

编译原理笔记

陈鸿峥

2020.05*

目录

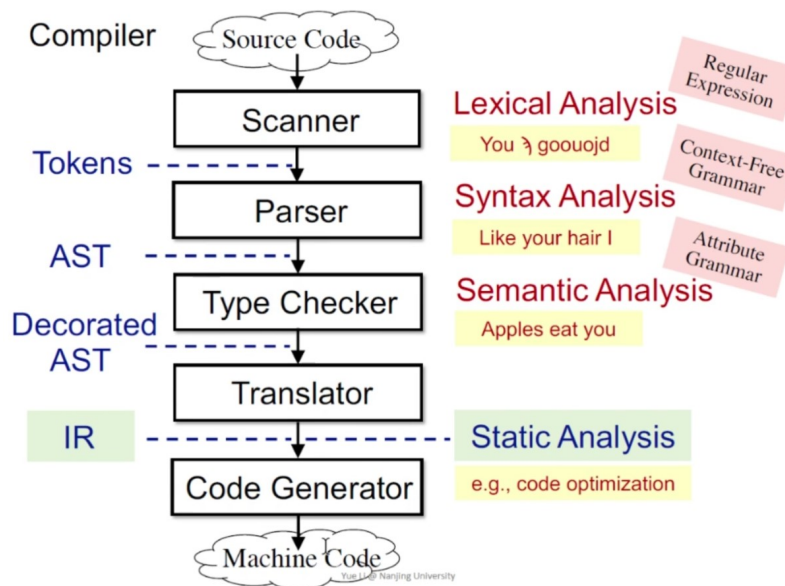
1	简介	1
2	词法分析	2
2.1	基本定义	2
2.2	正则表达式	3
2.3	有限自动机	4
2.4	Regex转DFA	7
2.5	最小化DFA	9

本课程采用书目Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques & Tools (2nd ed)*, 即大名鼎鼎的龙书。

1 简介

编译器的几个阶段如下，前端包括词法(lexical)、语法(syntax)、语义(semantic)分析，中端IR生成、优化，后端代码生成。

*Build 20200512



2 词法分析

分离词法分析和语法分析可以简化这两个任务，同时提升编译器的性能与兼容性。

2.1 基本定义

定义 1. 令牌(*token*)是一个令牌名字与可选属性值构成的对；模式(*pattern*)描述了每个词素(*lexeme*)要遵循什么规则；而词素（最小意义单位）则是源程序中一连串满足模式的字母，作为令牌的实例化。

例 1. 考虑C语句

```
printf("Total = %d\n", score);
```

其中`printf`和`score`是匹配(*match*)上令牌`id`模式的词素，而`"Total = %d\n"`是匹配上字面值`literal`的词素。

简单来讲，令牌是一个更大的概念，是同类词素的集合。比如一个令牌`comparison`的样例词素可以有`<=`和`!=`。

定义 2 (字母表与语言). 字母表(*alphabet*) Σ 是有限符号(*symbol*)的集合，如ASCII就是一个字母表。字符串(*string*) s 是从字母表中抽取的有限符号的序列， $|s|$ 为字符串长度， ϵ 为空串。语言(*language*)是字符串的可数集合。

例 2. 字母表 $\Sigma = \{0, 1\}$ ，则 $\{001, 1001\}$ 和 $\{\}$ 都是定义在 Σ 上的语言。

定义 3 (字符串术语). 前缀(*prefix*)和后缀(*suffix*)都可以包括 ϵ 。子串(*substring*)可通过删除任意前缀和任意后缀（包括零个）获得。真(*proper*)子串则不包含 ϵ 。子序列(*subsequence*)是删除零个或多个不一定连续的字母得到的字符串。

语言是一种集合，故集合运算也适用于语言。

并集(union)	$L \cup M$
连接(concatenation)/交集	LM
柯林闭包(Kleene closure)	$L^* = \cup_{i=0}^{\infty} L^i$
正闭包(positive)	$L^+ = \cup_{i=1}^{\infty} L^i$

2.2 正则表达式

定义 4 (正则表达式(regular expression, regex)). 正则表达式 r 定义了语言 $L(r)$, 以递归形式定义:

1. 奠基:

- ϵ 是正则表达式, 即 $L(\epsilon) = \{\epsilon\}$
- $a \in \Sigma$ 是正则表达式, 即 $L(a) = \{a\}$ (这里用斜体代表符号, 粗体代表符号对应的正则表达式)

2. 推论(induction): 若 r 和 s 都是正则表达式给出了语言 $L(r)$ 和 $L(s)$, 则

- $(r)|(s)$ 是正则表达式, 表示 $L(r) \cup L(s)$
- $(r)(s)$ 是正则表达式, 表示 $L(r)L(s)$
- $(r)^*$ 是正则表达式, 表示 $(L(r))^*$
- (r) 是正则表达式, 表示 $L(r)$

正则表达式表示的语言叫做正规集。

有以下运算规定:

- 一元运算符 $*$ 有最高优先级, 左结合
- 连接优先级次之, 左结合
- $|$ 优先级最低, 左结合

定义 5 (正则定义). $d_i \rightarrow r_i$, 其中 d_i 都是名字, 且各不相同。每个 r_i 是 $\Sigma \cup \{d_1, \dots, d_{i-1}\}$ 中符号上的正则表达式。

例 3. 比如C语言的标识符可记为

$$letter_ \rightarrow A|B|\dots|Z|a|b|\dots|z|_$$

$$digit \rightarrow 0|1|\dots|9$$

$$id \rightarrow letter_ (letter_ | digit)^*$$

正则表达式的拓展¹:

- r^+ 代表一个或多个
- $r?$ 代表零或一个
- $[a - z]$ 字母类

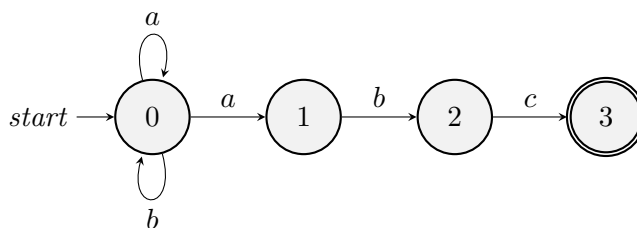
¹更多可参见[Regex101](#)

2.3 有限自动机

2.3.1 确定性/非确定性有限自动机

确定有限自动机(DFA)不可对 ϵ 进行移动, 而且对于每一状态 s , 输入符号 a , 只有唯一一条出边标记为 a ; 而非确定性有限自动机(NFA)可能有多种转换路径。有限状态集 S , 状态 $s_0 \in S$ 为初始状态(start/initial), $F \subset S$ 为终止状态(accepting/final)。

例 4. 识别语言 $L((a|b)^*abb)$, 下面为一个 NFA



判别字符串能否被DFA识别很简单, 只需要读入字符按照状态转移表跳转, 判断末态是不是终态即可。

Algorithm 1 基于DFA的识别算法

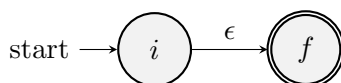
```
1:  $s = s_0$ 
2:  $c = nextChar()$ 
3: while ( $c \neq eof$ ) do
4:    $s = move(s, c)$ 
5:    $c = nextChar()$ 
6: if  $s \in F$  then
7:   return "yes"
8: elsereturn "no"
```

时间复杂度为 $O(|str|)$ 。

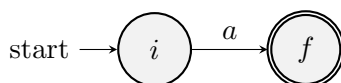
2.3.2 正则表达式转NFA

1. 奠基

- 对于表达式 ϵ , 构建NFA

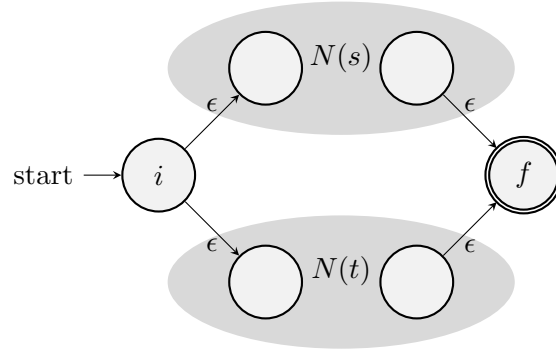


- 对于任意子表达式 $a \in \Sigma$, 构建NFA

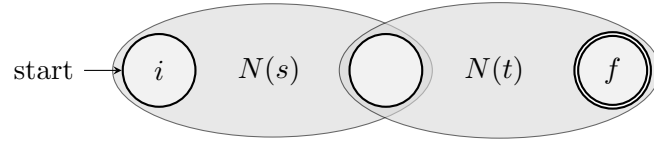


2. 推论

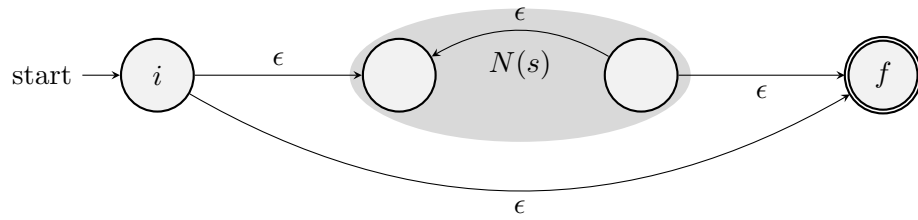
- $r = s|t$, 取并集



- $r = st$, 取连接



- $r = s^*$, Kleene闭包



2.3.3 NFA转DFA

定义 6 (ϵ 闭包及 $move$). ϵ 闭包是可通过NFA的 ϵ 边转换的状态。 $move(T, a)$ 为状态 $s \in T$ 通过输入符号 a 可到达的新的状态。

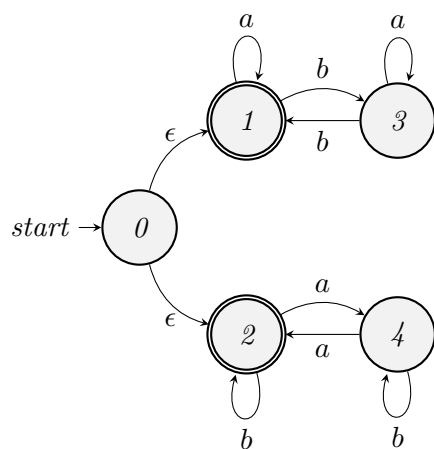
Algorithm 2 子集构造 (NFA转DFA)

Require: NFA N

Ensure: DFA D (与 N 接受相同的语言)

- 1: ϵ -closure(s_0)是 $Dstates$ 的唯一状态, 且未被标记(unmarked)
 - 2: **while** 在 $Dstates$ 中还有未被标记的状态 T **do**
 - 3: 标记 T
 - 4: **for** 每一个输入符号 a **do**
 - 5: $U = \epsilon$ -closure($move(T, a)$)
 - 6: **if** $U \notin Dstates$ **then**
 - 7: 将 U 作为未标记的状态加入 $Dstates$
 - 8: $Dtran[T, a] = U$
-

例 5. 考虑以下NFA:



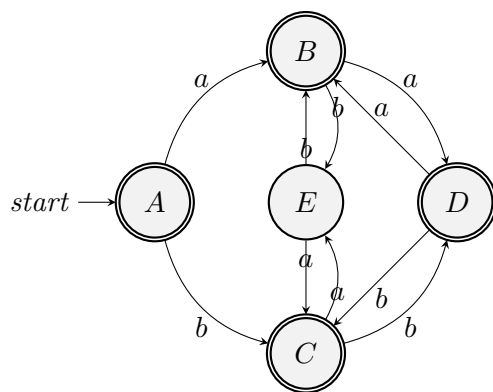
1. 这一NFA接受什么语言（用自然语言描述）？

2. 构造接受同一语言的DFA.

分析. 1. 含有偶数个 a 或偶数个 b 的由 a 、 b 构成的字符串，或者全是 a 或全是 b

2. 由subset construction算法构造如下

NFA	DFA	a	b
$\{0, \underline{1}, 2\}$	A	$\{1, 4\}$	$\{2, 3\}$
$\{\underline{1}, 4\}$	B	$\{1, 2\}$	$\{3, 4\}$
$\{\underline{2}, 3\}$	C	$\{3, 4\}$	$\{1, 2\}$
$\{\underline{1}, \underline{2}\}$	D	$\{1, 4\}$	$\{2, 3\}$
$\{3, 4\}$	E	$\{2, 3\}$	$\{1, 4\}$



直接用NFA识别语言算法如下，需要每次算所有当前可能状态执行动作 c 后的 ϵ 闭包。

Algorithm 3 用NFA识别语言

```
1:  $S = \epsilon\text{-closure}(s_0)$ 
2:  $c = \text{nextChar}()$ 
3: while  $c \neq \text{eof}$  do
4:    $S = \epsilon\text{-closure}(\text{move}(S, c))$ 
5:    $c = \text{nextChar}()$ 
6: if  $S \cap F \neq \emptyset$  then
7:   return “yes”
8: else
9:   return “no”
```

定理 1. *DFA*, *NFA*和正则表达式三者的描述能力是一样的。

但从NFA转为DFA可能导致状态数的指数增长。

例 6. $L_n = (a \mid b)^* a (a \mid b)^{n-1}$, 与此*NFA*等价的*DFA*状态数必不少于 2^n 。

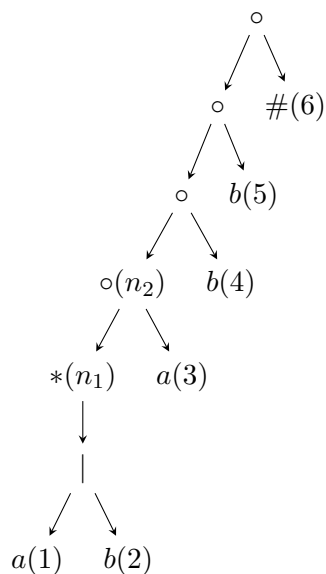
分析. 反证法。假设存在一个*DFA* D 接受语言 L_n , 且状态数少于 2^n 。构造 2^n 个长度为 n 的字符串

$aa \cdots a$
 $aa \cdots 1$
 \cdots
 $bb \cdots a$
 $bb \cdots b$

由于 D 的状态数少于 2^n , 故上面必存在两个不同的字符串 s 和 t , 它们在*DFA*上会走到同一状态。因为 s 和 t 不等, 因此总存在 i , 使得 $s[i] \neq t[i]$ 。不妨设 $s[i] = 0$, $t[i] = 1$, 令 $s' = s + (n-1)$ 个 a , $t' = t + (n-1)$ 个 a 。由 L_n 的表达式, s' 应该走到接受状态, 而 t' 应该走到非接受状态。但由于 s 和 t 走到同一状态, 那么它们再走 $(n-1)$ 个 a 也应该到达同一状态, 但这个状态既是接受状态又是非接受状态, 因此矛盾。

2.4 Regex转DFA

构造正则表达式的语法树, 以#结尾



*为star, |为or, o为cat

- $nullable(n)$: ϵ 包含在子树中则为真
- $firstpos(n)$: 符合regex子树的字符串中第一个字符可能出现的位置
- $lastpos(n)$: 符合要求字符串最后一个字符可能出现的位置
- $followpos(p)$: 紧跟 p 可能的位置

例 7. 结点 n_1 代表 $(a|b)^*$, 结点 n_2 代表 $(a|b)^*a$

- $nullable(n_1) = true$
- $firstpos(n_2) = \{1, 2, 3\}$
- $lastpos(n_2) = \{3\}$
- $followpos(1) = \{1, 2, 3\}$

计算 $followup$ 的两条法则:

- 若 n 为cat结点, 则对于左子树 c_1 的所有 $i \in lastpos(c_1)$, 有右子树 $firstpos(c_2) \in followpos(i)$
- 若 n 为star结点, 则对于所有 $i \in lastpos(n)$, 有 $firstpos(n) \in followpos(i)$

构建 $followpos$ 的过程实际上是深搜的过程, 可构造出一个有向图表示状态迁移。

Algorithm 4 Regex转DFA

Require: 正则表达式 r

Ensure: DFA D 可识别 $L(r)$

- 1: 语法树 T 的根节点为 n_0 , $firstpos(n_0)$ 是 $Dstates$ 的唯一状态, 且未被标记(unmarked)
 - 2: **while** 在 $Dstates$ 中还有未被标记的状态 S **do**
 - 3: 标记 S
 - 4: **for** 每一个输入符号 a **do**
 - 5: $U = \bigcup_{\text{对应}a\text{的位置}p \in S} followpos(p)$
 - 6: **if** $U \notin Dstates$ **then**
 - 7: 将 U 作为未标记的状态加入 $Dstates$
 - 8: $Dtran[S, a] = U$
-

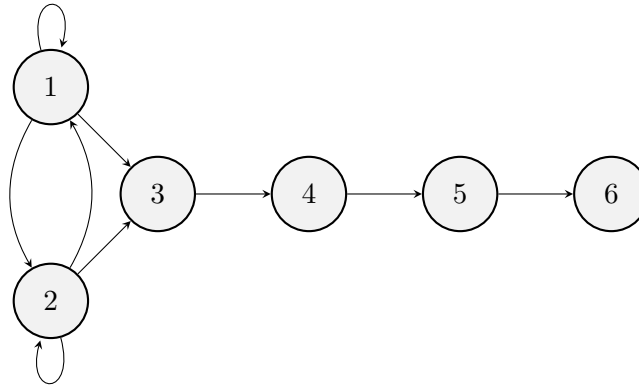
例 8. 考虑正则表达式 $(a | b)^* a b b \#$

1 2 3 4 5 6

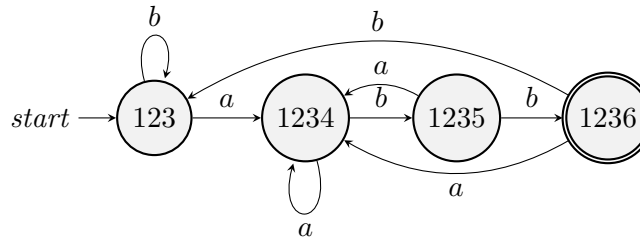
分析. 由表达式其实可以直接得到 $followpos$ 函数

<i>position</i>	<i>followpos(i)</i>
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	\emptyset

进而构造出一个有向图



然后可得DFA



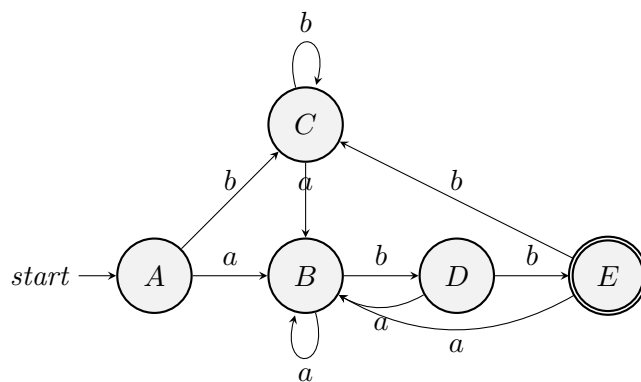
2.5 最小化DFA

定义 7 (区别(distinguish)). 字符串 w 区别状态 s 和 t , 如果DFA M 从状态 s 出发, 对输入串 w 进行状态转换, 最后停在某个接受状态; 从 t 出发, 对输入串 w 进行状态转换, 停在一个非接受状态; 反之亦然。

定理 2. 每一个正则集都可以唯一由一个状态数最少的DFA识别。

分析. 反证法。设算法得到的DFA为 D , 假设存在另一个DFA D' , D' 和 D 接受同一语言, 并且 D' 的状态数比 D 更少。设 D 的起始状态为 S , D' 的起始状态为 S' , 则 S 与 S' 不可区分。如果对于 S 和输入符号 a , 在 D 中迁移到状态 A ; 对于 S' 和输入符号 a , 在 D' 中迁移到状态 A' , 则 A 与 A' 不可区分。依此类推可知对于 D 中的任一状态 T , 在 D' 中都有一个状态 T' 与 T 不可区分。又由于 D 的状态数多于 D' 的状态数, 所以 D 中至少存在两个状态 T_1 和 T_2 , 使得 D' 中的一个状态 T 与它们均不可区分。因此 T_1 和 T_2 也不可区分, 于是矛盾。

例 9. 如下状态转移图



分析. 初始划分 Π 包括两个组: 接受状态组(E)和非接受状态组($ABCD$)。构造 Π_{new} , 先考虑(E), 仅一个状态, 不可划分, 仍将(E)放回 Π_{new} 。然后考虑($ABCD$), 对于输入 a , 这些状态都转换到 B , 分组($ABCD$)不变; 但对于输入 b , A 、 B 和 C 都转换到状态组($ABCD$)的一个成员, 而 D 转换到另一组成员 E 。因此, 在 Π_{new} 中, 状态组($ABCD$)需要分裂为两个新组(ABC)和 D , $\Pi_{new} = (ABC)(D)(E)$ 。继续执行下一轮操作, 最终得到 $\Pi_{final} = (AC)(B)(D)(E)$ 。因此选择 A 作为(AC)的代表, 其他不变, 可得到简化的自动机。

	a	b
A	B	A
B	B	D
D	B	E
E	B	A