编译原理笔记

陈鸿峥

2020.05*

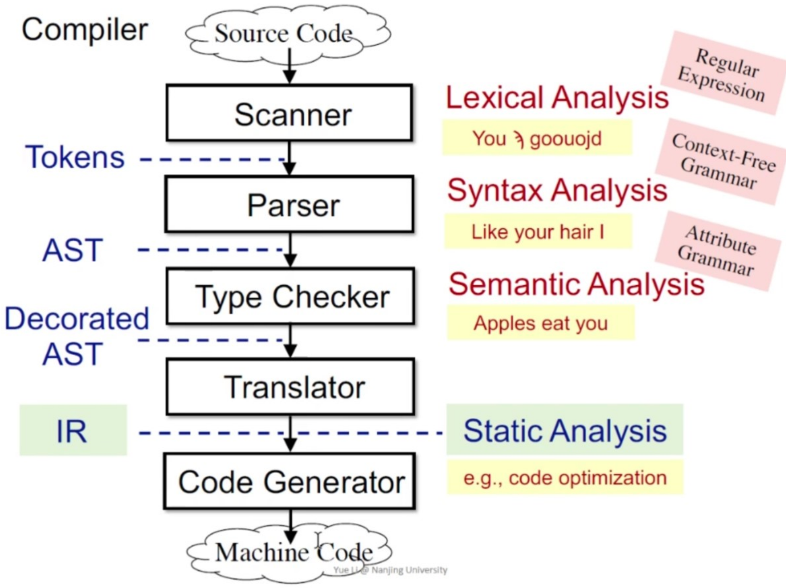
目录

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 20200517



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。字串(substring)可通过删除任意前缀和任意后缀（包括零个）获得。真(proper)字串则不包含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$L(\mathbf{a}) = \{a\}$（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)**.** *$d_i \to r_i$，其中d_i都是名字，且各不相同。每个r_i是$\Sigma \cup \{d_1, \ldots, d_{i-1}\}$中符号上的正则表达式。*

例 3. *比如C语言的标识符可记为*

$$\begin{aligned} letter\_ &\to A|B|\cdots|Z|a|b|\cdots|z|\_ \\ digit &\to 0|1|\cdots|9 \\ id &\to letter\_(letter\_|digit)^* \end{aligned}$$

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

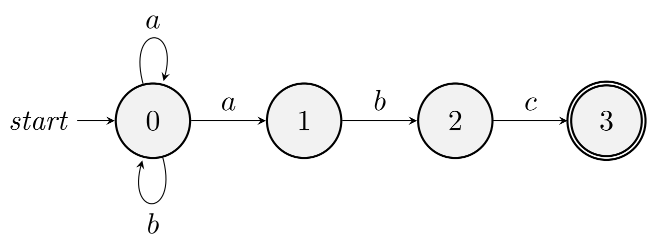
[1]更多可参见Regex101

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

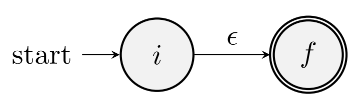
$s = s_0$
$c = nextChar()$
while $(c$!=**eof**$)$ **do**
  $s = move(s, c)$
  $c = nextChar()$
if $s \in F$ **then**
  return “yes”
else **return** “no”

时间复杂度为$O(|str|)$。

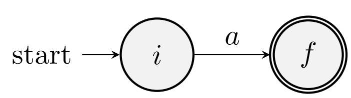
2.3.2 正则表达式转NFA

1. 奠基

   - 对于表达式ϵ，构建NFA

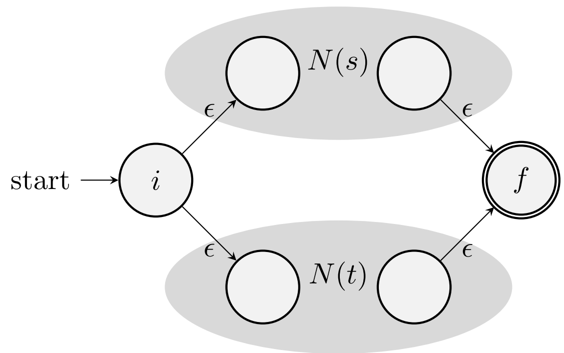


   - 对于任意子表达式$a \in \Sigma$，构建NFA

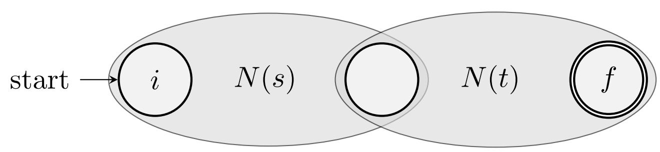


2. 推论

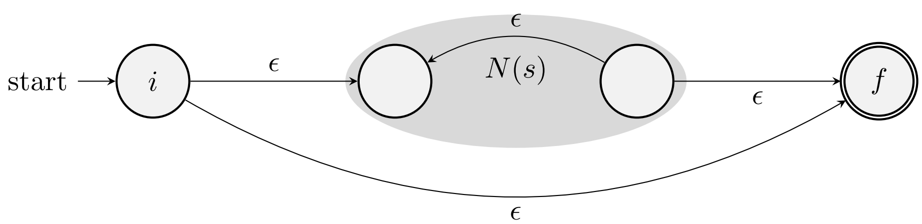
- $r = s|t$，取并集



- $r = st$，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。

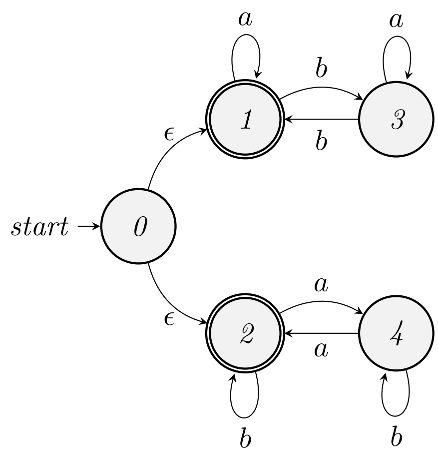
Algorithm 2 子集构造（NFA转DFA）

Require: NFA N

Ensure: DFA D（与N接受相同的语言）

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

例 5. 考虑以下NFA:

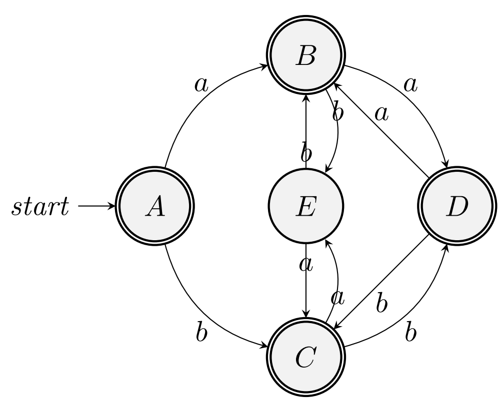


1. 这一NFA接受什么语言（用自然语言描述）？
2. 构造接受同一语言的DFA.

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	$\{3,4\}$	$\{1,2\}$
$\{\underline{1,2}\}$	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

```
S = ϵ-closure(s_0)
c = nextChar()
while c!=eof do
    S = ϵ-closure(move(S, c))
    c = nextChar()
if S ∩ F!=∅ then
    return "yes"
else
    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

$aa \cdots a$

$aa \cdots 1$

$\cdots$

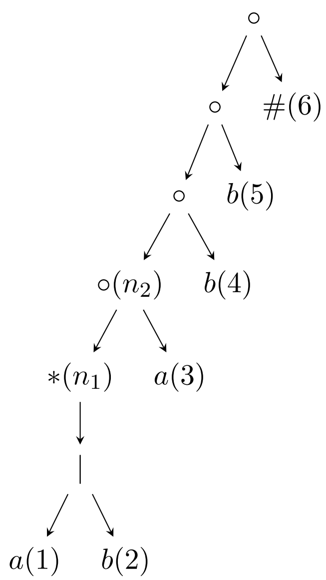
$bb \cdots a$

$bb \cdots b$

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，$\circ$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$L(r)$

```
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)
while 在Dstates中还有未被标记的状态S do
    标记S
    for 每一个输入符号a do
        U = ⋃_{对应a的位置p∈S} followpos(p)
        if U ∉ Dstates then
            将U作为未标记的状态加入Dstates
        Dtran[S, a] = U
```

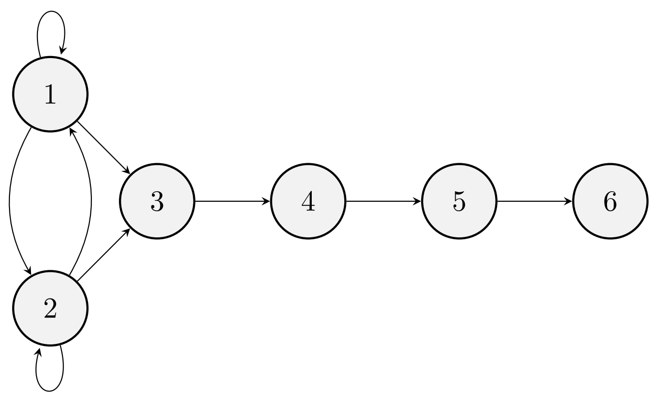
例 8. 考虑正则表达式

(	a	|	b	$)^*$	a	b	b	#
	1		*2*		*3*	*4*	*5*	*6*

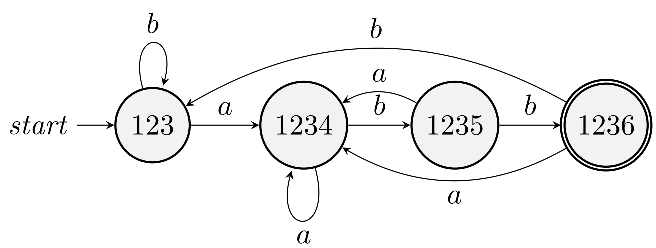
分析. 由表达式其实可以直接得到$followpos$函数

$position$	$followpos(i)$
1	$\{1,2,3\}$
2	$\{1,2,3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	$\varnothing$

进而构造出一个有向图



然后可得DFA



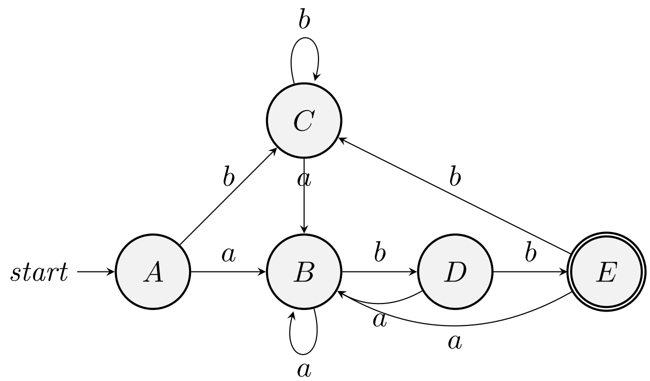
2.5 最小化DFA

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

3 语法分析

3.1 上下文无关法

语法分析需要解决：从词法分析中获得的每个属性字(token)在语句中承担什么角色，同时检查语句是否符合程序语言的语法。

定义 8 (上下文无关法(context-free gramma, CFG)). 包括四部分

- 终端符号*(terminal)*的集合T
- 非终端符号的集合N
- 唯一的开始符号$S \in N$
- 若干以下形式的产生式*(production)*

$$X \to Y_1 Y_2 \dots Y_n$$

其中$X \in N$且$Y_i \in T \cup N \cup \{\epsilon\}$。多个左侧相同的产生式右侧可用|合并。

定义 9 (推导(derivation)). 从开始符号开始，每一步推导就是用一个产生式的右方取代左端的非终端符号。

CFG定义语言的能力比正则表达式强很大原因是它引入了**递归**的因素。

例 10. 用上下文无关文法定义下列语言：

- $L = \{0^n1^n \mid n \geq 1\}$：$E \to 0E1 \mid 01$
- 只含有0和1的回文串：$S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \epsilon$
- 只含有(和)的匹配括号串：$E \to (E) \mid EE \mid \epsilon$
- 最左推导：每步推导都替换最左侧的非终端符号

$$E \overset{lm}{\Longrightarrow} -E \overset{lm}{\Longrightarrow} -(E) \overset{lm}{\Longrightarrow} -(E+E) \overset{lm}{\Longrightarrow} -(id+E) \overset{lm}{\Longrightarrow} -(id+id)$$

- 最右推导：每步推导都替换最右侧的非终端符号

$$E \implies -E \implies -(E) \implies -(E+E) \implies -(E+id) \implies -(id+id)$$

定义 10 (二义性)**.** 如果对于一个文法，存在一个句子，对这个句子可以构造两棵不同的分析树，那么我们称这个文法为二义的。

看语法分析树的叶子结点能不能连成句子。

例 11. 对于文法$E \to E+E \mid E*E \mid -E \mid (E) \mid id$及句子$id+id*id$，有以下两种推导：

$$\begin{aligned} E &\implies E+E \\ &\implies id+E \\ &\implies id+E*E \\ &\implies id+id*E \\ &\implies id+id*id \end{aligned} \qquad\qquad \begin{aligned} E &\implies E+E \\ &\implies id+E \\ &\implies id+E*E \\ &\implies id+id*E \\ &\implies id+id*id \end{aligned}$$

文法二义性的消除可通过引入更多的产生式。

例 12. $E \to E+E \mid E*E \mid (E) \mid id$是有二义的，因为不知道应该先算加法还是乘法。可将其改为

$$\begin{aligned} &E \to E+T \mid T \\ &T \to T*F \mid F \\ &F \to (E) \mid id \end{aligned}$$

其中E为$Expression$，T为$Term$，F为$Facotr$，即可消除二义性（必然得先算乘法）。

并不是所有上下文无关文法都可以做到无二义，也无法判断一个上下文无关文法是否是二义的。

3.2 NFA转CFG

1. 对于NFA的每一状态i，创建非终态A_i
2. 若状态i在输入a上有转换边到状态j，则添加生成式$A_i \to aA_j$；若状态i在输入ϵ上转换到状态j，则添加生成时$A_i \to A_j$

3. 若i是接受状态，则添加$A_i \to \epsilon$
4. 若i是初始状态，则令A_i为语法的初始符号

定义 11 (右线性文法). 如果每个产生式都属于下列形式之一

$$A \to aB \qquad A \to a \qquad A \to \epsilon$$

则这样的文法称为右线性文法

定义 12 (左线性文法). 如果每个产生式都属于下列形式之一

$$A \to Ba \qquad A \to a \qquad A \to \epsilon$$

则这样的文法称为左线性文法

在处理程序时，上下文无法文法存在局限性，无法解决诸如以下问题：

- 变量先声明，再使用
- 调用函数时，实参个数和形参个数一致

都得留到语义分析阶段才解决。