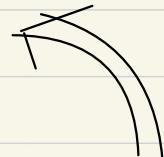


1

■ 串
■ 语言：串的集合

语言上的运算 (集合上的运算)

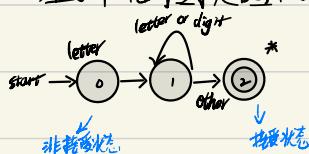
并
连接
闭包
正闭包



■ 正则表达式：描述所有通过对“给定字母表”上的符号应用这些运算符得到的语言

2

基于状态转换图的词法分析器



\Rightarrow “确定的”，意味在任意状态对于给定的任一符号最多只有一条边
从该状态离开的
的标号包含该符号

3 Lex (FLEX)

□ Lex：将输入的模式转换成状态转换图，并生成能够模拟该状态转换图的代码。

□ Lex 的结构

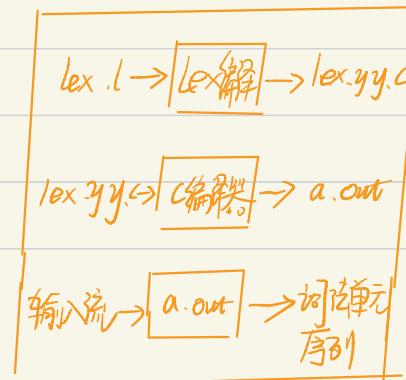
声明部分

%%

转换规则

%%

辅助函数



□ Lex 的冲突解决

1) 总是选择最长的前缀 (在后面讲实现部分有体现)

2) 如果最长的可能前缀与多个模式匹配, 总是选择先被列出的模式

④ 有穷自动机

□ NFA、DFA、正则表达式

1) NFA 状态、符号、转换表

2) DFA { 没有 ϵ 输入 }

确定 → 对每个状态 s 和每个输入符号 a 有且仅有一条标记

3) NFA、DFA、正则表达式 为 a 的边离开状态 s

描述能力等价 (即它们描述的语言集合相同)

⑤ 从正则表达式到有穷自动机

□ $NFA \rightarrow DFA$ [子集构造法]

算法思想: 不断地构建 DFA 的转换表, 使 DFA 模拟 NFA

“状态” \Rightarrow “状态集” $\xrightarrow{\text{NFA}}$ { ϵ }
不确定

过程描述:

$S' = \epsilon\text{-close}(S_0)$, S' 放入 D_{states}

while (D_{states} 中未访问的状态 S') {

 while (符号集中每个符号 a) {

$T = \text{move}(S', a)$

$S'' = \epsilon\text{-close}(T)$, S'' 放入 D_{states}

$D_{tran}[S', a] = S''$

DFA 的接受状态
是那些至少包含一个 NFA 中接受状态
的状态“集合”

ϵ -close(T) 计算:

ret = T

结果, 状态集

把 T 中所有状态压入栈

while (栈不为空, 出栈得 s) {

 while (由状态 s 经 ϵ 可到达的状态 u) {

 if (状态 u 不在 ret 中) \rightarrow 也说明 u 不曾在栈中

 把 u 加入 T, 把 u 压栈;

}

}

□ NFA 的模拟 —— 在线的子集构造算法,

已知: NFA (开始状态 s_0 , 接受状态集 F, 转换函数 move)

输入: 输入串

(注: 在线的含义是逐步地读取输入串,

1) $S = \epsilon\text{-close}(s_0)$

然后依次记录“状态集”的变化,

2) $c = \text{nextChar}();$

最后通过判断最后的“状态集”与 F 是否存在

3) while ($c \neq \text{eof}$) {

交集来决定输入串是否所属 NFA 描述

4) $S = \epsilon\text{-close}(\text{move}(S, c))$; 的语言)

$c = \text{nextChar}();$ ↴

}

if ($S \cap F \neq \emptyset$) return "yes";

else return "no";

口在线子集构造法的高效实现(及时复分析)

• 主要考虑第 4 行的实现

• 这里使用两个栈 { oldS 存上一次所能到达的所有状态,

NFA | newS 存当前所能到达的所有状态,

使用一个以状态作为下标的数组 alreadyOn 记录某状态是否已在 newS

另用一二维数组 mov(s, a) 保存转换表 中出现

过程实现为:

for (oldS 中所有状态 s) {

 for (mov(s, a), s 经 a 可达的所有状态 u) {

 if (!alreadyOn[u])

 addStates(u);

 将 s 弹出 oldS;

= } }

 addStates(s);

 将 s 压入 newS; alreadyOn[s] = TRUE;

 for (mov(s, t), s 经 t 可达的所有状态 t) {

 if (!alreadyOn[t])

 addStates(t);

 }

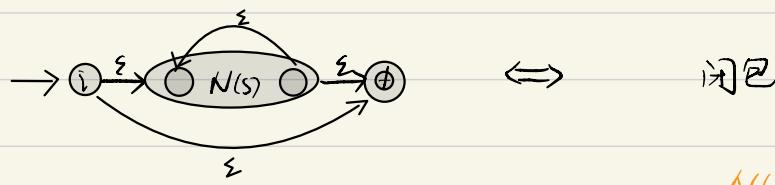
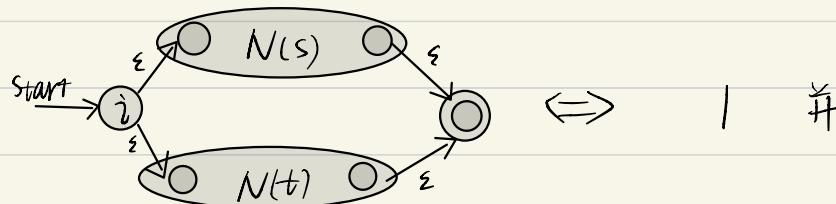
• 时复分析 [设总状态数(结点数) m, 转换数(边) n, 输入字符串长 k]

$O(k(m+n))$

6 从正则表达式构造NFA McNaughton-Yamada-Thompson 算法

思想：「正则表达式是一种表达式，所以可以把它转成语法分析树
 对于语法分析树中的每一结点（即正则表达式中的各种运算）
 考虑 如何转换成对应的NFA结点」

规则：



NFA 表示子 NFA

由正则表达式转换得到的NFA具有以下性质

- 1) 结点数不多于表达式中运算符和运算分量总和的2倍
- 2) NFA有且仅有一个开始状态和一个接受状态
- 3) 除接受状态外的状态要么有一条带号的出边，要么有2条空的出边

⑦ 正则表达式识别的效率

口 通过 NFA

构建 NFA 的开销

$O(|r|)$

识别每个串的开销

$O(k(m+n))$ k 表串的长度

$$= O(k|r|) \rightarrow \text{根据⑥中 NFA 的性质}$$

可得 $m \leq 2|r|$, $n \leq 4|r|$

口 通过 DFA

先构 NFA $O(|r|)$

NFA \rightarrow DFA

{ 针对一个状态一个输入需, $O(m+n) = O(|n|)$

针对一个状态(最多 $|n|$ 个输入), 需 $O(|k|^2)$

针对所有状态(假设有 S 个), 需 $O(S|k|^2)$

状态 \rightarrow 状态集

一般情况下, S 大约等于 H , 故需 $O(|H|^3)$

最坏情况下, 会得到 2^m 个状态, 需 $O(|H|^2 2^m)$

识别每个串的开销 $O(k)$

口 附: 利用 NFA/DFA 模式匹配过程

到达死状态!

以 NFA 为例, 匹配过程按 NFA 模拟过程持续执行。最后会进入一个没有后续状态的输入点。此时, 沿状态 $\xrightarrow{\text{集转换}}$ 路径回头寻找, 直至找到一个接受状态(最长前缀)。若此时状态集中含多个接受状态, 选择 Lex 程序中位置靠前的模式关联的那个接受状态。

另外, 需注意下一匹配过程, 字符串应该从接受状态对应的位置向后读开始输入。

⑦ 基于 DFA 的模式匹配器的优化 |

□ 直接把正则表达式生成对应的 DFA → 目的：优化 DFA 的状态数

[思路] ① 整体上还是先把正则表达式转为 NFA，再转为对应的 DFA

but，此~~处~~得到的 NFA 无~~ε~~状态，故能自动生成的 DFA 状态数↓

- 1) 正则表达式是个表达式，所以它有对应的抽象语法树，
树的叶子结点对应表达式中的运算符，而非叶结点则为运算符。
把这些叶子结点（除 ϵ 叶结点外）标号（被称为‘位置’）。

- 2) 对树中结点或位置求以下函数：

$\text{nullable}(n)$ ：返回 TRUE 则表示结点 n 可生成空串，否则返回 FALSE

$\text{firstpos}(n)$ ：返回一集合，集合中的元素都为位置，这些位置都可^以作为结点 n 代表的子表达式可匹配串的开头字符

$\text{lastpos}(n)$ ：类似 $\text{firstpos}(n)$ ，不过其中位置都可作为结尾字符。

$\text{followpos}(p)$ ：返回一集合，集合中的元素都为位置，这些位置都可作为位

置 p 的后续位置

→ $\begin{cases} \text{nullable} \\ \text{firstpos} \\ \text{lastpos} \end{cases}$ 都服务于 followpos 的计算

- 3) 位置实际与 NFA 的重要状态（重要状态指含非~~ε~~边的状态）相关，可根据

$\text{followpos}(p)$ 构建与对应正则表达式等价的 NFA.

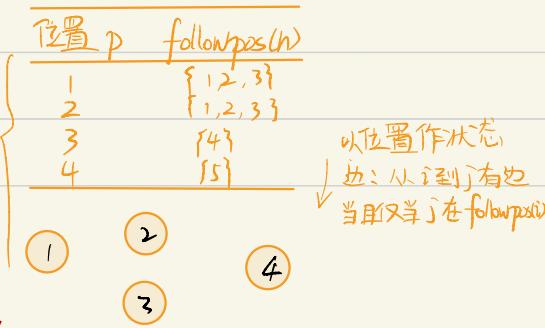
- 作 followpos 的有向图

• 以根结点的 firstpos 中的所

有位置设为开始状态

- 把和结尾~~非~~相关的~~位置~~作为唯一

正则表达式在最后添加~~#~~的接受状态
以示结束且使原接受状态可被视为主要状态



4) 把上述的NFA转为DFA(即子集构造法)

初始化 D_{vars} , 使之只包含未标记的状态 $\text{first pos}(n_0)$,

其中 n 是二叉树的结点数。

while (Dstates 中存在未标记的状态 s) {

标记S；

for (每个输入符号a) {

令 U 为 S 中和 α 对应的所有位置 p ($\text{followpos}(p)$) 的
并集；
且 (U 不在 D_{stares} 中)

将 U 作为未标记的状态，且加入 D_{states} 中；

$$D_{tran}[S, a] = U,$$

3

}

17 followpos(P) 的计算

{ 对。连接运算符： $\text{firstpos}(j)$ 都出现在 $\text{lastpos}(i)$ 中且置 followpos 中

对 * 闭包运算符：若 a 是 $\text{lastpos}(n)$ 中一位置，则 $\text{firstpos}(n)$ 都在 $\text{followpos}(n)$ 中
 i.) 分别作为运算符对括结点的左、右子结点， n 为此运算符结点

□ 时复分析

构建语法分析树 $O(n)$ \rightarrow 针对结点

nullable、firstpos、lastpos
O(|T|)

followpos \rightarrow 针对值(动态) $O(H)$

$NFA \rightarrow DFA$ $O(S \cdot H)$ \rightarrow DFA 的状态数

8 DFA的优化2

□ 最小化 DFA 的状态数

→ 已知 DFA 来缩减其状态数

思路：① 状态 s 和状态 t 是否能被串 a 区分，若分别从 s 、 t 出发经 a 后到达的两状态一个接收、另一个非接收，则状态 s 和状态 t 是可区分的。

② 给状态分组，^{要求}组中任意两状态对任意串不可区分。最后一组代表一状态，组合它们便成了状态最少的 DFA

原理：把 DFA 的状态集 Π 不断地划分成多个组，每个组中的状态不可区分，组间状态可区分因而划分。

过程：

- 1) 初始状态被分为两组，一个是接受状态组，另一为非接收状态组
- 2) 分割得 Π_{new} ，直至 Π_{new} 未发生变化

$$\text{令 } \Pi_{new} = \Pi$$

for (Π 中的每个组 G) {

- 把 G 分为更小的组，^{使得}两状态 s, t 在同一小组中当且仅当对于所有的输入符号 a ，状态 s, t 在 a 上的转换都到达 Π 中的同一组。
- 在 Π_{new} 中将 G 替换为对 G 进行划分得到的那些小组。

}

- 3) 令 Π_{final} 为最后得到的 Π_{new} ，按如下构建 D'

• 组作为 D' 的状态

• D' 的开始(结束)状态是含原 D 的开始(结束)状态的组

• D 中有 $s \xrightarrow{a} t$ 的转换，则在 D' 有 ‘ s 对应组’ \xrightarrow{a} ‘ t 对应组’ 的转换