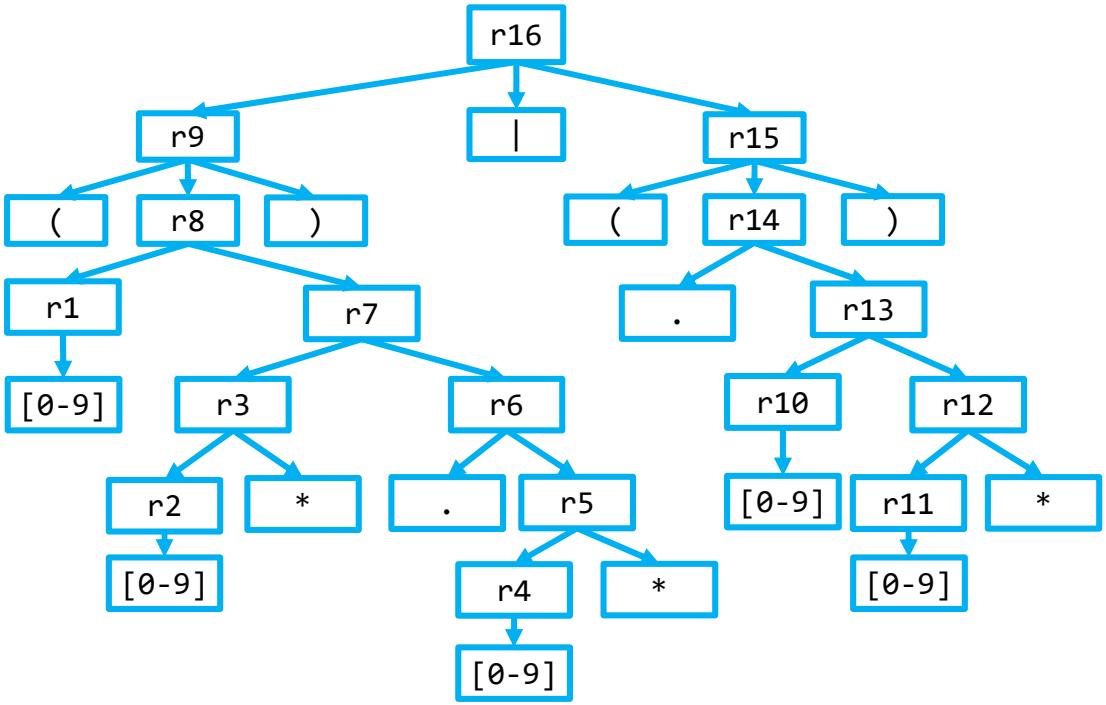






算法实现：迭代过程

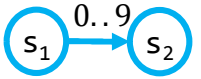
([0-9][0-9]*.[0-9]*)|(.[0-9][0-9]*)



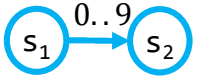
语法解析树

后序遍历

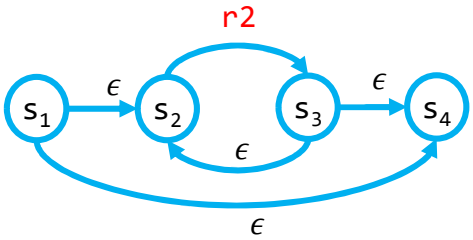
r1

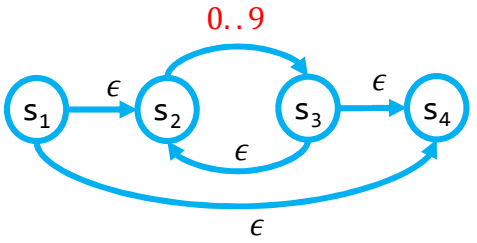


r2

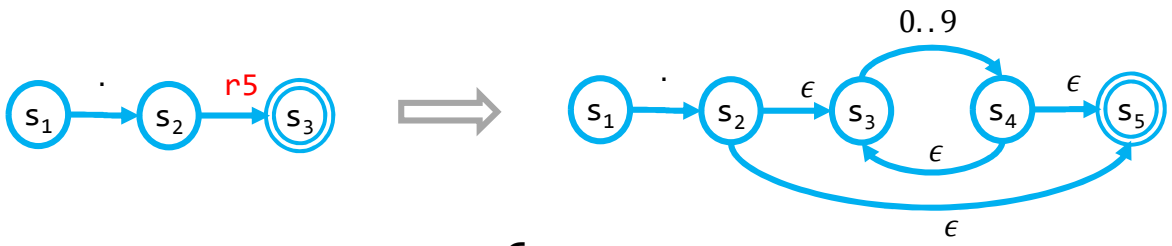


r3

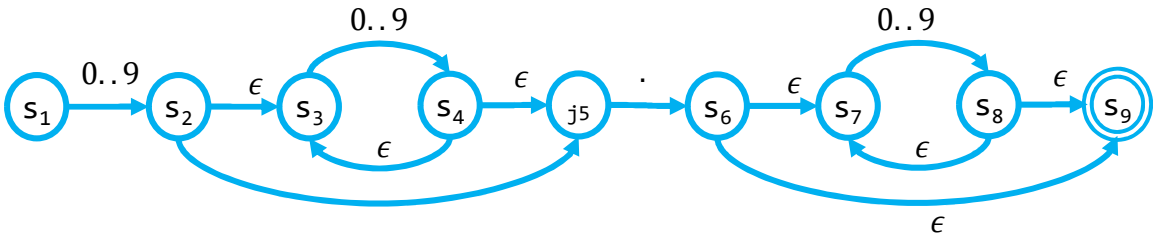




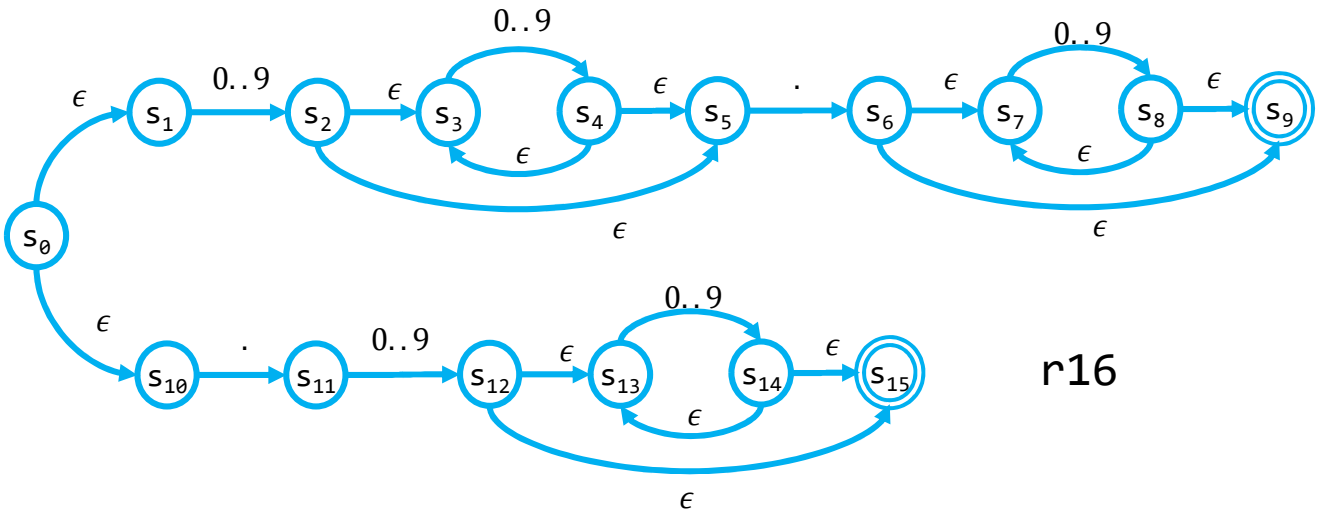
算法实现：迭代过程



r6



r8-r9



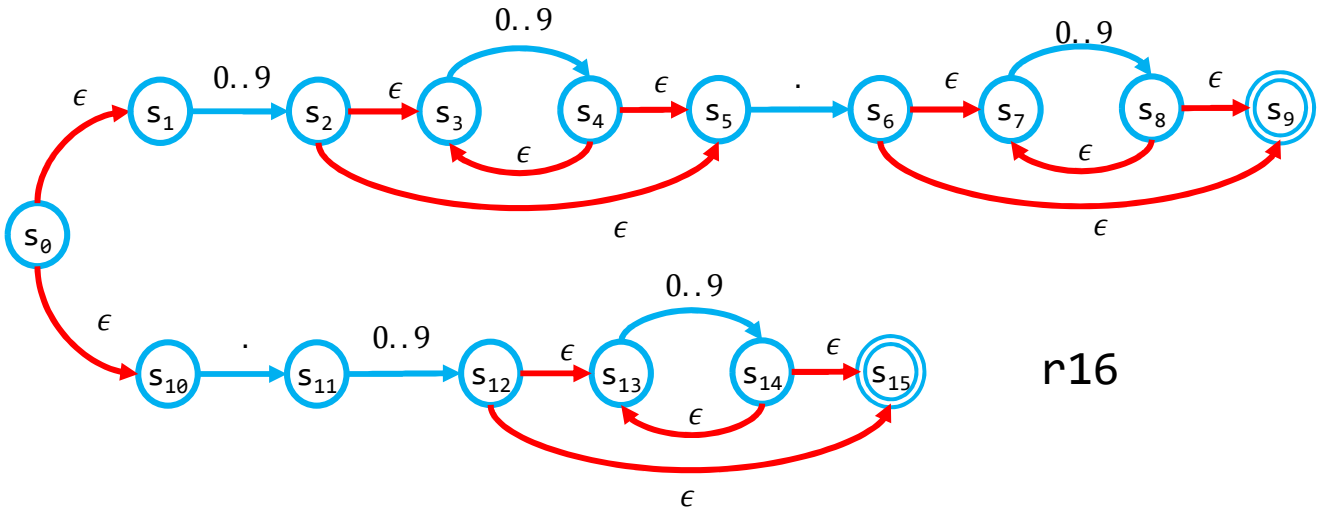
r16

NFA转换为DFA：子集构造法 Powerset Construction

- 给定一个字符集Σ上的NFA $(N, \Delta, n_0, N_{acc})$，它对应的可接受同一语言的DFA $(D, \Delta', d_0, D_{acc})$定义如下：
  - D中的所有状态d_i都是N的一个子集，$D \subseteq 2^N$
  - $d_0 = Cl^{\epsilon}(n_0)$ //ϵ闭包，同理假设d_i都为ϵ闭包
  - $\Delta' = \{d_i \times c \times d_j\}, \forall n_j \in d_j, \exists n_i \in d_i \text{ and } c \in \Sigma, (n_i, c, n_j) \in \Delta\}$
  - $D_{acc} = \{d_i \subseteq D : d_i \cap N_{acc} \neq \emptyset\}$

```
d0 = eclosure(n0);
D = d0;
worklist ={d0};
While (worklist!=null) do:
   worklist.remove(d);
   for each c in alphabets do:
      t = eclosure(d,c)
      if D.find(t) = null then:
         worklist.add(t);
         D.add(t);
```

ϵ闭包

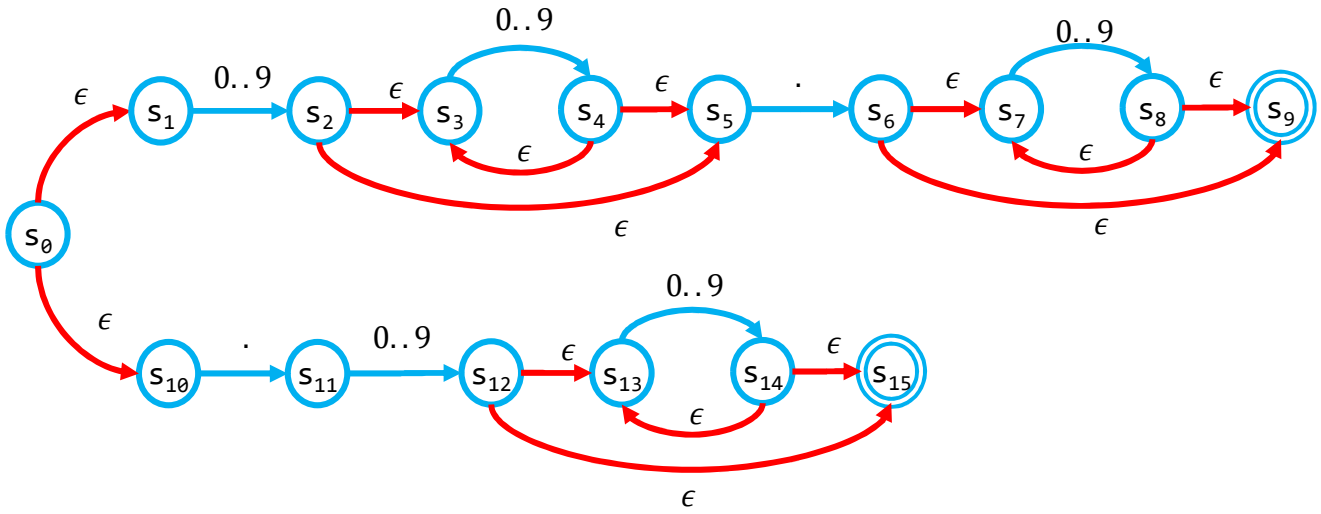


r16

- 状态s_i的ϵ闭包指的是s_i的ϵ-transition的状态集合
  - $Cl^{\epsilon}(s_i) = \cup\{s_j : (s_i, \epsilon) \rightarrow^* (s_j, \epsilon)\}$
  - $Cl^{\epsilon}(s_0) = \{s_0, s_1, s_{10}\}$
- 状态集S的ϵ闭包指的是S中所有状态的ϵ-transition的状态集合
  - $Cl^{\epsilon}(S) = \cup_{q \in S}\{q' : (q, \epsilon) \rightarrow^* (q', \epsilon)\}$
  - $Cl^{\epsilon}(\{s_0, s_1, s_2, s_{10}\}) = \{s_0, s_1, s_2, s_{10}, s_3, s_5\}$

a-transition

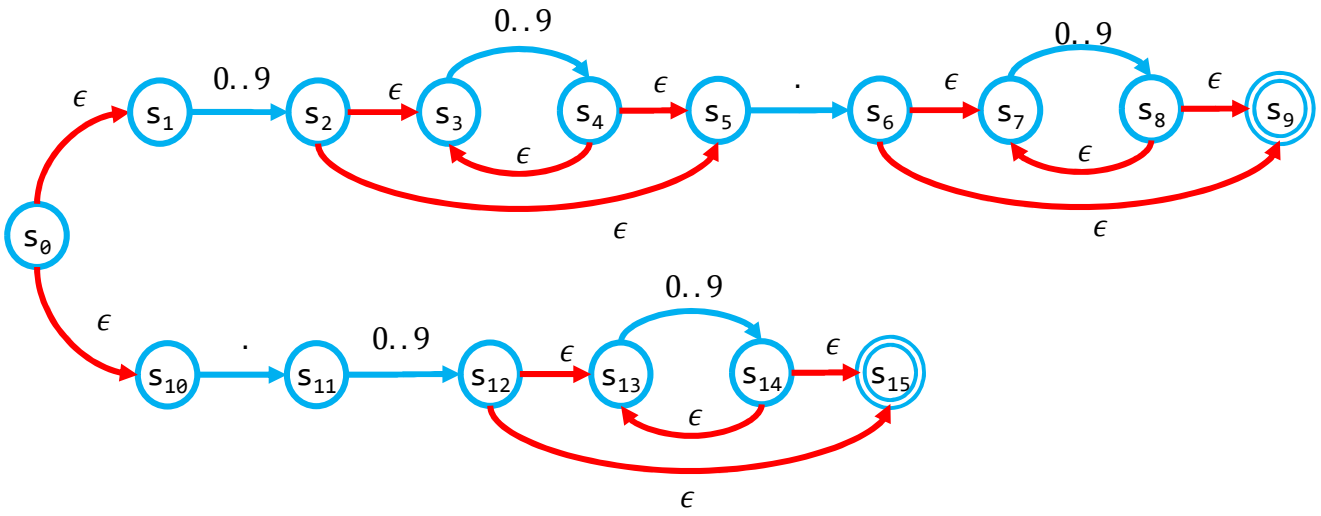
- 计算状态集S读取字符a后的状态集的ϵ闭包
  - $\delta(S,a) = Cl^{\epsilon}(\{s_j : (s_i, a) \to s_j \text{ and } s_i \in S\})$



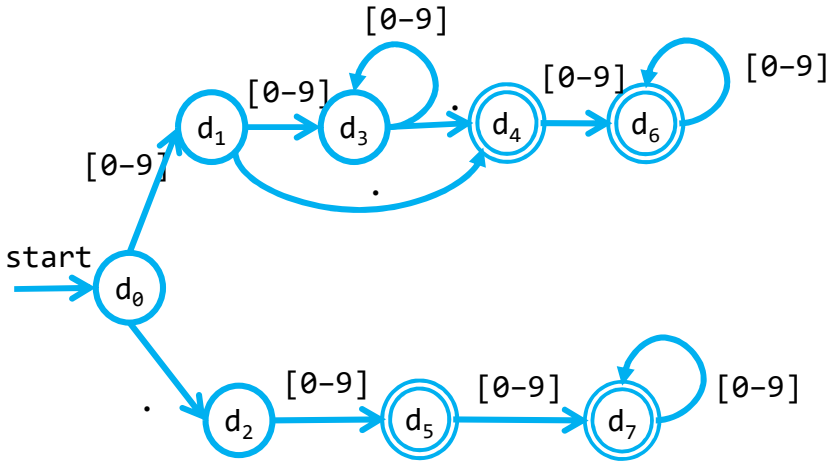
$\delta(\{s_0, s_1, s_{10}\}, 0) = \{s_2, s_3, s_5\}$

$\delta(\{s_2, s_3, s_5\}, 0) = \{s_3, s_4, s_5\}$

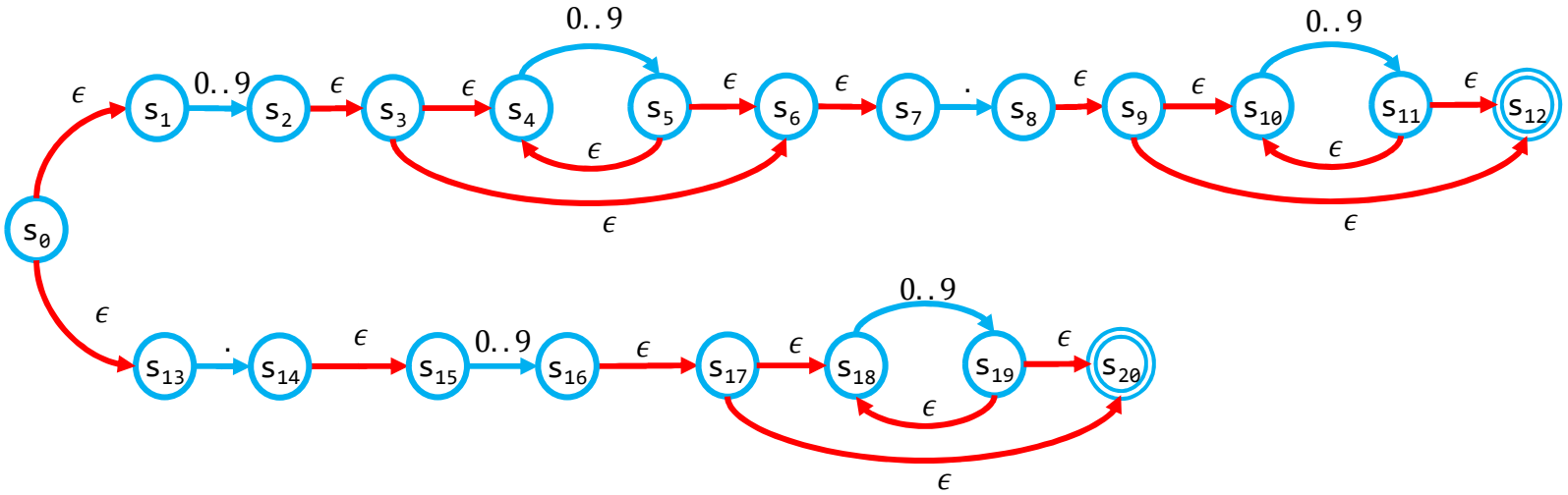
子集构造算法



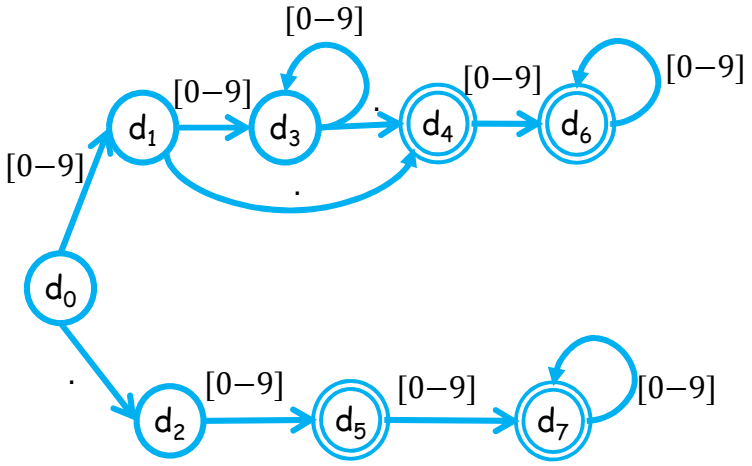
DFA 状态	NFA状态	$Cl^{\epsilon}(\delta(d,*))$	
		0 … 9	.
d_0	$\{s_0, s_1, s_2\}$	$\{s_2, s_3, s_5\}$ d_1	$\{s_{11}\}$ d_2
d_1	$\{s_2, s_3, s_5\}$	$\{s_3, s_4, s_5\}$ d_3	$\{s_6, s_7, s_9\}$ d_4
d_2	$\{s_{11}\}$	$\{s_{12}, s_{13}, s_{15}\}$ d_5	-
d_3	$\{s_3, s_4, s_5\}$	$\{s_3, s_4, s_5\}$ d_3	$\{s_6, s_7, s_9\}$ d_4
d_4	$\{s_6, s_7, s_9\}$	$\{s_7, s_8, s_9\}$ d_6	-
d_5	$\{s_{12}, s_{13}, s_{15}\}$	$\{s_{13}, s_{14}, s_{15}\}$ d_7	-
d_6	$\{s_7, s_8, s_9\}$	$\{s_{10}, s_{11}, s_{12}\}$ d_6	-
d_7	$\{s_{13}, s_{14}, s_{15}\}$	$\{s_{18}, s_{19}, s_{20}\}$ d_7	-



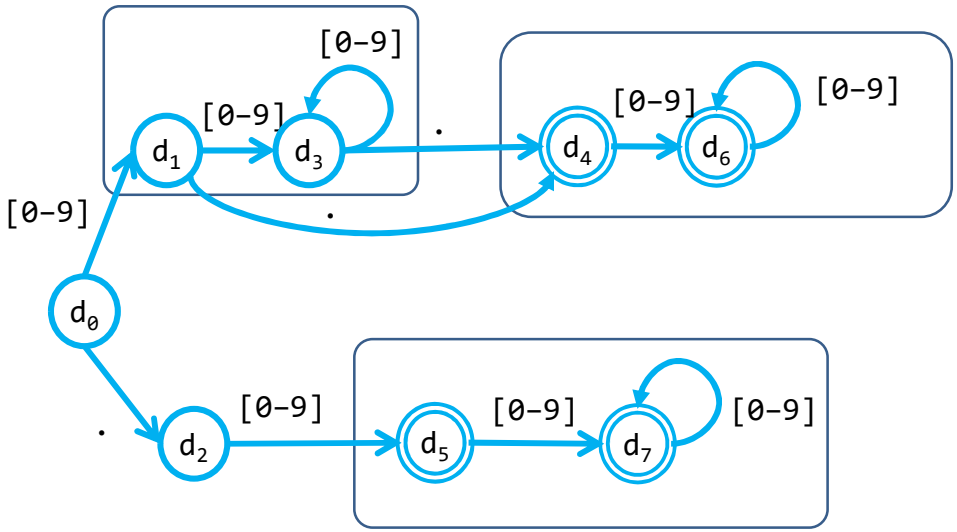
子集构造算法



DFA状态	NFA状态	$Cl^{\epsilon}(\delta(d,*))$	
		0 … 9	.
d_0	$\{s_0, s_1, s_2\}$	$\{s_2, s_3, s_4, s_6, s_7\}$	$\{s_{14}, s_{15}\}$
d_1	$\{s_2, s_3, s_4, s_6, s_7\}$	$\{s_4, s_5, s_6, s_7\}$	$\{s_8, s_9, s_{10}, s_{12}\}$
d_2	$\{s_{14}, s_{15}\}$	$\{s_{16}, s_{17}, s_{18}, s_{20}\}$	-
d_3	$\{s_4, s_5, s_6, s_7\}$	$\{s_4, s_5, s_6, s_7\}$	$\{s_8, s_9, s_{10}, s_{12}\}$
d_4	$\{s_8, s_9, s_{10}, s_{12}\}$	$\{s_{10}, s_{11}, s_{12}\}$	-
d_5	$\{s_{16}, s_{17}, s_{18}, s_{20}\}$	$\{s_{18}, s_{19}, s_{20}\}$	-
d_6	$\{s_{10}, s_{11}, s_{12}\}$	$\{s_{10}, s_{11}, s_{12}\}$	-
d_7	$\{s_{18}, s_{19}, s_{20}\}$	$\{s_{18}, s_{19}, s_{20}\}$	-

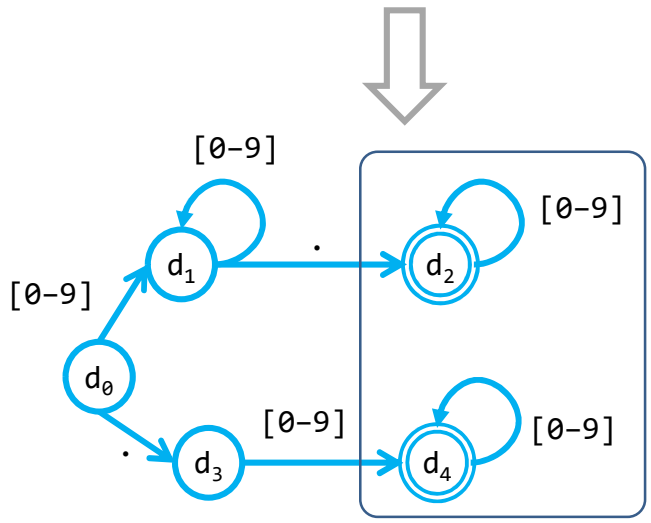


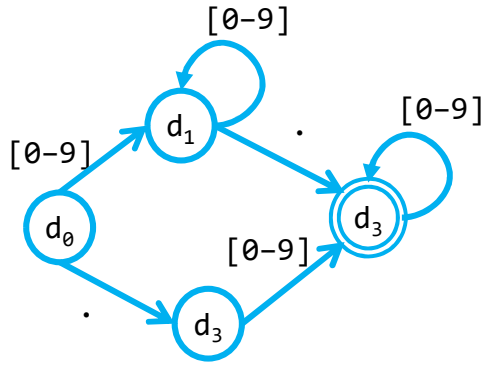
DFA优化思路：合并同类项



对于两个同类型节点d_i和d_j，可以合并的条件是：

$$\forall c \in \Sigma, \delta(d_i, c) = \delta(d_j, c)$$





d_2和d_4表示相同的词素，也可以合并

DFA优化思路：Hopcroft分割算法

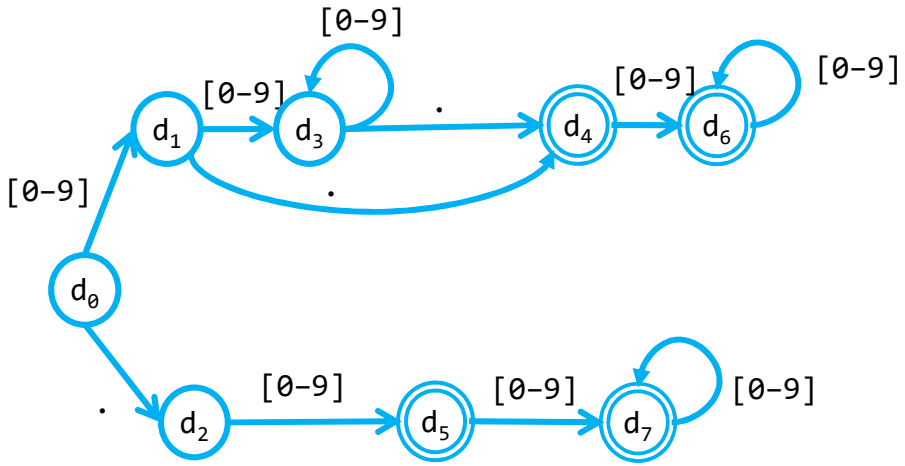
将DFA的状态集合D划分为两个子集：接受状态D_{ac}和普通状态$D\backslash D_{ac}$。

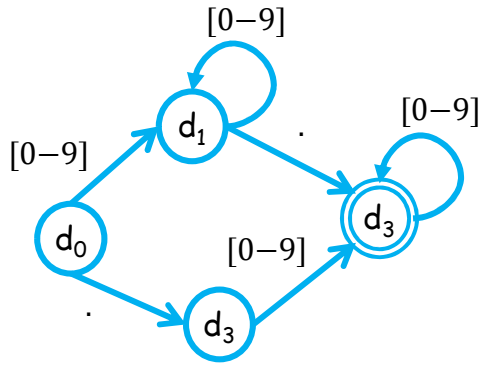
```
D = {D_ac, D\D_ac};
S = {}
While (S!=D) do:
   S = D;
   D = {};
   foreach s_i ∈ S do:
      D = D ∪ Split (s_i)

Split(s) {
   foreach c in Σ
      if c splits s into {s_1, s_2}
         return {s_1, s_2}
   return s
}
```

- 两个节点d_i和d_j不用split的条件是：
  - $\forall c \in \Sigma, \delta(s_i, c) = \delta(s_j, c)$
- 如果不同的接受状态分别对应不同词素应如何改进算法？

Hopcroft分割算法应用示例





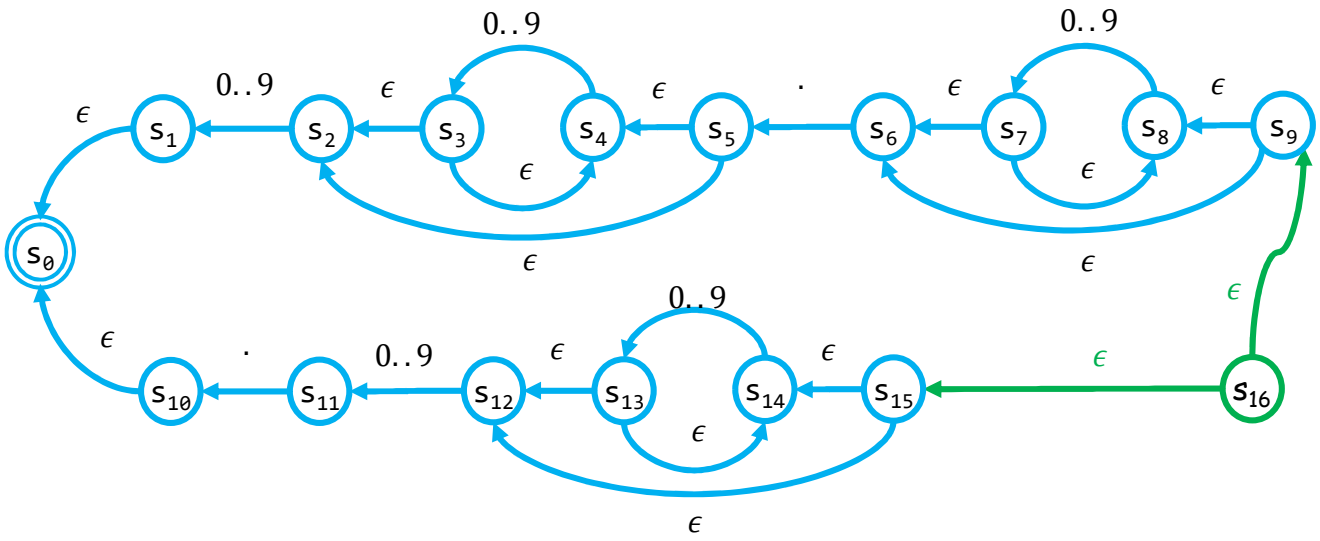
步骤	当前划分	集合	字符	Split
1	$\{\{d_0..d_3\},\{d_4..d_7\}\}$	$\{d_0..d_3\}$	0..9	$\{d_2\}\{d_0,d_1,d_3\}$
2	$\{\{d_2\}\{d_0,d_1,d_3\},\{d_4..d_7\}\}$	$\{d_0,d_1,d_3\}$	.	$\{d_0\},\{d_1,d_3\}$
3	$\{\{d_0\}\{d_2\},\{d_1,d_3\},\{d_4..d_7\}\}$	ALL	ALL	-

Brzozowski算法直接构造DFA

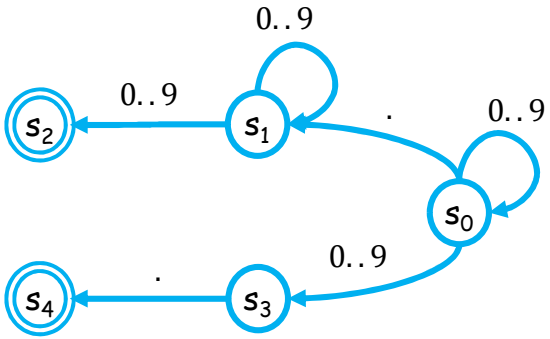
- 给定NFA，其等价的最小DFA是：

```
reachable(
   subset(
      reverse(
         reachable(
            subset(
               reverse(nfa)
)))))
```

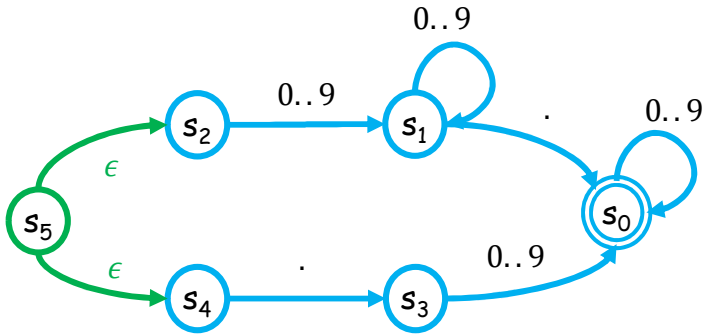
Brzozowski算法应用示例



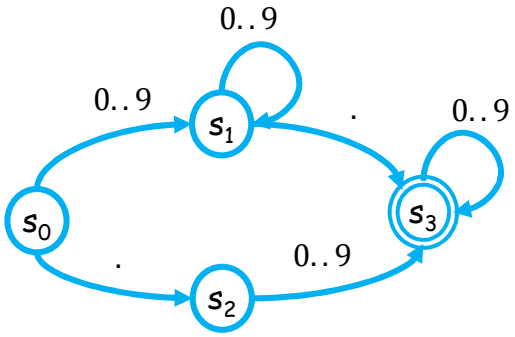
第一步reverse



第二步subset



第三步reverse



第四步subset

NFA/DFA复杂度分析

- 对于正则表达式r来说，如果采用Thompson构造法，其NFA状态最多有$|2r|$个，边最多有$|4r|$个。假设词素的长度为$|x|$，因为每个输入均可能激活$|r|$个状态，所以解析单个词素的时间复杂度为$O(|x| \times |r|)$。
- 如果采用DFA，虽然DFA的状态最多有$|2^{|2r|}|$个，但解析单个词素的时间复杂度为$O(|x|)$。
- NFA构造较快，但运行效率低；
- DFA构造耗时，但运行效率高。

练习

- 使用Thompson算法将正则表达式UNUM转化为NFA；
- 应用子集构造法将UNUM的NFA转化为DFA；
- 化简上一步得到的DFA。

$$
\begin{aligned}
\text{DIGIT} &\equiv [0\text{-}9] \\
\text{DIGITS} &\equiv \text{DIGIT DIGIT}^* \\
\text{FRACTION} &\equiv .\text{DIGITS} \mid \epsilon \\
\text{EXPONENT} &\equiv (\text{e}(+ \mid - \mid \epsilon)\text{DIGITS}) \mid \epsilon \\
\text{UNUM} &\equiv \text{DIGITS FRACTION EXPONENT}
\end{aligned}
$$

练习

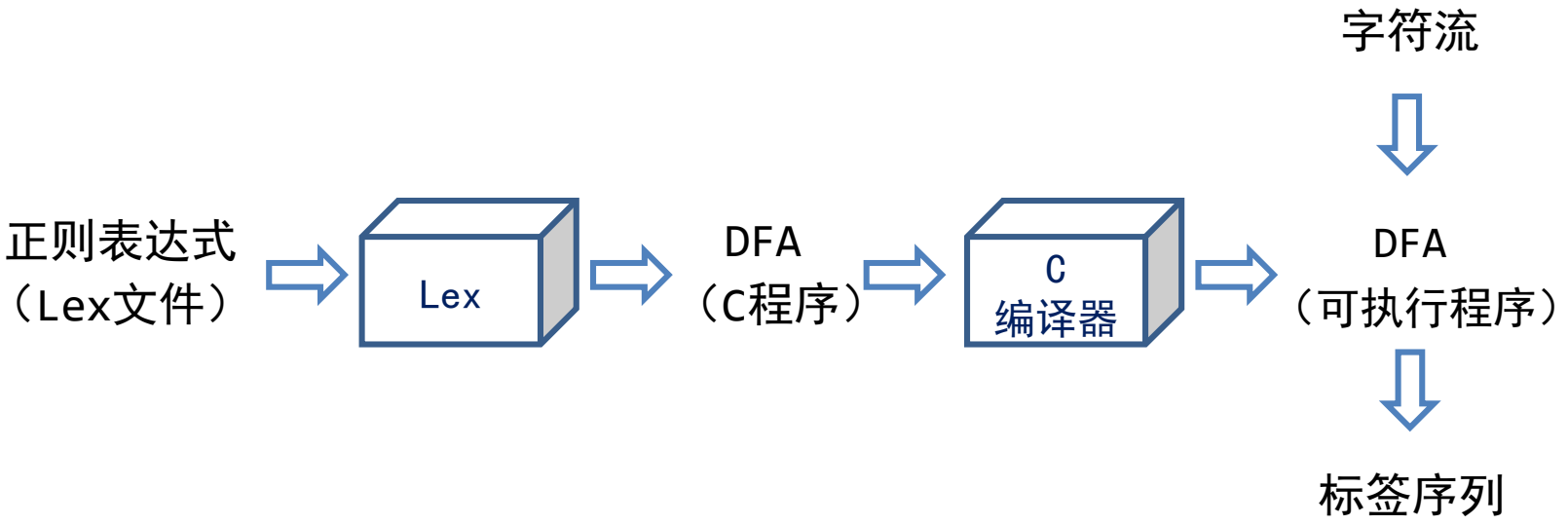
- 使用Thompson算法将下列正则表达式转化为NFA；
- 应用子集构造法将上一步得到的NFA转化为DFA；
- 化简上一步得到的DFA。

$$
\begin{aligned}
\text{IF} &\equiv \text{if} \\
\text{ELSE} &\equiv \text{else} \\
\text{FOR} &\equiv \text{for} \\
\text{WHILE} &\equiv \text{while} \\
\text{IDENFIFIER} &\equiv [a\text{-}z]([a\text{-}z]|[0\text{-}9])^*
\end{aligned}
$$

四、词法分析工具

Lex

- 词法分析器生成工具：Lex (POSIX)/Flex(GNU)
- 通常和语法分析工具YACC(POSIX)/Bison (GNU)配合使用。



示例

```
%{
#include “yy.tab.h”//定义PLUS、 MINUS等常量
%}
//词法正则定义
digit          [0-9]
letter         [a-zA-Z]
identifier     {letter}({letter}|{digit})*
number         {digit}+

//词法识别和转换规则定义
%%
"+"                  { return PLUS;       }
"-"                  { return MINUS;      }
"*"                  { return TIMES;      }
"/"                  { return SLASH;      }
";"                  { return SEMICOLON;  }
"="                  { return EQL;        }
if                   { return IFSYM;      }
else                 { return ELSESYM;     }
{identifier}         {
                       yylval.id = strdup(yytext);
                       return IDENT;      }
{number}             {
                       yylval.num = atoi(yytext);
                       return NUMBER;     }
%%
```

冲突处理

- 一个输入与多个词素模式匹配时，选择最长的匹配
  - <=不会识别为<和=
  - ifabc不会识别为保留字if和标识符abc
- 如果匹配长度相同，则优先匹配前面的规则
  - 保留字优先级高于标识符

五、应用扩展

日志解析问题

问题：给定API定义，识别日志属于哪个API

API定义由常量和变量组成：白色表示常量，黄色表示变量

```
GET /projects/:id/repository/branches
```

```
GET /projects/:id/repository/branches/:branch
```

```
GET /projects/:id/repository/commits/:sha
```

日志为访问记录

```
2021-07-04 16:43:47.193: Sending: 'GET
/api/v4/projects/XXXmyfuzzingstringXXX/repository/branches?search=fuzzstring
HTTP/1.1\r\nAccept: application/json\r\nHost:
10.177.75.243\r\n_OMITTED_AUTH_TOKEN_\r\nContent-Length: 0\r\nUser-Agent:
restler/7.5.0\r\n\r\n'
2021-07-04 16:43:49.761: Sending: 'GET /api/v4/projects/ XXXmyfuzzingstringXXX
/repository/commits?ref_name=fuzzstring&since=fuzzstring&until=fuzzstring&path=fuzzstring&all
=true&with_stats=true&first_parent=true&order=fuzzstring HTTP/1.1\r\nAccept:
application/json\r\nHost: 10.177.75.243\r\n_OMITTED_AUTH_TOKEN_\r\nContent-Length: 0\r\nUser-
Agent: restler/7.5.0\r\n\r\n'
```

https://docs.gitlab.com/ee/development/documentation/restful_api_styleguide.html

方法

- 方法一：正则表达式+For循环
  - 复杂度：O(|字符串长度| × 表达式个数)
- 方法二：正则表达式优化+For循环
  - 一般只能优化单个含义的正则
    - 'xaz|xbz|xcz’ => x[a-c]z
  - 如使用regex-opt工具
- 方法三：基于子集构造法（Lex工具）

https://bisqwit.iki.fi/source/regexopt.html

课后阅读

- 《编译原理（第2版）》第三章：Lexical Analysis;
- 《编译器设计（第2版）》第二章：词法分析器。