CHAPTER 2

在上一章末，想必你已体会到了表达式和机器就表达语言而言是同一的。由表达式出发来定义一门语言，与从等价机器出发来定义一门语言，效果别无二致。所以，这章我们索性把次序逆转，先趁热打铁地介绍一种新型表达式，再由它定义一门语言，接着再介绍其等价的机器模型。

我们要介绍的新型表达式叫做「上下文无关文法（CFG）」，很容易证明它~~[illegible]~~比正则表达式具有更强的表达能力。凡是能被其表达的语言，都叫「上下文无关语言（CFL）」。即便现在还没切入正题，但你也应清楚，和正则语言、正则表达式之间的关系类似，CFL是「绝对的」，而实现CFL的CFG集可以有成千上万种。

下面，先看一个引例。

e.g. 我们可以用~~[illegible]~~迭代的方法来生成任意 $s \in \{0^n1^n \mid n \in \mathbb{N}_0\}$。首先，写下初始字符A。然后，把A改写成0A1，并可以如此反复，比如继续写成00A11、000A111。直到某时刻我们想要停止迭代了，则把A换成空串ε。

简要地说，我们允许两种改写规则：$A \to 0A1$ 和 $A \to \varepsilon$。0、1、ε一旦写定，就再也无法改动。不难看出，这套规则能生成 $\{0^n1^n \mid n \in \mathbb{N}_0\}$ 中的任一字符串。

不难用上一章的pumping lemma说明：$\{0^n1^n \mid n \in \mathbb{N}_0\}$ 不是正则语言。所以，上面介绍的如此简单的迭代规则似乎有超越

正则表达式之处。不妨再看一个复杂些的例子：

e.g. 初始字符为A，允许五条~~覆~~改写规则：

① $A \rightarrow BC$
② $B \rightarrow 0B0$
③ $B \rightarrow C$
④ $C \rightarrow 1C$
⑤ $C \rightarrow \varepsilon$

$\boxed{\begin{array}{l}\text{也可缩写成} \\ A \rightarrow BC \\ B \rightarrow 0B0 \mid C \\ C \rightarrow 1C \mid \varepsilon\end{array}}$

它所能表达的是 $\{0^i 1^j 0^i 1^k \mid i \in \mathbb{N}, j,k \in \mathbb{N}_0\}$

比如 000110001可循 $A \overset{①}{\Rightarrow} BC \overset{②}{\Rightarrow} 0B0C \overset{②}{\Rightarrow} 00B00C \overset{②}{\Rightarrow} 000B000C \overset{③}{\Rightarrow} 000C000C \overset{④}{\Rightarrow} 0001C000C \overset{④}{\Rightarrow} 00011C000C \overset{⑤}{\Rightarrow} 00011000C \overset{④}{\Rightarrow} 000110001C \overset{⑤}{\Rightarrow} 000110001$ 得到。

我们把类似的迭代规则定义成「上下文无关文法」。

def 上下文无关文法（CFG）：
一个四元组 (V, Σ, R, S)，其中

1° V是一个有限集合。它指定了允许出现的变量（即能够被改写的符号）。

2° Σ是一个有限集合。它指定了中止符（即不能再被改写的符号）。

3° R是一个有限集合。它包含了所有改写规则。每条规则都~~[illegible]~~形如"$A \rightarrow w$"，其中 $A \in V$，$w \in (V \cup \Sigma_\varepsilon)^*$.

4° $S \in V$是初始变量.

譬如，在上面例子中，$V = \{A, B, C\}$，$\Sigma = \{0, 1\}$，$R = \{A \rightarrow BC, B \rightarrow 0B0, B \rightarrow C, C \rightarrow 1C, C \rightarrow \varepsilon\}$，$S = A$.

之所以称其为「上下文无关」，是因为~~[illegible]~~改写规则只看眼前，不看方位及上下文。

另外还请注意：CFG天生就有「不确定性」~~比如~~一个变量B既可以被改写为0B0，也可以被改写为C，二者均被允许。

def CFG ~~[illegible]~~ 生成字符串的流程：

设 $G = (V, \Sigma, R, S)$是CFG，w是字符串。如果存在一系列字符串 $u_1, u_2, \dots, u_m$ 使得 $S \overset{R}{\Rightarrow} u_1 \overset{R}{\Rightarrow} u_2 \overset{R}{\Rightarrow} \cdots \overset{R}{\Rightarrow} u_m \overset{R}{\Rightarrow} w$,

那么称其为 w 的生成流程，并说 w 可被 G 生成，记为 $S \overset{*}{\Rightarrow} w$.

(其中，"$\overset{R}{\Rightarrow}$"的含义是指：选取某一处变量和 R 中某一条规则，对该处变量进行改写).

def CFG生成的语言.

设 G 是 CFG. G 生成的语言为

$\mathcal{L}(G) := \{ w \mid w 可被 G 生成 \}$

def 上下文无关语言(CFL).

设 A 是一个集合。若存在某CFG G 使 $\mathcal{L}(G) = A$，则称 A 是上下文无关语言.

remark. 这些定义与Chapter1之中的极相类似，不再过多解释。

Theorem 1 正则语言类是上下文无关语言类的真子集。

proof. 因前面已举例说明二者不相等，故只须证明正则语言类是上下文无关语言类的子集即可。

想法是把任意正则表达式转置成CFG。对 $R_1 \cup R_2$，转成两条规则 $A \to A_1$ 与 $A \to A_2$；对 $R_1 \circ R_2$，转成规则 $A \to A_1 A_2$；对 R^*，转成规则 $A \to AA$。依此即可把长的正则表达式切分成小的。细节留作练习。■

Theorem 2 CFL类对 $\cup, \circ, *$ 运算封闭.

proof. 这与Theorem1是类似的。下面给出证明，视作Theorem1的解答。

$\mathcal{L}_1$ 与 $\mathcal{L}_2$ 为CFL.

设 $G_1 = (V_1, \Sigma_1, R_1, S_1)$ 与 $G_2 = (V_2, \Sigma_2, R_2, S_2)$ 为CFG且分别生成 $\mathcal{L}_1$ 与 $\mathcal{L}_2$.

构造 $G_3 = (V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_3, S_3)$

$G_4 = (V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_4, S_4)$

$G_5 = (V_1, \Sigma_1, R_5, S_1)$

$R_3 := R_1 \cup R_2 \cup \{ S_3 \to S_1, S_3 \to S_2 \}$

$R_4 := R_1 \cup R_2 \cup \{ S_4 \to S_1 S_2 \}$

$R_5 := R_1 \cup \{ S_1 \to S_1 S_1, S_1 \to \varepsilon \}$

则 $\mathcal{L}(G_3) = \mathcal{L}_1 \cup \mathcal{L}_2$，$\mathcal{L}(G_4) = \mathcal{L}_1 \circ \mathcal{L}_2$，

$\mathcal{L}(G_5) = \mathcal{L}_1^*$. ■

有趣的是，CFL类并不对∩与补操作封闭，比如 $\mathcal{L}_1=\{0^n1^n0^*\mid n\in\mathbb{N}\}$，$\mathcal{L}_2=\{0^*1^n0^n\mid n\in\mathbb{N}\}$，$\mathcal{L}_1\cap\mathcal{L}_2=\{0^n1^n0^n\mid n\in\mathbb{N}\}$，前两者是CFL，而后者不是。证明将放在以后。

有时候，化简CFG为特定形式（范式）是必要的。Chomsky范式即为一种相当精简的形式。

Theorem3 任何CFG G都可被化简成Chomsky范式 G'，即满足

1° $\mathcal{L}(G)=\mathcal{L}(G')$

2° G'中任何规则都只能形如"$A\to a$"或"$A\to BC$"，$A,B,C\in V'$，$a\in\Sigma'$

3° 除了初始变量S'以外，任何别的变量都不允许改写成ε或S'.

e.g.
$S\to AB$
$A\to 0A\mid\varepsilon$
$B\to B11$

可转为

$S\to AB\mid BC$
$A\to DA\mid 0$
$B\to BC$
$C\to EE$
$D\to 0$
$E\to 1$

proof. 先分析：如果我们有一条长规则如"$A\to BCDaEF\cdots$"，那么我们把它迂回地写成多条规则 $A\to BA_1$，$A_1\to CA_2$，$A_2\to Da$，… 即可，再对终结符作适当包装就大功告成了，这没什么困难的。

真正的难点在于"$A\to\varepsilon$"与"$A\to B$"一类的规则：它们不够长，不能用上面的「分步」法来修补，而应考虑其余方法。既是要延长之，那么不妨为其找些「宿主」，寄居其上，借宿主之力而延长自己。比如，有规则 $A\to ABC$，$B\to A\mid\varepsilon$，那么不妨让"$B\to\varepsilon$"寄生在"$A\to ABC$"上，使之变为 $A\to ABC\mid AC$，$B\to A$。

根据上述分析，我们依照下述路径来将G转成符合Chomsky范式的G'：

1° 把形如"$A\to\varepsilon$"的规则挪移到别处寄生.

2° 把形如"$A\to B$"的规则挪移到别处寄生.

3° 通过分步法迂回地把长规则拆解为短规则。

1° $\forall f\in R$，若 $f: A\to\varepsilon$，则寻找出所有右侧出现"A"的 $g\in R$．设 $g:\ B\to w_1 A w_2 A w_3 A\cdots w_n A w_{n+1}$ (w_i是串)

往R中添加 2^n-1 条规则，枚举出每一种A被ε代替的可能。比如，

$B\to w_1 w_2 A w_3 A\cdots w_n A w_{n+1}$ (删去首个)，

$B\to w_1 A w_2 w_3 w_4 A\cdots w_n A w_{n+1}$

(删去2、3个)

这些规则完美地模拟出 $A\to\varepsilon$ 的情形，相当于把 $f\ A\to\varepsilon$ 寄生在了 g 上．

当对所有g都做过一遍时，规则f已被架空了，名存实亡。在我们如此对所有形如 $A\to\varepsilon$ 的规则f都操作过后，它们全都丧失了意义，故一并移除之将毫无影响。

2° 类似地，$\forall f\in R$，若 $f: A\to B$，则寻找出所有右侧出现"A"的 $g\in R$，并添入用B代之的 2^n-1 条规则。最后，将这些 $f: A\to B$ 形状的规则一并移除。

3° 现在，规则集R中只剩长规则了。$\forall f\in R$，若 $f: A\to x_1 x_2\cdots x_n$ (x_i是字符，既可为变量也可为终结符) 且 $n>2$，则把该规则替换为如下一串规则：

	(若x_i是终结符)
$A\to A_1' A_2$	$A_1'\to x_1$
$A_2\to A_2' A_3$	$A_2'\to x_2$
$A_3\to A_3' A_4$	$A_3'\to x_3$
$\vdots$	$\vdots$
$A_{n-1}\to A_{n-1}' A_n'$	$A_{n-1}'\to x_{n-1}$
	$A_n'\to x_n$.

$A_i A_i'$ 均为人造的新变量

至此，我们已让R中所有规则都呈"$A\to BC$"或"$A\to a$"的格式。为了让右侧不出现初始变量，不妨投机取巧，引入新变量 S' 并添加新规则 $S'\to S$ 即可。■

remark. 为了形式上的简单，Chomsky范式将造成极端的迂回和冗余（见1°、2°、3°构造过程），进而阻碍理解。它的价值在于理论上的便利。

Lemma4 若CFG G符合Chomsky范式，则对任何 $w \in \mathcal{L}(G)$，G恰好用 $2|w|-1$ 步生成 w.（$w \neq \varepsilon$）.

proof. 因为$w \neq \varepsilon$，故生成w至少得花一步，且 $|w| \geq 1$。设 $w = x_1x_2\cdots x_n$，$|w| = n \geq 1$。因为产生终止符的唯一可能是"$A \to a$"规则，故生成w的过程中必定要把n个变量花n步改写成n个中止符。我们进一步问：如何能由初始变量产生出这n个变量？惟有通过"$A \to BC$"形式的分裂。而且，一旦产生，则不能再撤回（因为不允许"$A \to \varepsilon$"）。于是，只能用恰好$(n-1)$步去产生n个变量。综合而言，由初始变量生成w需要恰好 $2n-1 = 2|w|-1$ 步。■

remark. 该引理为我们提供了一个步数界，用它可以说明：任给一个CFG G（未必是Chomsky范式），以及一个字符串w，我们总能用算法去判断 $w \in \mathcal{L}(G)$。

Lemma5（*Pumping Lemma*）

若$\mathcal{L}$是CFL，则$\exists p>0$，$\forall s \in \mathcal{L}$且$|s| \geq p$，存在一种把$s$切成五段的分割 $s = uvxyz$ 满足

1° $|vy| > 0$ （也就是v与y不能都为空）

2° $|vxy| \leq p$

3° $uv^ixy^iz \in \mathcal{L}$，$\forall i \in \mathbb{N}$。

有了正则语言*Pumping Lemma*的经验，这一引理的内涵应当不难理解。而且，显然要用鸽笼原理来证明它。问题是：怎样选择鸽子？

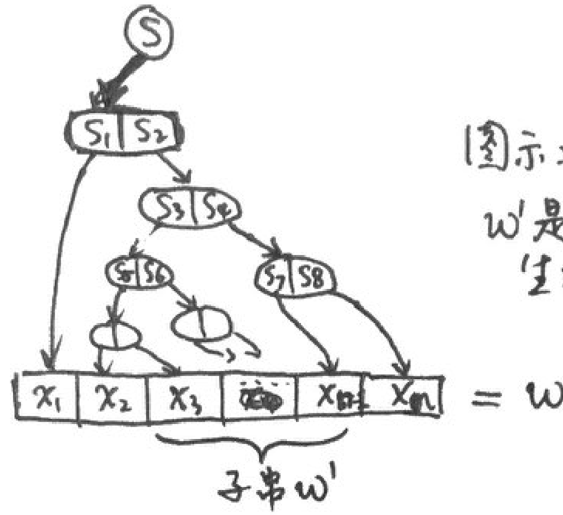
回想之前，我们利用DFA中重复遇到的状态来把字符串$s = xyz$的y部分任意延长。能这么做，是因为DFA的转移路径仅取决于当前状态和输入字符，只要我们复刻出完全一致的状态和输入字符，那么 复刻 DFA就会原样重复先前的行为。

在这里也是一样的。我们已知道，所谓「上下文无关」就是说CFG的生成字串路径仅取决于有什么变量、采取何种规则改写变量，那么，如果我们复刻出完全一致的变量和采用的规则序列，CFG自然也会生成同样的串。

所以，我们希望选的鸽子是「变量」。

proof. 设 $G=(V,\Sigma,R,S)$ 是CFG，$\mathcal{L}(G)=\mathcal{L}$ 且 G 符合Chomsky范式。由Lemma 4知，$\forall w\in\mathcal{L}$ $(w\neq\varepsilon)$ 且 $|w| = \frac{|V|}{2}+1$，G 恰好用 $2|w|-1 = |V|+1$ 步生成 w。

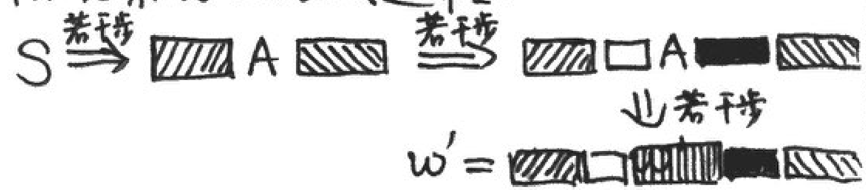
这样，我们不妨就取 $p := \frac{|V|}{2}+1$，那么，凡是 $|w|\geq p$ 的字符串 w，必有长为 p 的子串 w'，生成 w' 时必然经历 $|V|+1$ 步。可是我们一共只有 $|V|$ 个变量，所以生成 w' 过程中必定会遇到某变量A两次或以上。



图示：w的生成树。w'是由一部分子树生成的。

假设 $w = x_1x_2\cdots x_n$
$w' = x_ix_{i+1}\cdots x_{i+p-1}$

仅观察 w' 的生成过程：



既然A经若干步又得到A（以及此过程中附带生成的□与■），那么如果我们重复这中间的流程，则可以任意多地生成□与■。令 $v:=$□，$x:=$▥，$y:=$■，又令 w 中 v 左侧部分为 u，y 右侧部分为 z，则我们有

1° $|vy|>0$ （因为 $A \rightsquigarrow \square A$ 必须造成长度上升）

2° $|vxy| \le w'$

3° $uv^ixy^iz \in \mathcal{L}$. $\forall i \in \mathbb{N}_0$

现在我们可以来说明为什么$\{0^n1^n0^n \mid n \in \mathbb{N}_0\}=\mathcal{L}$不是CFL。假设它是，那么取$0^p1^p0^p \in \mathcal{L}$，据Pumping lemma，存在某种划分uvxyz满足引理条件。因$|vxy| \le p$，故vxy不能跨越在两个0之间。于是只有如下情形：

1° v和y都各自只含一种字符。那么uv^2xy^2z将不可能具有$0^n1^n0^n$的形式。

2° v和y中的某一个兼含0与1。不妨设v是如此。那么v的长像只能是

$$000\cdots\boxed{\cdot0111}\cdots1000\cdots0$$

或 $$000\cdots0111\cdot\boxed{\cdot\cdot1000}\cdots0$$

无论如何，它都无法包含全部的1。于是uv^2xy^2z必将造成01的乱序。↯

直接推论：CFL关于∩不封闭。又因$\mathcal{L}_1 \cap \mathcal{L}_2 = \overline{\bar{\mathcal{L}}_1 \cup \bar{\mathcal{L}}_2}$，故CFL也不能关于补封闭。

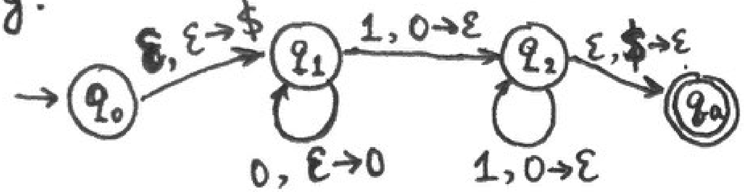
CFL奇异的性质引人深思：为何$\{0^n1^n \mid n \in \mathbb{N}_0\}$与$\{0^n1^n0^n \mid n \in \mathbb{N}_0\}$对于CFG来说有本质差别？CFG能力的瓶颈由什么所致？而它强于DFA/NFA/正则表达式的根本原因又在哪？还有，CFL关于∩不封闭有没有直观解释？上述问题，促使我们寻找CFG的等价模型——一种与DFA/NFA相仿的、直观的计算模型，以便我们窥其奥秘。

由于$\{0^n1^n \mid n \in \mathbb{N}_0\}$非正则，但却是CFL，因此我们猜想模型中需要提供不受限的存储容量。又由于$\{0^n1^n0^n \mid n \in \mathbb{N}_0\}$不是CFL，故存储容量又不是任人使用的，而具有一次性特征，用完一次

即丢失。分析至此，一种存储设备呼之欲出，那就是「栈」。

我们构想出如下计算模型：它是NFA的加强版，配备有一个无限容量的栈。状态转移不仅取决于当前状态以及当前输入字符，还取决于栈顶元素；状态转移时，栈顶元素会被弹出并读取，然后机器可依据定好的规则把新元素压入，或是什么也不压入。

e.g.



无论输入什么串，这台机器所做的第一步是读空字符~~[illegible]~~，弹出栈顶之空字符（即什么也不弹），并往栈内压入"$"字符。这一过程简写为 $\varepsilon, \varepsilon \to \$$.

然后，在状态q_1，若读入0，则往栈内压入0；若读入1且栈顶为0，则弹出栈顶元素并压入ε（即什么也不压）；若读入1且栈顶不为0，那么无法转移，「线程」结束。

在q_2，若读入0，则无法转移，线程结束；若读入1且栈顶为0，则弹出之；若读入1且栈顶非0，则结束；若遇栈顶为$，则弹出之并到达$q_a$。

在q_a，若仍有输入，则结束。

综上，这台机器可识别$\{0^n1^n \mid n\in\mathbb{N}\}$。

为将这样的模型规范化，我们作如下定义。

def 下推自动机（PDA）：

一个六元组$(Q, \Sigma, \Gamma, \delta, q_0, F)$，

1° Q是一个有限集合。它指定了该PDA所有可能的状态。

2° Σ是一个有限集合。它指定了该PDA所认识的字符集。

3° Γ是一个有限集合。它指定了该PDA在栈内允许存储的字符集。

4° $\delta: Q\times\Sigma_\varepsilon\times\Gamma_\varepsilon \to 2^{Q\times\Gamma_\varepsilon}$ 它指定了该PDA在各种状态下、读到各种字符、且栈顶元素为各种取值时，下一个状态可以是什么、栈顶元素被改写成什么。

例如 $\delta(q,a,x)\to\{(q',y),(q'',z)\}$

意为：处在状态 q 下读入 a，且当前栈顶元素为 x 的情形下，PDA可以转移为两者之一：状态 q' 并将栈顶改写为 y，或是状态 q'' 并将栈顶改写为 z。

注意 a、x、y，z 均可以为 ε。

5° $q_0\in Q$ 是该PDA的起始状态。

6° $F\subseteq Q$ 是该PDA「接纳」的状态集。

下面我们即定义何谓PDA的「计算」。

def PDA的计算流程。

设 $P=(Q,\Sigma,\Gamma,\delta,q_0,F)$ 是一台PDA. 对其输入字符串 $w=w_1w_2\cdots w_n$ ($w_i\in\Sigma_\varepsilon$). 若状态序列 $r_0r_1\ldots r_n$ 以及栈内容序列 $s_0s_1\ldots s_n$ 满足

1° $r_0=q_0$. $\forall i: r_i\in Q$

$s_0=\varepsilon$. $\forall i: s_i\in\Gamma_\varepsilon^*$ (即 s_i 保存的是此时刻栈顶到栈底的全部字符)

2° $\forall i\in\{0,1,\cdots,n-1\}$:

$\exists t: s_{i+1}=yt$ 且 $s_i=xt$ ($x,y\in\Gamma_\varepsilon$, $t\in\Gamma_\varepsilon^*$)

且 $(r_{i+1},y)\in\delta(r_i,w_i,x)$

(意即 s_{i+1} 等于 s_i 弹出 x 再压入 y，且 $x\to y$ 是符合 δ 规定的)

(当 $x=\varepsilon$ 时，相当于不弹出；当 $y=\varepsilon$ 时，相当于不压入)

则称 $(r_0,s_0),\ldots,(r_n,s_n)$ 是 P 对 w 的计算流程。r_n 称为终态。上面的定义虽则复杂，但本质就是在维护一个栈。

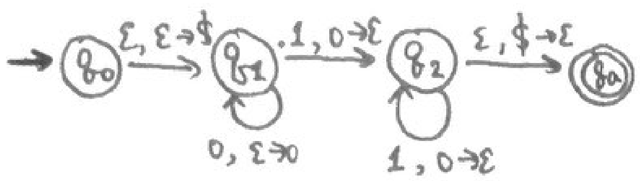
def PDA的接纳与拒绝。

对输入 w，若存在某一条计算流程的终态 $r_n\in F$，则称 w 被 P 接纳；否则称 w 被 P 拒绝。

def PDA识别的语言：所有被该PDA接纳的串.

我们用图示来巩固这些概念。
例子

e.g.



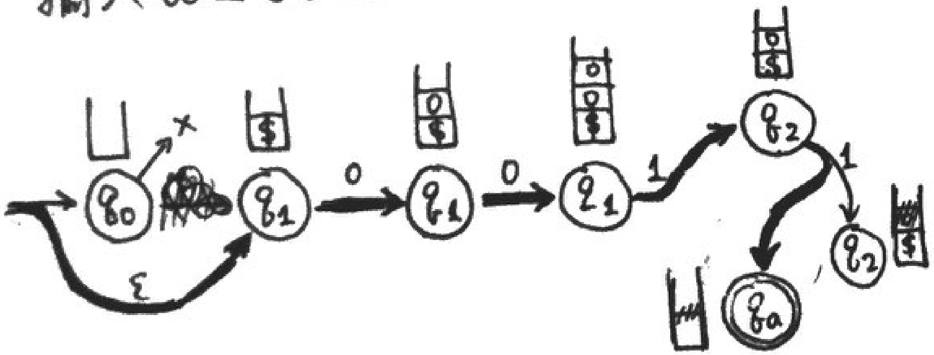
$Q=\{q_0, q_1, q_2, q_a\}$ $F=\{q_a\}$

$\Sigma=\{0,1\}$ $\Gamma=\{0,\$\}$

$\delta(q,a,x)$

q	a \ x	0	\$	ε
q_0	0	∅	∅	∅
	1	∅	∅	∅
	ε	∅	∅	$\{(q_1,\$)\}$
q_1	0	∅	∅	$\{(q_1,0)\}$
	1	$\{(q_2,\varepsilon)\}$	∅	∅
	ε	∅	∅	∅
q_2	0	∅	∅	∅
	1	$\{(q_2,\varepsilon)\}$	∅	∅
	ε	∅	$\{(q_a,\varepsilon)\}$	∅

输入 $w=0011$ 的运行情况：



接下来，我们证明PDA与CFG等价。

Lemma6 对任意CFG G，总存在某台PDA P：$\mathcal{L}(P)=\mathcal{L}(G)$.

proof. 我们留意到CFG生成字符串的过程，就是不断改写（展开）的过程。

我们希望构造的PDA，就是要模拟这改写（展开）过程。但又留意到改写的方式有许多，故我们要利用PDA的不确定产生许多的「线程」，试探所有的改写方式，并与输入作比较。无疑，我们改写时的「草稿」应留放在栈上，因为那是唯一一个容量无限的存储设备。

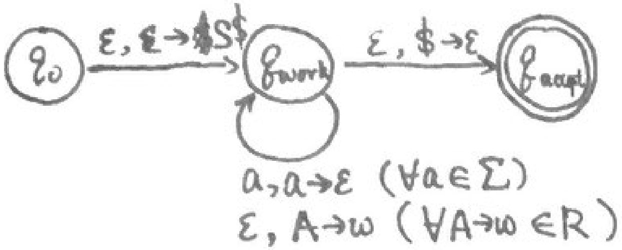
设 $G=(V,\Sigma,R,S)$

我们构造 $P:=(Q,\Sigma,V\cup\Sigma\cup\{\$\},\delta,q_0,F)$

其中 $Q:=\{q_0, q_{work}, q_{accept}\}$.

$F:=\{q_{accept}\}$.

δ 的定义见图示。



请注意：我们采用了偷懒的记号"$\varepsilon \to S\$$"与"$A \to w$"来把多步压栈压缩成一步到位的压栈。这么做显然是可以实现的，细节留作练习。

上面机器做的事情简述为：

1° 把符号"\$"放入栈底作为标记，然后把初始变量S压入栈中，转到 q_{work} 开始处理。

2° 在 q_{work}，有两种处理可能

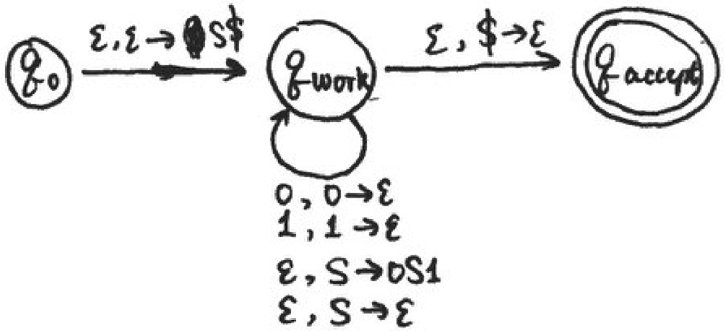
(1) 若栈顶是一个变量(如A)，那么根据规则把它改写成w并入栈。注意A可能可以改写成许多东西，比如 $A \to w_1$，$A \to w_2$，等等。这不要紧，因为PDA允许我们「一转多」。

(2) 若栈顶是一个中止符(如a)，那么仅当读入字符也是a时方能顺利转移，把栈顶元素弹出；否则，线程终结。这一步相当于比较了已展开完毕的字符与输入字符。

3° 如若栈顶为"\$"，则已到达底部，意味着输入串与展开串目前已完全匹配，转移至 q_{accept}。然而，若此时仍有输入，则无法匹配，线程终结。

易见：该机器P识别的语言正是 $\mathcal{L}(G)$。■

e.g. 设CFG为 $S \to 0S1 \mid \varepsilon$，则转成的PDA可以为：

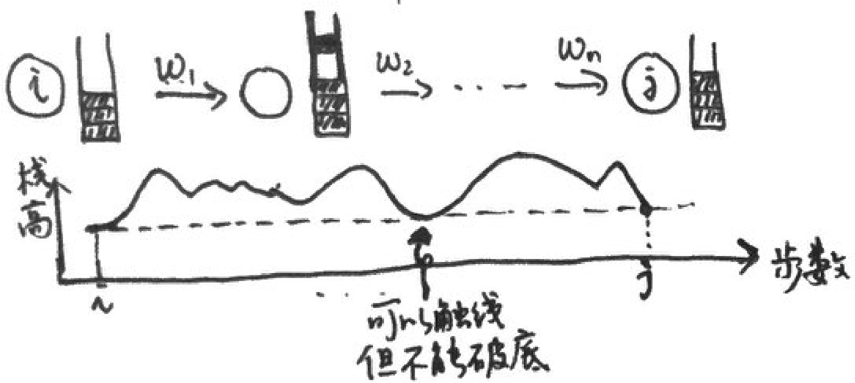


其中"$\varepsilon \to S\$$"等价于 $\xrightarrow{\varepsilon,\varepsilon\to\$}\bigcirc\xrightarrow{\varepsilon,\varepsilon\to S}$

"$S \to 0S1$"等价于 $\xrightarrow{\varepsilon,S\to 1}\bigcirc\xrightarrow{\varepsilon,\varepsilon\to S}\bigcirc\xrightarrow{\varepsilon,\varepsilon\to 0}$

反方向的讨论比较繁琐，为简化，先作一点铺垫。

def 好串. 设 P 是一台PDA，状态集为 Q。如果从状态 $i \in Q$ 出发（i 未必是 q_0，此时栈也未必为空），读入字符串 w，使得 P 依照某种流程来到状态 $j \in Q$，并且，全过程中栈的高度一直保持在原高度及以上，并在到达 j 时恰回到原高度，那么称 w 是 $i \to j$ 的好串。



remark. 好串之所以「好」，是因为它不理会原来处于栈中的元素，而且也不乱改其中内容。当好串使 P 从 i 依某种流程（注意：不一定是任何）转移至 j 后，栈就好似原封未动一样。这将有利于我们的后续「拼接」。

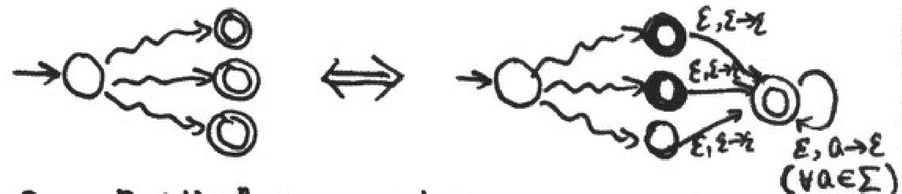
Lemma 7 对任意PDA P，总存在某种CFG G: $\mathcal{L}(G) = \mathcal{L}(P)$.

proof. 设 $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$.

不失一般性，作以下简化假定：

1° P 仅有一个接纳状态。若多于一个，则显然可引入一个新状态，把原接纳状态无条件指向它即可。

2° $\forall w \in \mathcal{L}(P)$，总存在一条计算流程，不仅使 w 被接纳，而且最终栈还是空的。这可以通过添加一个自环做到。



3° 凡是栈操作，只能是将非空字符入栈（$\varepsilon \to a$）或将非空字符出栈（$a \to \varepsilon$），而不允许改写（$a \to b$）或空操作（$\varepsilon \to \varepsilon$）。若出现后两者，拆成两步来完成即可。

$a \to b \iff a \to \varepsilon \to b$

$\varepsilon \to \varepsilon \iff \varepsilon \to \# \to \varepsilon$.

依照假定，我们设

$Q:=\{0,1,2,\cdots,n\}$，$q_0=0$，$F=\{n\}$.

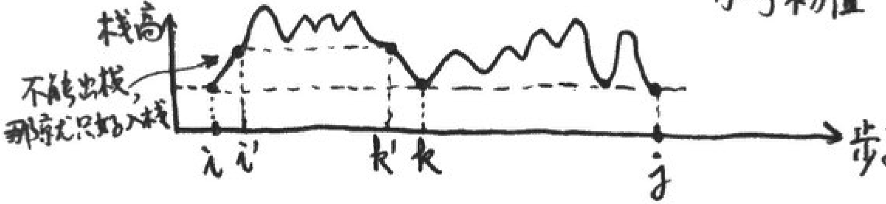
而 $\mathcal{L}(P)$ 正好就是所有 $0\to n$ 的好串之集合。我们希望做的是，找一个CFG $G=(V,\Sigma,R,S)$ 使得 $\mathcal{L}(G)$ 正好是所有 $0\to n$ 的好串之集合。换句话说，即从初始变量 S 出发，「生长出」所有 $0\to n$ 的好串。

这似乎不便入手，那么，能否分而治之呢？比如，假如已经知道变量 S_1 可以生长出所有 $0\to i$ 的好串，而 S_2 可以生长出所有 $i\to n$ 的好串，那么将二者拼接在一起，则仍为 $0\to n$ 的好串（这是由定义直接推知的）。当然，其间还有些细节，但我们的基本想法正是源于此。

令 $V:=\{A_{ij}\mid 0\le i,j\le n\}$；$\Sigma$ 与 P 中的 Σ 一致；$S:=A_{0n}$。我们期待建立一套规则，使初始变量 A_{0n} 得以生所有 $0\to n$ 的好串。而变量 A_{ij} 生成所有 $i\to j$ 的好串。R 的描述如下：

(1) 对 $\forall i\in Q$，添加规则 $A_{ii}\to\varepsilon$。原因很简单：ε 必定是 $i\to i$ 的好串。

(2) 对 $\forall i,j\in Q$，考虑 A_{ij} 应被怎样改写才能生成所有 $i\to j$ 的好串。既是好串，那么据定义则必在状态 j「触底线」，在 $i\to j$ 的途中也可能触若干次底线。如果记 $k\in Q$ 是它途中首次触线之状态（k 也许就为 j），那么，在 $i\to k$ 途中栈高总是严格大于初值，在 $k\to j$ 途中不小于初值。我们



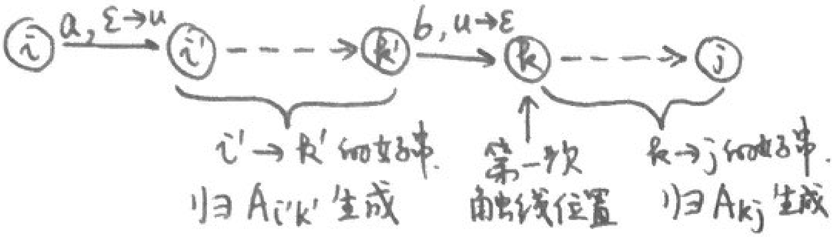
由此可知：i 的后继 $i'\to k$ 的前趋 k' 这一段对应的子串必为 $i'\to k'$ 的好串；$k\to j$ 这一段对应的子串必为 $k\to j$ 的好串。是故，要找 $i\to j$ 的好串，无非是枚举所有的 $k\in Q$，并把 $i\to i'$ 符号，$i'\to k'$ 的好串，$k'\to k$ 符号，$k\to j$ 的好串拼接起来罢了。据此，我们添加规则

$$A_{ij}\to aA_{i'k'}bA_{kj}$$

若 a,b,i',k' 满足

$(i', u) \in \delta(i, a, \varepsilon)$

$(k, \varepsilon) \in \delta(k', b, u)$ 对某 $u \in \Gamma$



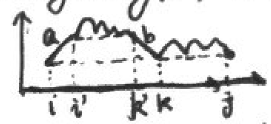
至此，我们构造G完毕。从直觉上看，这构造应当无误，但鉴于它比较复杂，故我们还是给出其正确性的证明，方比较稳妥。

claim 7.1 $\forall i, j \in Q$，以 A_{ij} 为初始变量而生成的任何字符串必为 $i \to j$ 的好串。

设 w 是任意满足 $A_{ij} \overset{*}{\Rightarrow} w$ 的字符串，对 w 的生成步数进行归纳。假设对于所有生成步数 $\leq n$ $(n \geq 1)$ 的 w，claim 7.1 成立。那么，对于生成步数为 $n+1$ $(n+1 \geq 2)$ 的 w，设其生成流程为

$$A_{ij} \Rightarrow \boxed{u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_n \Rightarrow w}$$

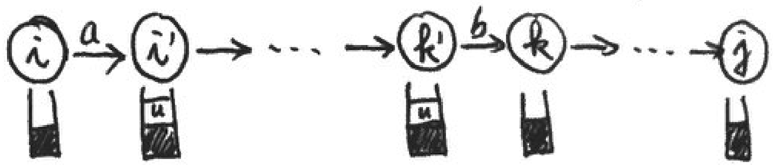
而我们知道，第一步改写只可能为 $A_{ij} \to a A_{i'k'} b A_{kj}$。而 $A_{i'k'}$ 与 A_{kj} 均在后 n 步改写中生成各自的串。据归纳假设，$A_{i'k'}$ 与 A_{kj} 能各自顺利生成 $i' \to k'$ 的好串、$k \to j$ 的好串，记之为 g_1 与 g_2，则 $w = a g_1 b g_2$ 必为 $i \to j$ 的好串。



作为基情况，$n=1$ 时，所有一步生成的串仅可能由 $A_{ij} \to \varepsilon$ $(i=j)$ 得来。显然 $w = \varepsilon$ 是 $i \to j=i$ 的好串。

claim 7.2 $\forall i, j \in Q$，任何 $i \to j$ 的好串均可由 A_{ij} 作初始变量而生成。

设 w 是任意 $i \to j$ 的好串，对其在 P 中计算的步数进行归纳。假设对于所有计算步数 $\leq n$ $(n \geq 0)$ 的 w，claim 7.2 成立。那，对于计算步数为 $n+1$ (≥ 2) 的 w，设其计算流程为



其中 k 是它首次「触底线」的状态。因为 w 是好串，所以 k 是存在的。而且，由 i 转移至下一步 i' 压入了 u，则由 k 的上一步 k' 转移至 k 必将弹出 u（不能是其它，因为 $i' \to k'$ 过程中不

输出底线的)。我们还~~的~~知道 $i' \rightsquigarrow k'$ 过程以及 $k \rightsquigarrow j$ 过程的计算步数严格小于 $n+1$，故由归纳假设，$i' \rightsquigarrow k'$ 过程所对应的串必能被 $A_{i'k'}$ 生成、$k \rightsquigarrow j$ 过程所对应的串必能被 A_{kj} 生成，也就是说串 w 必能由 ~~$A_{ij}\Rightarrow$~~ $aA_{i'k'}bA_{kj} \stackrel{*}{\Rightarrow} w$ 的途径生成。可是，"$A_{ij} \to aA_{i'k'}bA_{kj}$" 恰恰好是 G 中允许的规则，故 w 必能由 A_{ij} 生成。

作为基情况，$n=0$ 时，不经计算即可由 $i \to j$ 的好串有且仅有 ε，而且要求 $i=j$。这也恰恰好被规则 "$A_{ii} \to \varepsilon$" 捕捉。

（n 为奇数的情况是不存在的，因为好串有入栈必有出栈，二者成对出现）.

综合 claim 7.1，7.2，我们有：以 A_{ij} 为初始变量而生成的语言，正好等于 $i \to j$ 的好串集合。

特别地，以 A_{0n} 为初始变量而生成的语言，正好等于 $0 \to n$ 的好串集合。是故 $\mathcal{L}(G) = \mathcal{L}(P)$. ■

Theorem 8 语言 $\mathcal{L}$ 是 CFL 当且仅当存在某台 PDA P：$\mathcal{L} = \mathcal{L}(P)$.

proof. 由 Lemma 6，7 立得。 ■

remark. 回顾前面两个引理的证明，我以为 PDA 能表示 CFG 的关键在于栈的存在模拟了深度优先搜索的历程，提供了递归任意有限次的能力；CFG 能表示 PDA 的原因在于 CFG 迭代、递归的能力刻画了栈操作的特征，所谓「好串」无非是指它能让 PDA 完成一段深度优先搜索并返回原处。

有了 Theorem 8，我们更加明白 CFG 究竟比正则语言强在哪 —— 皆因有一个无限容量的栈存储，使我们能任意深地迭代。我们~~也~~同时也豁然开朗：CFG 表达不了诸如 $\{0^n1^n0^n \mid n \in \mathbb{N}\}$ 这样的语言，是因为无限容量的存储~~被~~访问受限，难由我们随意支配。尤其是，当栈顶被弹出并利用以后，我们无法再在别处为它保留副本，~~因~~而使其永远地被遗弃。背后的道理很深刻：即便有无限的容量，假若无法

释其潜能，则终究有很大的局限。

下面是两道很有意思的思考题，它们将进一步展现在PDA中什么是必要的、什么是多余的。

problems

1° 若把PDA定义中的转移函数 δ 改成 $Q\times\Sigma_\varepsilon\times\Gamma_\varepsilon \to 2^Q\times\Gamma_\varepsilon$ 的映射，所得模型的能力是否有衰退？证明之。

2° 若改成 $Q\times\Sigma_\varepsilon\times\Gamma_\varepsilon \to Q\times 2^{\Gamma_\varepsilon}$ 呢？