编译原理

Compilers

极夜酱

目录

Chapter 1 有限状态自动机

1.1　字母表

1.1.1　字母表（Alphabet）

字母表是一个非空的有限集合，一般用 Σ 表示，集合中的元素被称为符号/字符 (symbol)。

例如：

- $\Sigma = \{0, 1\}$：二进制数集合。
- $\Sigma = \{a, b, \cdots, z\}$：小写字母集合。
- $\Sigma = \{(,), [,], \{,\}\}$：括号集合。

1.1.2　串（String）

串是一个由字母表中的字符组成的有限序列。

例如：

- 0011 和 11 是 $\Sigma = \{0, 1\}$ 上的串。
- abc 和 bbb 是 $\Sigma = \{a, b, \cdots, z\}$ 上的串。
- (()) 和 (() 是 $\Sigma = \{(,), [,], \{,\}\}$ 上的串。

空串

空串使用 ϵ 表示。

串的长度

- $|0010| = 4$
- $|aa| = 2$
- $|\epsilon| = 0$

前缀（prefix）

- aa 是 aaabc 的前缀
- aaab 是 aaabc 的前缀
- aaabc 是 aaabc 的前缀

后缀（suffix）

- bc 是 aaabc 的后缀
- abc 是 aaabc 的后缀
- aaabc 是 aaabc 的后缀

子串（substring）

- ab 是 aaabc 的子串
- aaa 是 aaabc 的子串
- aaabc 是 aaabc 的子串

连接（concatenation）

当 $\omega = abd$，$\alpha = ce$，那么 $\omega\alpha = abdce$。

指数（exponentiation）

当 $\omega = abd$，那么 $\omega^3 = abdabdabd$，$\omega^0 = \epsilon$。

反转（reversal）

当 $\omega = abd$，那么 $\omega^R = dba$。

1.1.3 克林闭包（Kleene Closure）

Σ^k 用于表示所有在字母表 Σ 上的长度为 k 的串的集合。

例如，$\Sigma = \{a, b\}$，那么 $\Sigma^2 = \{ab, ba, aa, bb\}$，$\Sigma^0 = \{\epsilon\}$。

克林闭包 Σ^* 用于表示所有在字母表 Σ 上能够组成的串的集合。

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots = \bigcup_{k \geq 0} \Sigma^k \tag{1.1}$$

正闭包 Σ^+ 则是在 Σ^* 中除了空串以外的所有串的集合。

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots = \bigcup_{k > 0} \Sigma^k \tag{1.2}$$

1.2 语言

1.2.1 语言（Language）

语言是一个字母表中所构成串的集合。

例如，$\Sigma = \{a, b, c, \cdots, z\}$，那么所有英语单词所构成的集合 L 就是字母表 Σ 上的语言。

假设 $A = \{good, bad\}$ 和 $B = \{boy, girl\}$ 是两个语言，语言之间可以进行以下操作。

并集（union）

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\} \tag{1.3}$$

$A \cup B = \{good, bad, boy, girl\}$

连接（concatenation）

$$A \circ B = \{xy \mid x \in A \text{ or } y \in B\} \tag{1.4}$$

$A \circ B = \{goodboy, goodgirl, badboy, badgirl\}$

闭包

$$A^* = \{x_1, x_2, \cdots, x_k \mid k \geq 0 \text{ and each } x_i \in A\} \tag{1.5}$$

$A^* = \{\epsilon, good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, goodgoodbad, \cdots\}$

语法和语言与自动机理论密切相关，它们是许多软件实现的基础，例如编译器/解释器、文本编辑器、文本搜索、系统验证等。

在自动机理论中，要处理的问题就是判断一个给定的串是否属于某个语言。

例如：

- 0^*10^*：只包含一个 1 的串的集合。
- $\Sigma^*1\Sigma^*$：至少有一个 1 的串的集合。
- $\Sigma^*001\Sigma^*$：包含子串 001 的串的集合。
- $(\Sigma\Sigma)^*$：长度为偶数的串的集合。
- $(\Sigma\Sigma\Sigma)^*$：长度为 3 的倍数的串的集合。

1.3 DFA

1.3.1 DFA（Deterministic Finite Automaton）

有限状态机（FSM, Finite State Machine）用于决定程序当前状态和状态间的切换，状态机最终只能指向一个结果。

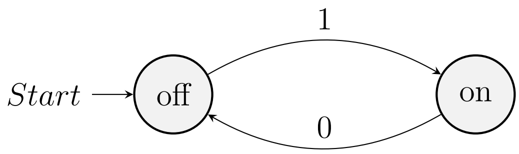


图 1.1: 有限状态机

确定性有限状态自动机 DFA 使用一个五元组 $(Q, \Sigma, \delta, q_0, F)$ 表示，其中

- Q：状态的集合
- Σ：字母表
- δ：状态转移函数（transition function）
- q_0：初始状态
- F：终结状态集合

例如 DFA 可以用来识别空串或者以 0 结尾的串：

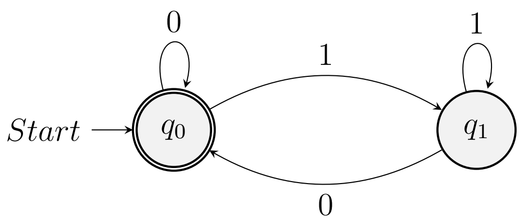


图 1.2: 识别空串或以 0 结尾的串的 DFA

其中 $Q = \{q_0, q_1\}$，$\Sigma = \{0, 1\}$，q_0 为初始状态，$F = \{q_0\}$，δ 为

状态	输入	
	0	1
q_0	q_0	q_1
q_1	q_0	q_1

能够被有限自动机接受的语言被称为正则语言（regular language）。

例如，构建一个能够识别所有包含子串 001 的串的 DFA：

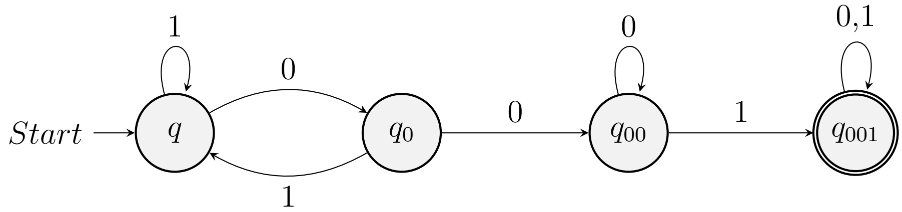
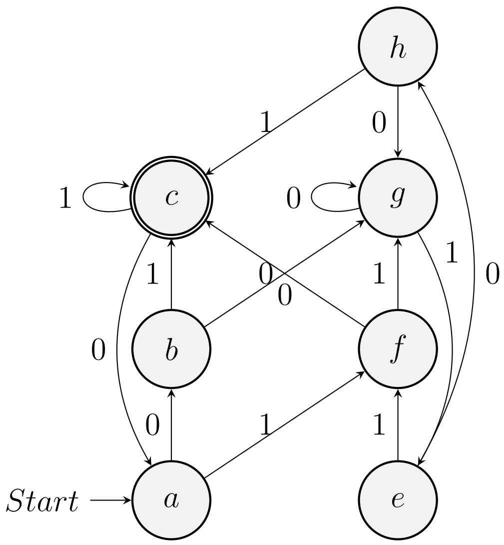


图 1.3: 识别包含子串 001 的串的 DFA

1.3.2 最小化 DFA

有限状态机的最小化，即将一个有限状态机转换为一个更小的有限状态机，使得状态的数目最少。

对于两个状态，如果它们之间的转移函数相同，则这两个状态可以合并为一个状态。



在这个 DFA 中，状态 b 和 h 是等价的，当接收 0 时都转移到状态 g，当接收 1 时都转移到状态 c。同时状态 a 和 e 也是等价的，状态 a 接收 0 转移到状态 b，状态 e 接收 0 转移到状态 h，状态 a 和 e 接收 1 时都转移到状态 f。

因此，状态 b 和 h 以及状态 a 和 e 可以进行合并。

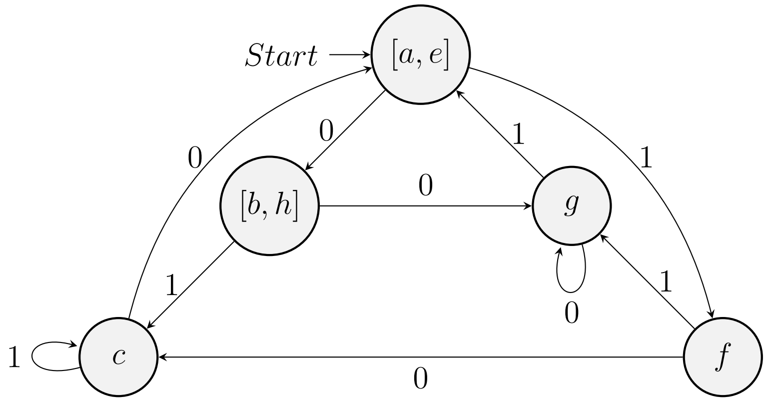


图 1.4: 最小化 DFA

1.4 NFA

1.4.1 NFA（Non-deterministic Finite Automaton）

在 DFA 中，每个状态的下一个状态都是唯一确定的，但是非确定性有限状态自动机 NFA 可能会存在多个下一状态。

例如在这个 NFA 中，状态 q_0 存在两个接收 1 的箭头，而状态 q_1 没有接收 1 的箭头。

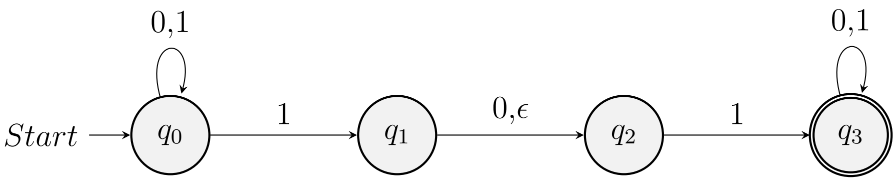


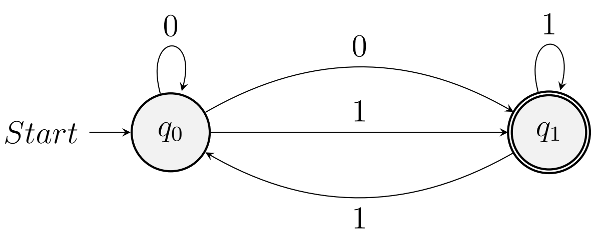
图 1.5: NFA

因此，在 NFA 中，每个状态允许对相同输入存在 0 个、1 个或多个转移的状态。如果存在一条能够到达终结状态的路径，那么就称当前的输入是被 NFA 接受的。

1.4.2 DFA 与 NFA 的转换

NFA 并不比 DFA 更加强大，理论证明 NFA 与 DFA 是等价的。

例如将一个 NFA 转换为 DFA：



构建一个与 NFA 等价的 DFA，只需将 NFA 中转换到的状态集合作为 DFA 中的一个状态即可。

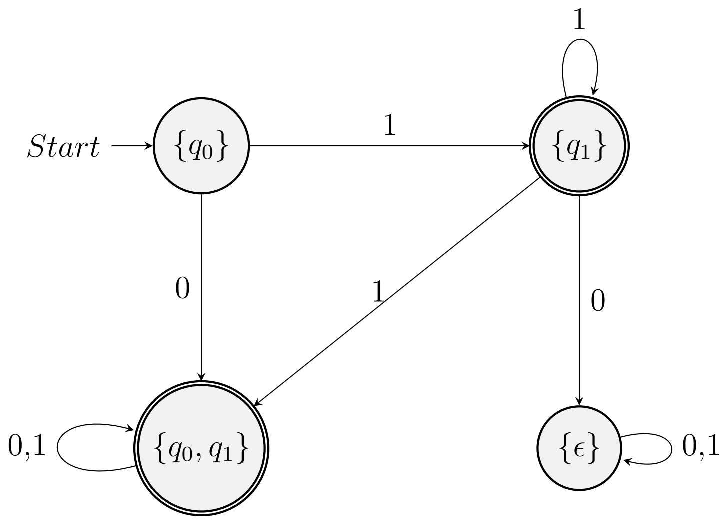


图 1.6: NFA 转换 DFA

1.4.3 ϵ-NFA

ϵ-NFA 允许不消耗输入字符在状态之间转移。

例如以下 ϵ-NFA 能够接受小数，如 +3.14、-0.12、.71、2. 等。

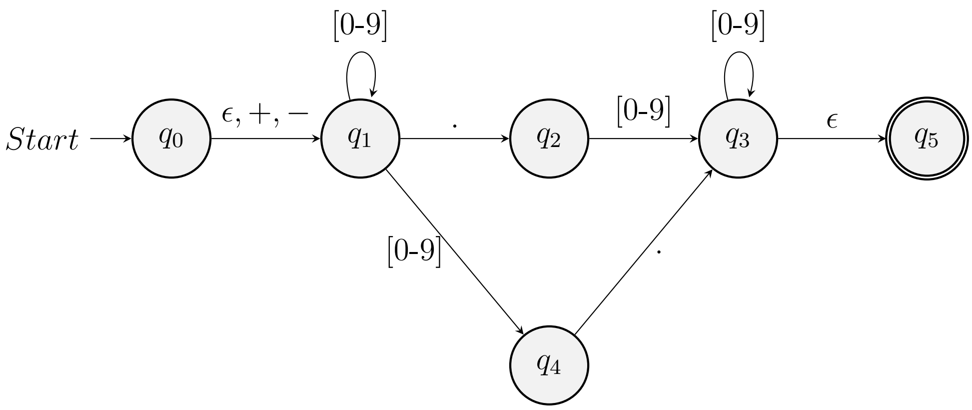


图 1.7: 接受小数的 ϵ-NFA

1.5 正则表达式

1.5.1 编译器（Compiler）

编译器是一种特殊的程序，可以将一种编程语言的源代码翻译成机器码、字节码或另一种编程语言。

编译器包含以下阶段：

1. 词法分析器（lexical analyzer）
2. 语法分析器（syntex analyzer）
3. 语义分析器（semantic analyzer）
4. 中间代码生成器（intermediate code generator）
5. 代码优化器（code optimizer）
6. 代码生成器（code generator）

1.5.2 词法分析

词法分析是编译器的第一步，它的主要任务是读取源代码，并生成能够被解析器（parser）进行语法分析的 tokens 和语法书（syntex tree）。

例如 time = hour * 60 + minute，经过词法分析后，将会得到：

- id(time)
- assignment(=)
- id(hour)
- op(*)
- num(60)
- op(+)

- id(minute)

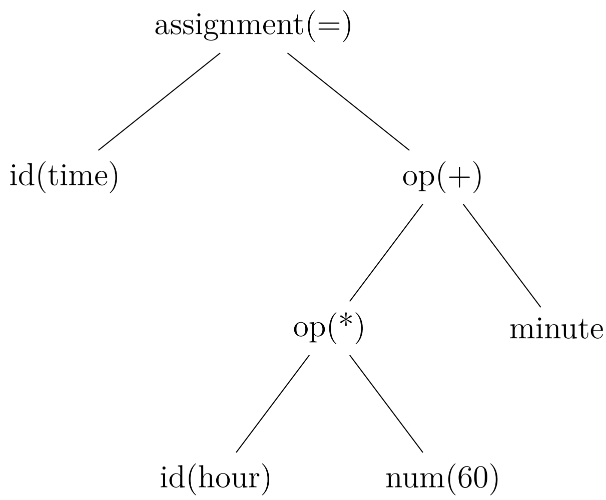


图 1.8: 语法树

1.5.3 正则表达式（Regex, Regular Expression）

正则表达式描述了字符串匹配的模式（pattern），可以用来检查一个串是否包含某个子串、替换子串、或提取符合条件的子串。像 grep、vi、python、lex 等工具都支持正则表达式的使用。

例如用于匹配一个合法的变量名的正则表达式为 [a-zA-Z_][a-zA-Z0-9_]*。即变量名只能由字母或下划线开头，后面可以是任意多个字母、数字或下划线。

正则表达式支持以下操作：

- 连接：ab 或 $a \cdot b$
- 选择：$a \mid b$
- 克林闭包：$a^* = \{\epsilon, a, aa, aaa, \cdots\}$
- 匹配至少 1 次：$a^+ = aa^*$
- 匹配 0 次或 1 次：$a? = a \mid \epsilon$
- 匹配任意字符：.

- 补集： $(a \mid b)$

其中克林闭包运算的优先级最高，其次是连接，最后是选择。

例如 $(a \mid b)^*aa(a \mid b)^*$ 用于匹配包含连续的 a 的串，$b^*(abb^*)^*(a \mid \epsilon)$ 用于匹配没有连续的 a 的串。

然而 $\{a^nb^n \mid n > 0\}$ 却不是正则语言，因为它无法用有限个状态来验证 a 和 b 的出现次数是相等的。

Chapter 2 上下文无关语言

2.1 上下文无关文法

2.1.1 上下文无关文法（CFG, Context Free Grammar）

CFG 能够描述某些具有递归结构的特征，它有足够强的语言表达力来表示大多数编程语言的语法。

CFG 由一个四元组 (V, T, P, S) 表示：

- V：变元（variable）/非终结符（non-terminal）集合，用大写字母表示。
- T：终结符（terminal）集合，用小写字母表示。
- P：产生式（production）集合。
- S：开始符号。

一个文法由一组替换规则产生。产生式集合

$$
\begin{gathered}
A \to \alpha_1 \\
A \to \alpha_2 \\
\cdots \\
A \to \alpha_k
\end{gathered}
$$

可以被写成 $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$ 的形式。

2.1.2 推导（Derivation）

推导用于确定符合文法规则的串的集合，即用来确定一个语言。

推导从开始符号开始，通过产生式进行替换，得到最终结果。

例如 $E \to E + E \mid E * E \mid (E) \mid id$，由开始符号 E 可以推导出 $(id + id) * id$。

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow (E) * E \\
&\Rightarrow (E) * id \\
&\Rightarrow (E + E) * id \\
&\Rightarrow (E + id) * id \\
&\Rightarrow (id + id) * id
\end{aligned}
$$

解析树（parse tree）是描述推导的一种直观方法。

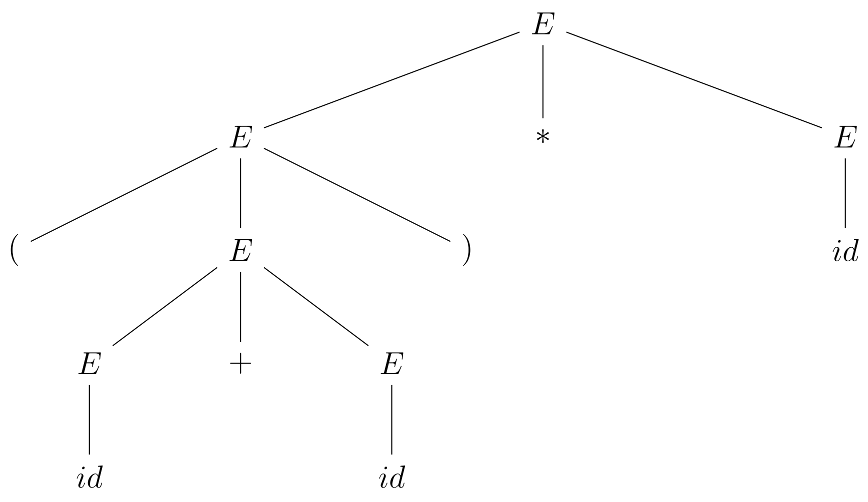


图 2.1: 分析树

如果只关注语义分析和代码生成所需的信息，可以将分析树简化为一棵抽象语法树（abstract syntex tree）。

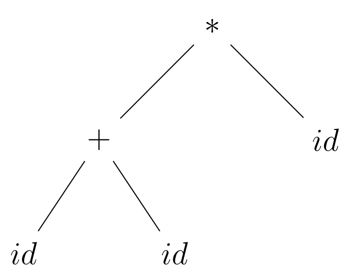


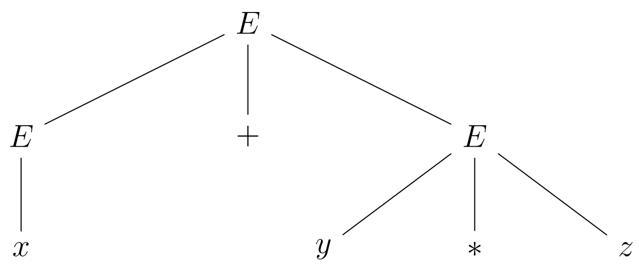
图 2.2: 抽象语法树

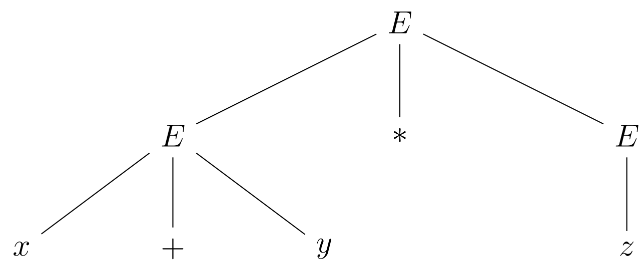
2.1.3 二义性（Ambiguity）

在推导的过程中涉及到同级别表达式的替换,因此按顺序可以分为最左推导(leftmost derivation）和最右推导（rightmost derivation)。

文法的二义性，是指对于符合文法规则的同一个句子，存在两种可能的分析树。

例如 $E \to E + E \mid E * E \mid (E) \mid x \mid y \mid z$，使用最左推导会对 $x + y * z$ 产生两个不同的分析树。



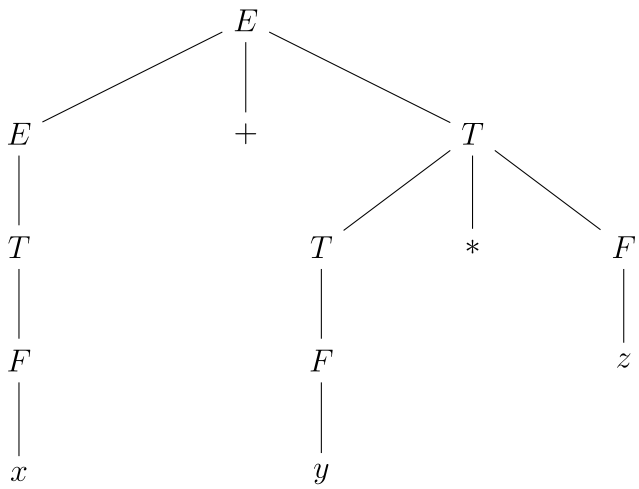


产生二义性的原因在于运算符之间的优先级在文法中并没有体现。消除二义性的办法就是在文法中引入一个中间量。

$$E \to E + T \mid T$$

$$T \to T * F \mid F$$

$$F \to id \mid (E)$$


E
E
+
T
T
T
*
F
F
F
z
x
y

2.2 CNF

2.2.1 上下文无关语言

CFG 可以用来表示语言 $\{a^n b^n \mid n > 0\}$:

$$
\begin{aligned}
S &\to aSb \\
S &\to ab
\end{aligned}
$$

例如根据 S 可以生成生成 aaabbb:

$$
\begin{aligned}
S &\Rightarrow aSb \\
&\Rightarrow aaSbb \\
&\Rightarrow aaabbb
\end{aligned}
$$

CFG 好还可以用于表示 a 和 b 出现相等次数的语言，例如 babaab:

$$
\begin{aligned}
S &\to aB \mid bA \\
A &\to a \mid aS \mid bAA \\
B &\to b \mid bS \mid aBB
\end{aligned}
$$

设计 CFG 需要一定的创造力，大部分复杂的 CFG 可以由多个简单的 CFG 并集组成。

例如设计一个能够表示语言 $\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$ 的 CFG。

这两个部分可以分别表示为:

$$
\begin{aligned}
S_1 &\to 0S_1 1 \mid \epsilon \\
S_2 &\to 1S_2 0 \mid \epsilon
\end{aligned}
$$

只需合并这两个部分，即可得到最终的 CFG:

$$S \to S_1 \mid S_2$$
$$S_1 \to 0S_11 \mid \epsilon$$
$$S_2 \to 1S_20 \mid \epsilon$$

2.2.2 乔姆斯基范式（CNF, Chomsky Normal Form）

CFG 可以被转换为一些更加简洁的形式，如 CNF。

CNF 规定每条 CFG 的每一条规则都必须满足：

1. $A \to BC$
2. $A \to a$

其中