CSCI 3130 形式语言与自动机理论

> Formal Languages and Automata Theory

①②③④⑤⑥⑦⑧⑨⑩⑪⑫⑬⑭⑮⑯⑰⑱⑲⑳㉑㉒㉓㉔㉕㉖㉗㉘㉙㉚㉛㉜㉝㉞㉟㊱㊲㊳㊴㊵㊶㊷㊸㊹㊺㊻㊼㊽㊾㊿

Universal Set　　全集

Lec 0 Logistics

> 2025.9.1

评分标准

不计 Attendance.

5 个 Assignments（30%）

> 从 Blackboard 上传.
>
> 答案必须手写.

期中（30%）

> 10.27

期末（40%）

禁用 AI.

课程大纲

计算理论（Theory of Computation）包含四个部分：

- 自动机理论（Automata Theory）

  > 研究抽象的计算模型（如有限自动机、下推自动机、图灵机）.
  >
  > 描述和模拟计算过程.

- 形式语言与文法（Formal Languages and Grammars）

  > 研究语言的结构和规则.
  >
  > 与自动机一一对应，不同的自动机识别不同类别的语言.

> 例：正则语言由有限自动机识别；上下文无关语言由下推自动机识别.

- 可计算性（Computability）

  > 研究哪些问题是可以计算的，哪些问题是不可计算的.
  >
  > 例：停机问题（Halting Problem）是不可判定的. 见 `Appendix 1. 停机问题` .

- 复杂性（Complexity）

  > 研究可计算问题的难度.
  >
  > 即使问题可解，我们也关心解它需要多少时间和空间.

1. 自动机

> Automata

主要分为三类：

- 有限自动机（Finite Automata）： no temporary memory

  > 和数字电路的 “有限状态机” 是同一个模型的不同叫法.
  >
  > 例：Vending Machines
  >
  > small computing power
  >
  > “有限” 指有限个状态，运行时只能依靠 “当前状态” 记忆有限信息，没有额外存储结构（注意：不是完全没有存储能力，状态本身就可以存储信息，只是很有限）.
  >
  > 只能识别正规语言（regular languages），无法处理需要无限计数或嵌套结构的问题，如检查括号是否匹配.

- 下推自动机（Pushdown Automata）：stack

  > Compilers for Programming Languages
  >
  > medium computing power

- 图灵机（Turing Machines）：random access memory

  > If any algorithm can be solved by computers, it can be solved by Turing Machines.
  >
  > 图灵机是通用计算模型，凡是现实中计算机可以通过算法解决的问题，图灵机也能够解决.
  >
  > 图灵机（理论上）有无限长的磁带作为存储，可以读写和移动读写头，可以保存和修改任意多的信息.

2. 形式语言

> Formal Languages

对编程语言的抽象. 不关心语义，只关心符号的合法组合方式.

> 研究 “哪些字符串是合法的，哪些不是”.
>
> 编程语言（C、Python、Java）就是一种形式语言.

> 编译器通过词法分析 & 语法分析，检查程序是否由合法的 “字符串” 组成合法的 “句子”.

一个形式语言由两部分构成：

- 符号集合（a set of symbols）
- 形成规则（grammar）

  > 定义如何把这些符号按照一定的规则组合成句子（sentence）.
  >
  > 编程语言的语法规则.

字符串（string）是从符号集合里任意挑选符号组成的序列（不一定合法）.

句子（sentence）是符合语法规则的字符串.

Lec 1 数学基础

> 2025.9.2 - 2025.9.21

1. 集合

> 详见 `大二 term 1 ENGG 2440 离散数学 6. Set Theory and Counting Principle` .

1.1 定义

A **set** is a collection of elements.

1.2 符号表示

$$
\begin{aligned}
1 \in A &\Rightarrow 1 \text{ is an element of the set } A \\
ship \notin B &\Rightarrow \text{ship is not an element of the set } B \\
\text{finite set} &\Rightarrow \begin{cases} C = \{a, b, c, d, e, f, g, h, i, j, k\} \\ C = \{a, b, \ldots, k\} \end{cases} \\
\text{infinite set} &\Rightarrow \begin{cases} S = \{2, 4, 6, \ldots\} \\ S = \{j : j > 0, \text{ and } j = 2k \text{ for some } k > 0\} \\ S = \{j : j \text{ is nonnegative and even}\} \end{cases} \\
\text{Universal Set} &\Rightarrow \text{全集} \\
\text{Complement} \overline{A} &\Rightarrow \text{补集} \\
\text{Union} \cup &\Rightarrow \text{并集} \\
\text{Intersection} \cap &\Rightarrow \text{交集} \\
\text{Difference} - &\Rightarrow \text{差集} \\
\text{Subset } \subset &\Rightarrow \text{子集} \\
\text{Proper Subset } \subseteq &\Rightarrow \text{真子集} \\
\text{Disjoint Sets} &\Rightarrow \text{互斥、不相交集 } A \cap B = \emptyset \\
\text{Set Cardinality} &\Rightarrow \text{势 } A = \{2, 5, 7\}, |A| = 3
\end{aligned}
$$

差集：$A = \{1, 2, 3\}$，$B = \{2, 3, 4, 5\}$，则 $A - B = \{1\}$，$B - A = \{4, 5\}$.

$\emptyset$：Empty, Null Set，空集

$$
\begin{aligned}
\emptyset &= \{\} \quad (\text{The set with no elements}) \\
S \cup \emptyset &= S \\
S \cap \emptyset &= \emptyset \\
S - \emptyset &= S \\
\emptyset - S &= \emptyset \\
\overline{\emptyset} &= \text{Universal Set}
\end{aligned}
$$

Powersets：幂集

A powerset is a set of sets. 记 $S = \{a, b, c\}$，则 Powerset of S = the set of all the subsets of S, 记作 2^S.

$$
2^S = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}
$$

If S is finite, then $|2^S| = 2^{|S|}$.

Partition：分割

A set can be divided by separating it into a number of subsets. Suppose that $S_1, S_2, \ldots, S_n$ are subsets of a given set S and that the following holds:

- 两两不相交：The subset $S_1, S_2, \ldots, S_n$ are mutually disjoint;
- $S_1 \cup S_2 \cup \ldots \cup S_n = S$;
- None of the S_i is empty.

Then, $S_1, S_2, \ldots, S_n$ is called a **partition** of S.

1.3 德摩根律

> DeMorgan's Laws

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$
$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

1.4 笛卡尔积

> Cartesian Product

$A = \{2, 4\}$，$B = \{2, 3, 5, 6\}$，则

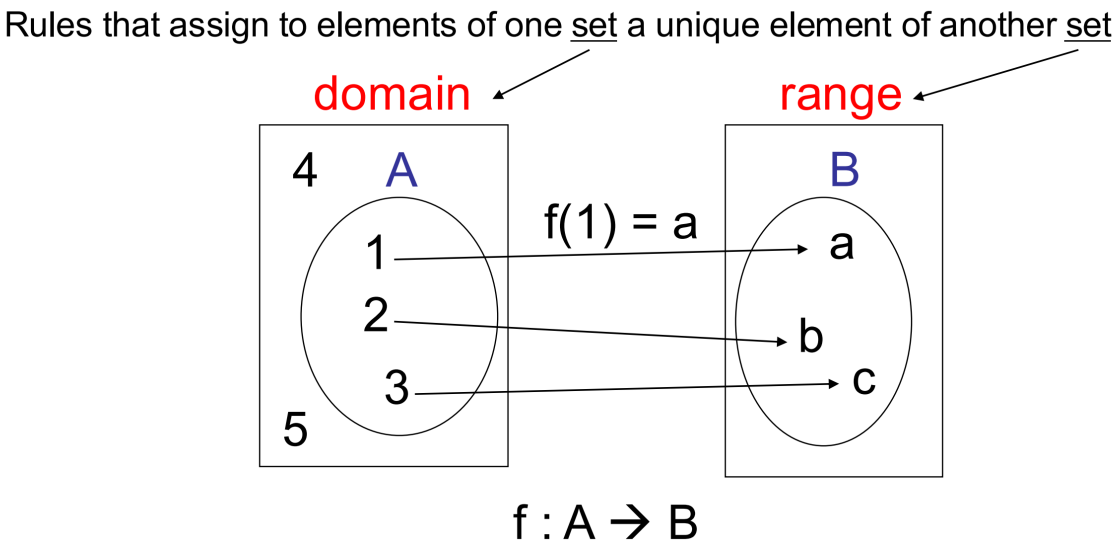
$$A \times B = \{(2,2), (2,3), (2,5), (2,6), (4,2), (4,3), (4,5), (4,6)\}$$
$$|A \times B| = |A||B|$$

方向重要，$A \times B \neq B \times A$.

2. 函数

> Functions

把一个集合（domain）中的元素分配给另一个集合（range）中的唯一元素的规则.



全函数（Total function）：定义域中的所有元素都被函数覆盖.

偏函数（Partial function）：定义域的某些元素没有映射.

3. 渐近分析

> Asymptotic Analysis
>
> 详见 `大二 term 1 ENGG 2440 离散数学 5. Asymptotics` .

Big-O：表示函数增长的上界.

$$f(x) = O(g(x)) \quad if \quad \exists\, c > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad f(x) \leq cg(x)$$

> 虽然一个函数可以有多个 Big-O，但通常选择**最紧的上界**，以提供更精确的增长描述.
>
> 注意：c 大于 0.

Big-Ω：表示函数增长的下界.

$$f(x) = \Omega(g(x)) \quad if \quad \exists\, c > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad f(x) \geq cg(x)$$

Theorem 1

$f(x) = O(g(x))$ is the same as $g(x) = \Omega(f(x))$.

$$\begin{aligned} f(x) = O(g(x)) &\Rightarrow \exists\, c > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad f(x) \leq cg(x) \\ &\Leftrightarrow \exists\, c > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad g(x) \geq \frac{1}{c} f(x) \end{aligned}$$

By taking $c' = \frac{1}{c} > 0,\ x_0' = x_0 \geq 0$, we have $g(x) = \Omega(f(x))$.

Θ：同时满足 O 和 Ω 的条件，表示函数增长的确切阶.

$$f(x) = \Theta(g(x)) \quad if \quad f(x) = O(g(x)) \quad and \quad g(x) = O(f(x)).$$

> $f(x) = \Theta(g(x)) \nRightarrow f(x) = g(x)$. It just means that
> $\exists\, c_1, c_2 > 0,\ x_0 \geq 0 \quad such\ that \quad \forall x \geq x_0, \quad c_1 g(x) \leq f(x) \leq c_2 g(x)$
>
> 只能说明 $f(x)$ 和 $g(x)$ 同阶.

Small-o：表示严格上界（严格更高阶），比 Big-O 更严格的界定.

$$f(x) = o(g(x)) \quad if \quad \lim_{x \to \infty} \frac{f(x)}{g(x)} = 0.$$

等价于

$$f(x) = o(g(x)) \quad if \quad \forall\, c > 0,\ \exists\, x_0 > 0\ such\ that\ 0 \leq f(x) < cg(x)\ for\ all\ x \geq x_0.$$

> 注意同阶不足以满足任意 c，$g(x)$ 要比 $f(x)$ 高阶才满足.

$$f(x) = o(g(x)) \Rightarrow f(x) = O(g(x))$$
$$f(x) = O(g(x)) \nRightarrow f(x) = o(g(x))$$

Small-ω：表示严格下界（严格更低阶），比 Big-Ω 更严格的界定.

$$f(x) = \omega(g(x)) \quad if \quad \lim_{x \to \infty} \frac{g(x)}{f(x)} = 0.$$

等价于

$$f(x) = \omega(g(x)) \quad if \quad \forall\, c > 0,\ \exists\, x_0 > 0\ such\ that\ 0 \leq cg(x) < f(x)\ for\ all\ x \geq x_0.$$

> 注意同阶不足以满足任意 c，$g(x)$ 要比 $f(x)$ 低阶才满足.

$$f(x) = \omega(g(x)) \Rightarrow f(x) = \Omega(g(x))$$
$$f(x) = \Omega(g(x)) \nRightarrow f(x) = \omega(g(x))$$

4. 关系

> Relations

关系是集合论中的重要概念，比函数更一般（general）.

4.1 定义

一个关系 R 是一组有序对：

$$R = \{(x_1, y_1), (x_2, y_2), \ldots\}$$

如果 $(x, y) \in R$，我们记作 xRy.

> 函数：定义域中的每个元素，有且仅有一个像.
>
> 关系：定义域中的一个元素，可对应任意多个值，甚至没有对应值.
>
> 关系更广泛，函数是关系的一种特殊情况.

A **relation** from a set A to a set B is a subset of $A \times B$. Hence, a relation R consists of ordered pairs (a, b), where $a \in A$ and $b \in B$. If $(a, b) \in R$, we say that a is **R**-elated to b, and we also write aRb.

例：若定义 $R =$ '$>$'，有 $2 > 1,\ 3 > 2,\ 3 > 1$.

4.2 等价关系

> Equivalence Relations ($\equiv$ or $\sim$)
>
> "$\equiv$" 也可以表示 "定义为"，不要混淆.

一个关系 R 如果满足以下三个条件，就称为等价关系：

- 自反性（Reflexive）

  > 对任意 x，有 xRx.

- 对称性（Symmetric）

  > 若 xRy，则 yRx.

- 传递性（Transitive）

  > 若 xRy 且 yRz，则 xRz.

例 1：等于号 '$=$'.

例 2：在非负整数集上，定义一个关系 R：

$$xRy \text{ if and only if } x \bmod 3 = y \bmod 3.$$

易得 R 是一个等价关系.

> 它实际上把非负整数分成了三个等价类（Equivalence Class）：
>
> - 余数为 0 的集合：$\{0, 3, 6, 9, \ldots\}$
> - 余数为 1 的集合：$\{1, 4, 7, 10, \ldots\}$
> - 余数为 2 的集合：$\{2, 5, 8, 11, \ldots\}$

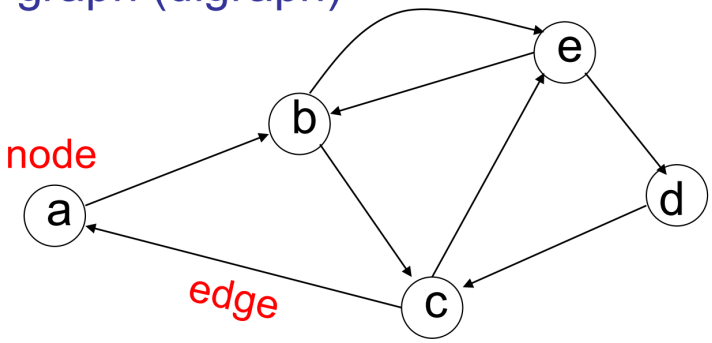
5. 图论

> Graphs
>
> 详见 `大二 term 1 ENGG 2440 离散数学 9. Introduction to Graph Theory`.

A directed graph：有向图

> Nodes (Vertices)：顶点 / 节点
>
> Edges：边

A directed graph (digraph)



- Nodes (Vertices)

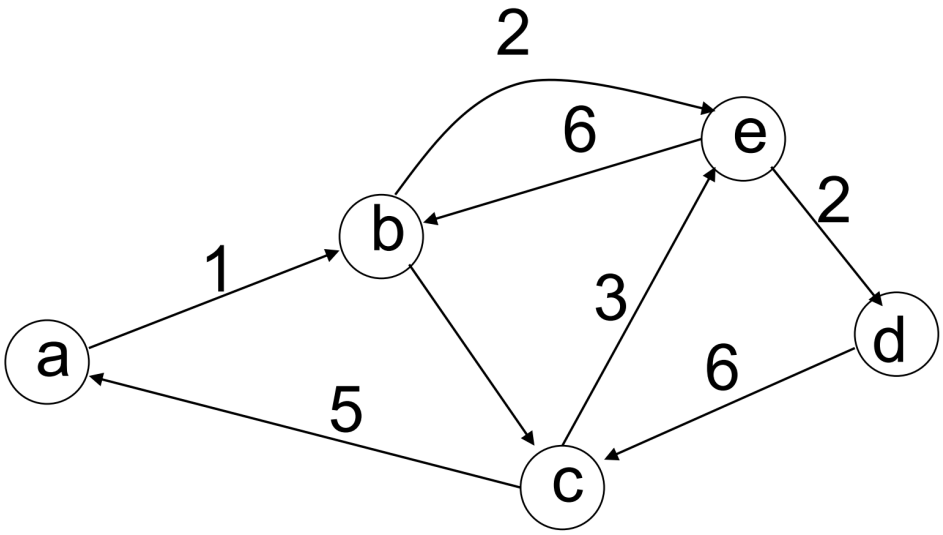
$$V = \{ a, b, c, d, e \}$$

- Edges

$$E = \{ (a,b), (b,c), (b,e), (c,a), (c,e), (d,c), (e,b), (e,d) \}$$

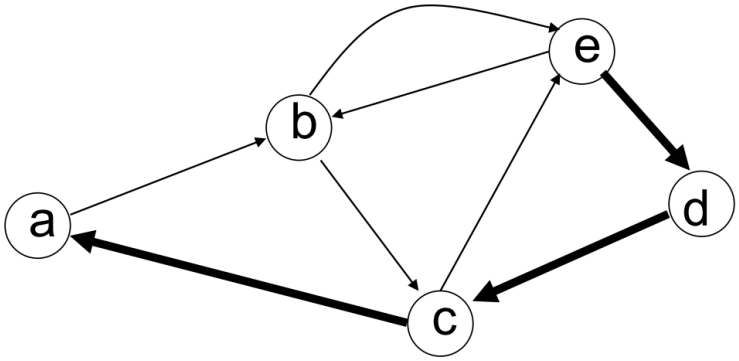
Labeled Graph：标记图

Labeled Graph



Walk：游走 / 步行.

Walk



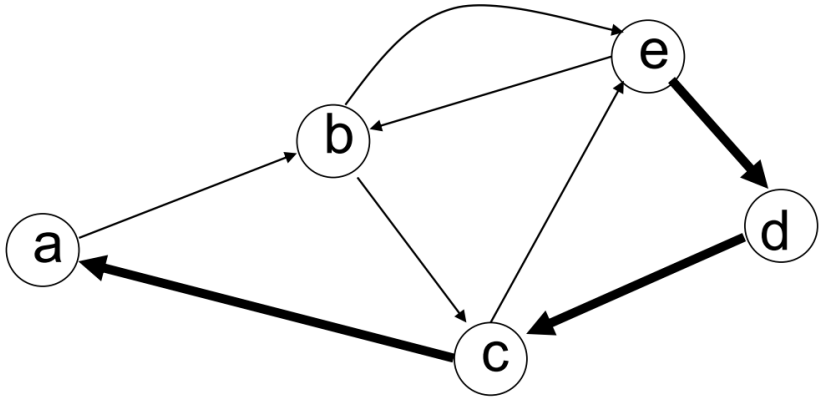
Walk is a sequence of adjacent edges

(e, d), (d, c), (c, a)

Path：路径

Simple Path：简单路径

Path



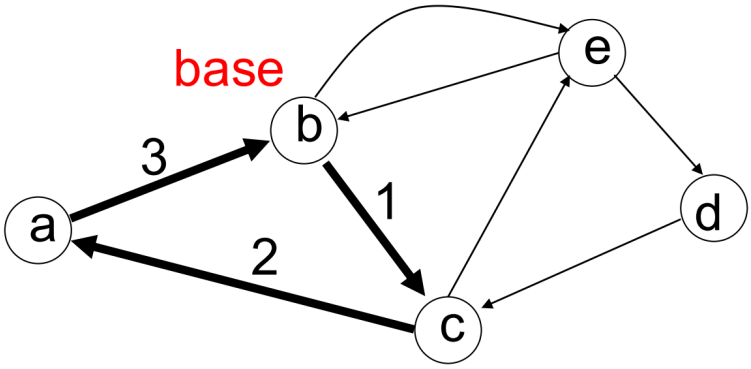
Path: a walk where no edge is repeated

Simple path: no node is repeated

Cycle：环

Simple Cycle：简单环

Cycle



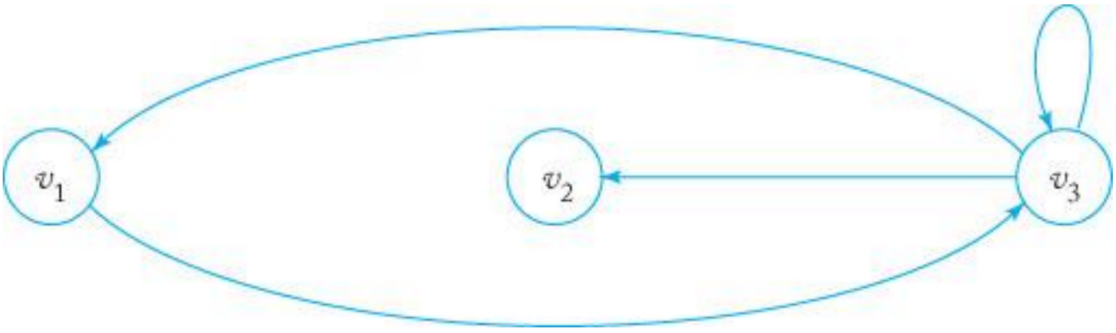
Cycle: a walk from a node (base) to itself

Simple cycle: only the base node is repeated

Loop：自环

Loop

An edge from a vertex to itself



- $(v_1, v_3), (v_3, v_2)$ is a simple path from v_1 to v_2
- $(v_1, v_3), (v_3, v_3), (v_3, v_1)$ is a cycle (not simple one)
- There is a loop on vertex v_3

Euler Tour：欧拉回路.

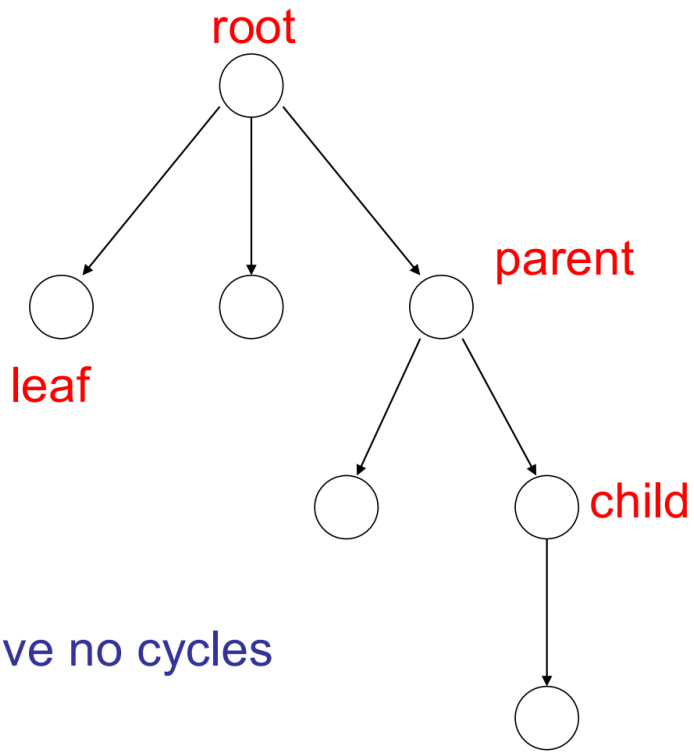
A cycle that contains each edge once. 每条边恰好走一次.

Hamiltonian Cycle：哈密顿回路.

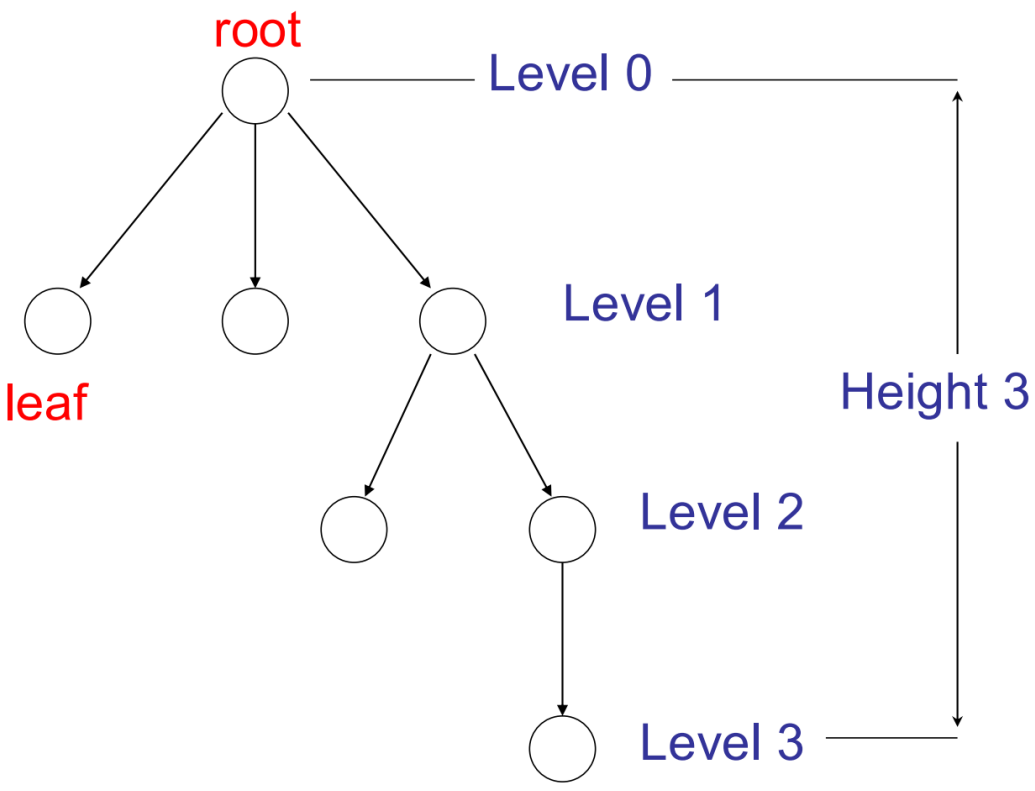
A simple cycle that contains all nodes. 每个顶点恰好走一次.

Trees：树

Trees



Trees have no cycles



6. 证明方法

> Proof Techniques
>
> 直接证明、反证法、数学归纳法.

Direct/Constructive Proof：直接证明.

例：For integers a, b, if a and b are odd, then ab is odd.

证明：

$$\begin{aligned} ab &= (2x+1)(2y+1) \\ &= 4xy + 2x + 2y + 1 \\ &= 2(2xy + x + y) + 1 \end{aligned}$$

$z = 2xy + x + y$, $ab = 2z + 1$.

Not only did you prove that a z exists, you constructed an "algorithm" for generating this z.

This is an example of a constructive proof.

Proof by Induction：数学归纳法.

Proof by Contradiction：反证法.

Lec 2 语言 语法 自动机

1. 语言

> Languages

1.1 字母表

> Alphabet

字母表（通常记作 Σ）是一个有限的符号集合（A Set of Symbols），其中每个符号称为字母（Symbol）.

$$\Sigma = \{a_1, a_2, \ldots, a_n\}, \quad n \geq 1$$

> 字母表是有限集合.
>
> 这些字母是基本单元，不能再拆分.
>
> 字母表 $\neq$ 字符串.

1.2 字符串

> String

给定一个字母表 Σ，一个字符串就是从 Σ 中取出若干个符号，按顺序排列得到的**有限**序列.

$$w = a_1 a_2 \dots a_n, \quad a_i \in \Sigma$$

> $|w| = n$ 是字符串的长度.
>
> （2025.10.18）注意，本课范围内讨论的字符串长度都是有限的. 但是一个语言中包含的字符串数量可以无限.

1.3 字符串操作

> String Operations

记 $w = a_1 a_2 \dots a_n$，$v = b_1 b_2 \dots b_m$.

1.3.1 连接

> Concatenation

$wv = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$

$|wv| = |w| + |v|$

1.3.2 逆序

> Reverse

$w^R = a_n \dots a_2 a_1$

1.3.3 幂运算

> String Power / Exponentiation

$$w^n = \underbrace{ww \cdots w}_{n}$$

定义 $w^0 = \lambda$.

1.3.4 Kleene 星号

> Kleene Star / Kleene Closure

给定一个字母表 Σ，Σ^* 表示由 Σ 中的符号组成的所有可能的有限字符串的集合，包括空字符串 λ.

$\Sigma = \{a, b\}, \Sigma^* = \{\lambda, a, b, ab, ba, aa, bb, aba, \dots\}$

1.3.5 Kleene 加号

> Kleene plus operator

Recall Kleene Star:

给定一个字母表 Σ，Σ^* 表示由 Σ 中的符号组成的所有可能的有限字符串的集合，包括空字符串 λ.

$\Sigma = \{a, b\}, \Sigma^* = \{\lambda, a, b, ab, ba, aa, bb, aba, \ldots\}$

Kleene plus 则是在 Kleene Star 的基础上去掉空字符串. 即 Σ^+ 表示由 Σ 中的符号组成的所有**非空**有限字符串的集合.

$\Sigma = \{a, b\}, \Sigma^+ = \{a, b, ab, ba, aa, bb, aba, \ldots\}$

即 $\Sigma^+ = \Sigma^* - \lambda$.

1.4 空字符串

> Empty String

A string with no letters: λ

$|\lambda| = 0$

$\lambda w = w\lambda = w$

1.5 子串

> Substring

在字母表 Σ 上，一个字符串 x 是另一个字符串 w 的子串，如果存在字符串 $u, v \in \Sigma^*$，使得 $w = uxv$.

> 给定一个字母表 Σ，Σ^* 表示由 Σ 中的符号组成的所有可能的有限字符串的集合，包括空字符串 λ.
>
> $\Sigma = \{a, b\}, \Sigma^* = \{\lambda, a, b, ab, ba, aa, bb, aba, \ldots\}$
>
> 见 `1.3.4 Kleene 星号运算`.

如果 $u = \lambda$，那么 x 是前缀（Prefix）

如果 $v = \lambda$，那么 x 是后缀（Suffix）

如果 $u = v = \lambda$，那么 $x = w$（字符串是自己的子串）

1.6 语言

> Languages

在形式语言中，语言（Language）定义为 Σ^* 的任意一个子集. 换句话说，语言是一组字符串，这些字符串由某个字母表中的符号拼成.

例：$\Sigma = \{a, b\}, \Sigma^* = \{\lambda, a, b, ab, ba, aa, bb, aba, \ldots\}$

Languages: $\{\lambda\}$, $\{a, aa, aab\}$, $\{\lambda, abba, baba, aa, ab, aaaaaa\}$

> 注意：语言可以为空集，且 $\emptyset \neq \{\lambda\}$.

语言可以有限（Finite），也可以无限（Infinite）：

An infinite language $L = \{a^n b^n : n \geq 0\}$.

> $\lambda \in L$, $ab \in L$, $aaabbb \in L$.
>
> $abb \notin L$.

（2025.10.18）这里的无限指的是字符串数量，不是长度.

1.7 语言操作

> Operations on Languages

1.7.1 交并补差

集合操作（交并补差）可用于语言：

$$\{a, ab, aaaa\} \cup \{bb, ab\} = \{a, ab, bb, aaaa\}$$
$$\{a, ab, aaaa\} \cap \{bb, ab\} = \{ab\}$$
$$\{a, ab, aaaa\} - \{bb, ab\} = \{a, aaaa\}$$

Complement: $\overline{L} = \Sigma^* - L$.

$$\overline{\{a, ba\}} = \{\lambda, b, aa, ab, bb, aaa, \ldots\}$$

1.7.2 逆序

> Reverse

Definition: $L^R = \{w^R : w \in L\}$.

> 语言中的所有字符串分别取逆序.

例：

$\{ab, aab, baba\}^R = \{ba, baa, abab\}$

$L = \{a^n b^n : n \geq 0\} \Rightarrow L^R = \{b^n a^n : n \geq 0\}.$

1.7.3 连接

> Concatenation

Definition: $L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$

例：$\{a, ab, ba\}\{b, aa\} = \{ab, aaa, abb, abaa, bab, baaa\}$

> 有点类似笛卡尔积，但是不一样.

1.7.4 幂运算

Definition: $L^n = \underbrace{LL\cdots L}_{n}$

特别地，$L^0 = \{\lambda\}$

例:

$L = \{a^n b^n : n \geq 0\} \Rightarrow L^2 = \{a^n b^n a^m b^m : n, m \geq 0\}$

$aabbaaabbb \in L^2.$

> Note that n and m in the above are unrelated.

1.7.5 Kleene 星号

> Star-Closure

Definition: $L^* = L^0 \cup L^1 \cup L^2 \cdots$

L 中的字符串任意组合得到的集合. 注意，L 的字符串不能从中打断.

> 包含空字符串 λ.

1.7.6 Kleene 加号

> Positive Closure，正闭包

Definition: $L^+ = L^1 \cup L^2 \cup \cdots$

$L^* = L^0 \cup L^+ = \{\lambda\} \cup L^+.$

> 当 $\lambda \notin L$ 时，可得 $L^+ = L^* - \{\lambda\}$. 注意如果 L 本身包含空字符串，则 L^+ 仍有空字符串.

2. 语法

> Grammars

语言 = 符号 + 语法.

2.1 定义

A grammar G is defined as a 4-tuple:

$$G = (V, T, S, P)$$

where

- V is a finite set of variables（变量集合 / 非终结符）

> 类似占位符，用来被产生式规则 P 替换.

- T is a finite set of terminals（终结符集合）

> 实际字符，不能继续被替换和推导.
>
> 某个文法中，如果终结符集合是 $\{a, b\}$, 最后生成的字符串只会由 a 和 b 组成.
>
> 实际上就是用该语法生成的语言对应的字母表.
>
> 注意，终结符集合里不用显式写出 λ，产生式也可以包含 λ.

- $S \in V$, called start variable（起始变量）
- P is a finite set of production rules（产生式规则）

> 规则的形式通常是 $A \to \alpha$. 其中 A 是变量，α 是变量和终结符组成的字符串.

生成字符串的过程：从起始变量出发，根据产生式规则不断把变量（非终结符）替换掉，直到只剩下终结符.

2.2 用语法定义语言

> 从生成论的角度下定义，相当于第二种定义.
>
> 第一种定义是集合论的角度，见 `1.6 语言`. 它把语言定义为字母表的 Kleene Star 的任意子集.
>
> 两种定义得到的产物相同（形式上等价），但是用语法生成来定义语言更符合我们对自然语言的认识（有一定规则，而非随机挑选一个子集）.
>
> 集合论定义强调 “什么可以是形式语言”，更抽象.
>
> 生成论定义强调 “形式语言是如何被构造出来的”，更符合直觉和逻辑.

Let $G = (V, T, S, P)$ be a grammar. Then the set

$$L(G) = \{w \in T^* : S \Rightarrow^* w\}$$

is the **language** generated by G.

> $\Rightarrow^*$ 表示 “经过有限步推导”.

例 1

例 1：$G = (\{S\}, \{a, b\}, S, P)$, $P : S \to aSb, S \to \lambda$

易得 $L(G) = \{a^n b^n : n \geq 0\}$.

2.3 句型 & 句子

> sentential forms & sentence
>
> 直接按字面意思理解即可. 句型就像英语里归纳总结的若干句型，句子则是用具体的单词填充句型得到的.

If $w \in L(G)$, then the sequence

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n \Rightarrow w$$

is a derivation of the **sentence** w.

$S, w_1, w_2, \ldots, w_n$ are called **sentential forms**.

> 包括起始变量在内的所有中间状态都叫做句型.
>
> 当推导到最后一步，完全由终结符组成，就变成了一个句子.

例 2

例 2：Find a grammar that generates

$$L = \{a^n b^{n+1} : n \geq 0\}$$

由 例 1 得,

$$\begin{aligned} & G = (\{S\}, \{a, b\}, S, P), \\ & P : S \to aSb, \\ & \quad S \to \lambda \\ \Rightarrow\ & L(G) = \{a^n b^n : n \geq 0\}. \end{aligned}$$

我们只需在此基础上进行构造，使每个句子末尾都多一个 b:

$$\begin{aligned} & G = (\{S, A\}, \{a, b\}, S, P), \\ & P : S \to Ab, \\ & \quad A \to aAb, \\ & \quad A \to \lambda \end{aligned}$$

> 起始的时候在末尾强制添加一个 b，然后按照例 1.

例 3

例 3：考虑语法

$$\begin{aligned} G &= (\{S\}, \{a, b\}, S, P), \\ P: S &\to SS, \\ S &\to \lambda \\ S &\to aSb, \\ S &\to bSa \end{aligned}$$

和语言

$$L = \{w \in \{a, b\}^* : n_a(w) = n_b(w)\}$$

> 所有 a 和 b 数量相等的字符串集合.

试证明：

$$L(G) = L.$$

> 即证明文法 G 恰好生成所有 a 和 b 数量相等的字符串.

首先，G 的产生式包含四种可能的推导步骤，每种都保持 a 和 b 数量的平衡. 显然 $L(G) \subseteq L$.

然后证明 $L \subseteq L(G)$：数学归纳法（Induction）.

① 归纳假设（Induction Hypothesis）

Assume $\forall w \in L$ with $|w| \leq 2n$ can be derived with G.

② 归纳步骤（Induction Step）

证明：$\forall w \in L$ with $|w| = 2n + 2$ can also be derived with G.

情况 1：形如 $w = aw_1b$（首尾不同）.

显然 $w_1 \in L$. 且 $|w_1| = 2n$. By assumption,

$$S \Rightarrow^* w_1$$

> 即 $w_1 \in L(G)$. 这是根据归纳假设得出的.

因此

$$S \Rightarrow aSb \Rightarrow^* aw_1b = w$$

> 即 $w \in L(G)$.
>
> $w = bw_1a$ 同理可得.

情况 2：形如 $w = \underbrace{a \cdots a}_{2n+2}$（首尾相同）.

引理：随机游走小结论. 见 `Appendix 2. 随机游走`.

从左到右扫描，每遇到一个 a 视为前进（$+1$），每遇到一个 b 视为后退（-1）. 若首尾相同，则必能把 w 分为长度 $\leq 2n$ 的两部分，这两部分分别满足 a 和 b 数量相等.

即必有 $w_1 \in L$ with $|w_1| \leq 2n$ 和 $w_2 \in L$ with $|w_2| \leq 2n$ such that $w = w_1 w_2$.

> 也可以构造 $\Delta = n_a(左) - n_b(左)$ 表示当前扫描间隔的左侧 a 与 b 数量差值. 扫描第一个间隔时 $\Delta = +1$，扫描最后一个间隔时 $\Delta = -1$. 因为每一步扫描 Δ 只会变化 ± 1，因此从 $+1$ 到 -1 必定经过 0.

By assumption,

$$S \Rightarrow^* w_1, \quad S \Rightarrow^* w_2$$

> 即 $w_1 \in L(G), w_2 \in L(G)$. 这是根据归纳假设得出的.

因此

$$S \Rightarrow SS \Rightarrow^* w_1 S \Rightarrow^* w_1 w_2 = w$$

> 即 $w \in L(G)$.
>
> $w = \underbrace{b \cdots b}_{2n+2}$ 同理可得.

③ 基例（Base Case）

正常情况取 $n = 0$ 验证即可. 但是证明过程包含情况 2：形如 $w = \underbrace{a \cdots a}_{2n+2}$（首尾相同）. 该情况要求 $n \geq 1$（即至少长度为 4，如 $abba$ 和 $baab$）. 因此取 $n = 1$ 验证，即验证 $\forall w \in L$ with $|w| \leq 2 \times 1$ can be derived with G.

- $w \in L, |w| = 0$：λ. 显然满足 $\lambda \in L(G)$（$S \Rightarrow \lambda$）.
- $w \in L, |w| = 2$: ab 或 ba. 以 ab 为例，$S \Rightarrow aSb \Rightarrow ab$. ba 同理. 因此 $w \in L(G)$.

> 注意 L 中的句子长度一定为偶数，不需要考虑奇数长度.

根据归纳，Assume $\forall w \in L$ with $|w| \leq 2n$ can be derived with G,

$\Rightarrow \forall w \in L$ with $|w| = 2n + 2$ can also be derived with G.

$n = 1$ 时假设成立，则 $\forall w \in L \Rightarrow w \in L(G)$. 即 $L \subseteq L(G)$.

综上所述，$L(G) \subseteq L$ 且 $L \subseteq L(G) \Rightarrow L(G) = L$.

2.4 语法等价

> Equivalent of Grammars

Two grammars G_1 and G_2 are **equivalent** if they generate the same language.

> 即 $L(G_1) = L(G_2)$.

例：

$$
\begin{aligned}
G_1 &= (\{S\}, \{a, b\}, S, P_1), \\
P_1 : S &\to aSb, \\
S &\to \lambda \\
\Rightarrow L(G_1) &= \{a^n b^n : n \geq 0\}.
\end{aligned}
$$

$$
\begin{aligned}
G_2 &= (\{S, A\}, \{a, b\}, S, P_2), \\
P_2 : S &\to aAb, \\
S &\to \lambda, \\
A &\to aAb, \\
A &\to \lambda \\
\Rightarrow L(G_2) &= \{a^n b^n : n \geq 0\}.
\end{aligned}
$$

3. 自动机

> Automaton（单数）
>
> Automata（复数）

自动机是一个抽象模型，可以理解为一台能自动运行的机器.

核心思想：状态（State）、转移（Transition）.

3.1 接收器 & 转换器

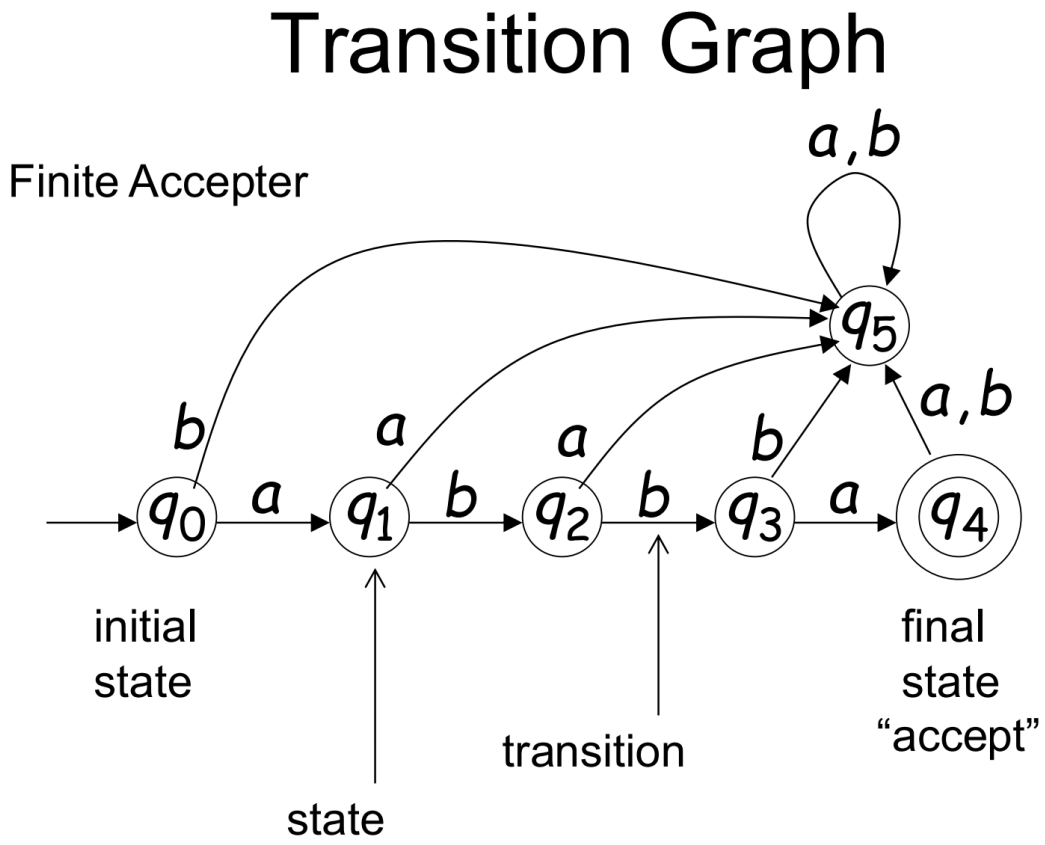
> Accepter & Transducer

自动机根据输出类型可分为两类：

接收器（Accepter）

An automaton whose output is YES or NO.

> 接收器是用来作判断和识别的机器. 它的主要任务是接收一个字符串，然后根据内部规则判断这个字符串是否 “合法” 或是否 “符合某种模式”.
>
> 输出 YES：意味着接受，表示输入的字符串符合预设的规则.
>
> 输出 NO：意味着拒绝，表示输入的字符串不符合规则.
>
> 本节介绍的 DFA 和 NFA 都指接收器.



从 q_0 出发，最终落在 q_4 就接受，不落在 q_4 就拒绝.

转换器（Transducer）

An automaton whose output are strings of symbols.

> 转换器是用来做转换和翻译的机器. 它不仅读取一个输入字符串，还会根据输入生成一个新的输出字符串.
>
> 例：编译器、编码器、解码器、Mealy 机、Moore 机.

3.2 DFA 确定有限自动机

> Deterministic Finite Accepters
>
> Deterministic Finite Automaton
>
> DFA
>
> 确定有限状态自动机 / 确定有限自动机

Deterministic Finite Accepter (DFA) is defined by the 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

其中，

状态集合 Q：a finite set of internal states.

> 自动机内部所有可能状态的有限集合.
>
> 包含初始状态、中间状态、接受状态（最终状态）.

输入字母表 Σ：a finite set of symbols called input alphabet.

> 有限的符号集合，任何要被这台机器处理的字符串，必须由这个字母表中的字符组成.
>
> 理论上，如果输入字符串中出现了一个不属于 Σ 的字符，那么转移函数 δ 对于这个输入是未定义的（undefined）. 此时这台机器会 “卡住” 或 “崩溃”.
>
> 但在实际实现中，通常会采用更鲁棒的方式来处理，例如引入一个 “陷阱状态 / 死亡状态”. Trap State 是一个非接受状态，一旦进入陷阱状态，任何后续的输入都无法离开. 所有从陷阱状态出发的转移都会指回它自己. 若用 `other` 来表示不在 Σ 中的字符，则设计转移函数 $\delta(q_i, \text{other}) = q_{\text{trap}}$，$\delta(q_{\text{trap}}, \text{any}) = q_{\text{trap}}$. 这样，任何包含非法字符的字符串都会被导向一个不可逃脱的非接受状态，从而被明确拒绝.

转移函数 δ：$Q \times \Sigma \to Q$ called transition function (Total function).

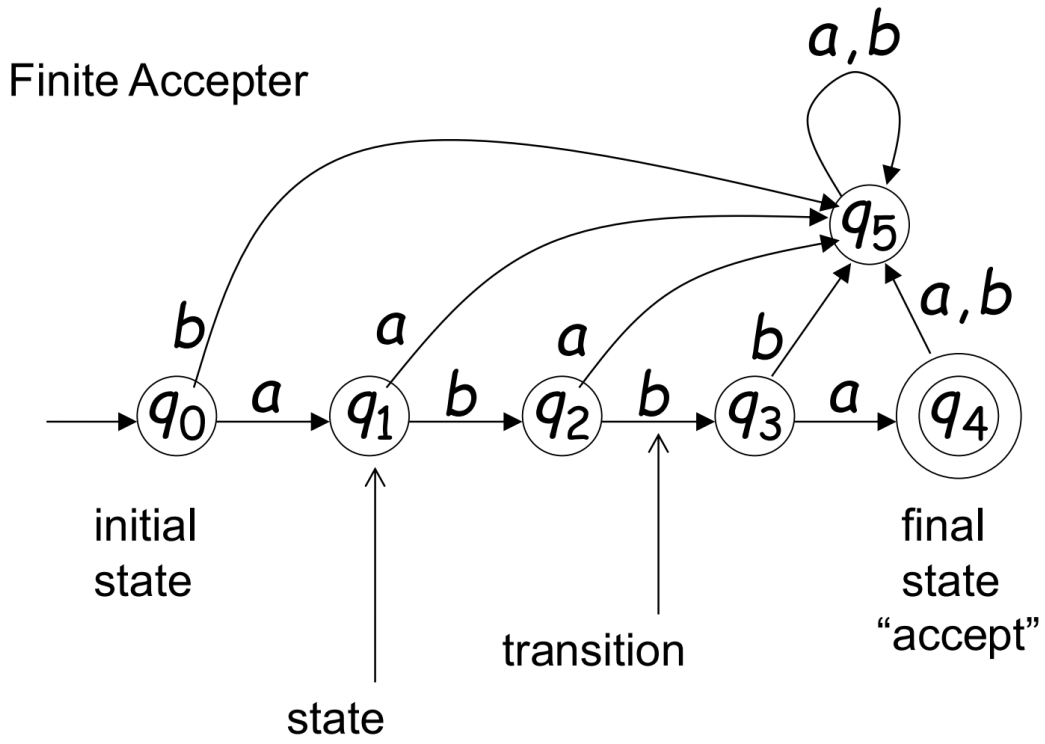
> 自动机的 "规则手册".
>
> $Q \times \Sigma \to Q$ 意思是，它接受一个当前状态和一个输入符号，然后给出**唯一**的下一个状态.
>
> Total function 意思是这个规则手册是完整的，对于任何可能的 “状态 + 输入” 组合，都有明确的指令，没有遗漏. 这是 DFA “确定性” 的来源.

初始状态 q_0：$q_0 \in Q$ is the initial state

> 每次给机器一个新的字符串进行分析时，它总是从初始状态出发.
>
> 注意，如果空字符串能被接受，则 $q_0 \in F$. 因为自动机处理空字符串时，不会进行任何状态转移，即读完后仍停留在初始状态 q_0.

最终状态集合 F：$F \subseteq Q$ is a set of final states

> DFA 接受 / 拒绝字符串判定：当机器读完整个输入字符串后，如果刚好停在 F 中的任何一个状态，那么这个字符串就被接受（输出 YES）. 如果停在非最终状态，那么字符串就被拒绝（输出 NO）.



以此图为例，

$$
\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3, q_4, q_5\} \\
\Sigma &= \{a, b\} \\
F &= \{q_4\} \\
\delta(q_0, a) &= q_1 \\
\delta(q_0, b) &= q_5 \\
&\vdots \text{（共12条）}
\end{aligned}
$$

转移函数可以用表格形式来表示：

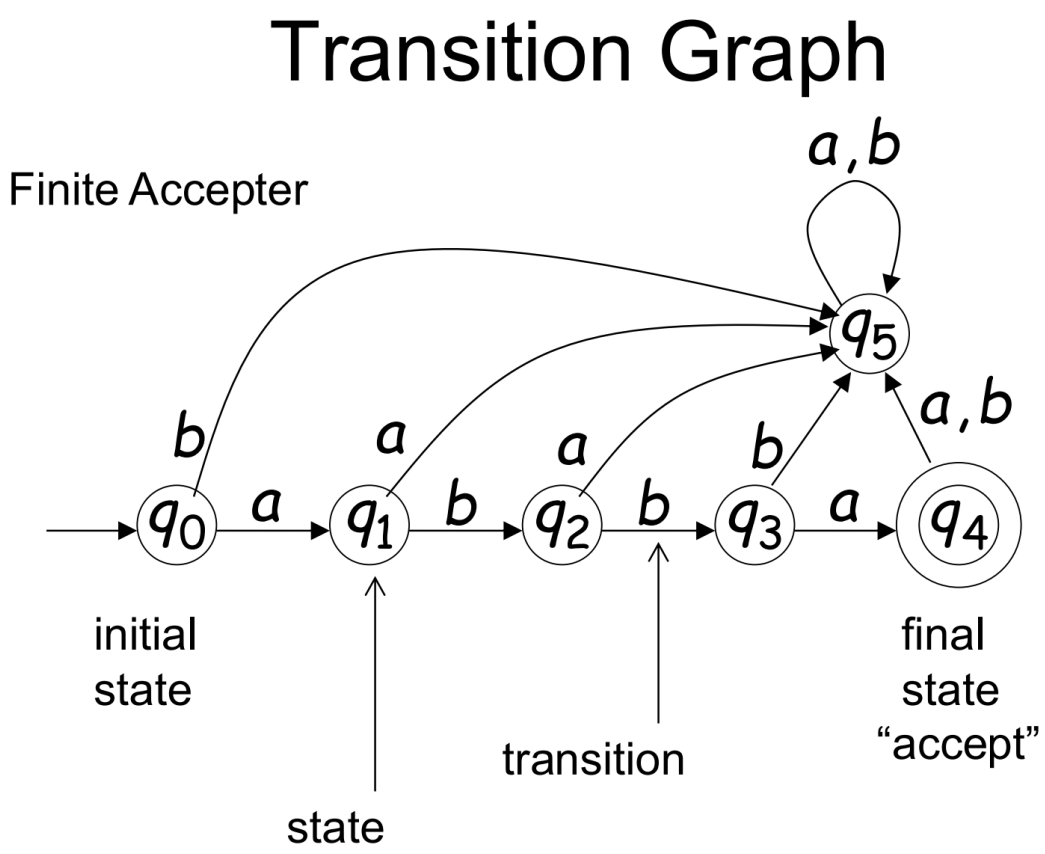
δ	a	b
q_0	q_1	q_5
q_1	q_5	q_2
q_2	q_5	q_3
q_3	q_4	q_5
q_4	q_5	q_5
q_5	q_5	q_5

3.2.1 扩展转移函数

> Extended Transition Function δ^*

计算从一个状态开始，在读取一整个字符串之后，最终会到达哪个状态.

$$\delta^* : Q \times \Sigma^* \to Q$$



$$\begin{aligned}\delta^*(q_0, ab) &= q_2\\ \delta^*(q_0, abba) &= q_4\\ \delta^*(q_0, abbbaa) &= q_5\end{aligned}$$

定理：There is a walk from q to q' with label $w \Leftrightarrow \delta^*(q, w) = q'$.

> 将 "在状态图中沿着一条路径行走" 这个直观行为，与 δ^* 这个数学函数严格等同.

扩展转移函数的递归定义（Recursive Definition）

$$\begin{aligned}\delta^*(q, \lambda) &= q\\ \delta^*(q, w\sigma) &= \delta\left(\delta^*(q, w), \sigma\right)\end{aligned}$$

递归定义通常包含两个部分：一个**基本情况**和一个**递归步骤**.

① 基本情况（Base Case）

$$\delta^*(q, \lambda) = q$$

这条规则是递归的停止点. 如果从状态 q 开始，要处理的字符串是空的，那么哪里也去不了，停在状态 q.

② 递归步骤（Recursive Step）

$$\delta^*(q, w\sigma) = \delta\left(\delta^*(q, w), \sigma\right)$$

这条规则定义了如何处理一个非空字符串. 它把字符串分解成 w, σ 两部分.

- w：字符串的初始部分（除最后一个字符以外的部分），可能是较长的字符串，也可能是空串.
- σ：字符串的最后一个字符.

递归定义的意思是，想要求出 $\delta^*(q, w\sigma)$ 的结果，第一步，递归地求出从 q 开始，只读完 w 部分会到达哪个状态. 这个结果就是 $\delta^*(q, w)$. 第二步，从上一步得到的状态 $\delta^*(q, w)$ 出发，再使用普通的单步转移函数 δ 读取最后一个字符 σ，看看会跳到哪里.

总结：处理长字符串的结果 $=$ 处理前面一小段的结果 $+$ 再走一步.

3.2.2 被 DFA 接受/拒绝的语言

> Language accepted/refused by a DFA

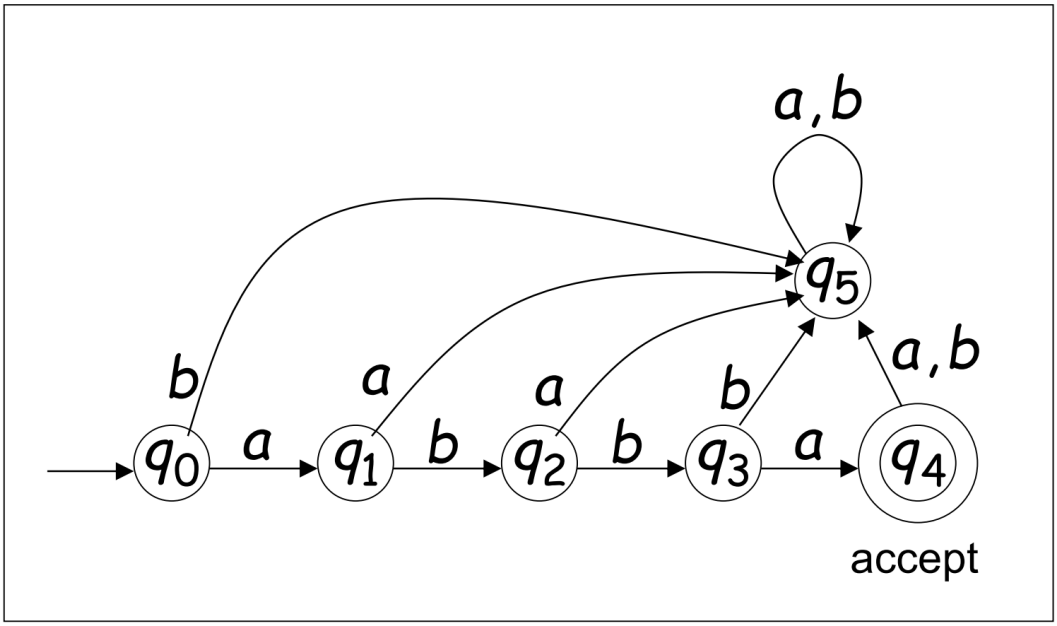
记一个 DFA 为 M.

Definition: The language $L(M)$ contains **all** input strings accepted by M.

> $L(M) = \{\text{strings that drive } M \text{ to a final state}\}.$

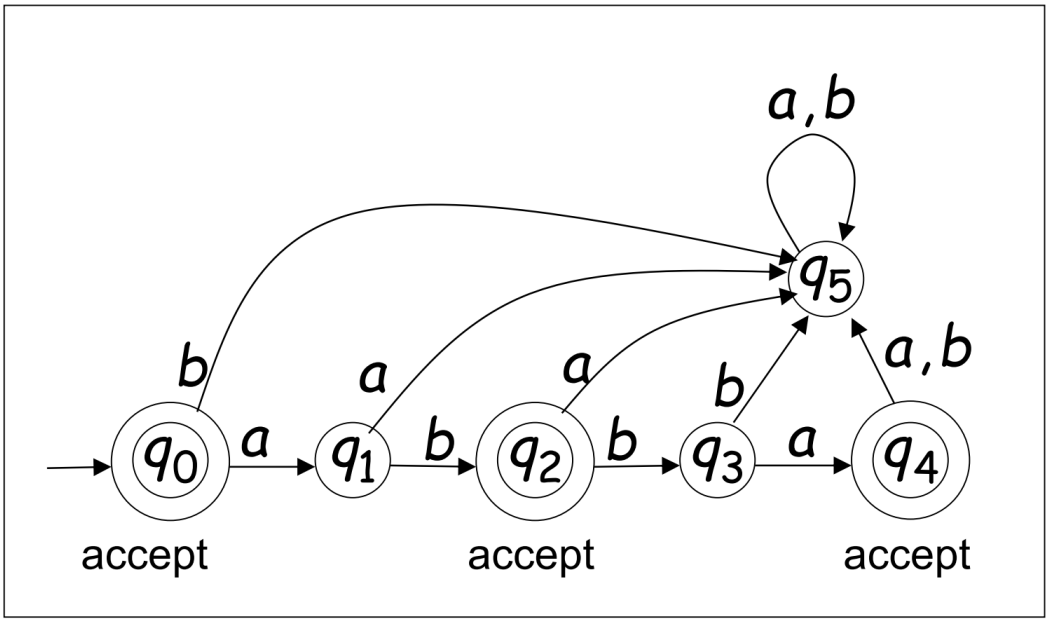
例 1

$L(M)=\{abba\}$ M



例 2

$L(M)=\{\lambda,ab,abba\}$ M



形式化定义：

For a DFA $M=(Q,\Sigma,\delta,q_0,F)$, language accepted by M:

$$L(M)=\{w\in\Sigma^*:\delta^*(q_0,w)\in F\}$$

> 注意，一个被 M 接受的语言，不仅它的所有字符串都能被 M 接受，而且它要包含**所有**被 M 接受的字符串.
>
> 如果只是部分被 M 接受的字符串构成的集合，不能称为被 M 接受的语言；如果语言中有一些字符串被 M 拒绝，也不能称为被 M 接受的语言.

一个 DFA 对应唯一一个语言，但一个语言可以对应多个不同的 DFA. 虽然一个语言可以对应多个 DFA，但对于任何一个正则语言，只存在唯一一个最小化的 DFA 来识别它.

> 见 `3.5 状态最小化` .

Language rejected by M:

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}$$

> 指的是所有不被 M 接受的字符串的集合. 注意被 M 拒绝的语言不是其中某些字符串被拒绝，而是**所有都被拒绝**；且包含了所有被 M 拒绝的字符串.
>
> 如果只是部分被 M 拒绝的字符串构成的集合，不能称为被 M 拒绝的语言；如果语言中有一些字符串被 M 接受，也不能称为被 M 拒绝的语言.

3.2.3 正则语言

> Regular Language

A language L is regular iff there exists some DFA M such that $L = L(M)$.

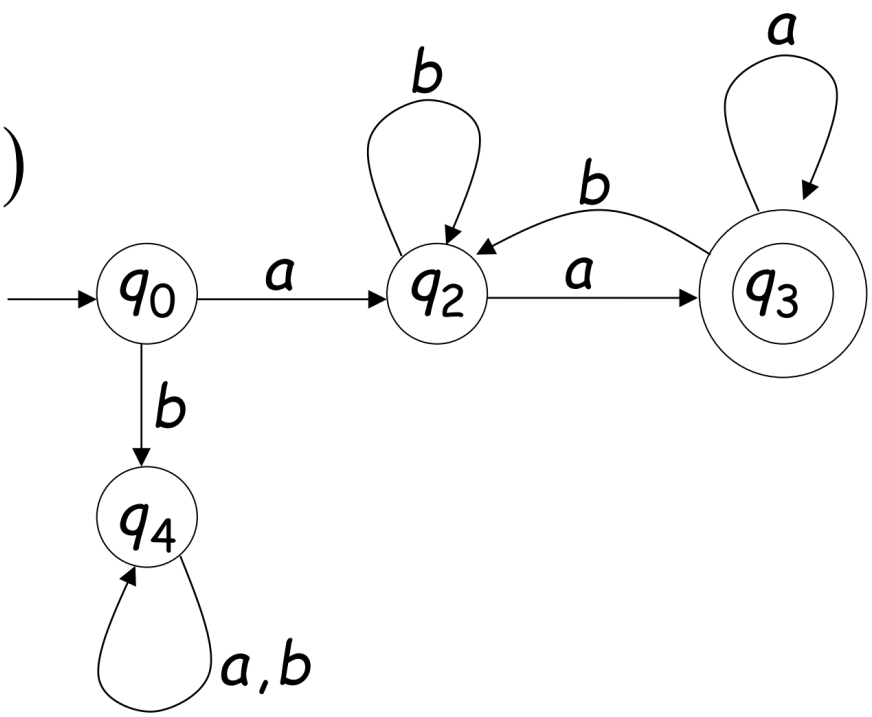
> 注意，定义使用的是 DFA，而不是 NFA.

All regular languages form a language family.

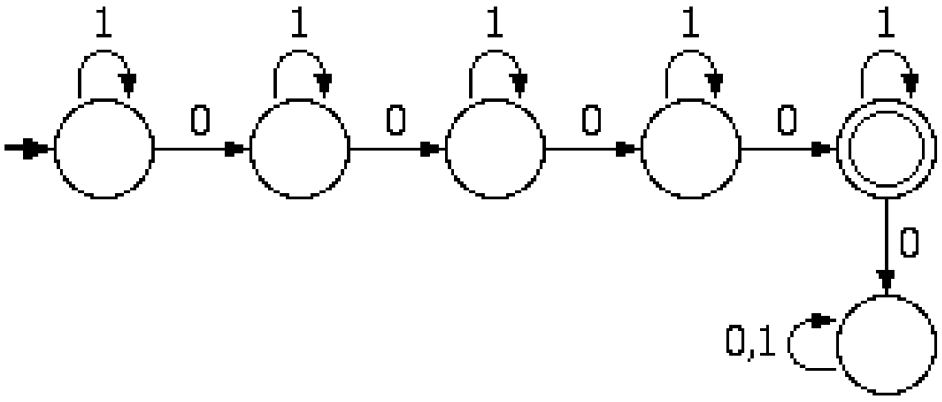
例 1

The language $L = \{awa : w \in \{a,b\}^*\}$ is regular:

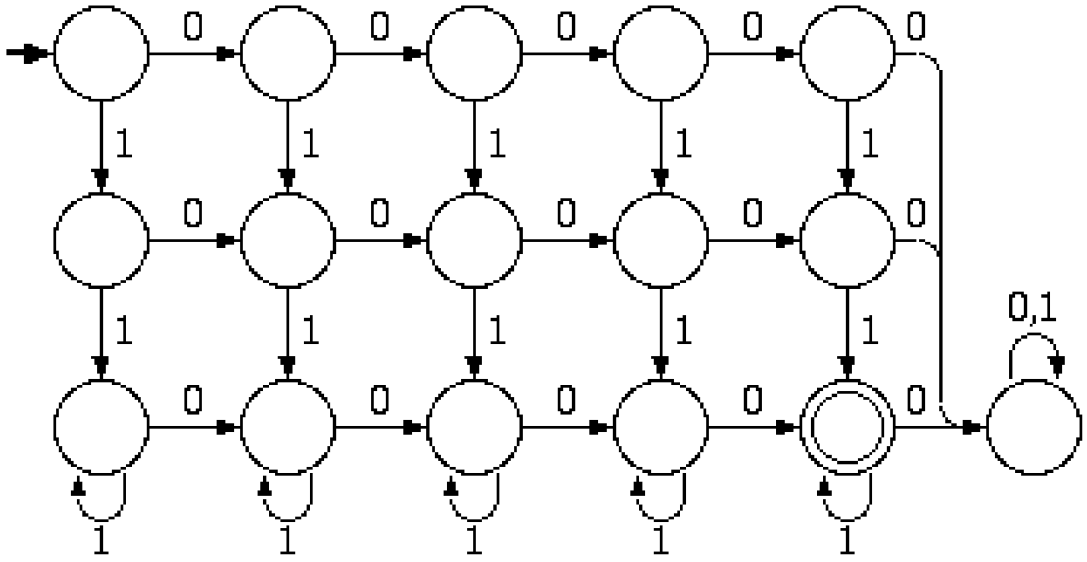
$L = L(M)$



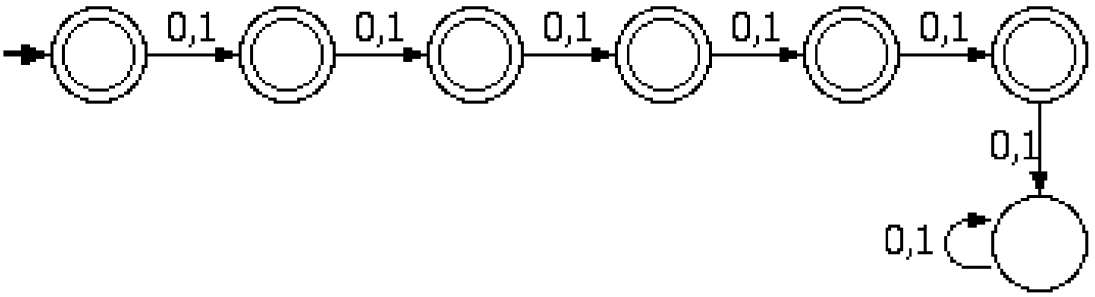
例 2：All strings that contain exactly 4 "0"s.



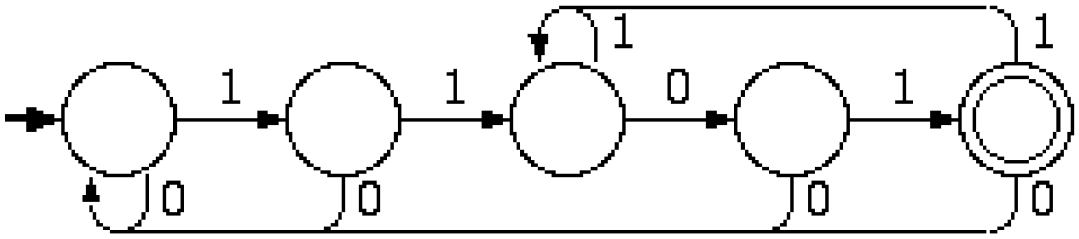
例 3：All strings containing exactly 4 "0" s and at least 2 "1" s.



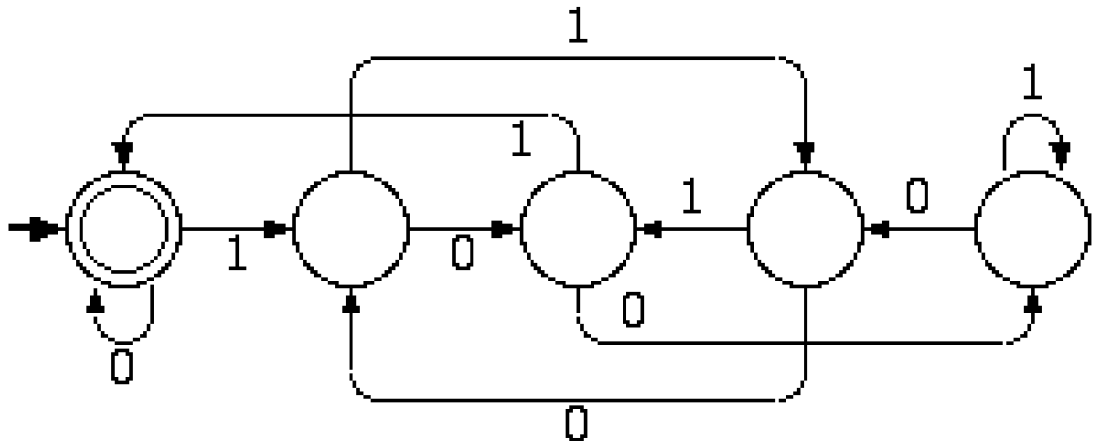
例 4：All strings of length at most five.



例 5：All strings ending in "1101".



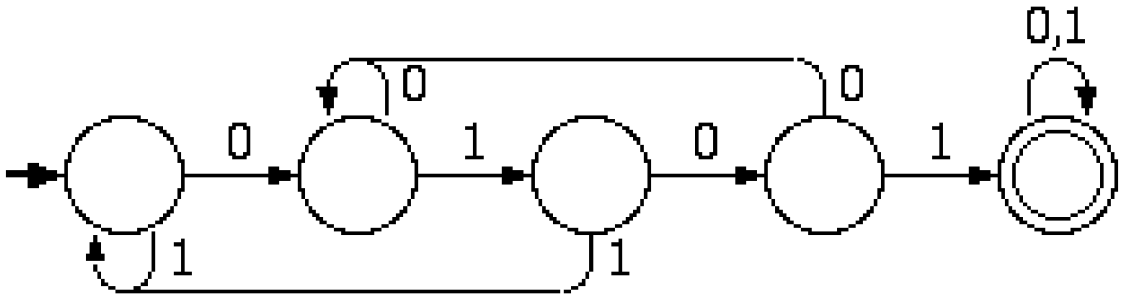
例 6：All strings whose binary interpretation is divisible by 5.



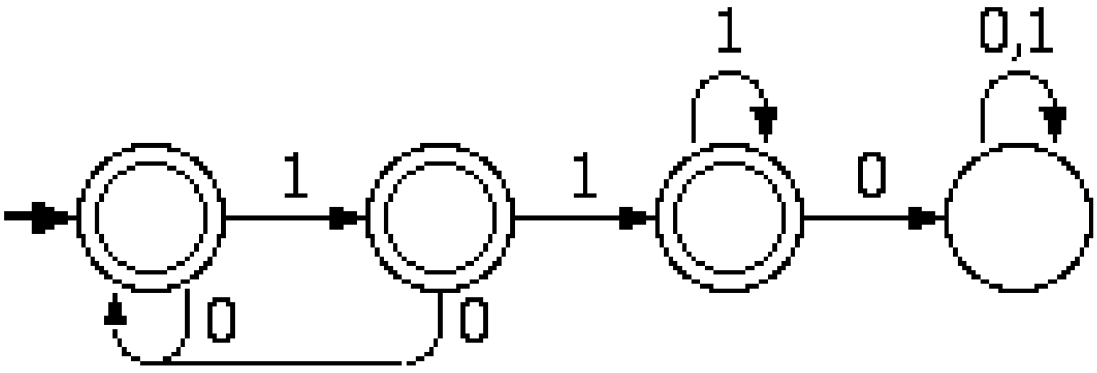
原理：如果一个数 N 除以 5 的余数是 r，那么在它的二进制末尾添加一个比特 b（0 或 1）后，新数的值是 $2N + b$，新数的余数 r' 是 $2r + b \mod 5$.

从左到右分别是余数为 $0, 1, 2, 3, 4$ 的状态.

例 7：All strings that contain the substring 0101.

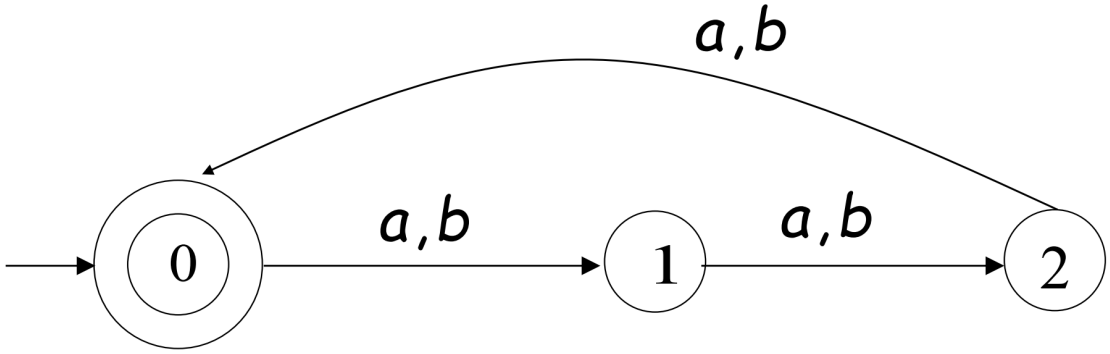


例 8：All strings that don't contain the substring 110.

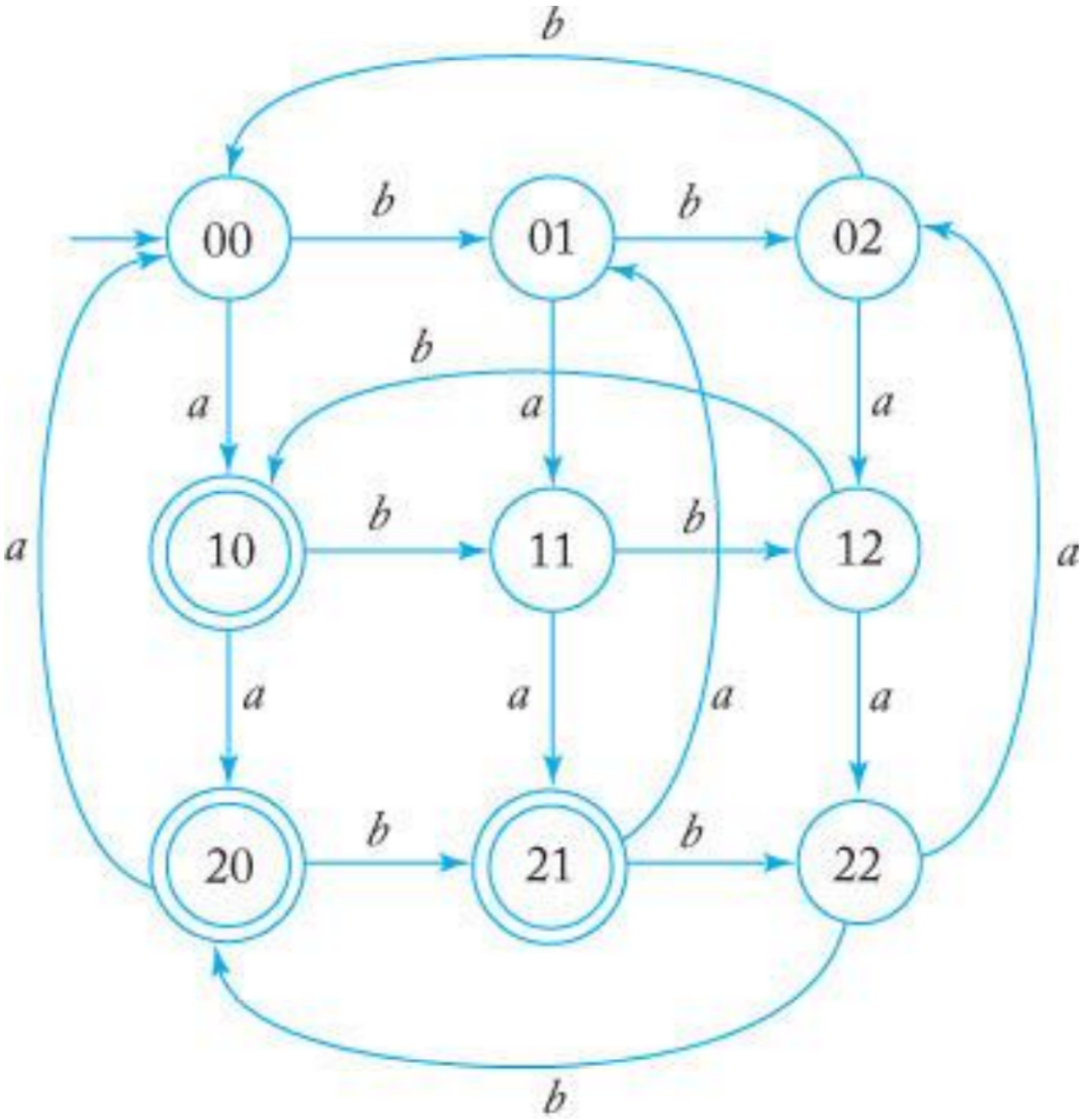


例 9：Find DFAs for the following languages on $\Sigma = \{a, b\}$

(a) $L = \{w : |w| \mod 3 = 0\}$

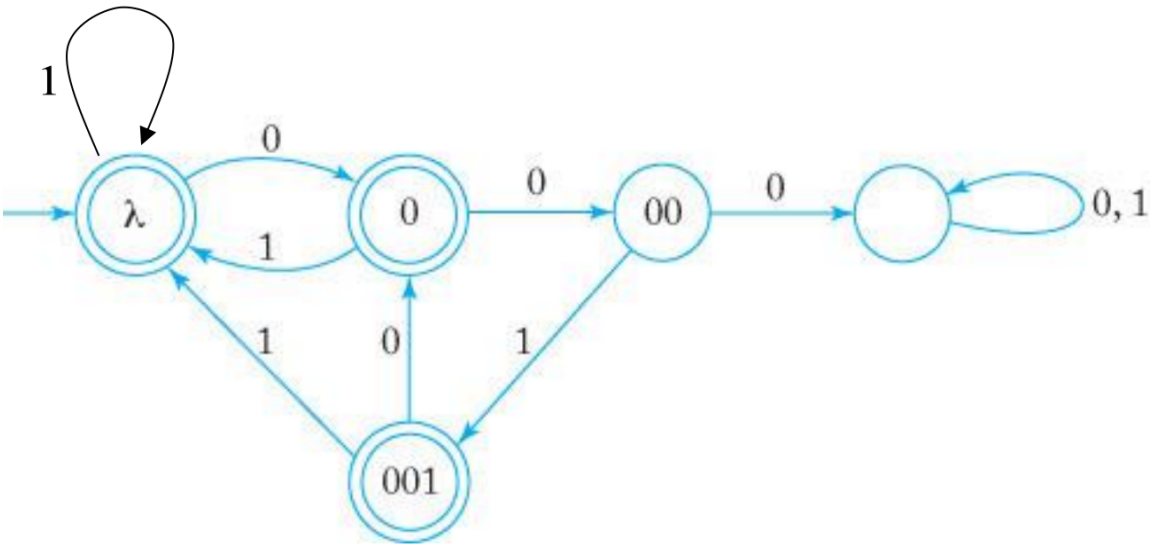


(b) $L = \{w : n_a(w) \mod 3 > n_b(w) \mod 3\}$

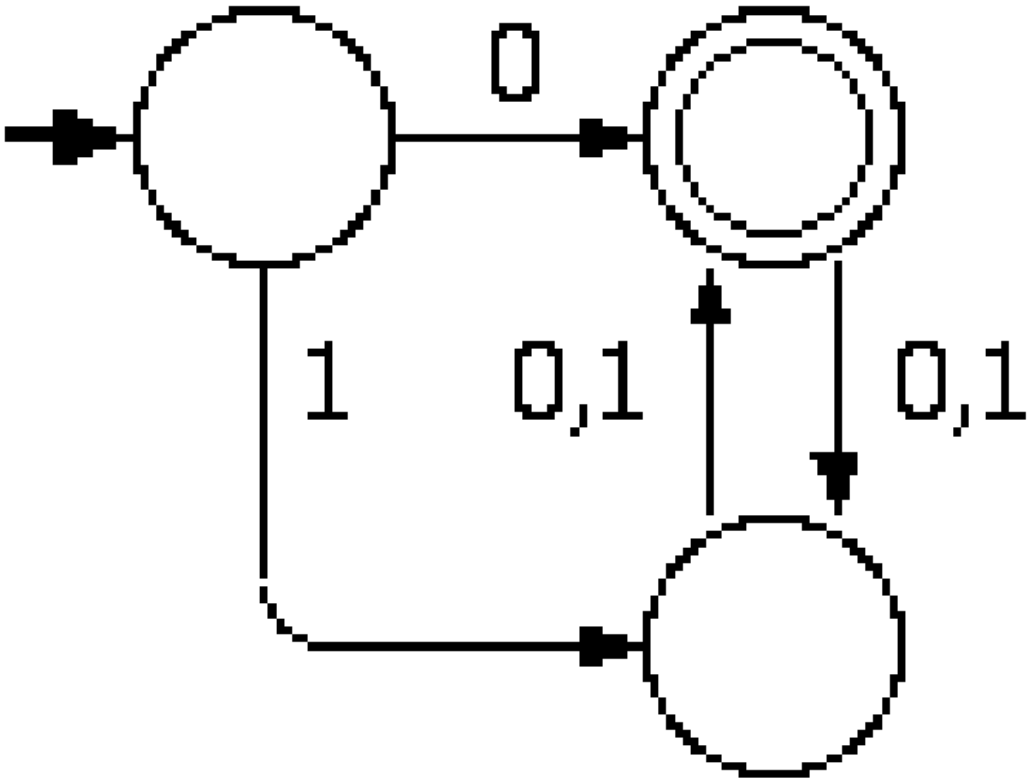


例 10：Every 00 is followed immediately by a 1.

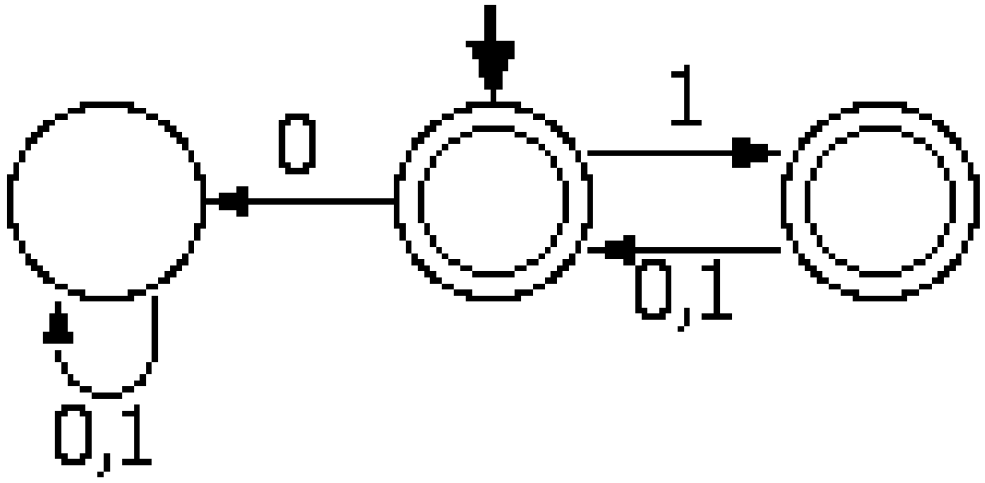
> 注意，不仅拒绝连续的三个 0，而且还拒绝末尾没有接 1 的 00.



例 11：All strings that start with 0 and have odd length or start with 1 and have even length.



例 12：All strings where every odd position is a 1.



3.3 NFA 非确定有限自动机

> Nondeterministic Finite Accepters
>
> Nondeterministic Finite Automaton
>
> NFA
>
> 非确定有限状态自动机 / 非确定有限自动机

Nondeterministic Finite Accepter (NFA) is defined by the 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

其中，

状态集合 Q：a finite set of internal states.

输入字母表 Σ：a finite set of symbols called input alphabet.

转移函数 δ：$Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$ called transition function.

> $Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$ 意思是，它接受一个当前状态和一个输入符号（可以接受空字符进行空转移），然后给出下一个可能状态的集合. The range of δ is in the powerset 2^Q，即 Q 的所有子集作为元素构成的集合，元素也包括空集.
>
> 注意，NFA 的转移函数不需要对所有当前状态加输入给出定义，如果转移函数对某个当前状态和输入符号未定义，则 $\delta(\text{current state}, \text{input}) = \emptyset$ - no transition - the automaton hangs，视为该路线拒绝了该字符串（the input cannot be consumed，还没读完就没地方转移了），但其他路线仍有可能接受.

初始状态 q_0：$q_0 \in Q$ is the initial state

最终状态集合 F：$F \subseteq Q$ is a set of final states

> NFA 接受 / 拒绝字符串判定：若存在一条路线，使得字符串的所有字符都被读取且自动机处于最终状态，则该字符串被该 NFA 接受；若所有路线都无法实现 “字符串的所有字符都被读取且自动机处于最终状态”，则该字符串被该 NFA 拒绝.

> 第二种记法：对于某条特定路线，若字符串所有字符都被读取但不处于最终状态，该字符串在这条路线会被拒绝；若字符串字符无法被全部读取（还没读完就没地方转移了），该字符串在这条路线也会被拒绝. 若字符串所有字符都被读取且处于最终状态，该字符串在这条路线被接受. 只要有一条路线接受字符串，该字符串就视为被该 NFA 接受；如果所有路线都拒绝字符串，则该字符串视为被该 NFA 拒绝.

3.3.1 空转移

> Lambda Transitions
>
> NFA 特有. DFA 不允许空字符作为输入字符.

$$\delta(q, \lambda) = \{q \text{ 的所有可以通过 } \lambda \text{ 边到达的状态，包括 } q \text{ 本身}\}$$

> $q_i \in \delta(q_i, \lambda)$ 恒成立. 读取空字符时一定有留在原地的选择.
>
> 实际上 The λ symbol never appears on the input tape. “读取空字符” 实际上是在上一个字符读取之后，下一个字符读取之前的一个没有任何读取的思考环节，在思考时可以沿着 λ 边跳转，也可以不进行任何操作直接读取下一个字符.

3.3.2 扩展转移函数

> Extended Transition Function δ^*

$q_j \in \delta^*(q_i, w)$: there is a walk from q_i to q_j with label w.

3.3.3 被 NFA 接受 / 拒绝的语言

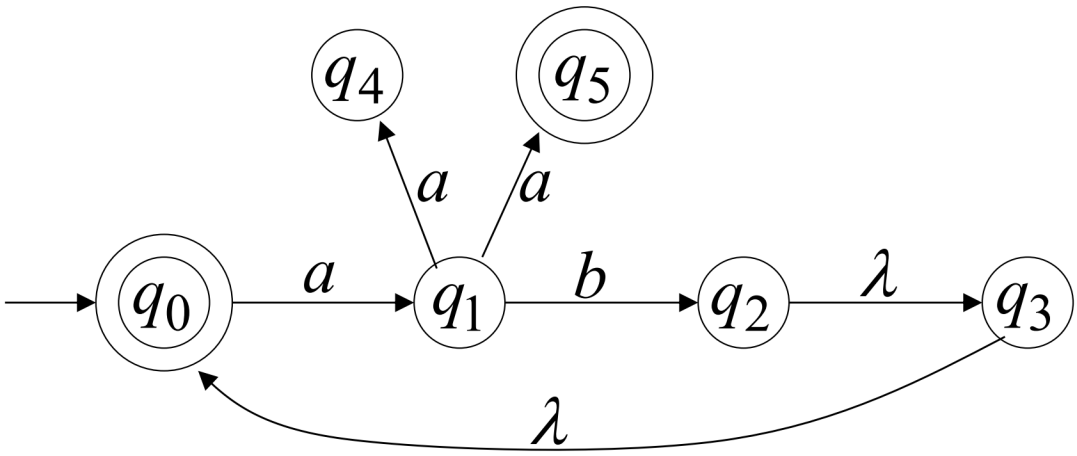
> Language accepted/refused by a NFA

The language L accepted by an NFA M is defined as the set of all accepted strings:

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \emptyset\}$$

> 只要有一条路径能接受字符串，该字符串就视为被 M 接受.

例:



$L(M) = \{ab\}^* \{aa\} \cup \{ab\}^*$

3.4 DFA 与 NFA 的等价性

> Equivalence of DFA and NFA

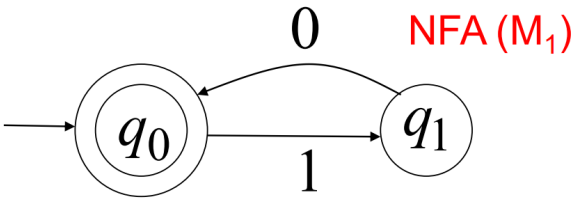
Two finite accepters M_1 and M_2 are said to be equivalent if

$$L(M_1) = L(M_2)$$

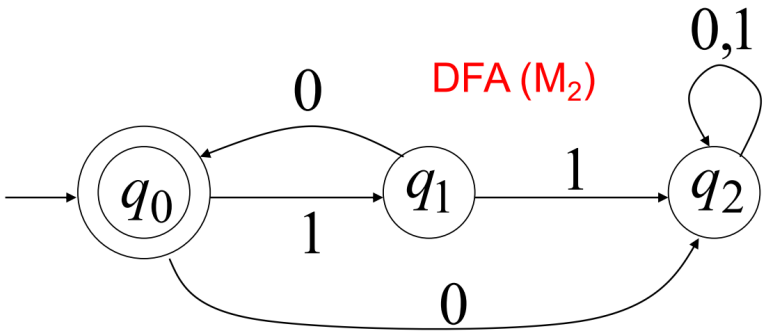
That is, if they both accept the same language.

例:

$$L(M_1) = \{10\}*$$



$$L(M_2) = \{10\}*$$



下面证明 DFA 与 NFA 的等价性. 即证明:

$$\{\text{Languages accepted by NFAs}\} = \{\text{Languages accepted by DFAs}\}$$

> "Languages accepted by DFAs" 即 Regular Languages.
>
> 这里证明的是 DFA 和 NFA 整体上等价，而不是某一个 DFA 和某一个 NFA.
>
> 如果证明它们接受的语言集相同，那么对于任意一个 DFA，都能找到至少一个 NFA 和它接受同一个语言，即这两个 finite accepters 等价；反之亦然.

Step 1：证明

$$\{\text{Languages accepted by NFAs}\} \supseteq \{\text{Languages accepted by DFAs}\}$$

Proof: Every DFA is trivially an NFA (DFA 在定义上全部满足 NFA 的要求，因此 DFA 本身就是一种特定的 NFA) $\Rightarrow$ Any language L accepted by a DFA is also accepted by an NFA.

> DFA 转 NFA 取本身就可以，不需要构建新的.

Step 2：证明

$$\{\text{Languages accepted by NFAs}\} \subseteq \{\text{Languages accepted by DFAs}\}$$

即证明 Any NFA can be converted to an equivalent DFA.

> Any NFA can be converted to an equivalent DFA $\Rightarrow$ Any language L accepted by a NFA is also accepted by an DFA.

Step 3：证明 "Any NFA can be converted to an equivalent DFA".

工程师式证明 - 直接给出转化的方法.

3.4.1 NFA 转 DFA

> Convert NFA to DFA

给定一个 NFA $M = (Q, \Sigma, \delta, q_0, F)$. 下面给出与其等价的 DFA M' 的构造方法.

Step 1：初始化 state of DFA.

假设 M 的初始状态为 q_0. 则 M' 的初始状态为 $\{q_0\}$.

> 注意，这里意思不是 M' 的初始状态为一个集合，而是用这个符号给初始状态打一个标签（标签只是个名字，在构建完整个 DFA 后可以随便换）

Step 2：逐状态构造

For every DFA's state $\{q_i, q_j, \dots, q_m\}$, 根据给定的 NFA 计算：

$$\begin{cases} \delta^*(q_i, a) \\ \delta^*(q_j, a) \\ \quad \vdots \\ \delta^*(q_m, a) \end{cases}$$

全部取并集，得到一个新的集合 $\{q'_i, q'_j, \dots, q'_n\}$.

Add transition to DFA

$$\delta\left(\{q_i, q_j, \dots, q_m\}, a\right) = \{q'_i, q'_j, \dots, q'_n\}$$

Repeat Step 2 for all letters in alphabet, until no more transitions can be added.

注意：这里 $\delta^*(q_i, a), \delta^*(q_j, a), \dots, \delta^*(q_m, a)$ 的结果可能是包含多个状态元素的集合，也可能是不包含任何状态的空集（即 NFA 没有对这一步转移作出定义）. 如果所有全部是空集，取完并集后仍然为空集，但是 DFA 必须对每一个可能的转移都作出定义，因此仍然需要设置 $\delta\left(\{q_i, q_j, \dots, q_m\}, a\right) = \emptyset$, 在图中表示为一个圈里面写着 $\emptyset$ 作为一个状态. 并且，这个状态仍然要对可能的转移进行考虑，而显然 NFA 也没有定义这样的转移，因此这个状态的所有转移都指向自己（相当于一个陷阱状态，不能不标出来）.

Step 3：构造最终状态

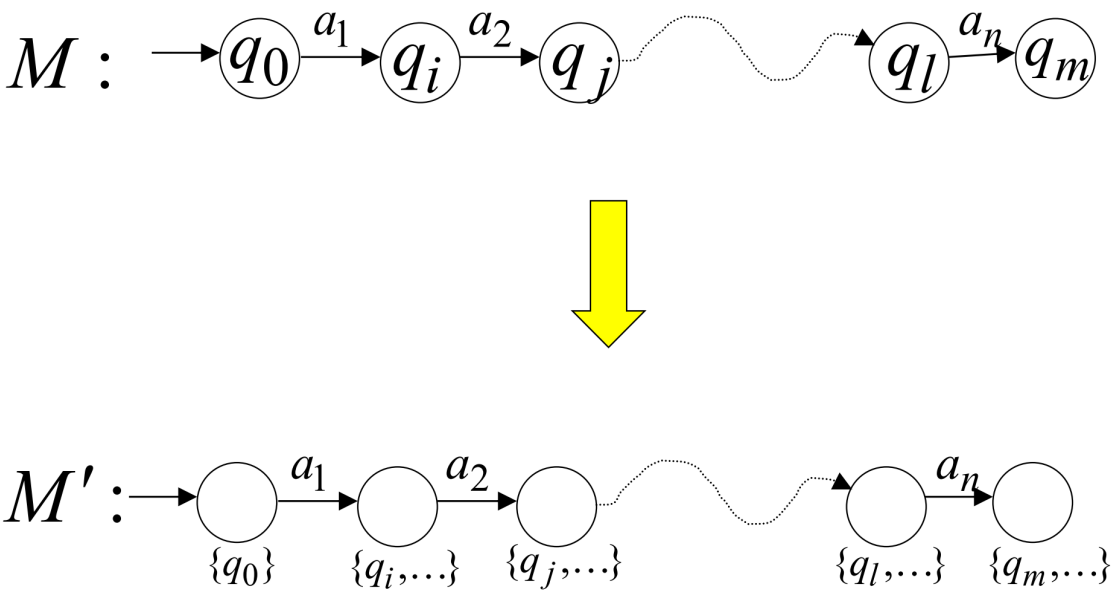
For any DFA state $\{q_i, q_j, \dots, q_m\}$, if some q_j is a final state in the NFA, then, $\{q_i, q_j, \dots, q_m\}$ is a final state in the DFA

以上便是构造方法. 下面证明该方法构造出的 DFA 与原 NFA 等价. 即 Take a NFA M, apply procedure to obtain a DFA M'，证明

$$L(M) = L(M')$$

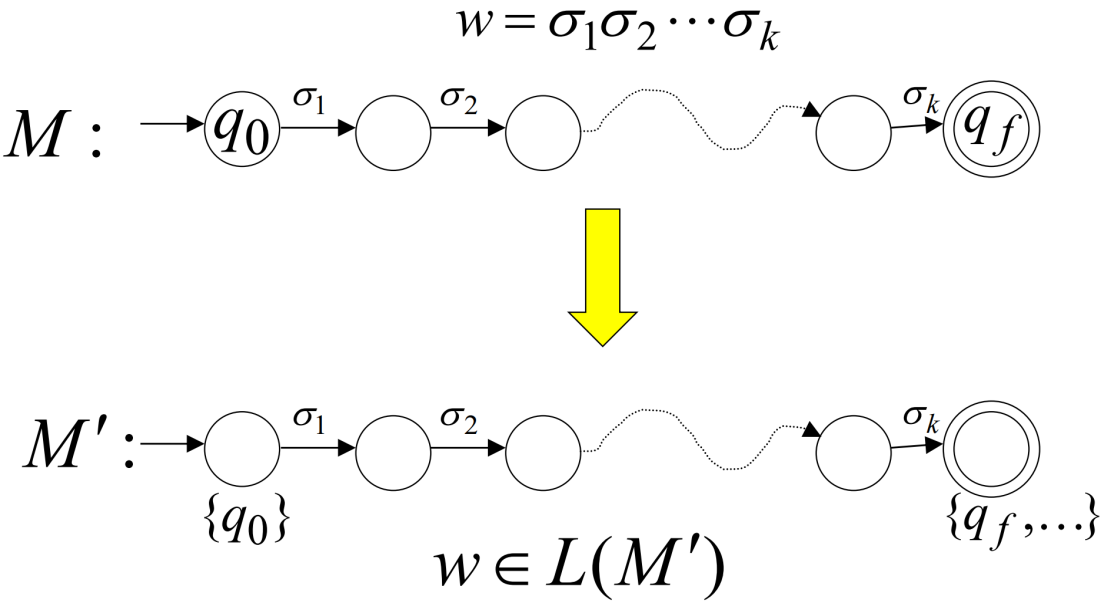
① 证明：$L(M) \subseteq L(M')$.

首先，任取 $v = a_1 a_2 \cdots a_n$, 可证明



> 用数学归纳法，待补充.

因此，对于任意 $w \in L(M)$，有



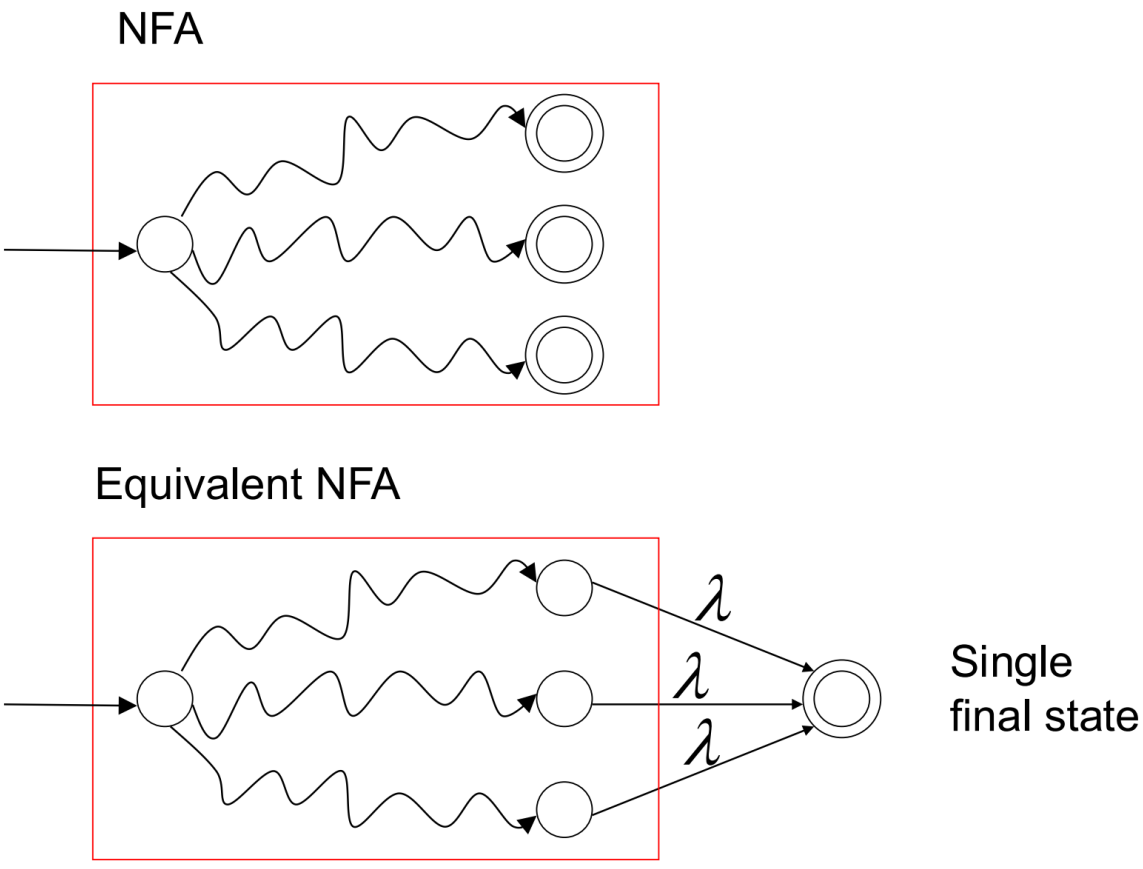
② 证明：$L(M) \supseteq L(M')$.

> 易证，待补充.

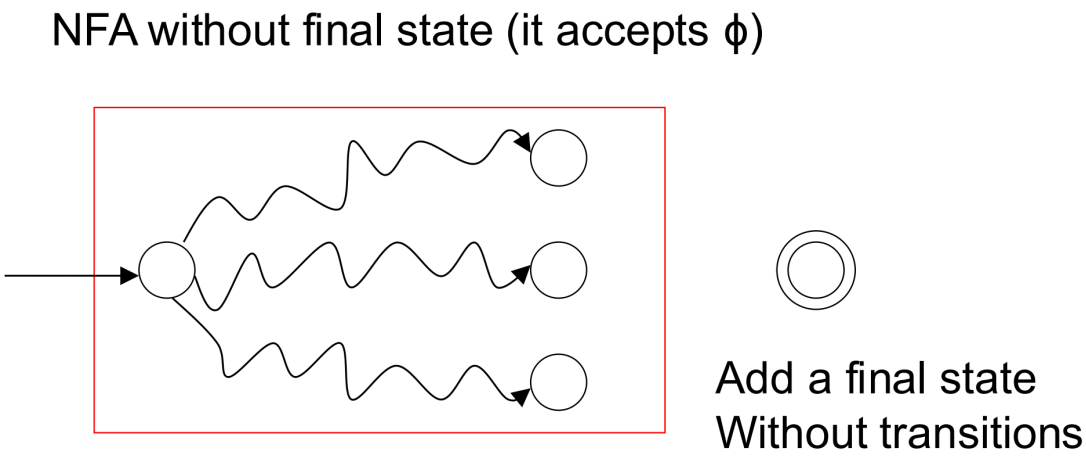
例：待补充.

3.4.2 NFA 转单最终状态

Any NFA can be converted to an equivalent NFA with a single final state.



极端情况



3.5 状态最小化

> Reduction of the Number of States in FA

3.5.1 可区分性

在一个 DFA 中，有两个状态 p 和 q.

无论输入什么字符串，p 和 q 要么同时达到接受状态，要么同时达到非接受状态（表现一致），则称它们是不可区分的（indistinguishable）.

形式化定义

Two states p and q of a DFA are called indistinguishable if

$$\begin{cases}\delta^*(p,w)\in F \Leftrightarrow \delta^*(q,w)\in F\\ \delta^*(p,w)\notin F \Leftrightarrow \delta^*(q,w)\notin F\end{cases}$$

For all $w\in\Sigma^*$.

Lec 3 正则表达式 正则文法

在 `Lec 2 语言 语法 自动机 1.6 语言` 中，给出了语言的定义.

> 语言（Language）定义为 Σ^* 的任意一个子集.

但是语言有很多，对于某个具体的语言，要把它和其他语言作区分，就要规定更详细的表示方法. 例如，

- If a language is finite, you can list all of its strings.

  > $L = \{a, aa, aba, aca\}$

- Descriptive

  > $L = \{x | n_a(x) = n_b(x)\}$

- Using basic Language operations

  > $L = \{aa, ab\}^* \cup \{b\}\{bb\}^*$
  >
  > 正则语言就可以这样表示.

1. 正则表达式

> Regular Expressions (RE)

正则表达式用于描述正则语言. 它是由

- 字母表 Σ 中的字符串
- 圆括号 $()$

  > 圆括号：Parentheses

- 运算符 $+, \cdot, *$

组合而成的符号串.

注意：正则表达式**不是**语言本身，而是**描述**语言的工具.

> It is incorrect to say that for a language L, $L = (a + b + c)^*$
>
> But it's okay to say that L is described by $(a + b + c)^*$

所有正则语言都可以用正则表达式描述.

例：

$$(a + b \cdot c)^* \xrightarrow{\text{描述}} \{a, bc\}^* \\ \Leftrightarrow \{\lambda, a, bc, aa, abc, bca, \ldots\}$$

1.1 概念定义

1.1.1 原始正则表达式

> primitive regular expressions

原始正则表达式（primitive regular expressions）包括

- 空集 $\emptyset$
- 空字符串 λ
- 字母表中的某个符号（如 $a \in \Sigma$）

若 r_1 和 r_2 是正则表达式，则

- $r_1 + r_2$
- $r_1 \cdot r_2$
- r_1^*
- (r_1)

也是正则表达式.

> 这条规则定义了所有非原始正则表达式.
>
> A string is a regular expression iff it can be derived from the primitive regular expressions by a **finite** number of applications of the rules in (2).
>
> 注意，是有限次操作.

例：

正则表达式：$(a + b \cdot c)^* \cdot (c + \emptyset)$

不是正则表达式：

- $(a + b+)$
- a^n
- a^+

1.1.2 正则表达式的语言

> Languages of Regular Expressions

$L(r)$: language of regular expression r

例：

$$L((a+b\cdot c)^*)=\{\lambda,a,bc,aa,abc,bca,\ldots\}$$

For primitive regular expressions,

- $L(\emptyset)=\emptyset$
- $L(\lambda)=\{\lambda\}$
- $L(a)=\{a\}$

For regular expressions r_1 and r_2

- $L(r_1+r_2)=L(r_1)\cup L(r_2)$
- $L(r_1\cdot r_2)=L(r_1)L(r_2)$
- $L(r_1^*)=(L(r_1))^*$
- $L((r_1))=L(r_1)$

> 注意，这七条是正则表达式所表示的语言的递归定义.
>
> 例如，$L(r_1+r_2)=L(r_1)\cup L(r_2)$ 是定义，而不是推导得到的，因此不要问为什么左边等于右边.

例:

$$\begin{aligned}L((a+b)\cdot a^*)&=L((a+b))L(a^*)\\&=L(a+b)L(a^*)\\&=(L(a)\cup L(b))(L(a))^*\\&=(\{a\}\cup\{b\})(\{a\})^*\\&=\{a,b\}\,\{\lambda,a,aa,aaa,\ldots\}\\&=\{a,aa,aaa,aaaa\ldots,b,ba,baa,baaa,\ldots\}\end{aligned}$$

1.1.3 运算符优先级

> Priority of Operators

Star closure (∗) precedes concatenation (·) precedes union (+).

1.1.4 正则表达式等价

Regular expressions r_1 and r_2 are equivalent if $L(r_1) = L(r_2)$

2. 正则表达式与正则语言

> Connection Between REs and Regular Languages

2.1 Kleene 定理

> Kleene Theorem

Regular expressions and Finite Automata are equivalent (w.r.t. the languages they describe/accept)

$$\{\text{Language Described by Regular Expressions}\} = \{\text{Regular Languages}\}$$

For every regular language there is a regular expression;

For every regular expression there is a regular language.

证明：

① For any regular expression r, the language $L(r)$ is regular.

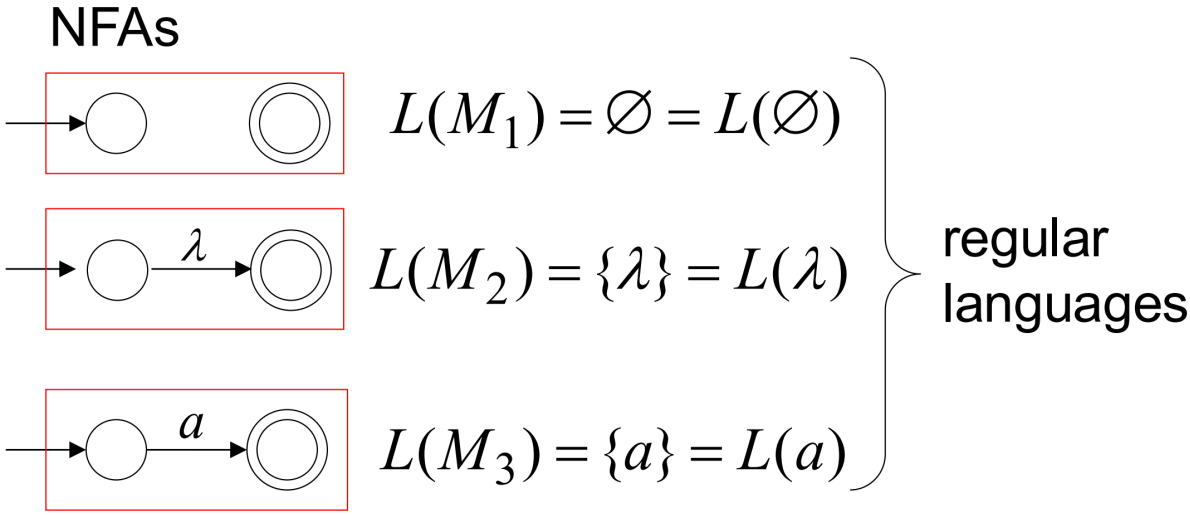
即证 If we have any regular expression r, we can construct an NFA (DFA) that accepts $L(r)$.

证法：对 size of r 进行数学归纳.

Induction Basis

Primitive Regular Expressions: $\emptyset, \lambda, \alpha$.

这三个都很简单，直接给出构建：



Inductive Hypothesis

Assume for regular expressions r_1 and r_2 that $L(r_1)$ and $L(r_2)$ are regular languages.

We will prove:

$$\begin{cases} L(r_1 + r_2) \\ L(r_1 \cdot r_2) \\ L(r_1^*) \\ L((r_1)) \end{cases}$$

are regular languages.

Recall `1.1.2 正则表达式的语言`.

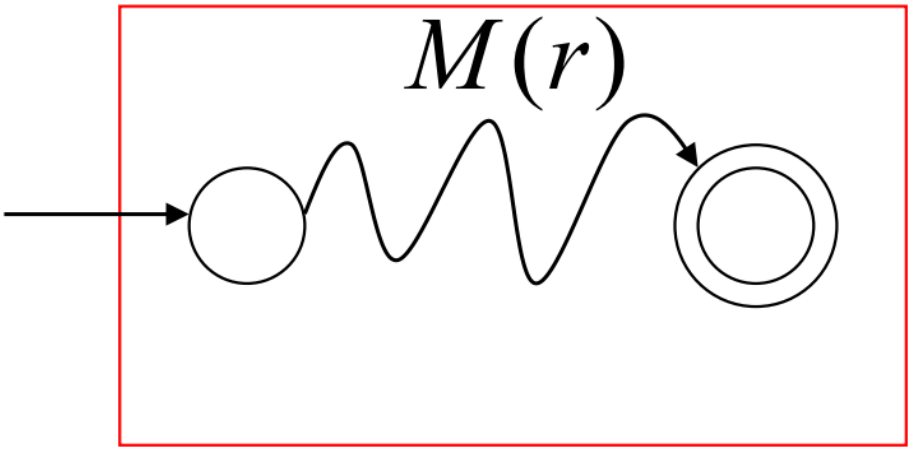
For regular expressions r_1 and r_2

- $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
- $L(r_1 \cdot r_2) = L(r_1)L(r_2)$
- $L(r_1^*) = (L(r_1))^*$
- $L((r_1)) = L(r_1)$

首先易得 $L((r_1)) = L(r_1)$ 是正则语言.

其次，由 `3.4.2 NFA 转单最终状态` 可知，Any NFA can be converted to an equivalent NFA with a single final state.

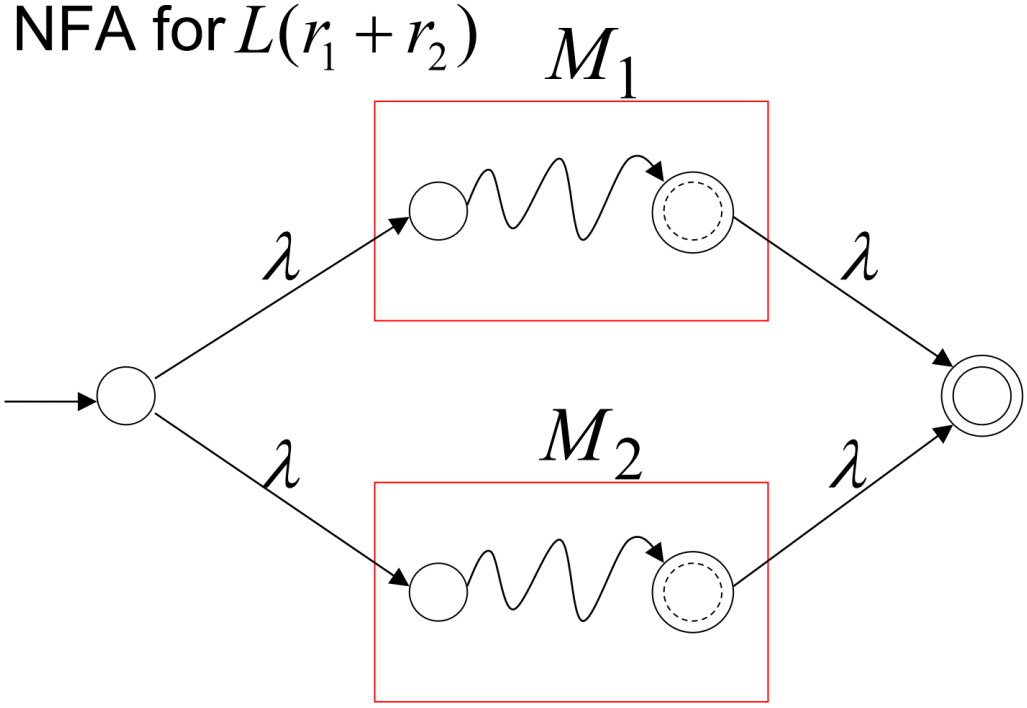
因此，Schematic representation（示意图） of an NFA $M(r)$ accepting $L(r)$ 可以表示为



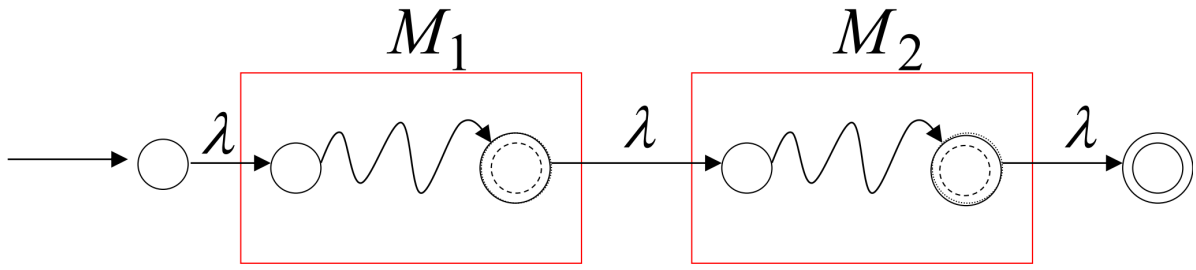
2.1.1 RE $\rightarrow$ NFA：分而治之

> 准确地说是 “找到一个 NFA that accepts RE 所描述的语言".
>
> 核心思想：分而治之. 把复杂的正则表达式 ”分“ 成几个简单的 subproblem，分别求出 subproblem 的 NFA. 然后通过下方给出的三种构造方式来 ”治“，组合 subproblem 的 NFA 得到最终的 NFA.

- NFA for $L(r_1 + r_2)$

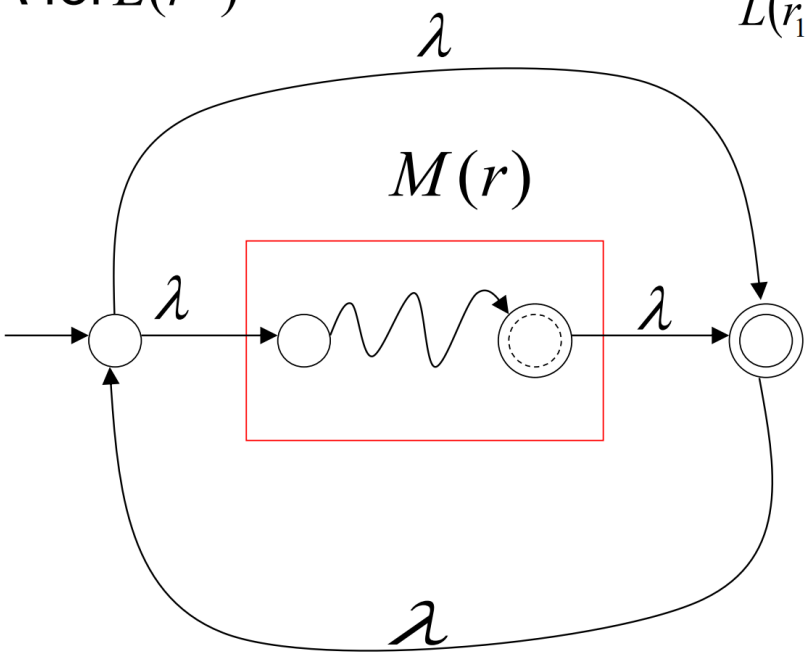


- NFA for $L(r_1 r_2)$



- NFA for $L(r^*)$

$$L(r_1^*) = (L(r_1))^*$$

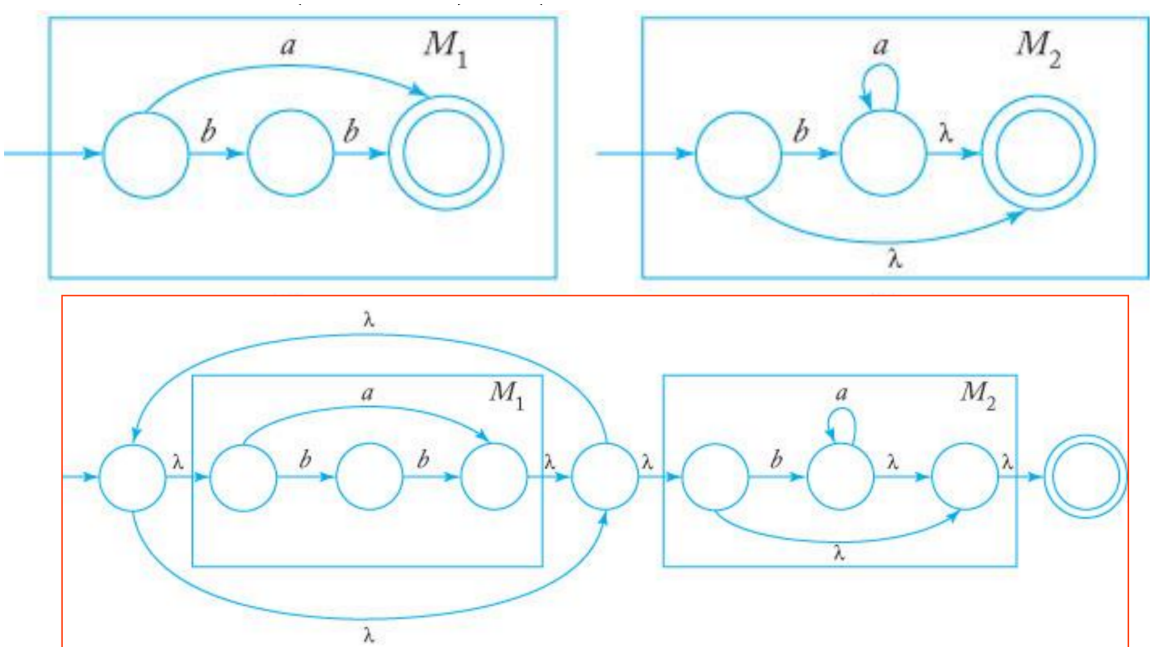


综上，For any regular expression r, the language $L(r)$ is regular.

注意，这里的三种 NFA 构造可以用来出题：

Find an NFA that accepts $L(r)$, where

$$r = (a + bb)^*(ba^* + \lambda)$$

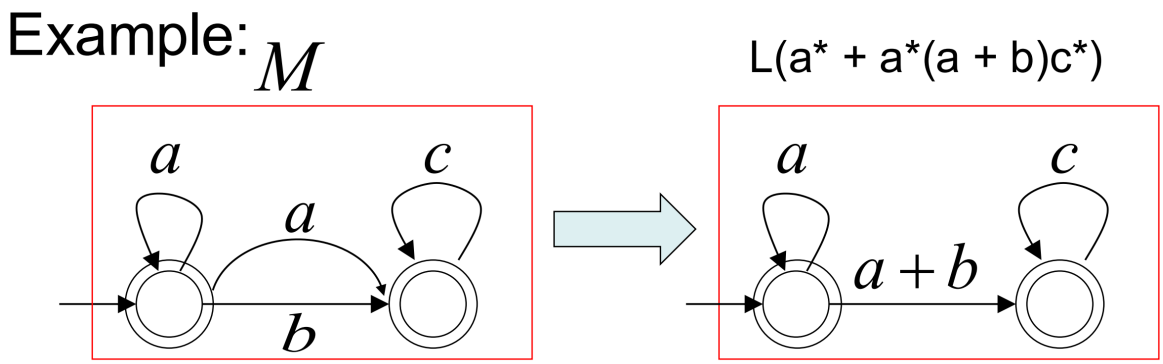


② For any regular language L, there is a regular expression r with $L(r) = L$.

Since any regular language has an associated NFA and hence a transition graph, all we need to do is to find a regular expression capable of generating the labels of all the walks from q_0 to any final state.

2.1.2 NFA $\rightarrow$ RE：广义转移

从一个 NFA M 的转移图可以 construct the equivalent 广义转移图（Generalized Transition Graph），in which transition labels are regular expressions.

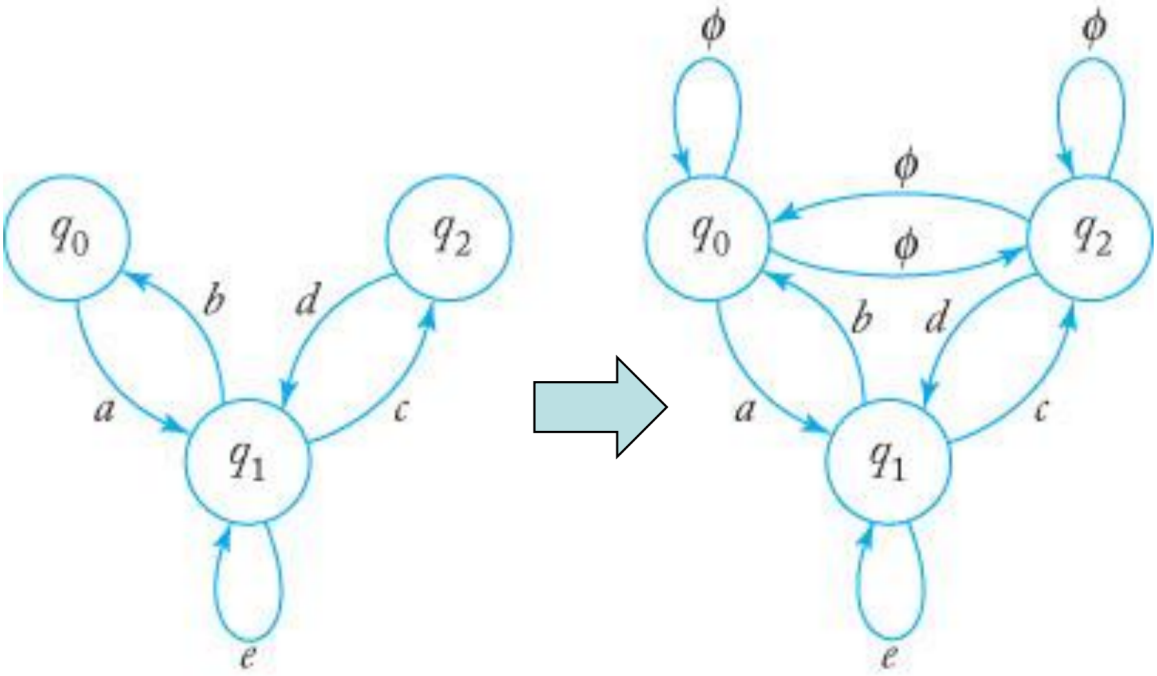


> 注意，狭义的转移图也符合广义转移图的定义，是一种特殊的广义转移图.

GTG 仍然可能有很多状态. 列举所有的 walks 还是很耗时.

下面介绍一个标准流程，把任意 ≥ 3 个状态的转移图简化为 2 个状态的广义转移图.

首先，介绍 Complete GTG：



Complete GTG 要求每个顶点有 1 条指向所有顶点的转移. 如果缺失，用 $\emptyset$ 补齐.

注意与 DFA 的转移图区分：DFA 要求每个状态对所有合法字符（即字母表中的字符）都定义一条转移，a 个状态和 b 种合法字符共 $a \times b$ 条转移的边. 而 Complete GTG 是每个顶点都有指向所有顶点的 1 条转移，包括自己在内（自环）. 未定义的转移用 $\emptyset$ 作标签表示. 因此 $|V|$ 个顶点有 $|V|^2$ 条转移的边. 和合法字符的种数无关.

NFA $\rightarrow$ RE 标准流程：

① Convert the NFA (with single final state) into a complete GTG. Let r_{ij} stand for the label of the edge from q_i to q_j.

② If the GTG has only 2 states with $q_i = q_0$ and $q_j \in F$.

> 注意，这里不讨论其他情况，因为任意 NFA 都可以转化为等效的单 final state 的 NFA.

Its associated RE is

$$r = r_{ii}^* r_{ij} (r_{jj} + r_{ji} r_{ii}^* r_{ij})^*$$

> 这里输出整个流程的结果. 即无论哪种情形都能转化到这里.

③ If the GTG has three states with $q_i = q_0$, $q_j \in F$, and $q_k \in Q$, update edges:

$$\begin{cases} r_{ii} \leftarrow r_{ii} + r_{ik} r_{kk}^* r_{ki} \\ r_{jj} \leftarrow r_{jj} + r_{jk} r_{kk}^* r_{kj} \\ r_{ij} \leftarrow r_{ij} + r_{ik} r_{kk}^* r_{kj} \\ r_{ji} \leftarrow r_{ji} + r_{jk} r_{kk}^* r_{ki} \end{cases}$$

> 剩余 2 个节点 4 条边，通过更新把删去节点的作用加上，实现等效.

> 这里只能删去 初始状态节点 和 最终状态节点 之外的第三个节点.

When this is done, remove vertex q_k and its associated edges.

移除后，剩余 2 个状态，回到步骤 ②.

④ If the GTG has four or more states, pick a state q_k to be removed. Apple step ③ for all pairs of states $(q_i, q_j), i \neq k, j \neq k$. At each update, apply the simplifying rules

$$\begin{cases} r + \emptyset = r \\ r \cdot \emptyset = \emptyset \\ \emptyset^* = \lambda \end{cases}$$

> 注意，$r_1 \emptyset^* r_2 = r_1 \lambda r_2 = r_1 r_2$ 可以由 $\emptyset^* = \lambda$、`1.1.2 正则表达式的语言` 的递归定义、`1.1.4 正则表达式等价` 得到.
>
> $L(r_1\lambda) = L(r_1)L(\lambda) = L(r_1)\{\lambda\} = L(r_1)$，因为对于任意字符串 $w = a_1 a_2 \dots a_n, a_i \in \Sigma$，有 $\lambda w = w\lambda = w$.
>
> 因为 $L(r_1\lambda) = L(r_1)$，所以 $r_1\lambda = r_1$.

wherever possible. When this is done, remove state q_k.

> 注意，在一次 remove 中，所有自环 r_{ii} 只需要更新一次.
>
> 因为 $r_{ii} \leftarrow r_{ii} + r_{ik} r_{kk}^* r_{ki}$ 与另一个节点 j 的选取无关.
>
> 若删除后剩余 $|V|$ 个节点，则只需要进行 $|V|^2$ 次更新计算.
>
> 注意，这里选取的删除节点必须是 初始状态节点 和 最终状态节点 之外的某个节点.

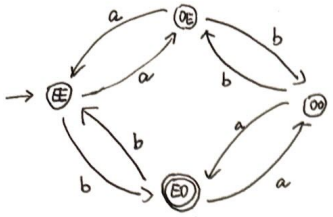
⑤ 重复 ④，直到剩余 3 个状态，回到步骤 ③.

通过该流程，任意正则语言可以求出描述它的正则表达式. 即 For any regular language L, there is a regular expression r with $L(r) = L$.

> 先作出接受 L 的 NFA 的转移图，然后按流程转为 RE.

例：

$L = \{w \in \{a,b\}^* : n_a(w) \text{ is even and } n_b(w) \text{ is odd}\}$. 求描述 L 的 RE.

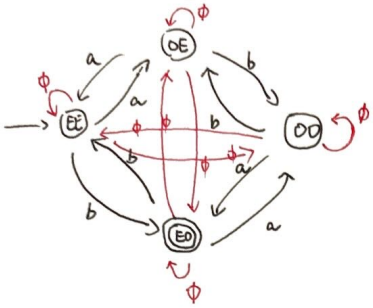


Step 0: 作出 NFA that accepts L 的 Transition Graph.

注意：要有 1 个 初 (EE) 和 1 个 final state (EO).

↓

若有多个，都为 1 个 (用空转移).



Step 1: NFA → Complete GTG

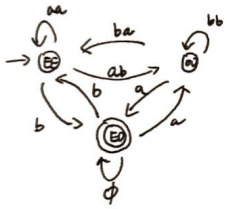
Step 2: Delete OE.

交叉 $\gamma_{EE\to OO} \leftarrow \gamma_{EE\to OO} + \gamma_{EE\to OE}\,\gamma^*_{OE\to OE}\,\gamma_{OE\to OO} = a\phi^*b = ab$

交叉 $\gamma_{OO\to EE} \leftarrow \gamma_{OO\to EE} + \gamma_{OO\to OE}\,\gamma^*_{OE\to OE}\,\gamma_{OE\to EE} = b\phi^*a = ba$

自环 $\gamma_{EE\to EE} \leftarrow \gamma_{EE\to EE} + \gamma_{EE\to OE}\,\gamma^*_{OE\to OE}\,\gamma_{OE\to EE} = \phi + a\phi^*a = aa$

自环 $\gamma_{OO\to OO} \leftarrow \gamma_{OO\to OO} + \gamma_{OO\to OE}\,\gamma^*_{OE\to OE}\,\gamma_{OE\to OO} = \phi + b\phi^*b = bb$



EE 交叉 only $\gamma_{EE\to EO} \leftarrow \gamma_{EE\to EO} + \gamma_{EE\to OE}\,\gamma^*_{OE\to OE}\,\gamma_{OE\to EO} = b + a\phi^*\phi = b$

EE 自环无需再算 $\gamma_{EE\to EE} \leftarrow \gamma_{EE\to EE} + \gamma_{EO\to OE}\,\gamma^*_{OE\to OE}\,\gamma_{OE\to EE} = b + \phi\phi^*a = b$

EO 新自环 $\gamma_{EO\to EO} \leftarrow \gamma_{EO\to EO} + \gamma_{EO\to OE}\,\gamma^*_{OE\to OE}\,\gamma_{OE\to EO} = \phi + \phi\phi^*\phi = \phi$

EO、EO 交叉 only
$\gamma_{EO\to OO} \leftarrow \gamma_{EO\to OO} + \gamma_{EO\to OE}\,\gamma^*_{OE\to OE}\,\gamma_{OE\to OO} = a + \phi\phi^*b = a$
$\gamma_{OO\to EO} \leftarrow \gamma_{OO\to EO} + \gamma_{OO\to OE}\,\gamma^*_{OE\to OE}\,\gamma_{OE\to EO} = a + b\phi^*\phi = a$

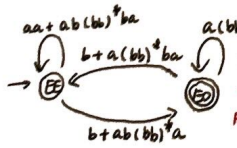
Step 3: Delete OO (起始和最终之外)

$\gamma_{EE\to EO} \leftarrow \gamma_{EE\to EO} + \gamma_{EE\to OO}\,\gamma^*_{OO\to OO}\,\gamma_{OO\to EO} = b + ab(bb)^*a$

$\gamma_{EO\to EE} \leftarrow \gamma_{EO\to EE} + \gamma_{EO\to OO}\,\gamma^*_{OO\to OO}\,\gamma_{OO\to EE} = b + a(bb)^*ba$

$\gamma_{EE\to EE} \leftarrow \gamma_{EE\to EE} + \gamma_{EE\to OO}\,\gamma^*_{OO\to OO}\,\gamma_{OO\to EE} = aa + ab(bb)^*ba$

$\gamma_{EO\to EO} \leftarrow \gamma_{EO\to EO} + \gamma_{EO\to OO}\,\gamma^*_{OO\to OO}\,\gamma_{OO\to EO} = \phi + a(bb)^*a = a(bb)^*a$



因为是新的一轮 remove. 所以要重新算自环

只剩 2 个 states

Step 4: 根据 2 states 的 Complete GTG 直接写出 RE:

$$\gamma = \gamma^*_{EE\to EE}\,\gamma_{EE\to EO}\left(\gamma_{EO\to EO} + \gamma_{EO\to EE}\,\gamma^*_{EE\to EE}\,\gamma_{EE\to EO}\right)^*$$

$$= (aa + ab(bb)^*ba)^*\,(b + ab(bb)^*a)\left(a(bb)^*a + (b + a(bb)^*ba)\,(aa + ab(bb)^*ba)^*(b + ab(bb)^*a)\right)^*$$

3. 正则文法

> Regular Grammars

Recall `Lec 2 语言 语法 自动机 2.语法`:

A grammar G is defined as a 4-tuple:

$$G = (V, T, S, P)$$

where

- V is a finite set of variables.
- T is a finite set of terminals.
- $S \in V$ called start variable.
- P is a finite set of production rules.

Let $G = (V, T, S, P)$ be a grammar. Then the set

$$L(G) = \{w \in T^* : S \overset{*}{\Rightarrow} w\}$$

is the language generated by G.

If $w \in L(G)$, then the sequence

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n \Rightarrow w$$

is a derivation of the **sentence** w.

$S, w_1, w_2, \ldots, w_n$ are called sentential forms.

3.1 线性文法

> Linear Grammars

产生式右侧至多只有一个变量的文法.

Grammars with at most one variable at the right side of a production.

注意，线性文法可能生成正则语言，也可能生成非正则语言. 它的范围比正则文法更广.

例 1:

$$\begin{aligned} G &= (\{S\}, \{a, b\}, S, P), \\ P : S &\to aSb, \\ S &\to \lambda \end{aligned}$$

例 2:

$$
\begin{aligned}
&G = (\{S, A\}, \{a, b\}, S, P), \\
P: &S \to Ab, \\
&A \to aAb, \\
&A \to \lambda
\end{aligned}
$$

例 3:

$$
\begin{aligned}
&G = (\{S, A, B\}, \{a, b\}, S, P), \\
P: &S \to A, \\
&A \to aB \mid \lambda, \\
&B \to Ab
\end{aligned}
$$

$L(G) = \{a^n b^n : n \geq 0\}$

反例：A Non-Linear Grammar

$$
\begin{aligned}
&G = (\{S\}, \{a, b\}, S, P), \\
P: &S \to SS, \\
&S \to \lambda, \\
&S \to aSb, \\
&S \to bSa
\end{aligned}
$$

$L(G) = \{w : n_a(w) = n_b(w)\}$

> 证明见 `Lec 2 语言 语法 自动机 2.3 例 3`

3.2 右线性文法

> Right-Linear Grammars

所有产生式具有 $A \to wB$ 或 $A \to w$ 的形式，其中 w 是终结符串（string of terminals）.

例:

$$
\begin{aligned}
&G = (\{S\}, \{a, b\}, S, P), \\
P: &S \to abS, \\
&S \to a
\end{aligned}
$$

3.3 左线性文法

> Left-Linear Grammars

所有产生式具有 $A \to Bw$ 或 $A \to w$ 的形式，其中 w 是终结符串（string of terminals）.

例:

$$
\begin{aligned}
&G = (\{S, A, B\}, \{a, b\}, S, P),\\
P: &S \to Aab,\\
&A \to Aab \mid B,\\
&B \to a
\end{aligned}
$$

3.4 正则文法

> Regular grammar

A regular grammar is either right-linear or left-linear grammar.

A regular grammar is always linear, but not all linear grammars are regular.

Theorem：任何正则文法都能生成一个正则语言；任何正则语言都可以找到一个生成它的正则文法.

$$\{\text{Languages Generated by Regular Grammars}\} = \{\text{Regular Languages}\}$$

证明分为两步.

第一步：证明 “任何正则文法都能生成一个正则语言”；

见 `3.4.1 正则文法生成正则语言` .

第二步：证明 “任何正则语言都可以找到一个生成它的正则文法”.

见 `3.4.2 正则语言一定可以由正则文法生成` .

3.4.1 正则文法生成正则语言

$$\{\text{Languages Generated by Regular Grammars}\} \subseteq \{\text{Regular Languages}\}$$

The language $L(G)$ generated by any regular grammar G is a regular language.

分类讨论：

① 右线性文法

> The case of Right-Linear Grammars

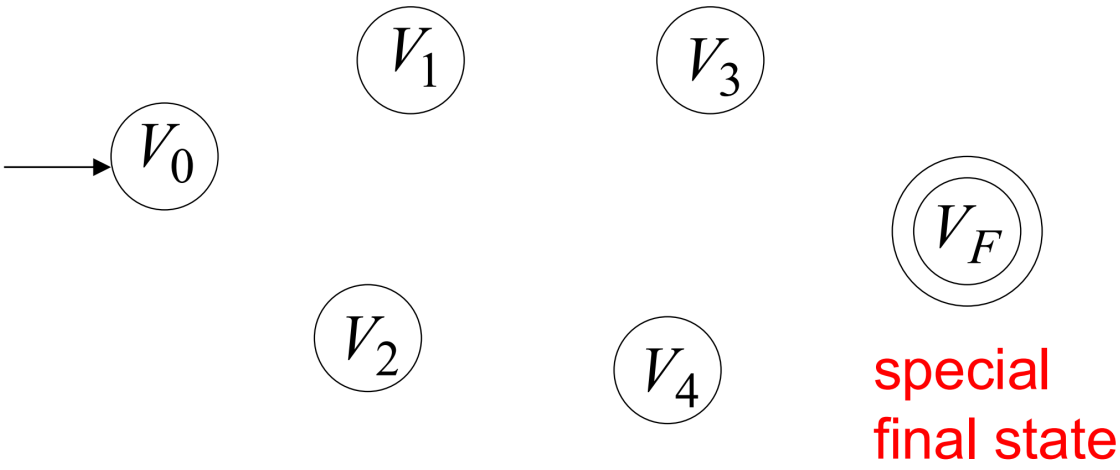
Let G be a right-linear grammar.

We will prove: $L(G)$ is regular.

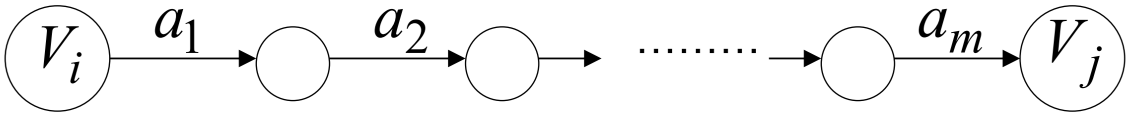
思路：构建一个 NFA M with $L(M) = L(G)$.

构建流程：

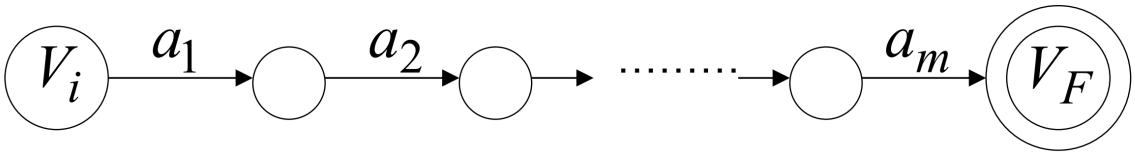
考虑 a right-linear grammar G has variables $V_0, V_1, V_2, \ldots$ and productions $V_i \to a_1 a_2 \cdots a_m V_j$ or $V_i \to a_1 a_2 \cdots a_m$. We construct the NFA M such that each variable V_i corresponds to a node (state).



For each production $V_i \to a_1 a_2 \cdots a_m V_j$, we add transitions and intermediate nodes



For each production $V_i \to a_1 a_2 \cdots a_m$, we add transitions and intermediate nodes



② 左线性文法

> The case of Left-Linear Grammars

Let G be a left-linear grammar.

We will prove: $L(G)$ is regular.

思路：构建一个 right-linear grammar G' with $L(G) = L(G')^R$

> 这里的上标 R 指反转/逆序（Reverse）.
>
> 见 `Lec 2 语言 语法 自动机 1.7.2 逆序`.

然后有：

$$\begin{aligned} & L(G') \text{ is Regular Language} \\ \Rightarrow\ & L(G')^R \text{ is Regular Language} \\ \Rightarrow\ & L(G) \text{ is Regular Language} \end{aligned}$$

> 第一个箭头，证明见 `Lec 4 正则语言的封闭性 1.4 Reverse`

构建流程：

$$\text{Left-linear } G \begin{cases} A \to Bw \\ A \to w \end{cases} \Rightarrow \text{Right-linear } G' = \begin{cases} A \to w^R B \\ A \to w^R \end{cases}$$

易得 $L(G) = L(G')^R$.

3.4.2 正则语言一定可以由正则文法生成

$$\{\text{Languages Generated by Regular Grammars}\} \supseteq \{\text{Regular Languages}\}$$

思路：先写出接受该正则语言的 NFA（因为正则，NFA 一定存在，但这步没有标准流程，通常凭感觉写出），然后从 NFA 构造正则文法.

NFA 构造正则文法流程：

Let M be the NFA with $L = L(M)$. Construct from M to a regular grammar G such that $L(M) = L(G)$.

For any transition $q_i \xrightarrow{a} q_j$, add production $A_i \to aA_j$.

For any final state q_f, add production $A_f \to \lambda$.

> 统一构造成右线性文法.
>
> 这里 q_i, q_j, q_f 是 NFA 的状态，A_i, A_j, A_f 是语法的变量.
>
> $\xrightarrow{a}$ 的 a 是 q_i 接受合法字符 a 后转移到 q_j，aA_j 的 a 是产生式中的终结符.

Since G is right-linear grammar, G is also a regular grammmar, with $L(G) = L(M) = L$.

注意，由于一个语言可以由不同的语法生成，因此上述证明过程只证明了对于任意正则语言 L，至少存在一个正则文法 G 使得 $L(G) = L$. 而不能证明所有生成 L 的语法都是正则的. 事实上，L 即能由正则语法生成，也可能由线性但非正则的文法生成.

例：

$$L = \{a^n \mid n \geq 1\}$$

可以由正则文法（右线性）

$$\begin{aligned} G &= (\{S\}, \{a\}, S, P), \\ P &: S \to aS \mid a \end{aligned}$$

生成. 也可以由线性但非正则的文法

$$\begin{aligned} G &= (\{S, A\}, \{a\}, S, P), \\ P &: S \to aA, \\ & \quad A \to Aa \mid \lambda \end{aligned}$$

生成.

4. 正则语言的四种描述

> 4 Standard Representations of Regular Languages

We now have several ways of describing regular languages:

- DFA
- NFA
- RE
- RG

以 NFA 为桥梁，这四个可以互相转换：

DFA 转 NFA：取自己.

NFA 转 DFA：见 `Lec 2 语言 语法 自动机 3.4.1 NFA 转 DFA`.

RE 转 NFA：见 `2.1.1 分而治之`.

NFA 转 RE：见 `2.1.2 广义转移`.

RG 转 NFA：见 `3.4.1 正则文法生成正则语言`.

NFA 转 RG：见 `3.4.2 正则语言一定可以由正则文法生成`.

When we say: we are given a Regular Language L

We mean: Language L is in a standard representation

Question: Given regular language L, and string w, how can we check if $w \in L$?

Answer: Take the DFA that accepts L and check if w is accepted

Question: Given regular language L, how can we check if L is empty ($L = \emptyset$)?

Answer: Take the DFA that accepts L, check if there is any path from the initial state to a final state

Question: Given regular language L, how can we check if L is finite?

Answer: Take the DFA that accepts L, check if there is a walk with **cycle** from the initial state to a final state

Question: Given regular languages L_1 and L_2, how can we check if $L_1 = L_2$?

Answer: Find if $(L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2) = \emptyset$

当我们用集合论定义语言（即用集合描述），要证明这个语言正则，要靠直觉写出四种描述的任意一种（通常写 RE 或 NFA）. 如果写不出来，则可以尝试使用泵引理来证明这个语言不是正则.

> 见 `Lec 4 2.泵引理`.

Lec 4 正则语言的封闭性 泵引理

1. 正则语言的封闭性

> Closure Properties of Regular Languages

封闭性：

如果对集合 S 中的任意元素执行某种运算 O，其结果仍然是集合 S 中的一个元素，那么我们就说集合 S 对运算 O 是封闭的.

例如，正整数集 $\mathbb{Z}^+$ 对加法是封闭的，对减法是不封闭的.

例如，实数集 $\mathbb{R}$ 对加、减、乘、除（除数不为 0）都是封闭的.

接下来我们要证明正则语言的封闭性：

For regular languages L_1 and L_2, we will prove that

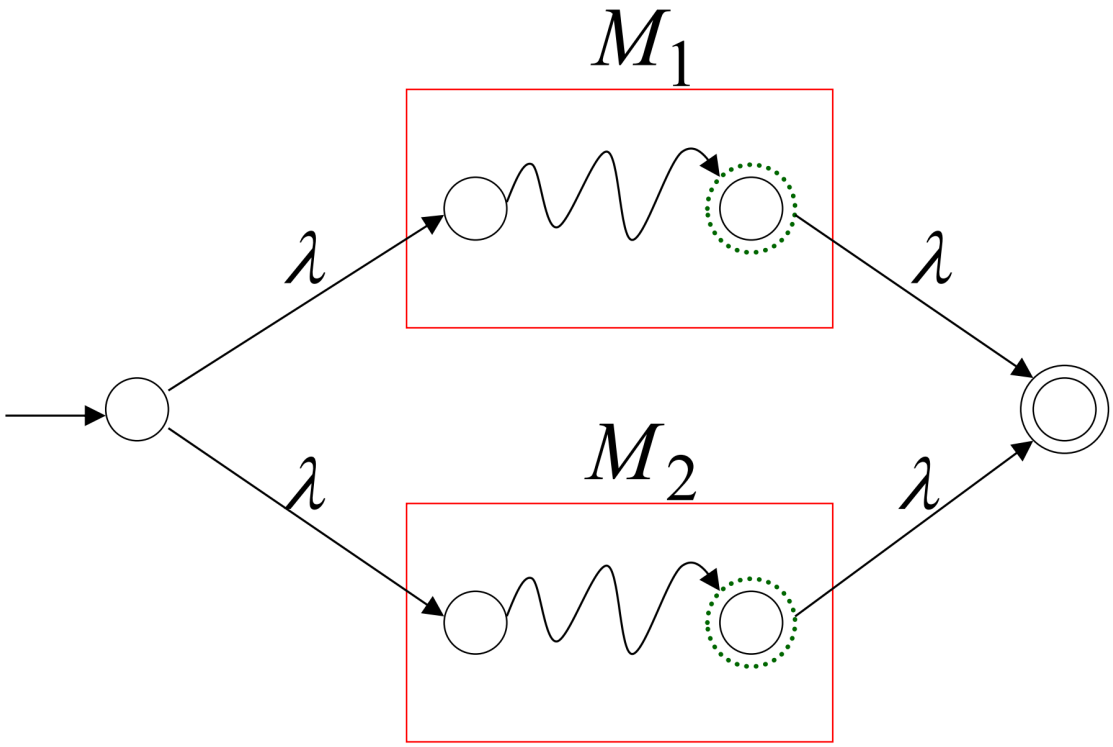
$$\begin{cases} \text{Union: } L_1 \cup L_2 \\ \text{Concatenation: } L_1 L_2 \\ \text{Star: } L_1^* \\ \text{Reversal: } L_1^R \\ \text{Complement: } \overline{L_1} \\ \text{Intersection: } L_1 \cap L_2 \\ \text{Difference: } L_1 - L_2 \end{cases}$$

Are regular languages.

In other words, regular languages are **closed under** Union / Concatenation / Star / Reversal / Complement / Intersection / Difference.

这意味着正则语言具有极强的稳定性，无论进行哪些基本的语言操作，结果都不会超出 “正则” 这个范畴. 因此，我们不需要发明新的理论或工具来处理运算后的新语言，因为它们依然可以被有限自动机识别.

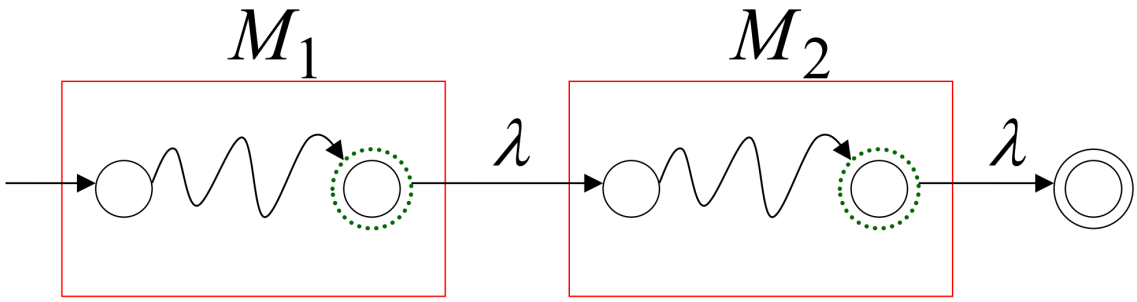
1.1 Union



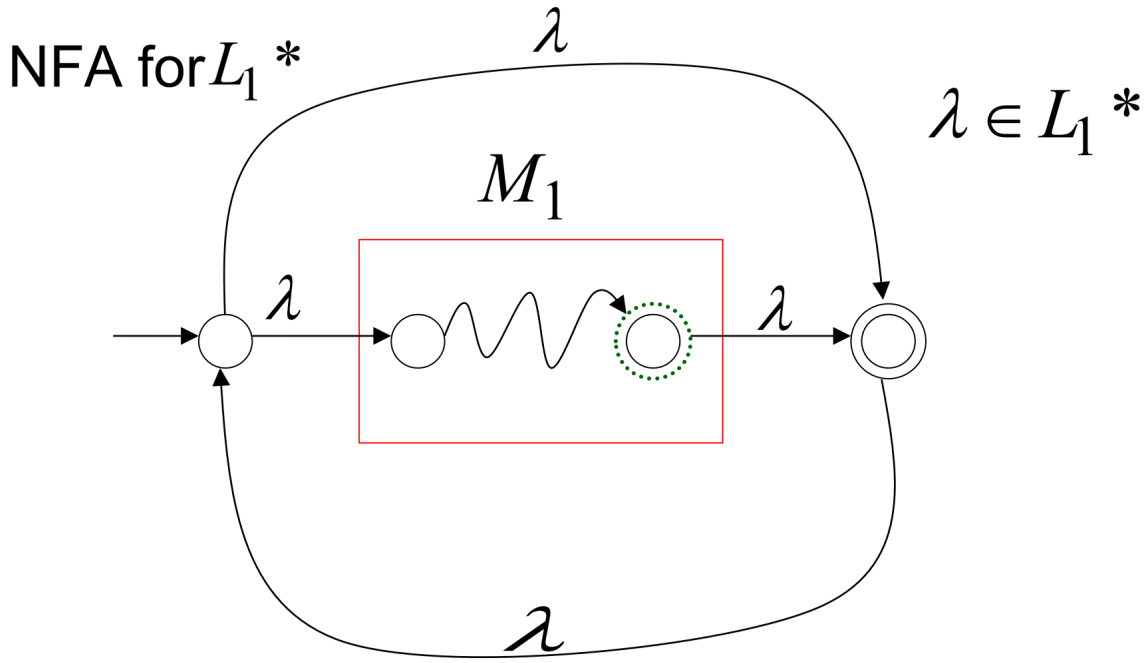
注意，也可以使用 Product Construction，并把 Final state 设计为 $(q_1, q_2) \in F$ iff $q_1 \in F_1$ **or** $q_2 \in F_2$.

> 关于 Product Construction，见 `1.6 Intersection`.

1.2 Concatenation

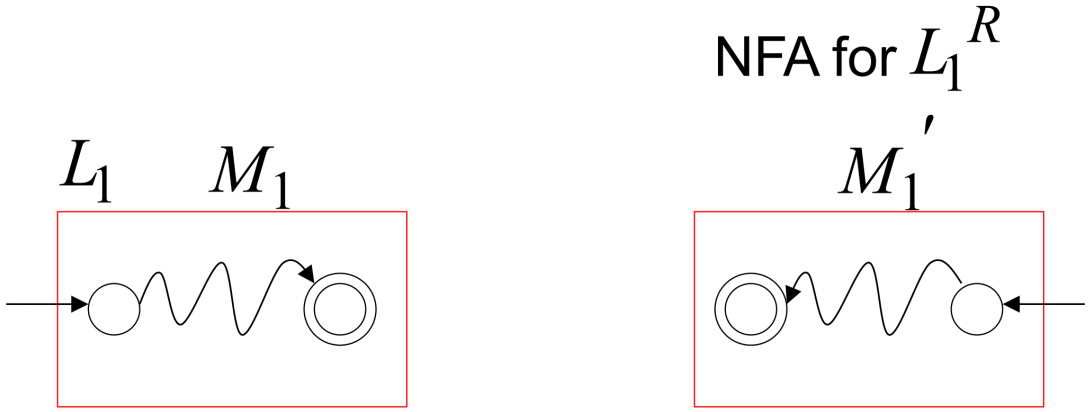


1.3 Star Operation



1.4 Reverse

起始状态与最终状态对调，内部箭头全部反转：



下面证明正则语言在反转操作上是封闭的.

假设一个语言 L 是正则语言，则它可以被一个 NFA M 接受（转为等价的单最终状态）. 把 M 的起始状态和最终状态对调，箭头全部反转，得到一个新的 NFA M^R.

易得任何被 M 接受的字符串 w，其反转 w^R 一定会被 M^R 接受. 即 $L(M)^R \subseteq L(M^R)$.

反之，任何被 M^R 接受的字符串 w，其反转 w^R 一定被 M 接受. 即
$L(M) \supseteq L(M^R)^R \Leftrightarrow L(M)^R \supseteq L(M^R)$.

则 $L(M)^R = L(M^R)$. 因为 $L(M^R)$ 是正则语言，所以 $L(M)^R$ 是正则语言.

综上，正则语言的反转仍然是正则语言，即正则语言在反转操作上是封闭的.

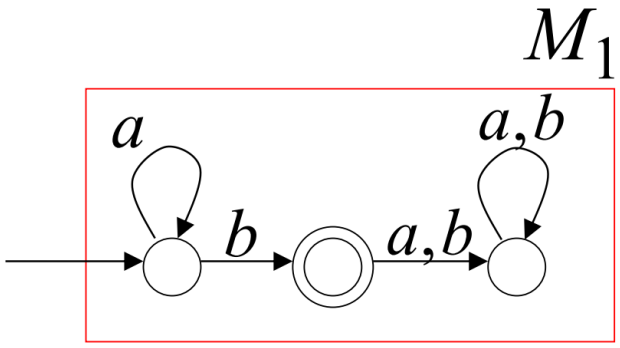
1.5 Complement

Take the DFA that accepts L_1, make final states non-final, make non-final state final

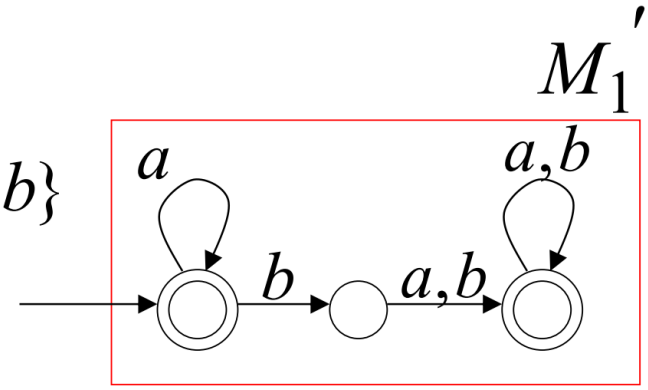
> 注意和 Reverse 区分，Reverse 的构造不要求 DFA，且只对调了起始和最终. Complement 要求 DFA，且把所有 final state 改为非 final state，把所有非 final state 改为 final state.
>
> 先取 DFA 的原因：接受 L_1 的 NFA 可能对某些输入没有作定义（当然也就不接受相关的字符串），如果基于这个 NFA 反转状态，原先未定义的仍然未定义. 然而新构造的状态机应该定义和接受这些原先未定义的输入，因为我们的目标是构造接受 L_1 补集的状态机，所有原先被拒绝的都要被新构造的接受.
>
> 因此，我们取 DFA，对所有可能的输入都给出了定义. 那么，当我们反转状态后，得到的新 DFA 仍然对所有可能输入都给出了定义，只不过原先接受的（落在 final state）现在不接受（落在 non-final state），原先不接受的（落在 non-final state）现在接受（落在 final state）.

例:

$$L_1 = \{a^n b\}$$



$$\overline{L_1} = \{a,b\}^* - \{a^n b\}$$



1.6 Intersection

给出两种封闭性证明：

① 逻辑推导

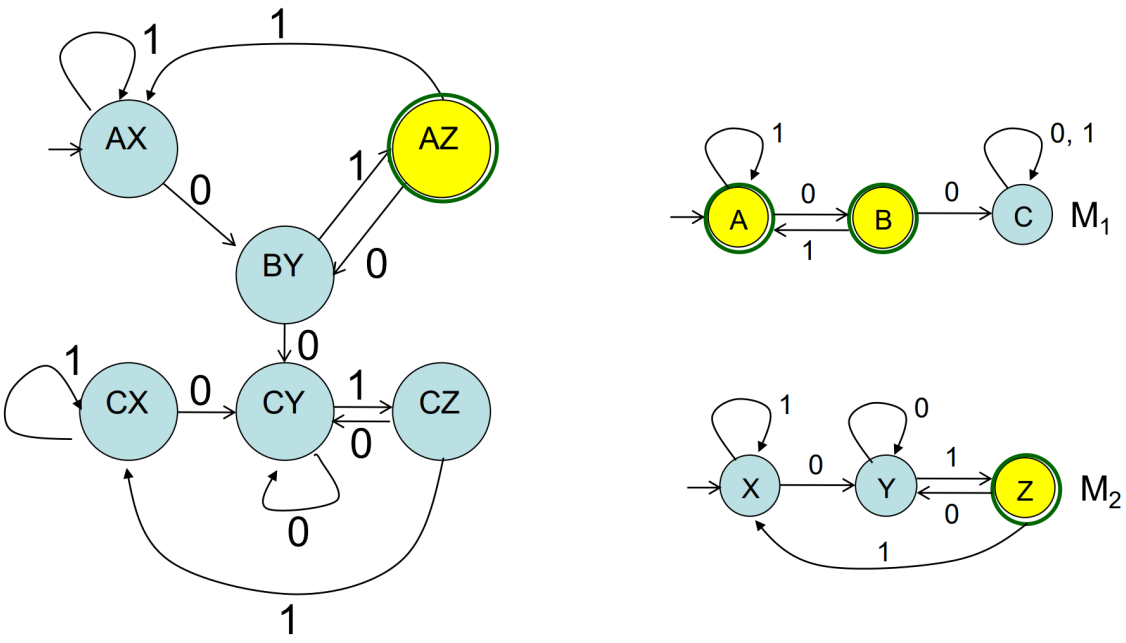
If L_1, L_2 are regular,

$$\begin{aligned} &\Rightarrow \overline{L_1}, \overline{L_2} \text{ are regular} \\ &\Rightarrow \overline{L_1} \cup \overline{L_2} \text{ is regular} \\ &\Rightarrow \overline{\overline{L_1} \cup \overline{L_2}} \text{ is regular} \\ &\Rightarrow L_1 \cap L_2 \text{ is regular} \end{aligned}$$

> 核心是 DeMorgan's Law: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.

② NFA 构造

例：



这个方法通常称为 Product Construction.

如果 L_1 和 L_2 是正则语言，分别由自动机 $M_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, F_1)$ 和 $M_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, F_2)$ 接受，那么它们的交集 $L_1 \cap L_2$ 也是正则语言，可以由一个新的自动机 $M_{1\cap 2}$ 接受.

$M_{1\cap 2}$ 构造思路：模拟 M_1 和 M_2 同步运行. 它的状态是一个有序对 (q_1, q_2)，其中 q_1 是 M_1 中的一个状态，q_2 是 M_2 中的一个状态.

> 上例的左图把括号和逗号省略了，如 AX 即表示 (A, X).

$$M_{1\cap 2} = (Q, \Sigma, \delta, q_0, F)$$

- $Q = Q_1 \times Q_2 = \{(q_1, q_2) \mid q_1 \in Q_1, q_2 \in Q_2\}$.

  > 笛卡尔积.

- Σ：不变.
- $q_0 = (q_{0,1}, q_{0,2})$.
- δ

对于任何状态 $(q_1, q_2) \in Q$ 和任意输入 $a \in \Sigma$，新的转移函数定义为

$$\delta\left((q_1, q_2), a\right) = (\delta_1(q_1, a), \delta_2(q_2, a))$$

> 若为非 DFA 的 NFA，则 $\delta\left((q_1, q_2), a\right) = \delta_1(q_1, a) \times \delta_2(q_2, a)$.

- $F = F_1 \times F_2$

> 一个状态 $(q_1, q_2) \in F$ 当且仅当 $q_1 \in F_1$ and $q_2 \in F_2$.
>
> 这是交集的定义.

1.7 Difference

同样给出两种封闭性证明：

① 逻辑推导

If L_1, L_2 are regular,

$$\begin{aligned} &\Rightarrow \ \overline{L_2} \text{ is regular} \\ &\Rightarrow \ L_1 \cap \overline{L_2} \text{ is regular} \\ &\Rightarrow \ L_1 - L_2 \text{ is regular} \end{aligned}$$

核心是 Difference 的定义：

$$\begin{aligned} & x \in L_1 - L_2 \\ \Leftrightarrow \ & x \in L_1 \text{ and } x \notin L_2 \\ \Leftrightarrow \ & x \in L_1 \text{ and } x \in \overline{L_2} \\ \Leftrightarrow \ & x \in L_1 \cap \overline{L_2} \end{aligned}$$

即 $L_1 - L_2 = L_1 \cap \overline{L_2}$.

② NFA 构造

同样 Product Construction. 并把 Final state 设计为 $(q_1, q_2) \in F$ iff $q_1 \in F_1$ **and** $q_2 \notin F_2$.

> 其余构造步骤和 `1.6 Intersection` 完全相同.

1.8 同态

> homomorphism

Suppose Σ and Γ are alphabets. Then a function

$$h : \Sigma \to \Gamma^*$$

is called a homomorphism. In other words, a homomorphism is a substitution in which a single letter is replaced with a string.

> 这里的 h 是人为定义的，题目通常会给出.

同态是一种替换操作，其中 Σ 中的单个字母被 Γ^* 中的一个字符串替代.

同态的定义可以自然地扩展到字符串和语言.

字符串的同态：如果 $w = a_1a_2 \dots a_k$ 是 Σ^* 中的一个字符串，那么 w 的同态映射 $h(w)$ 定义为

$$h(w) = h(a_1)h(a_2) \dots h(a_k)$$

> 将字符串中的每个字母依次应用同态映射，并将结果字符串连接起来.

语言的同态：If L is a language on Σ, then its homomorphic image is defined as

$$h(L) = \{h(w) : w \in L\}$$

> 对语言 L 中的每个字符串应用同态映射.

正则语言对同态映射是封闭的：如果 L 是一个正则语言，并且 h 是一个同态映射，那么 $h(L)$ 仍然是一个正则语言. 这意味着，即使经过这种字符串替换操作，我们仍然可以使用 DFA / NFA 来识别新的语言 $h(L)$.

2. 泵引理

> Identifying Nonregular Languages - Pumping lemma

How can we prove that a language L is not regular?

- Prove that there is no DFA that accepts L
- Problem: this is not easy to prove
- Solution: the Pumping Lemma

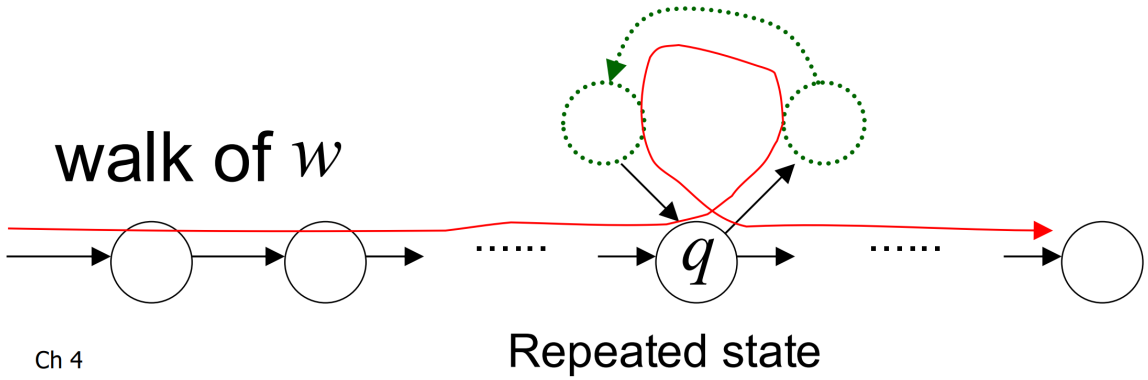
2.1 抽屉原理

> The Pigeonhole Principle（鸽笼原理）

有 n 个鸽子和 m 个鸽笼，$n > m$. 则 There is a pigeonhole with at least 2 pigeons.

应用于 DFA，易得：

for any DFA, if string w has length $\geq$ number of states, a state q must be repeated in the walk of w.



2.2 泵引理

> 口诀：构造 w，归类 xyz，泵

泵引理：Given an infinite regular language L, there **exists** an integer m, for **any** string $w \in L$ with length $|w| \geq m$, we can write $w = xyz$ with $|xy| \leq m$ and $|y| \geq 1$, such that $xy^iz \in L$, $i = 0, 1, 2, \ldots$

> 注意几个关键点：
>
> 如果 L 正则，符合描述的 m **一定存在**；
>
> 符合描述的 m：对**任意** $w \in L$ 且 $|w| \geq m$，都**存在**符合描述的分解.
>
> 符合描述的分解：$w = xyz$，$|xy| \leq m$ 且 $|y| \geq 1$, such that $xy^iz \in L$ 对**任意** $i = 0, 1, 2, \ldots$ 都成立.

注意这里的 infinite 指的是 L 中字符串的数量，而非长度. 本课范围内的字符串均为有限长，不需要单独强调.

为什么要强调 infinite？因为 finite 的语言必然是正则（有限数量的字符串，可以用简单的正则表达式 $w_1 + w_2 + \ldots$ 描述），不需要用到泵引理.

此外，只有 infinite 的语言才能保证 $|w| \geq m$ 的字符串是存在的.

泵引理博弈：

① 对手：声称一个符合描述的 m

> 符合描述的 m：对**任意** $w \in L$ 且 $|w| \geq m$，都**存在**符合描述的分解.
>
> 注意，泵引理博弈是基于玩家和对手都必须承认泵引理的正确性（符合描述的 m 必然存在）. 因为对手声称 L 是正则的，那它必须拿出至少一个符合描述的 m. 当然这里的 m 是对手单方面声称符合描述，实际上符合描述的 m 可能不存在.

② 玩家：Pick a string $w \in L$ of length $|w| \geq m$.

> 虽然 w 有很多构造可选，但是不够巧妙的构造可能导致挑战失败，即使对手是错的.

③ 对手：声称一个符合描述的 decomposition xyz

> 符合描述的分解：$w = xyz$，$|xy| \leq m$ 且 $|y| \geq 1$, such that $w_i = xy^iz \in L$ 对**任意** $i = 0, 1, 2, \ldots$ 都成立.
>
> 同理，这里只是对手的声称，不一定存在.

④ 玩家：Pick i that the pumped string $w_i \notin L$.

> 注意，由于对手是一个假想敌，并不真实存在，参与博弈的只有牛马大学生玩家. 这个苦逼现实导致了两个困难的挑战：
>
> - 挑战一：挑战 m 的存在性.
>
> 因为不会真的有个对手给你一个 m 的具体值让你挑战，因此玩家必须证明所有符合描述的 m 都不存在. 但玩家不需要为 $m = 1, m = 2, \ldots$ 一一写出证明，因为所有具体值都可以用抽象的符号 m 来统一表示. 因此玩家只需要根据 m 构造一个合适的 w 进入第二个挑战（这个 w 的构造考验玩家的思维能力，如果玩家的构造不够巧妙，那么挑战可能会失败，即使语言 L 确实不是正则语言）.
>
> - 挑战二：挑战分解的存在性.
>
> 因为不会真的有个对手给你一个具体分解 $w = xyz$ 让你挑战，因此玩家必须证明所有符合描述的分解都不存在. 但玩家不需要穷举所有的分解，最常见的技巧是利用 $|xy| \leq m$ 和 w 的巧妙构造来简化分解（归类）. 例如，对于像 $L = \{a^nb^n : n \geq 0\}$ 这样的语言，玩家可以构造 $w = a^mb^m$，这样 w 的前 m 个字符全是 a，所有分解方式都归入了一个类型——y 是 a^m 的子串，即 $y = a^k, k = 1, 2, \ldots, m$. 只要挑战这一类分解即可. 对于更复杂的非正则语言，得益于 $|xy| \leq m$ 的强限制，通常也可以归纳为少数几类，然后对每一类分别挑战，全部成功才算成功.
>
> - 挑战二若成功，说明每一类分解都不符合描述，即不存在任何符合描述的分解. 由于挑战一中玩家使用抽象符号 m 表示对手可能给出的所有 m，即对于**任意** m，玩家总能**构造出一个** w（即存在至少一个 w）使得**任何符合描述的分解都不存在**，即每一个 m 都不符合描述，即**不存在任何符合描述的** m，挑战一也成功了.

泵引理及泵引理博弈有效性的形式证明：假设 L 是一个正则语言，那么一定存在一个 DFA 能够接受它. 并且这个 DFA 的状态数量是有限的（计为 p，$p \geq p_{\min}$，其中 $p_{\min}$ 是所有可能 DFA 的最小状态数，p 是我们假设存在的某个特定 DFA 的状态数）.

> 注意，因为该 DFA 有可能不存在，所以我们不需要构造一个实际的 DFA 并得到它的状态数. 我们只要假设存在这么一个 DFA，并用一个字母 p 表示它的状态数即可.

这时我们只要能找到一个足够长（大于等于泵长度 m，通常取 $m = p$）且属于 L 的字符串 w，那么必然有环.

> 泵长度是随意设置的，只需要保证 $m \geq p_{\min}$ 就一定在某个存在的 DFA 上可泵. 但是按照反证法的逻辑，因为我们假设的那个特定 DFA 状态数为 p，所以应该取 $m \geq p$ 才在我们假设的 DFA 上可泵. 注意，泵引理的核心是如果 L 正则，则符合描述的 m 一定存在. 这很好证明：因为 p 是有限的，满足 $m \geq p$ 的 m 肯定存在（例如 $m = p$）.
>
> 实际表述上无需这么严谨，通常不区分 p 和 $p_{\min}$. 并且泵引理通过我们的严谨讨论被证明有效，实际使用时直接取 m 即可（$m \geq p$，但是不需要假设一个 p，只需要知道泵长度 m）.

此时该字符串可以分解为三部分：$w = xyz$.

- x：环路开始前的部分.
- y：环路本身（$|y| \geq 1$，$|xy| \leq m$）.

> 路径 xy 包含从初始状态 q_0 开始，直到第一次出现状态重复的所有输入符号. 若 $|xy| > m$，则 xy 本身可以继续泵，即在 xy 路径上有其他状态更早重复，不符合我们想要的 “第一次” 状态重复.
>
> 为什么 xy 一定要是起始状态到 “第一次” 状态重复（y 只环了一圈）？ 实际上多重复几次（例如 xy^2）仍然可泵（xy^4，xy^6 等），但是这是给对手降低难度. 玩家要求对手对玩家提供的 w 给出 $|xy| \leq m$ 的可泵片段，这可以把对手可能提供的分解局限到局部，更方便玩家逐个击破. 倘若不对 $|xy|$ 作出限制，要讨论非常多种情况，这对玩家很不利，因为玩家只有证明所有分解都无法泵出，才能获胜.
>
> 而对于正则语言，一定存在 $|xy| \leq m$ 的可泵片段，这是最严格的限制：玩家给一个**较大的**满足要求的 w（例如，$a^m b^m c^m$ 而非 $a^{\frac{m}{3}} b^{\frac{m}{3}} c^{\frac{m}{3}}$），既然对手声称这是正则语言，请拿出证据证明这么小（全在 a 中）的可泵片段都存在.

- z：环路结束后的部分.

对于任意符合描述的分解 xyz，既然 y 对应一个环路，那么在这个环路中循环任意次，DFA 都会回到同一个状态，并继续处理 z 到达终态. 即所有 “泵出” 的字符串 xy^iz（$i \geq 0$）都必须被该 DFA 接受.

对于任意符合描述的分解 xyz，如果我们能找到一个特定的 i，使得字符串 xy^iz 明显不满足 L 的定义，就自相矛盾，即假设是错误的（符合描述的分解 xyz 不存在，则符合描述的 m 不存在，则接受 L 的 DFA 不存在，则 L 不是正则语言）.

> 通常从 $i = 0$ 或 $i = 2$ 入手.

注意：”泵引理博弈“ 和 ”形式证明“ 的写法有所不同.

”泵引理博弈“ 只是一轮博弈，玩家对对手的说辞进行挑战，若挑战成功，只能说明对手无法证明该语言是正则语言，并不能证明该语言是非正则语言.

”形式证明“ 则是用全称量词对所有可能的博弈进行了推演，如果反证成功，则能直接说明该语言是非正则语言.

做题时，如果题目是 ”Prove the following languages are not regular“，则是要求我们用形式证明的写法. 核心就是考察 w 的构造，使得所有可能的分解 xyz 能被归类为尽量少的几类，并且能够一一击败.

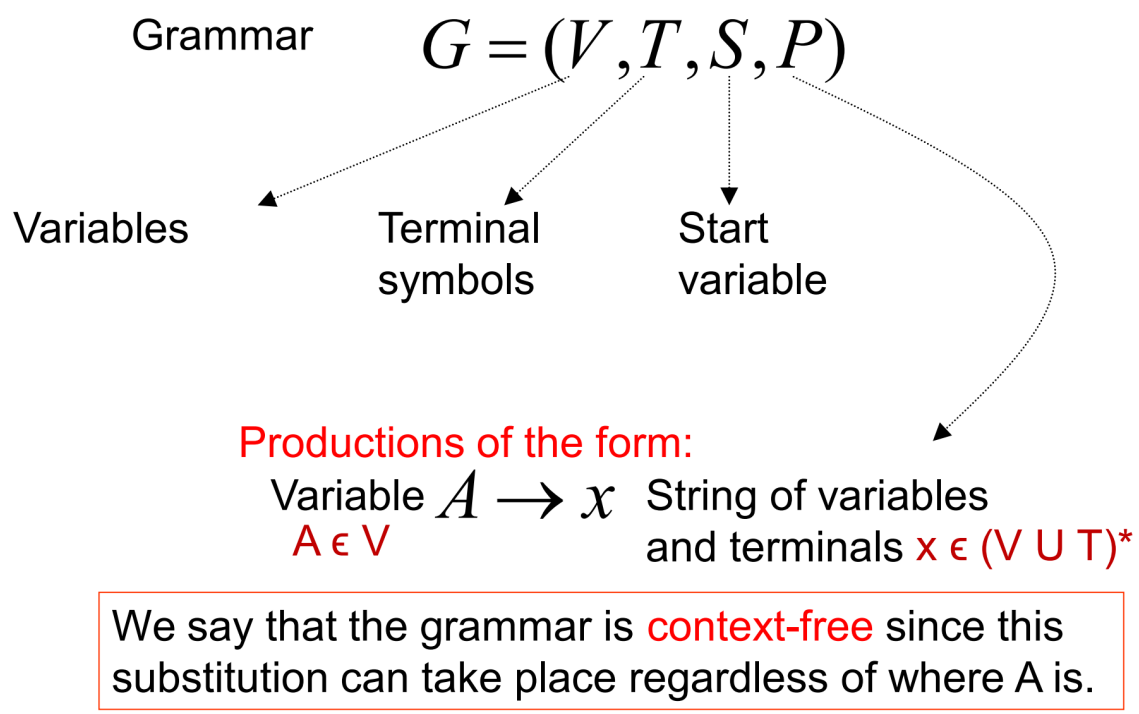
Lec 5 上下文无关 解析与歧义

1. 上下文无关

1.1 上下文无关文法

> Context-Free Grammars

定义：即 `Lec 2 2. 语法` 中对 Grammar 的定义.



与之相对的是上下文有关文法（Context-Sensitive Grammar, CSG），比上下文无关文法更广泛、强大.

> CSG：产生式规则形如 $\alpha A \beta \to \alpha x \beta$

Regular and linear grammars are clearly context-free; But a context-free grammar is not necessarily linear.

1.2 上下文无关语言

> Context-Free Languages

A language L is context-free iff there is a context-free grammar G with $L = L(G)$.

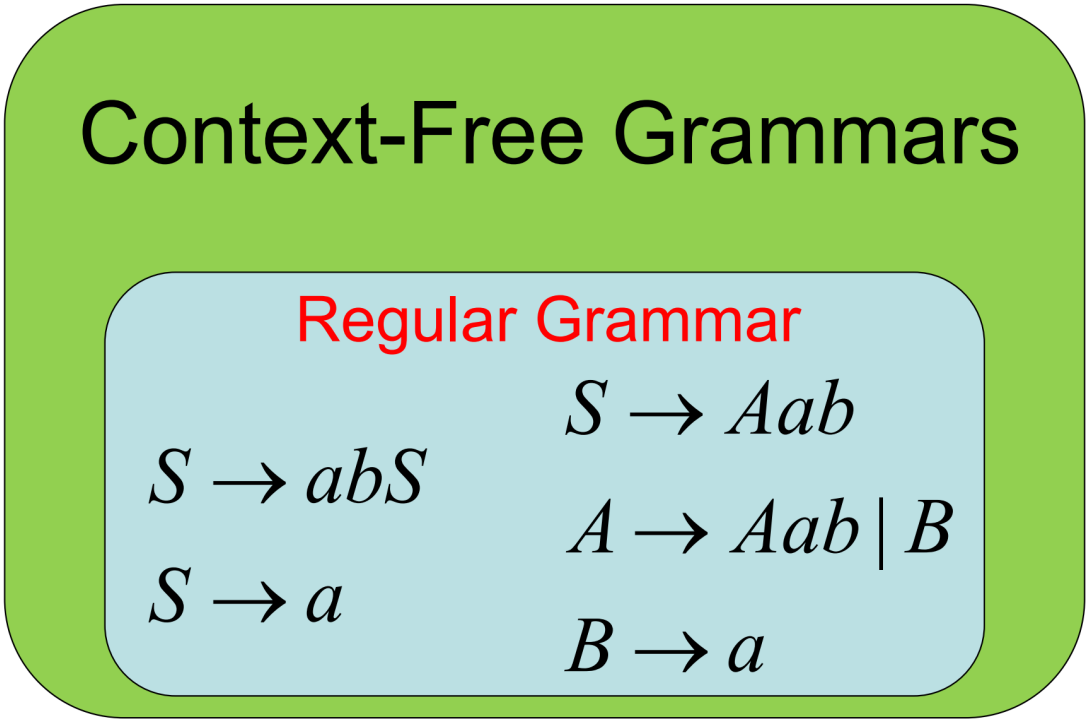
例:

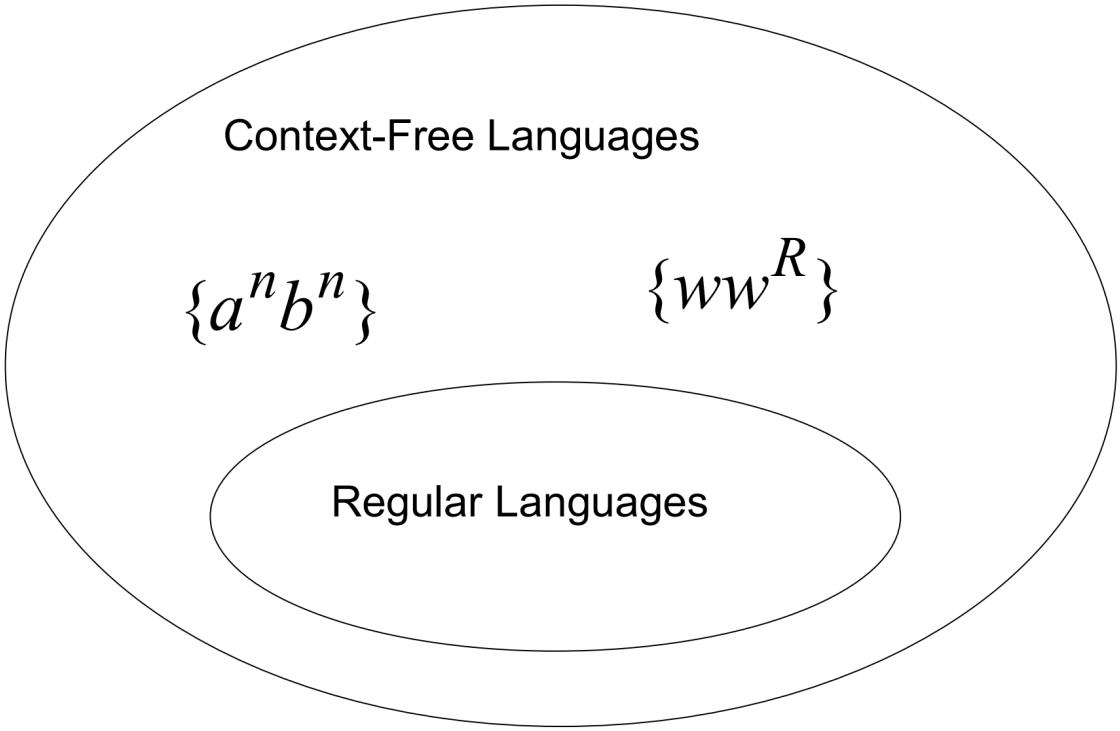
$$
\begin{aligned}
G &= (\{S\}, \{a, b\}, S, P), \\
P &: S \to aSb, \\
&\quad S \to \lambda \\
\Rightarrow L(G) &= \{a^n b^n : n \geq 0\}.
\end{aligned}
$$

注意，上下文无关语言通常是无限集合，我们不一定能在有限时间里找到它的所有字符串（没有通用的算法）.

因此，判定两个 CFG 是否等价是非常困难的.

1.3 关系图





1.4 推导顺序

> Derivation Order

In CFGs that are not linear, a derivation may involve sentential forms with more than one variable. 这样，同一个字符串就有不同的推导顺序.

例:

$$
\begin{aligned}
&G = (\{S, A, B\}, \{a, b\}, S, P),\\
P: &S \to AB,\\
&A \to aaA,\\
&A \to \lambda,\\
&B \to Bb,\\
&B \to \lambda\\
\Rightarrow &L(G) = \{a^{2n}b^m : n \geq 0, m \geq 0\}.
\end{aligned}
$$

以字符串 aab 为例，介绍最左推导和最右推导.

1.4.1 最左推导

> Leftmost derivation

$S \Rightarrow \color{red}{A}B \Rightarrow aa\color{red}{A}B \Rightarrow aa\color{red}{B} \Rightarrow aa\color{red}{B}b \Rightarrow aab$

1.4.2 最右推导

> Rightmost derivation

$S \Rightarrow A\color{red}{B} \Rightarrow A\color{red}{B}b \Rightarrow \color{red}{A}b \Rightarrow aa\color{red}{A}b \Rightarrow aab$

1.4.3* 最左最右推导定理

① 对于任意一个由 CFG 生成的字符串，它**至少有一个**最左推导和一个最右推导.

② 对于任意一个由**无歧义**的 CFG 生成的字符串，它有唯一的最左推导和最右推导，且它们对应同一棵解析树.

③ 对于任意一个由**有歧义**的 CFG 生成的字符串，它对应多棵推导树，但每棵特定的推导树有唯一的最左推导和最右推导.

1.5 推导树

> Derivation (Parse) Trees

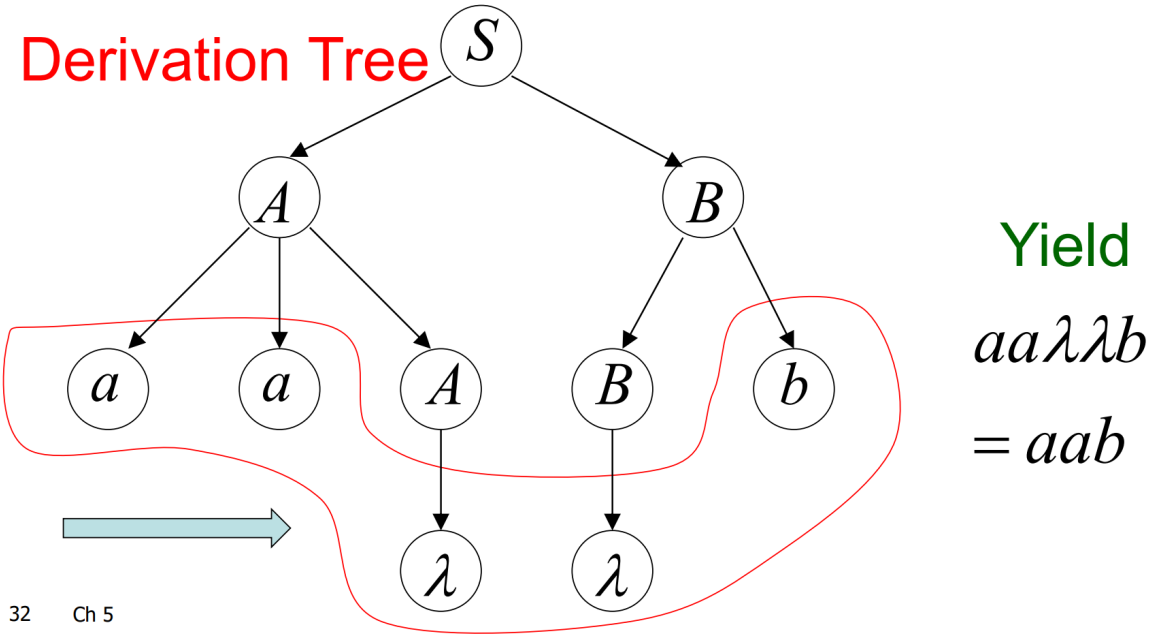
An ordered tree in which nodes are labeled with the left sides of productions and in which the children of a node represent its corresponding right sides.

> 从父节点到子节点对应产生式从左到右.

例：

$$
\begin{aligned}
&G = (\{S, A, B\}, \{a, b\}, S, P),\\
&P : S \to AB,\\
&\quad A \to aaA,\\
&\quad A \to \lambda,\\
&\quad B \to Bb,\\
&\quad B \to \lambda\\
&\Rightarrow L(G) = \{a^{2n}b^m : n \geq 0, m \geq 0\}.
\end{aligned}
$$

$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$ 的推导树：



形式化定义：

Let $G = (V, T, S, P)$ be a CFG. An **ordered** tree is a derivation tree for G iff it has the following properties:

① The root is labeled S

② Every leaf has a label from $T \cup \{\lambda\}$

③ Every internal vertex has a label from V

④ If a vertex has label $A \in V$, and its children are labeled $a_1, a_2, \ldots, a_n$, then P must contain a production $A \to a_1 a_2 \ldots a_n$

⑤ A leaf labeled λ has no sibling

> 这里的 ordered 指子节点的顺序是重要的，不能随意调换. 例如，对于产生式 $A \to abABc$，A 的五个子节点从左到右的顺序必须是 a, b, A, B, c.

1.5.1 产出

> yield

推导树的产出：把 leaf 从左到右拼接.

Depth-first, leftmost.

1.5.2 部分推导树

形式化定义：

- A tree that has properties ③, ④ and ⑤.

> ③ Every internal vertex has a label from V
>
> ④ If a vertex has label $A \in V$, and its children are labeled $a_1, a_2, \ldots, a_n$, then P must contain a production $A \to a_1 a_2 \ldots a_n$
>
> ⑤ A leaf labeled λ has no sibling

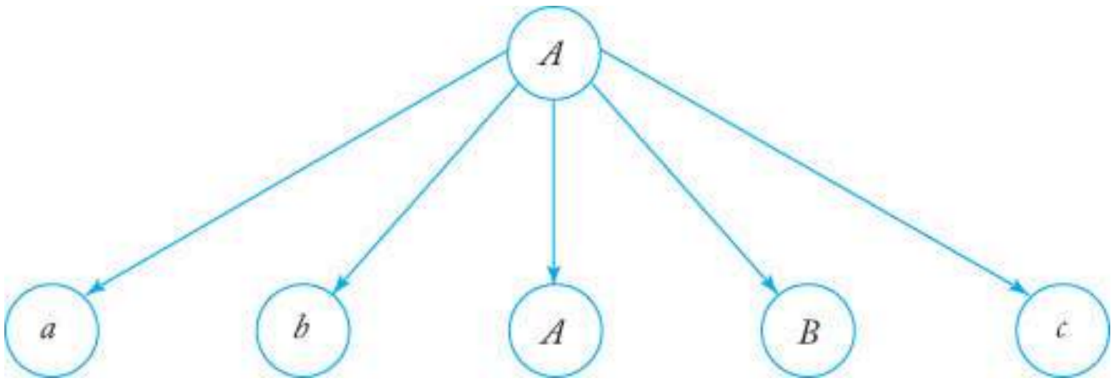
- ① does not necessarily hold.

> ① The root is labeled S

- ② is replaced by: Every leaf has a label from $V \cup T \cup \{\lambda\}$

> ② Every leaf has a label from $T \cup \{\lambda\}$

例：$A \to abABc$



Theorem:

- Let $G = (V, T, S, P)$ be a CFG. Then for every $w \in L(G)$, there exists a derivation tree of G whose yield is w. Conversely, the yield of any derivation tree is in $L(G)$.
- Also, if t_G is any partial derivation tree for G whose root is labeled S, then the yield of t_G is a sentential form of G.

注意，有时不同的推导顺序可能对应同一个推导树：

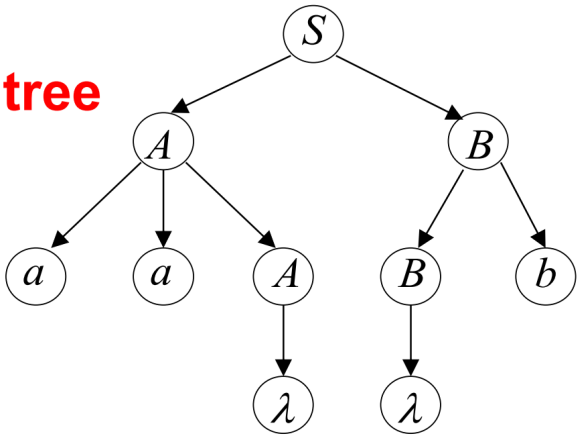
Leftmost:

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aaBb \Rightarrow aab$$

Rightmost:

$$S \Rightarrow AB \Rightarrow ABb \Rightarrow Ab \Rightarrow aaAb \Rightarrow aab$$

Same derivation tree



2. 解析与歧义

> Parsing and Ambiguity

2.1 解析

解析（Parsing）旨在确定一个终结符字符串 w 是否属于 $L(G)$，如果属于，找到导出 w 的产生式序列（derivation）.

2.1.1 暴力解析

> Exhaustive Search Parsing (Brute Force Parsing)
>
> 穷举搜索解析 / 暴力解析

探索所有可能性，直到找到一条可以导出 w 的路径. 常用策略为广度优先搜索和深度优先搜索.

例 1

例 1：$S \to SS|aSb|bSa|\lambda$. Find derivation of $aabb$.

广度优先搜索：按阶段前进，Phase n 表示经过 n 次推导能够到达的句型. 如果其中一些明显推不出要找的字符串，可以直接删去.

Phase 1：

$$
\begin{aligned}
&S \Rightarrow SS \\
&S \Rightarrow aSb \\
&\cancel{S \Rightarrow bSa} \\
&\cancel{S \Rightarrow \lambda}
\end{aligned}
$$

> 第三个删去，因为 b 在 a 前面不可能推出 $aabb$；第四个删去，因为 λ 无法再推，且 $\lambda \neq aabb$.

Phase 2：保留 Phase 1 中合理的结果，继续推导

$$
S \Rightarrow SS \Rightarrow \begin{cases} SSS \\ aSbS \\ \cancel{bSaS} \\ SaSb \\ \cancel{SbSa} \\ S \end{cases}
$$

$$
S \Rightarrow aSb \Rightarrow \begin{cases} aSSb \\ aaSbb \\ \cancel{abSab} \\ \cancel{ab} \end{cases}
$$

Phase 3：

$$
\begin{aligned}
&S \Rightarrow SS \Rightarrow SSS \Rightarrow \cdots \\
&S \Rightarrow SS \Rightarrow aSbS \Rightarrow \cdots \\
&S \Rightarrow SS \Rightarrow SaSb \Rightarrow \cdots \\
&S \Rightarrow SS \Rightarrow S \Rightarrow \cdots \\
&S \Rightarrow aSb \Rightarrow aSSb \Rightarrow \cdots
\end{aligned}
$$

$$
S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \begin{cases} aaSSbb \\ \cancel{aaaSbbb} \\ \cancel{aabSabb} \\ aabb \end{cases}
$$

找到了一个 derivation：$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

缺点

- Tediousness (bad for efficiency)
- It is possible that it never terminates for strings not in $L(G)$

> 若文法包含形如 $A \to \lambda$ 或 $A \to B$ 的产生式，则无法用字串 w 的长度 $|w|$ 来限制推导的最大深度（可能在变长和变短之间反复循环，或者在 A 和 B 之间反复循环），则可能永远无法中止.

Theorem: Suppose a CFG does not have any rules of the form $A \to \lambda$ or $A \to B$, then the exhaustive search parsing can be made into an algorithm which, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

不能变短；如果希望长度不变，有且只有一种情况：使用形如 $A \to a$ 的产生式把一个变量转为终结符. 这样的转换最多 $|w|$ 次，因为终结符一旦出现就不能删除，且最多能出现 $|w|$ 个.

除了变短和长度不变，只剩一种情况：长度变长. 变长至少要变长 1，则这种情况也最多发生 $|w|$ 次.

所以长度为 $|w|$ 的字符串最多只需要 $2|w|$ phases 即可解析.

最困难的情况有可能发生：

$$\begin{aligned} S &\Rightarrow SS \\ &\Rightarrow SSS \\ &\Rightarrow SSS\cdots \\ &\Rightarrow \underbrace{SSS\cdots SSS}_{|w|个} \\ &\Rightarrow aSS\cdots SSS \\ &\Rightarrow aaS\cdots SSS \\ &\Rightarrow aaa\cdots \\ &\Rightarrow \underbrace{aaa\cdots aaa}_{|w|个} \end{aligned}$$

所以在没有任何信息之前，$2|w|$ 是最紧的上界.

For grammar with P rules

Phase 1: we have no more than $|P|$ sentential forms

Phase 2: we have no more than $|P|^2$ sentential forms

...

Phase $2|w|$: we have no more than $|P|^{2|w|}$ sentential forms

Total time (一次推导计为 1 个时间单位) needed for string w:

$$|P| + |P|^2 + \cdots + |P|^{2|w|}$$

复杂度：$O(P^{2|w|+1})$

> 极其差，随 P 幂增长，随 $|w|$ 指数增长.

2.1.2 CYK 算法

For general context-free grammars, there exists a parsing algorithm that parses a string w in time $O(|w|^3)$.

> CYK algorithm. 此处不展开.
>
> 这里是 general CFG, 没有上述 theorem 的限制.

2.1.3 简单文法

> Simple grammar (s-grammar)

There exist faster algorithms for specialized grammars.

> 例如简单文法（simple grammar）

定义：

一个上下文无关文法 G 被称为简单文法，如果它满足以下条件：

- 所有产生式都是以下两种形式之一：
  - $A \to aX$
  - $A \to \lambda$
- 唯一性条件：$A \to aX_1$ 和 $A \to aX_2$ 不能同时存在. 即 Pair (A, a) 最多 appears once.

> $A \in V$ 是变量，$a \in \Sigma$ 是任意终结符，$X \in V^*$ 是 string of variables（若干变量组成的字符串，只含变量）.
>
> 唯一性条件和 X 只含变量，确保在解析时，只需要查看输入的下一个符号，就能唯一确定要使用的产生式.

For S-grammars, in the exhaustive search parsing, there is only one choice in each phase (按 input 从左向右，每生成一个终结符算一个 phase).

Time for a phase: 1

Total time for parsing string w: $|w|$

注意，线性复杂度. 这说明只要对文法稍作限制，解析效率就会快很多. 这也启发我们，可否通过某些流程，把复杂文法转化为更易于解析的文法（如简单文法）？

2.2 歧义性

> Ambiguity

A context-free grammar G is ambiguous if some string $w \in L(G)$ has two or more derivation trees / leftmost derivations / rightmost derivations

2.2.1 固有歧义性

Some context free languages have only ambiguous grammars.

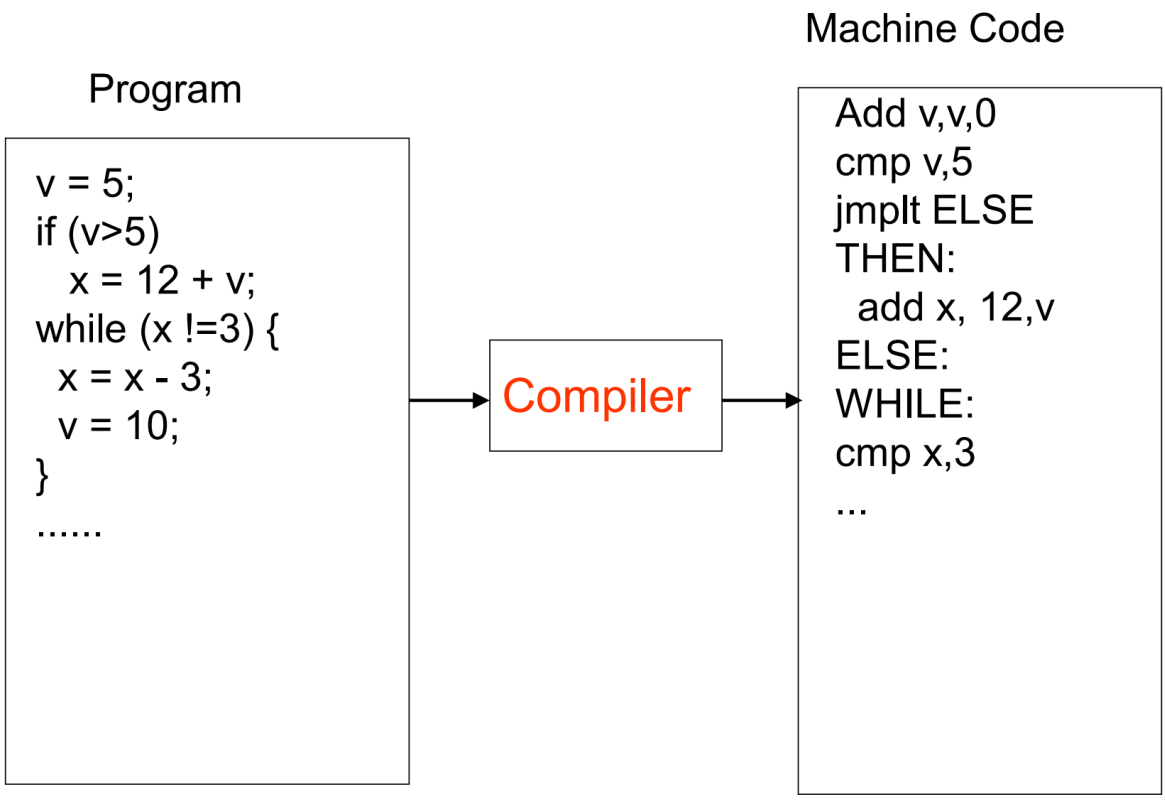
证明一个 CFL 是固有歧义的，非常困难，因为要证明**不存在任何**能够生成 L 的非歧义性 CFG.

证明一个 CFL 不是固有歧义的，相对简单，只需要找到一个能够生成 L 的非歧义性 CFG 即可.

3. 编程语言

> Context-Free Grammars and Programming Languages

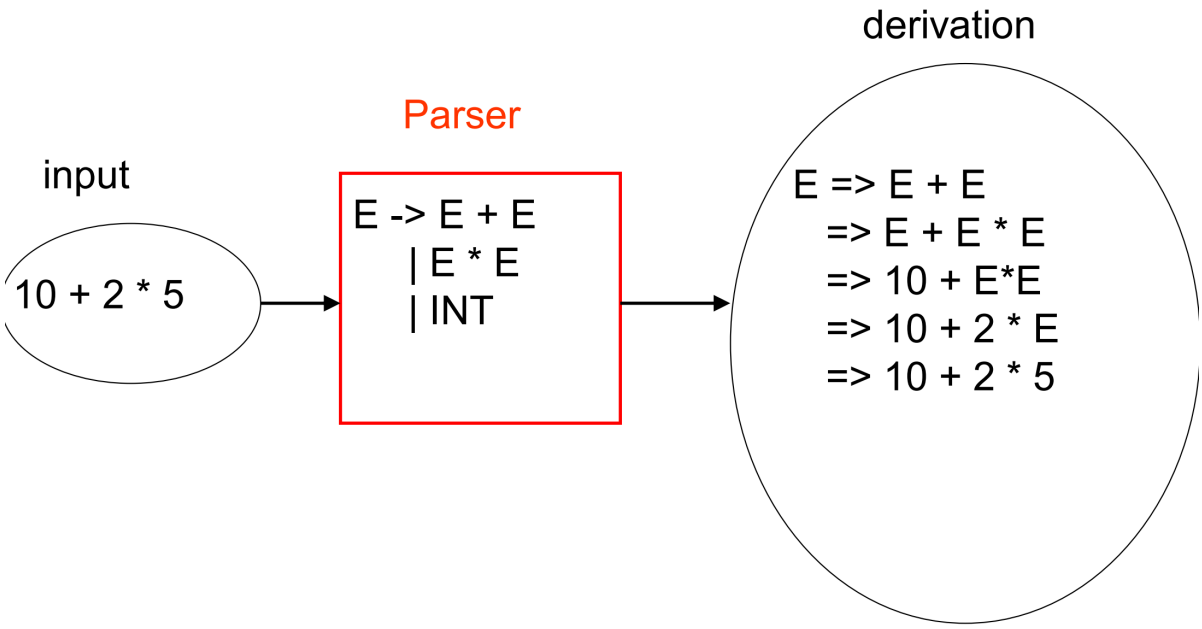
3.1 编译器



编译器工作方式：

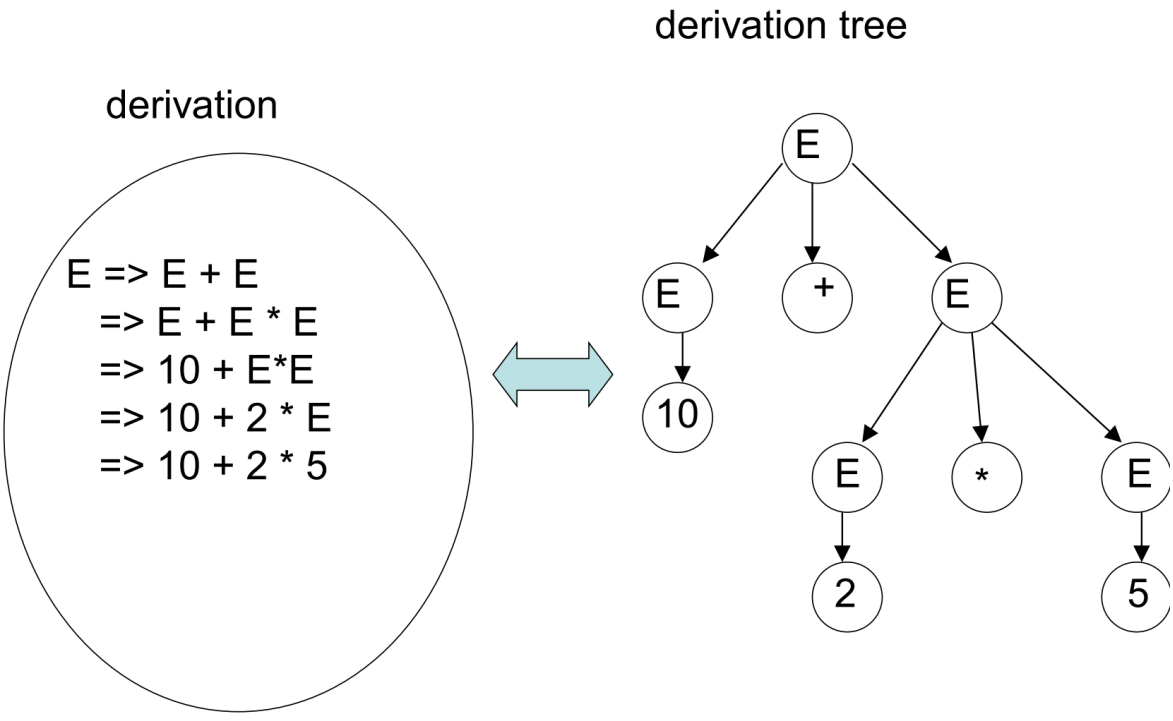
源代码（source file）$\xrightarrow{\text{lexer / scanner / tokenizer 词法分析器}}$ Stream of tokens $\xrightarrow{\text{parser 解析器 / 语法分析器}}$ Parse Tree（解析树 / 推导树）$\xrightarrow{\text{code generator}}$ Object code / Machine code

解析器知道编程语言的文法，它会根据文法找到输入字符串的推导过程.

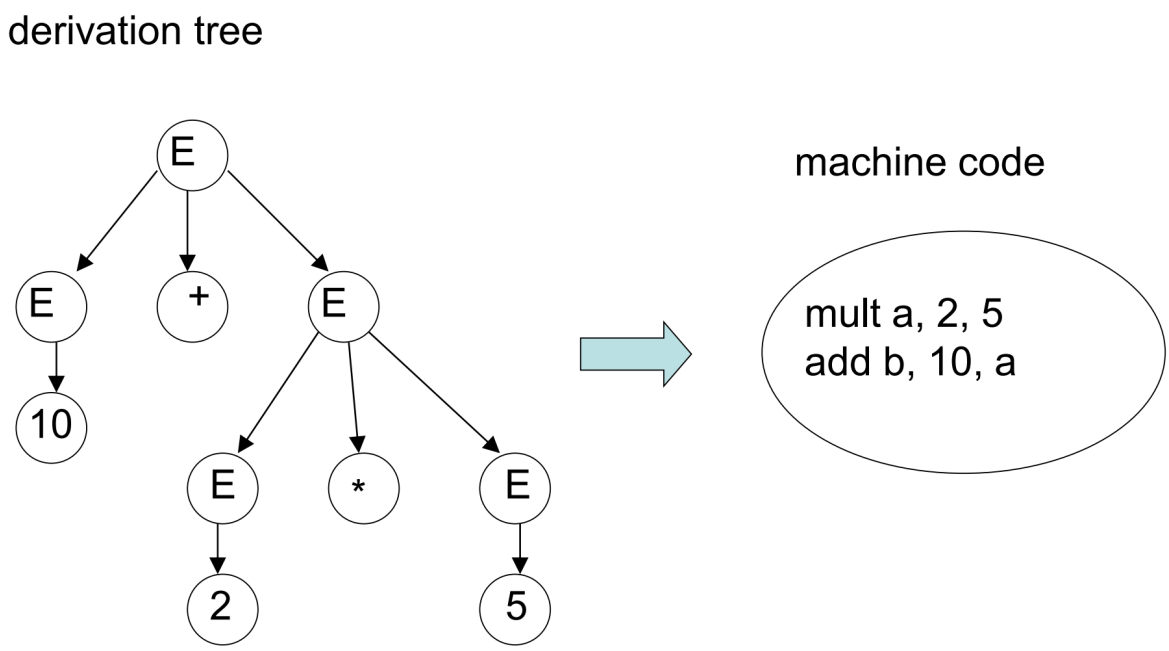


> LL and LR grammar: parse in linear time

然后转为推导树：



最后转为机器码：



3.2 Backus-Naur 范式

> Backus-Naur Form (BNF): 编程语言中常用的文法形式

例：

$$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \mid \langle \text{expr} \rangle + \langle \text{term} \rangle$$
$$\langle \text{term} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{term} \rangle * \langle \text{factor} \rangle$$

Lec 6 语法转换

1. 语法转换方法

1.1 完全替换

Theorem: Let $G = (V, T, S, P)$ be a context-free grammar. Suppose that P contains a production of the form

$$A \to x_1 B x_2$$

Assume that A and B are different variables and that

$$B \to y_1 | y_2 | \dots | y_n$$

is the set of all productions in P which have B as the left side.

Let $\hat{G} = (V, T, S, \dot{P})$ be the grammar in which $\dot{P}$ is constructed by deleting

$$A \to x_1 B x_2$$

from P, and adding to it

$$A \to x_1 y_1 x_2 | x_1 y_2 x_2 | \dots | x_1 y_n x_2$$

Then

$$L(\hat{G}) = L(G)$$

> 注意，删除的是 $A \to x_1 B x_2$（准确地说是替换）. $B \to y_1 | y_2 | \dots | y_n$ 本身不能随意删除，只有当所有右侧含 B 的产生式都被替换后，即 B 变成 useless variable，才能把 $B \to y_1 | y_2 | \dots | y_n$ 删除. 所以应用该定理一定要仔细检查所有右侧含 B 的产生式都被替换后才能删掉 $B \to y_1 | y_2 | \dots | y_n$
>
> 之所以提到这点，是因为根据 `1.3 简化流程`，想要进行某个替换的出发点通常都是删掉原产生式来简化语法，所以如果用到 “完全替换” 规则，通常就是把所有右侧含 B 的产生式都替换，然后删掉 $B \to y_1 | y_2 | \dots | y_n$.
>
> 还有一个需要注意的地方，在 `1.3 简化流程` 中，通常是根据不理想的产生式的类型按顺序逐类删除（先删空产生式，再删单元产生式，最后删无用产生式）. 而 “完全替换” 定理的前提是，$B \to y_1 | y_2 | \dots | y_n$ 是所有 B 在左侧的产生式. 按照简化流程，这个前提很难满足，通常是拿其中一部分（一条）出来替换，因此只能采用 `1.2 部分替换`.

1.2 部分替换

注意，`1.1 完全替换` 有个重要前提，即 $B \to y_1 | y_2 | \dots | y_n$ 是所有左边是 B 的产生式.

核心逻辑：$A \to x_1Bx_2 \to x_1y_1x_2|x_1y_2x_2|\dots|x_1y_nx_2$，当 $A \to x_1Bx_2$ 后只有进一步转为 $x_1y_1x_2|x_1y_2x_2|\dots|x_1y_nx_2$ 的可能，因此可以删去中间步骤 $A \to x_1Bx_2$. 如果我们只取了一部分左侧为 B 的表达式，则只能在 $A \to x_1Bx_2$ 的基础上添加，而不能删除 $A \to x_1Bx_2$.

即如下的 “部分替换” 规则（而非 `1.1 完全替换`）：

$$\begin{cases} A \to x_1Bx_2 \\ B \to y_1|y_2|\dots|y_n|y_{n+1} \end{cases} \Leftrightarrow \begin{cases} A \to x_1Bx_2|x_1y_1x_2|x_1y_2x_2|\dots|x_1y_nx_2 \\ B \to y_{n+1} \end{cases}$$

> 这里取 $B \to y_1|y_2|\dots|y_n$ 进行替换，还有一条 $B \to y_{n+1}$ 没有取. 因此，不能删掉 $A \to x_1Bx_2$，逻辑是如果删掉这条，则 $A \to x_1Bx_2 \to x_1y_{n+1}x_2$ 无法实现，语法不等价.
>
> 同样注意，这里能删掉 $B \to y_1|y_2|\dots|y_n$ 也是因为所有右侧含 B 的产生式（这里只有一条）都被 “部分替换”.

部分替换的逻辑：

$$\begin{cases} A \Rightarrow x_1Bx_2 \Rightarrow x_1y_1x_2 \\ A \Rightarrow x_1Bx_2 \Rightarrow x_1y_2x_2 \\ \quad\vdots \\ A \Rightarrow x_1Bx_2 \Rightarrow x_1y_nx_2 \\ A \Rightarrow x_1Bx_2 \Rightarrow x_1y_{n+1}x_2 \end{cases} \Leftrightarrow \begin{cases} A \Rightarrow x_1y_1x_2 \\ A \Rightarrow x_1y_2x_2 \\ \quad\vdots \\ A \Rightarrow x_1y_nx_2 \\ A \Rightarrow x_1Bx_2 \Rightarrow x_1y_{n+1}x_2 \end{cases}$$

例：

$$\begin{cases} A \to x_1Bx_2 \\ C \to x_1Bx_2 \\ B \to y_1|y_2|\dots|y_n|y_{n+1} \end{cases} \Leftrightarrow \begin{cases} A \to x_1Bx_2|x_1y_1x_2|x_1y_2x_2|\dots|x_1y_nx_2 \\ C \to x_1Bx_2 \\ B \to y_1|y_2|\dots|y_n|y_{n+1} \end{cases}$$

> 这里不能删除 $B \to y_1|y_2|\dots|y_n$：因为 $C \to x_1Bx_2$ 右侧也含 B，如果删除 $B \to y_1|y_2|\dots|y_n$ 则 $C \to x_1Bx_2 \to x_1y_1x_2|x_1y_2x_2|\dots|x_1y_nx_2$ 无法实现.
>
> 这里不能删除 $A \to x_1Bx_2$：因为是部分替换.
>
> 这只是思想实验，实际按照简化流程不应该出现这种情况.

例：

$$\begin{cases} A \to x_1Bx_2 \\ C \to x_1Bx_2 \\ B \to y_1|y_2|\dots|y_n|y_{n+1} \end{cases} \Leftrightarrow \begin{cases} A \to x_1Bx_2|x_1y_1x_2|x_1y_2x_2|\dots|x_1y_nx_2 \\ C \to x_1Bx_2|x_1y_1x_2|x_1y_2x_2|\dots|x_1y_nx_2 \\ B \to y_{n+1} \end{cases}$$

> 这里可以删除 $B \to y_1|y_2|\dots|y_n$：所有右侧含 B 的表达式都被 “部分替换”.
>
> 这里不能删除 $A \to x_1Bx_2$ 和 $C \to x_1Bx_2$：部分替换.

> 这是比较常见的情况，尤其在移除空表达式时. 移除单位表达式也可能见到这种情况（移除单位表达式还可以用 dependency graph）.

1.3 简化流程

Let L be a CFL that does not contain λ. Then, there exists a CFG that generates L and that does not have any useless-, unit-, or λ-production.

因此，所有不含 λ 的上下文无关语言都可以简化.

但要按顺序：先移除 λ-production，再移除 unit production，再移除 useless prodcution.

1.3.1 移除空产生式

λ-production: $A \to \lambda$

Nullable Variable: $A \Rightarrow \ldots \Rightarrow \lambda$

移除空产生式实际上要移除所有 Nullable Variable. 否则，只移除直接产生 λ 的产生式，可能有新的 λ-production 生成.

移除空产生式：把所有形如 $A \to \lambda$ 的 λ-production 都删掉，然后把所有右侧包含 Nullable Variable 的 production **添加**右侧的 Nullable Variable 用 λ 替代的 production（类似 `1.2 部分替换`）. 这一步添加是把 λ 的作用直接体现在 production 上，显式的 $A \to \lambda$ 删除就没有影响了. 注意我们删除的是空产生式，但添加的是所有右侧 Nullable Variable 用 λ 替代的产生式. 其中某些 Nullable Variable 可能并没有被删除动作直接影响到，但还是要添加.

Find a CFG without λ-productions equivalent to the grammar G:

$$
\begin{array}{ll}
Grammar\ G & \\
S \to ABaC & S \to ABaC \mid BaC \mid AaC \mid ABa \mid aC \mid Aa \mid Ba \mid a \\
A \to BC & A \to B \mid C \mid BC \\
B \to b \mid \lambda & B \to b \\
C \to D \mid \lambda & C \to D \\
D \to d & D \to d
\end{array}
$$

A,B, and C are nullable variables

1.3.2 移除单位产生式

形如 $A \to B$

法一：逐个解决.

- 形如 $A \to A$ 的产生式直接移除.
- 形如 $A \to B$ 的，用完全替换/部分替换.

法二：Dependency graph

把所有 unit production 单独抽出来画图. 这些变量在图中的转化不会产生任何终结符. 那么，如果 A 指向 B，那么 B 能产生的 non-unit production A 也可以共享.

所以在原先的 non-unit production 基础上，加上各自可以共享其他人的 non-unit production，就是去掉 unit production 的结果. 因为大家都共享了，所以互相之间的转换被删掉没有影响.

$S \to Aa \mid B$

$B \to A \mid bb$

$A \to a \mid bc \mid B$

Example 6.6

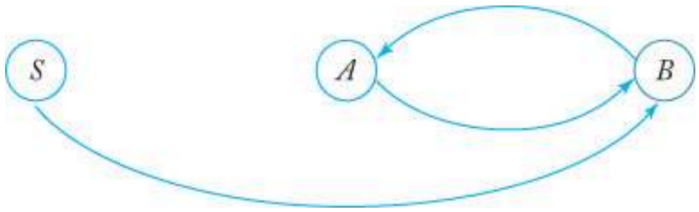
$S \to Aa$
$B \to bb$
$A \to a \mid bc$

Non-unit production

$S \to a \mid bc \mid bb$
$B \to a \mid bc$
$A \to bb$

New rules

$S \to a \mid bc \mid bb \mid Aa$
~~$B \to a \mid bb \mid bc$~~
$A \to a \mid bb \mid bc$



dependency graph

1.3.3 移除无用产生式

无用变量：从 S 无法到达它，或从它无法导出终结符串的变量.

无用产生式：含无用变量的产生式.

If $S \Rightarrow \ldots \Rightarrow xAy \Rightarrow \ldots \Rightarrow w$, where $w \in L(G)$ contains only terminals, then variable A is useful.

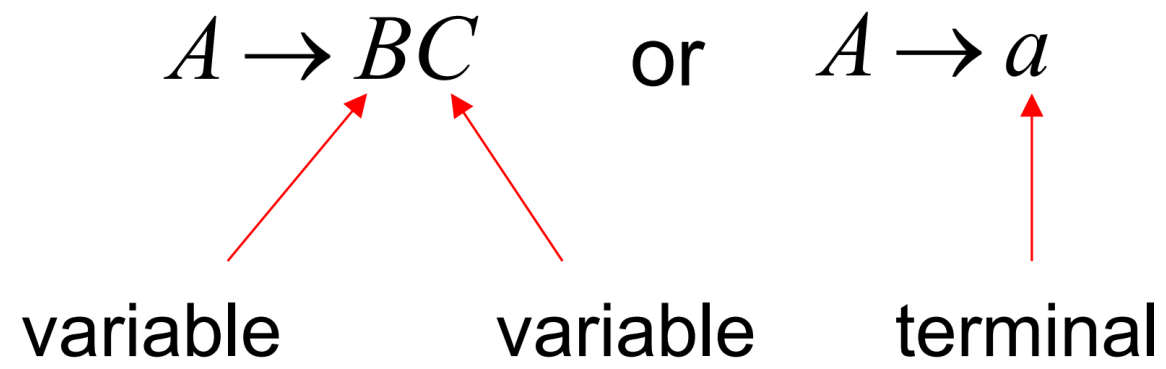
Otherwise, variable A is useless.

无用产生式可以直接移除.

2. 两个范式

2.1 Chomsky 范式

> Chomsky Normal Form (CNF)



CNF 文法的产生式只有两种形式：$A \to BC$（右侧是两个变量）和 $A \to a$（右侧是一个终结符）. 解析树中的每个非终结符节点最多只能分出两个子节点，这意味着**推导树是二叉树**.

对于一个长度为 n 的字符串，其在 CNF 文法中推导树高度至多为 $2n-1$.

注意：B 和 C 可以相同（待补充）.

2.1.1 CFG 转 CNF

定理：对于任何不产生 λ 的 CFG，存在一个等价的 CNF 文法.

> 这里说的 “不产生 λ” 指的是该 CFG 生成的语言中不包含 λ. 只要语言不包含 λ，即使文法中包含形如 $A \to \lambda$ 的产生式，该定理仍适用（见步骤 ①），仍能把该 CFG 转为 CNF.
>
> 定理要求语言中不包含 λ 的原因是 CNF 无法生成 λ.

步骤：

① 移除空产生式和单位产生式.

> 同 `1.3 简化流程` .
>
> 为什么不用移除无用产生式：因为空产生式和单位产生式必然不满足 CNF 的要求，直接移除；无用产生式有可能满足 CNF 的要求，即使是那些不满足的，只要它既不是空产生式也不是单位产生式（这两个都被移除了），那么在后续步骤它可以被转为符合要求的多条产生式（即使无用，也不违反 CNF 的定义）. 所以没必要删除.
>
> 不过处于化简考虑，推荐删除无用产生式.

② 对于右侧长度大于 1 的产生式中的每种终结符 a，引入一个新变量 T_a，将原产生式的 a 替换为 T_a，并添加规则 $T_a \to a$.

> 右侧长度等于 1 的已经满足要求（单位产生式被移除，剩下的只剩形如 $A \to a$ 的产生式）.
>
> 类似于 `1.1 完全替换` 的逆过程，因为完全替换操作前后的文法等价，因此从后到前也能产生等价的文法. 这里说类似完全替换而不是部分替换，因为每引入一个新变量 T_a，只会有一个 T_a 在左侧的产生式 $T_a \to a$，这就是它的全部.

③ 现在，右侧长度大于 1 的产生式不含任何终结符. 处理右侧过长的产生式，确保所有右侧长度大于 1 的产生式只能形如 $A \to BC$（即右侧长度只能为 2）.

Replace any production $A \to C_1C_2 \cdots C_n$ $(n > 2)$ with

$$\begin{aligned} A &\to C_1V_1 \\ V_1 &\to C_2V_2 \\ &\cdots \\ V_{n-3} &\to C_{n-2}V_{n-2} \\ V_{n-2} &\to C_{n-1}C_n \end{aligned}$$

注意，不能分解为 $A \to C_1C_2$，$C_2 \to C_2C_3$，.....，因为这样会无限递归，不等价. 且最后一条可以直接 $V_{n-2} \to C_{n-1}C_n$，不需要再引入 V_{n-1}，因为 $V_{n-2} \to C_{n-1}C_n$ 右侧长度已经为 2 了.

> 这一步类似一系列连环的 `1.1 完全替换` 的逆过程.

这里，$V_1, V_2, \ldots, V_{n-2}$ 是新引入的中间变量.

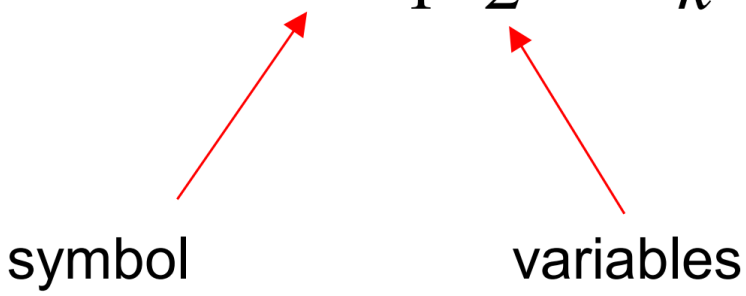
这样，就把任意一个不产生 λ 的 CFG，转化为了一个等价的 CNF 文法.

2.2 Greibach 范式

> Greibach Normal Form (GNF)

若一个 CFG 的所有产生式满足以下形式

$$A \to a\,V_1V_2 \cdots V_k \qquad k \geq 0$$



> 注意，k 可以等于 0.
>
> 注意，右侧必须以**单个**终结符开头. 不能没有，也不能 ≥ 2

> 注意，右侧若干变量可以彼此相同（待补充）

则称为 Greibach 范式. 注意，Greibach 范式无法产生 λ.

注意区分：

① `Lec 3 3.2 右线性文法`

所有产生式具有 $A \to wB$ 或 $A \to w$ 的形式，其中 w 是终结符串（string of terminals）.

例：

$$\begin{aligned} G &= (\{S\}, \{a, b\}, S, P), \\ P : S &\to abS, \\ S &\to a \end{aligned}$$

② `Lec 5 2.1.3 简单文法`

一个上下文无关文法 G 被称为简单文法，如果它满足以下条件：

- 所有产生式都是以下两种形式之一：
  - $A \to aX$
  - $A \to \lambda$
- 唯一性条件：$A \to aX_1$ 和 $A \to aX_2$ 不能同时存在. 即 Pair (A, a) 最多 appears once.

> $A \in V$ 是变量，$a \in \Sigma$ 是任意终结符，$X \in V^*$ 是 string of variables（若干变量组成的字符串，只含变量）.
>
> 唯一性条件和 X 只含变量，确保在解析时，只需要查看输入的下一个符号，就能唯一确定要使用的产生式.

2.2.1 CFG 转 GNF

定理：对于任何不产生 λ 的 CFG，存在一个等价的 GNF 文法.

> 注意：不产生 λ 很重要.

获取这个等价 GNF 相对 CNF 要困难，课件只给了些具体例子：

$$
\begin{aligned}
&S \to AB \\
&A \to aA \mid bB \mid b \\
&B \to b
\end{aligned}
\quad\Rightarrow\quad
\begin{aligned}
&S \to aAB \mid bBB \mid bB \\
&A \to aA \mid bB \mid b \\
&B \to b
\end{aligned}
$$

$$
\begin{aligned}
&S \to abSb \\
&S \to aa
\end{aligned}
\quad\Rightarrow\quad
\begin{aligned}
&S \to aT_bST_b \\
&S \to aT_a \\
&T_a \to a \\
&T_b \to b
\end{aligned}
$$

Greibach
Normal Form

待补充：具体的转化方法.

3. CYK 算法

Recall：

成员资格问题（Membership Question，即 `Lec 5 2.1 解析`）：for context-free grammar G, find if string $w \in L(G)$

成员资格算法（Membership Algorithms）：即 Parsers.

- Exhaustive search parser: $O(P^{2|w|+1})$

  > P 是产生式数量，见 `Lec 5 2.1.1 暴力解析`.

- CYK parsing algorithm: $O(|w|^3)$

> J. **C**ocke
>
> D. H. **Y**ounger
>
> T. **K**asami

算法步骤：待补充.

复杂度：$O(|w|^3)$

> 待补充.
>
> 准确来说是 $O(P \cdot |w|^3)$，但许多时候文法被视为固定（P 视为常数），所以简化为 $O(|w|^3)$.

期中考试

> Midterm

Lec 1 ~ Lec 6 总体分为两部分，Lec 1-4 是正则语言，Lec 5-6 是上下文无关语言. 其中，正则语言是上下文无关语言的一部分，即先从比较好研究的正则语言入手，再拓展到上下文无关语言.

cheatsheet:

解析与歧义

1. Parsing: $w \in L(G)$ or not?
   If yes, 找到导出 w 的 derivation.
   暴力解析: $O(p^{2|w|+1})$
   CYK: $O(|w|^3)$
2. Ambiguity: A CFG is Ambiguity if some $w \in L(G)$ has ≥ 2 derivation trees / leftmost / rightmost derivations

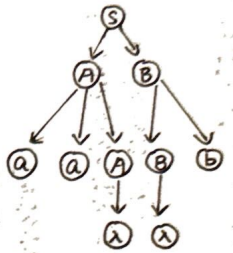
Context-Free Grammars (CFG)

上下文无关

$A \to x$ (单) $x \in (V \cup T)^*$

$\Rightarrow$ Context-Free Languages

推导树



$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$

简单文法 $\Rightarrow x \in V^*$ (变量串)

① $A \to aX$ or $A \to \lambda$

② $A \to aX_1$ 和 $A \to aX_2$ 不能同时存在.

解析: $O(|w|)$

Greibach 范式

$A \to aV_1V_2 \cdots V_k \quad k \geq 0$

语法简化

1. 移除 Nullable Variables (移除 λ-productions)
2. 移除 Unit Productions

陈替换法, 还可用 Dependency Graph

$S \to A \mid a \mid B$
$B \to A \mid bb$
$A \to a \mid bc \mid B^*$

$S \to Aa$	$S \to a \mid bc \mid bb$
$B \to bb$ +	$B \to a \mid bc$
$A \to a \mid bc$	$A \to bb$
原非 unit production	New rules

3. 移除 Useless Productions

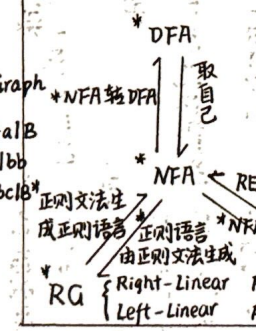
$S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} w \in L(G)$

↳ useful. 其他均 Useless. 含 useless 的 production 直接删.

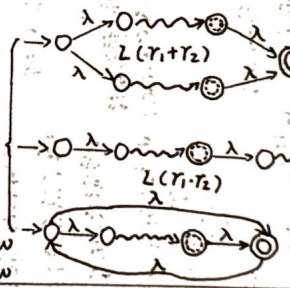
Regular Languages: 被某些 DFA 接受的语言

正则

4 种表示



封闭性



⇒ 证 L 非正则

泵引理 Pumping

O: 声称 m.

P: Pick $w \in L$, $|w| \geq m$.

O: 声称 decomposition $w = xyz$.

P: Pick i, $xy^iz \notin L$. (i 通常取 0 或 2)

Chomsky 范式 (CNF)

$A \to BC$ or $A \to a$

CFG 转 CNF:

① 移除 λ production 和 Unit production

② 右侧长度大于1的 production, 每种终结符 a 替换为变量 T_a, 并 add $T_a \to a$

③ $\forall A \to C_1C_2 \cdots C_n$ $(n>2)$ 转为 $\begin{cases} A \to C_1V_1 \\ V_1 \to C_2V_2 \\ \vdots \\ V_{n-2} \to C_{n-1}C_n \end{cases}$

CYK 算法

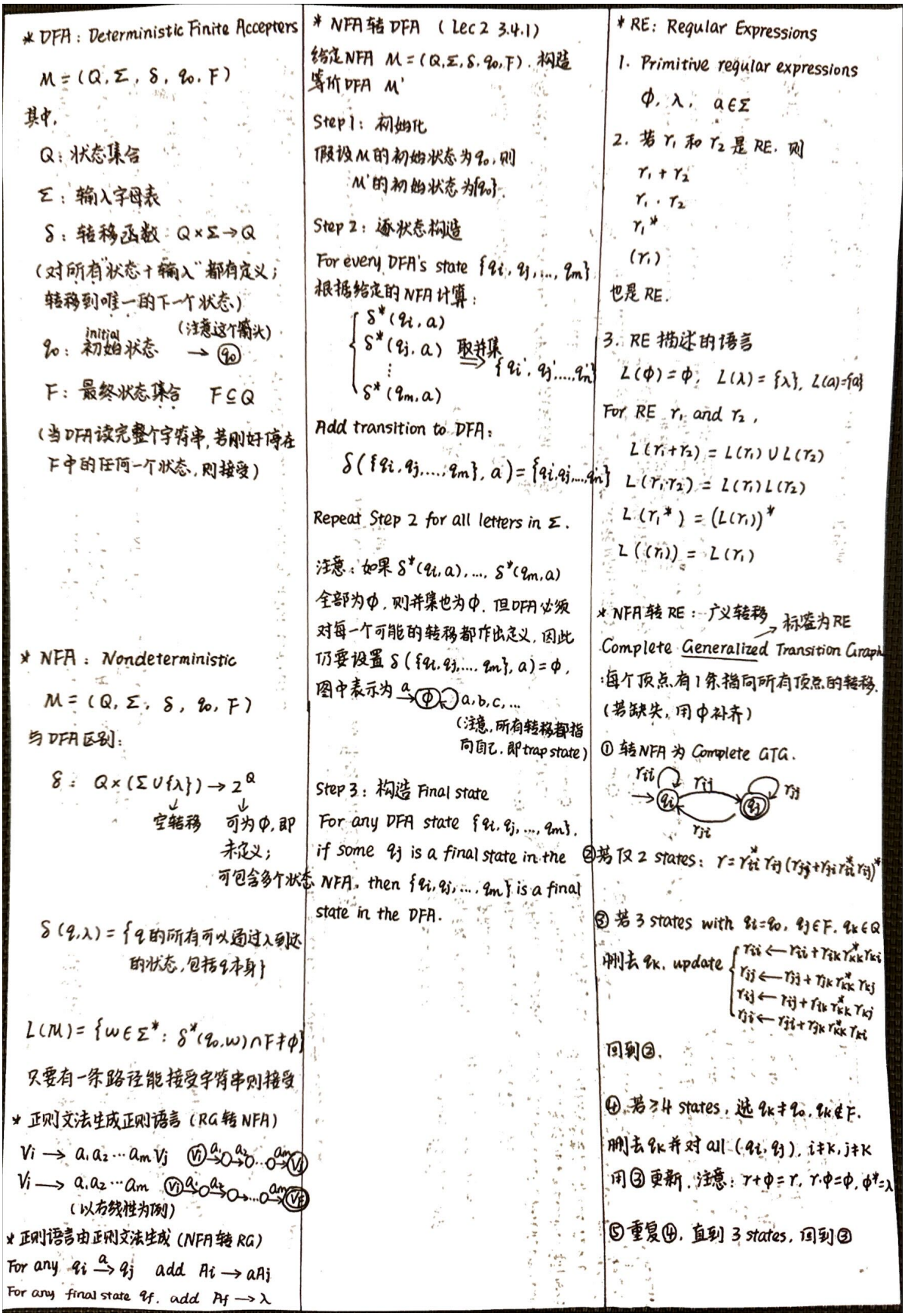
$O(|w|^3)$

例: Is $aabbb \in L(G)$?

G:
$S \to AB$
$A \to BB$
$A \to a$
$B \to AB$
$B \to b$

a	a	b	b	b
A	A	B	B	B
aa	ab	bb	bb	
	S,B	A	A	
	aab	abb	bbb	
	S,B	A	S,B	
	aabb	abbb		
	A	S,B		
	aabbb			

YES ← S, B


* DFA: Deterministic Finite Accepters
M = (Q, Σ, δ, q0, F)
其中,
Q: 状态集合
Σ: 输入字母表
δ: 转移函数 Q×Σ→Q
(对所有"状态+输入"都有定义;
转移到唯一的下一个状态.)
q0: initial 初始状态 (注意这个箭头)
F: 最终状态集合 F⊆Q
(当DFA读完整个字符串,若刚好停在
F中的任何一个状态,则接受)
* NFA: Nondeterministic
M = (Q, Σ, δ, q0, F)
与DFA区别:
δ: Q×(Σ∪{λ}) → 2^Q
空转移
可为φ,即未定义;
可包含多个状态
δ(q,λ) = {q的所有可以通过λ到达的状态,包括q本身}
L(M) = {ω∈Σ*: δ*(q0,ω)∩F≠φ}
只要有一条路径能接受字符串则接受
* 正则文法生成正则语言 (RG转NFA)
Vi → a1a2…amVj
Vi → a1a2…am
(以右线性为例)
* 正则语言由正则文法生成 (NFA转RG)
For any qi →a qj add Ai → aAj
For any final state qf. add Af → λ
* NFA转DFA (Lec2 3.4.1)
给定NFA M = (Q,Σ,δ,q0,F). 构造
等价DFA M'
Step 1: 初始化
假设M的初始状态为q0,则
M'的初始状态为[q0].
Step 2: 逐状态构造
For every DFA's state {qi, qj, ..., qm}
根据给定的NFA计算:
δ*(qi,a)
δ*(qj,a) 取并集 ⇒ {qi', qj', ..., qn'}
δ*(qm,a)
Add transition to DFA:
δ({qi,qj,...,qm}, a) = {qi',qj',...,qn'}
Repeat Step 2 for all letters in Σ.
注意:如果 δ*(qi,a), ..., δ*(qm,a)
全部为φ,则并集也为φ. 但DFA必须
对每一个可能的转移都作出定义,因此
仍要设置 δ({qi,qj,...,qm},a) = φ,
图中表示为 a, b, c, ...
(注意,所有转移都指
向自己,即trap state)
Step 3: 构造 Final state
For any DFA state {qi, qj, ..., qm},
if some qj is a final state in the
NFA, then {qi, qj, ..., qm} is a final
state in the DFA.
* RE: Regular Expressions
1. Primitive regular expressions
φ, λ, a∈Σ
2. 若 r1 和 r2 是 RE, 则
r1 + r2
r1 · r2
r1*
(r1)
也是 RE.
3. RE 描述的语言
L(φ) = φ, L(λ) = {λ}, L(a) = {a}
For RE r1 and r2,
L(r1+r2) = L(r1) ∪ L(r2)
L(r1r2) = L(r1)L(r2)
L(r1*) = (L(r1))*
L((r1)) = L(r1)
* NFA转RE: 广义转移 → 标签为RE
Complete Generalized Transition Graph
每个顶点有1条指向所有顶点的转移.
(若缺失,用φ补齐)
① 转NFA为 Complete GTG.
② 若仅2 states: r = r11* r12 (r22 + r21 r11* r12)*
③ 若3 states with qi=q0, qj∈F, qk∈Q
删去qk, update
rii ← rii + rik rkk* rki
rij ← rij + rik rkk* rkj
rji ← rji + rjk rkk* rki
rjj ← rjj + rjk rkk* rkj
回到②.
④ 若≥4 states, 选 qk≠q0, qk∉F.
删去qk并对 all (qi, qj), i≠k, j≠k
用③更新. 注意: r+φ=r, r·φ=φ, φ*=λ
⑤ 重复④, 直到3 states, 回到②

Lec 7 下推自动机

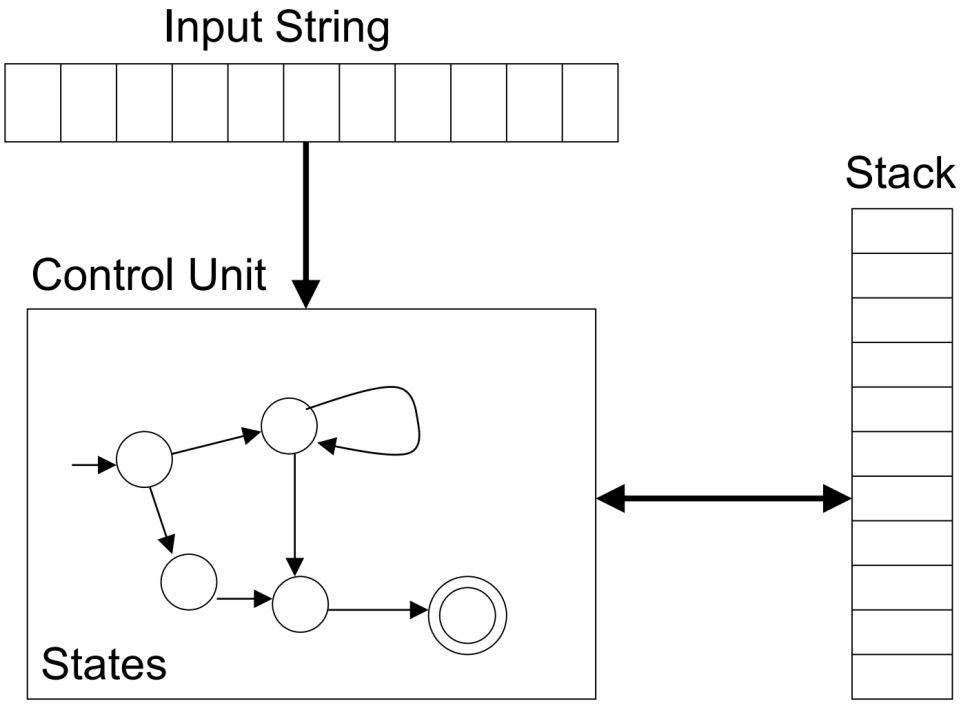
7.1 NPDA 非确定性下推自动机

> Nondeterministic Pushdown Automata

7.1.1 PDA 下推自动机

> Pushdown Automaton

Pushdown Automaton（PDA）的结构可视化：

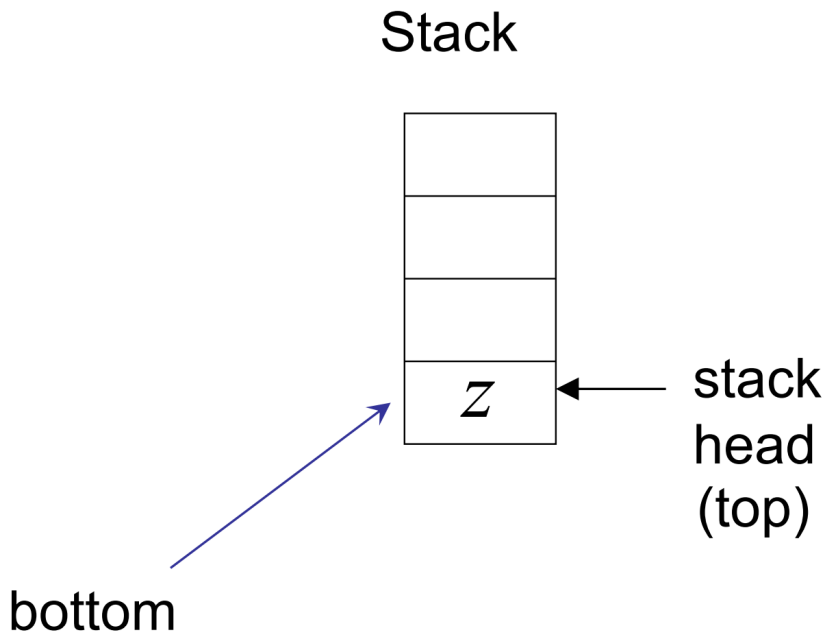


关键点：PDA 使用一个栈（Stack）来无限地计数（Count without limit），从而能识别如 $L = \{a^n b^n : n \geq 0\}$ 这种需要匹配数量的语言，同时检查所有 a 都在第一个 b 之前出现.

PDA 结构：

- 输入字符串（Input String）
- 控制单元 / 状态（Control Unit / States）
- 栈（Stack）

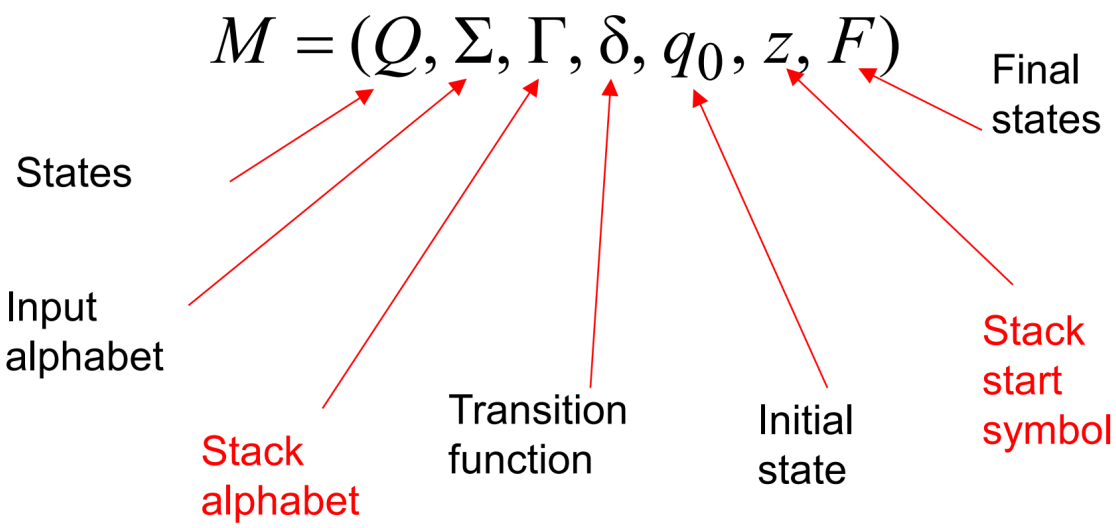
7.1.2 栈



栈有一个栈底（bottom），栈头 / 栈顶（stack head / top），以及一个初始栈符号 z（Initial Stack Symbol）.

7.1.3 NPDA 非确定性下推自动机

> Non-Deterministic Pushdown Automaton (NPDA)



栈字母表 Γ（Stack alphabet）：包含自动机在栈中存储和操作的所有符号. 通常，Input alphabet Σ 是 Stack alphabet Γ 的子集（$\Sigma \subseteq \Gamma$，但定义中没有强调这一点）. Γ 还会包含一些 Σ 中没有的特殊符号，如栈底符号 z.

① 转移函数

转移函数（Transition function）：

$$\delta(q_1, u, s_1) = \{(q_2, s_2)\}$$

- $q_1 \in Q$：当前状态（Current state）
- $u \in \Sigma \cup \{\lambda\}$：当前输入**符号**（Current input symbol）
- $s_1 \in \Gamma$：当前栈顶**符号**（Current symbol on top of the stack）
- $q_2 \in Q$：下一状态（Next state）
- $s_2 \in \Gamma^*$：代替 s_1 的栈顶**字符串**.

> 下推自动机的转移分为两个部分：状态的移动和栈的变化. 其中，状态的移动和 NFA 类似，栈的变化可以理解为弹出 s_1，压入 s_2.
>
> 本课程关注理论模型，不关注编程实现. 在实际编程中，可以用链表实现一个栈结构，此时 s_1 即栈头地址所指向节点中存储的值.
>
> s_2 和 s_1 的定义不同，是为了用一个转移步骤实现多种功能：
>
> 替换（Replace）：如果 s_2 是单个符号，则把 s_1 替换为另一个符号.
>
> 弹出（Pop）：如果 $s_2 = \lambda$，则弹出 s_1，没有新符号压入.
>
> 压入（Push）：如果 s_2 是一个多字符的字符串，它允许机器在一次转移中压入多个符号，对模拟上下文无关文法的推导非常有用. 特别地，如果 s_2 是 s_1 后跟一个字符串， 则效果等价于不弹出 s_1，直接压入后面跟着的字符串.

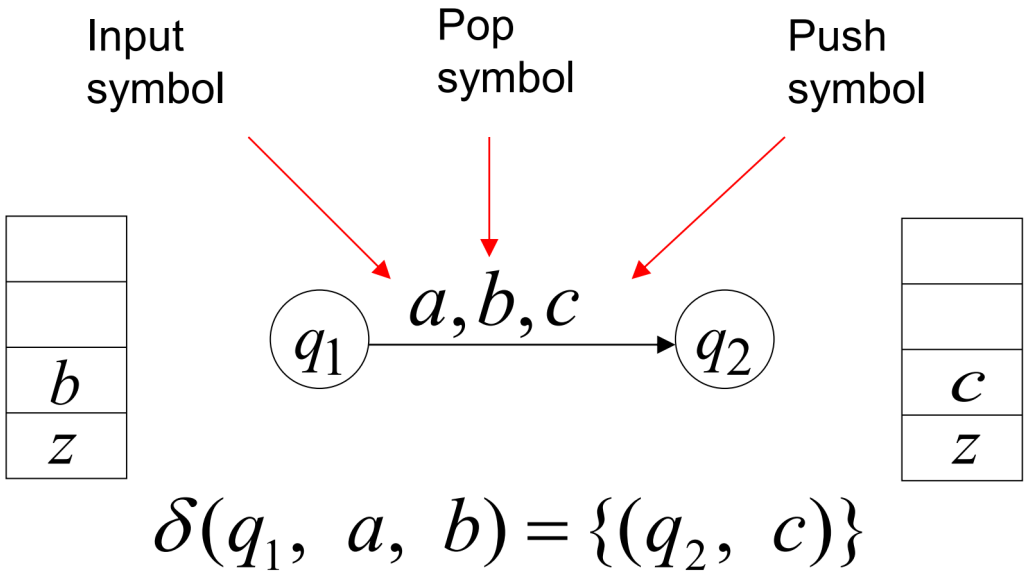
- $\delta(q_1, u, s_1) = \{(q_2, s_2)\}$ 等号右侧是一个集合，因为 NPDA 对于给定的 q_1, u, s_1 可能有零个、一个或多个下一步的移动选择. 集合的每个元素 (q_2, s_2) 都是一个可能的下一状态和压入字符串 pair.

② 转移图

> Labels on the Edges of Transition Graphs

和 NFA 类似，转移图由状态和有向边组成. 每条边表示一种可能的转移.

其中，每条边的标签由三部分构成：



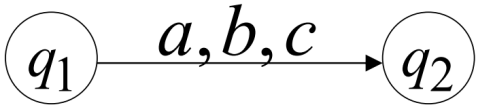
$$\delta(q_1,\ a,\ b) = \{(q_2,\ c)\}$$

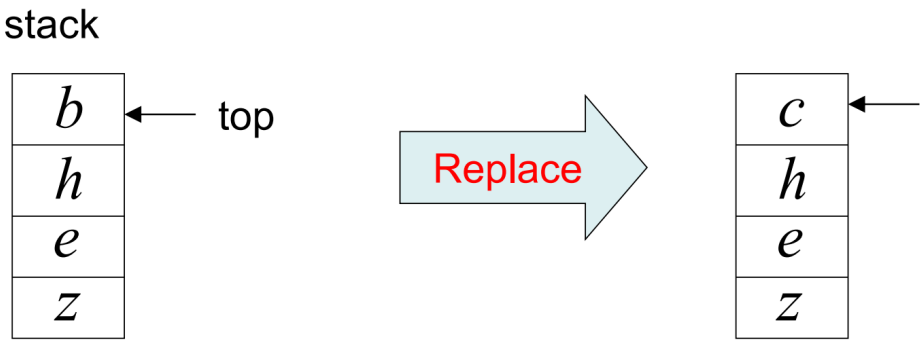
> 注意位置不要搞混. 分别对应定义中的 u, s_1, s_2

一次转移可以弹 1 推 1、弹 1 推多、只推不弹（不规范）、只弹不推、不弹不推（不规范）.

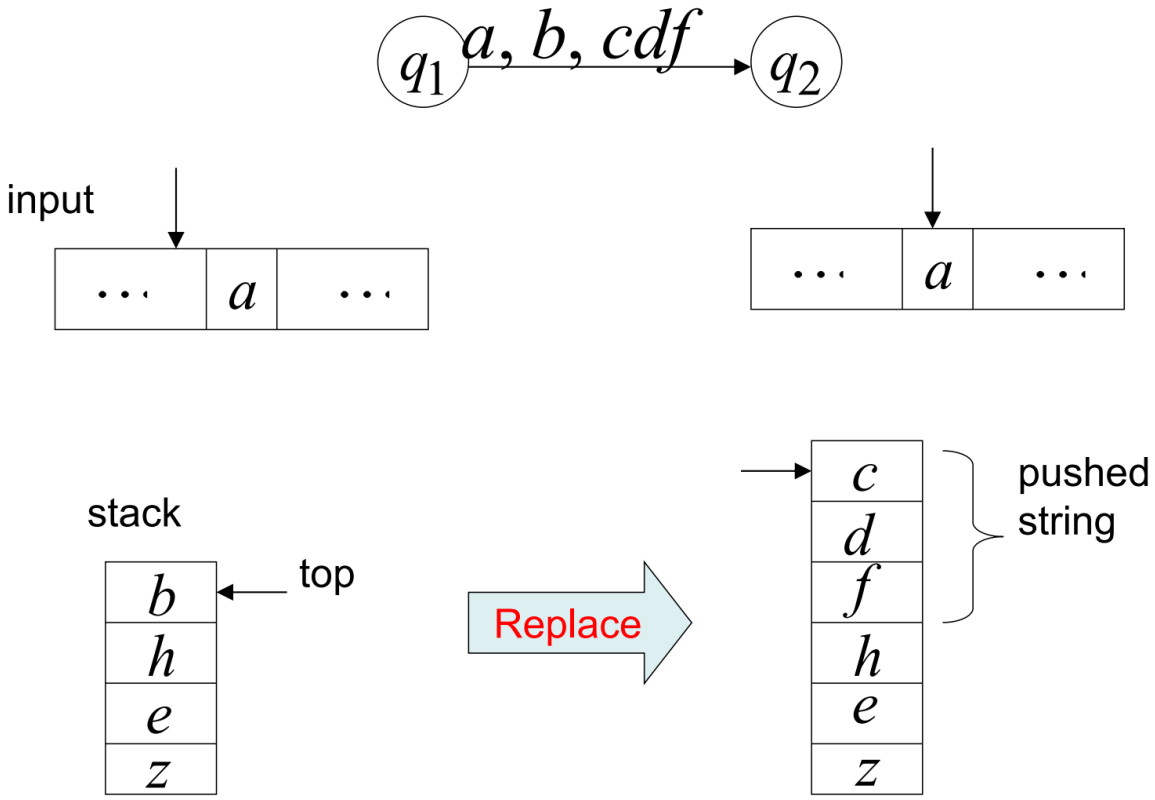
> 注意：不允许一次 pop 多个字符.

弹 1 推 1：



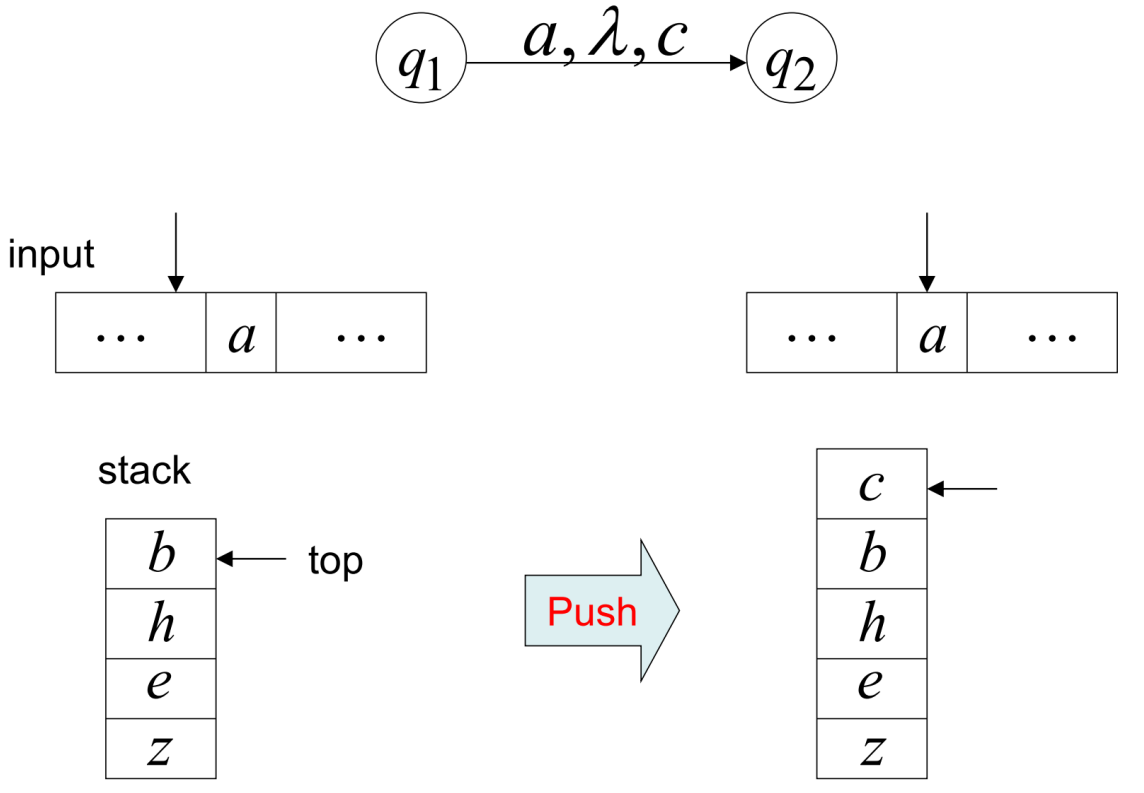


也可以弹 1 推多：



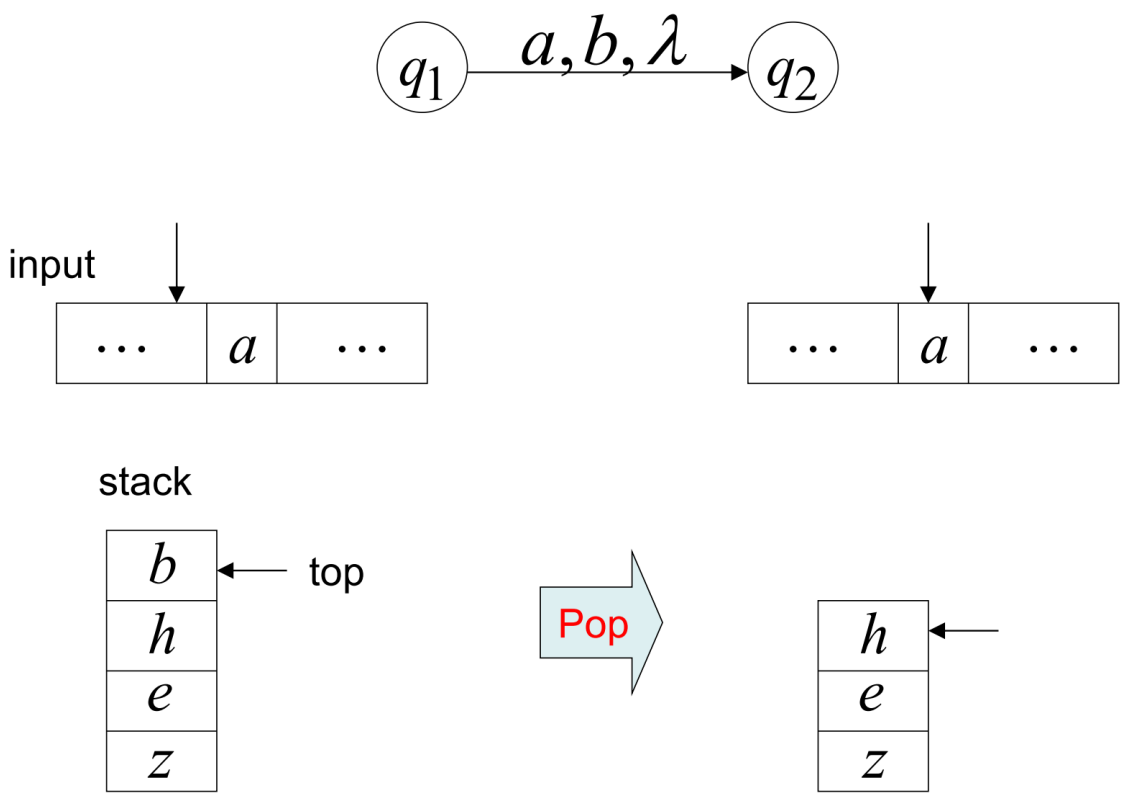
> 注意顺序，如果推入 $s_2 = cdf$，则新的栈顶为 c，然后是 d 和 f.

也可以只推不弹：



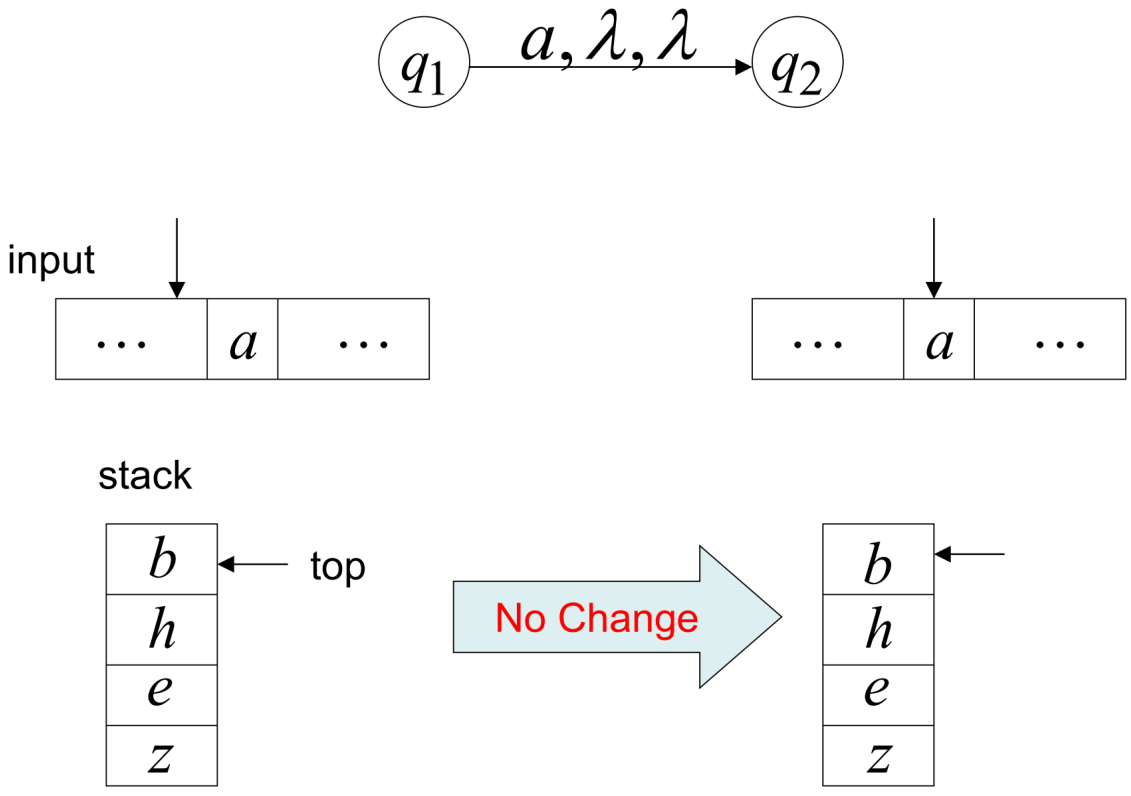
> 注意：这里不是标准写法，因为栈字母表 Γ 通常不包含 λ.
>
> 标准写法：$q_1 \xrightarrow{a,b,cb} q_2$，即 $\delta(q_1, a, b) = \{(q_2, cb)\}$

也可以只弹不推：



> 这里是标准写法，因为 $s_2 \in \Gamma^*$ 可以为 λ.

不弹不推：

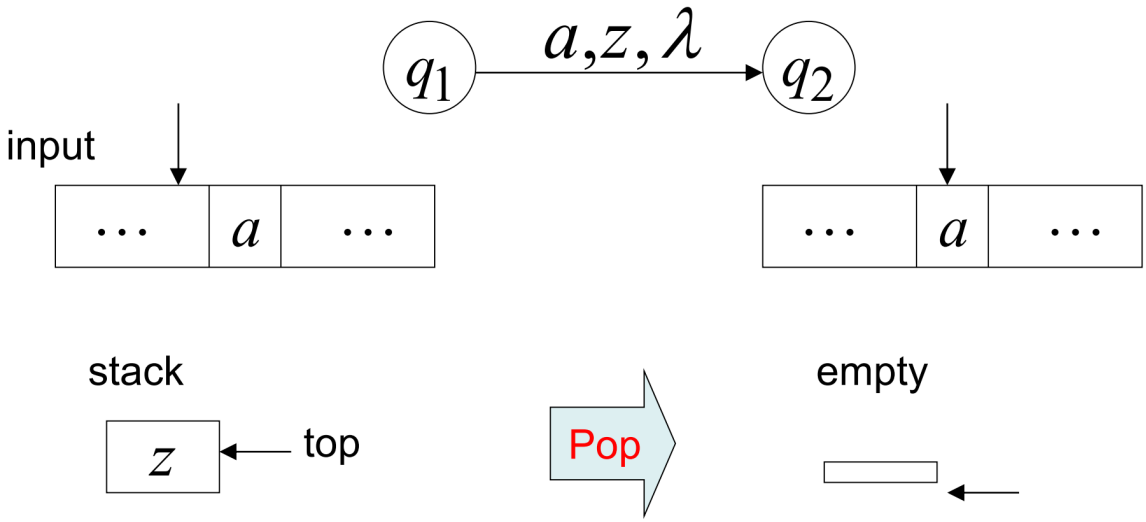


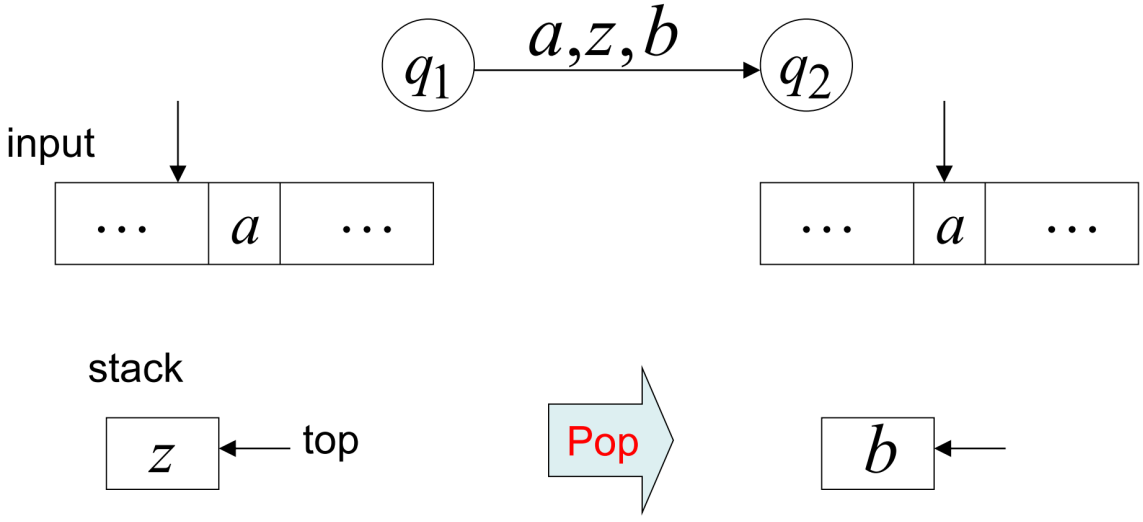
> 这里不是标准写法.

> 标准写法：$q_1 \xrightarrow{a,b,b} q_2$，即 $\delta(q_1, a, b) = \{(q_2, b)\}$

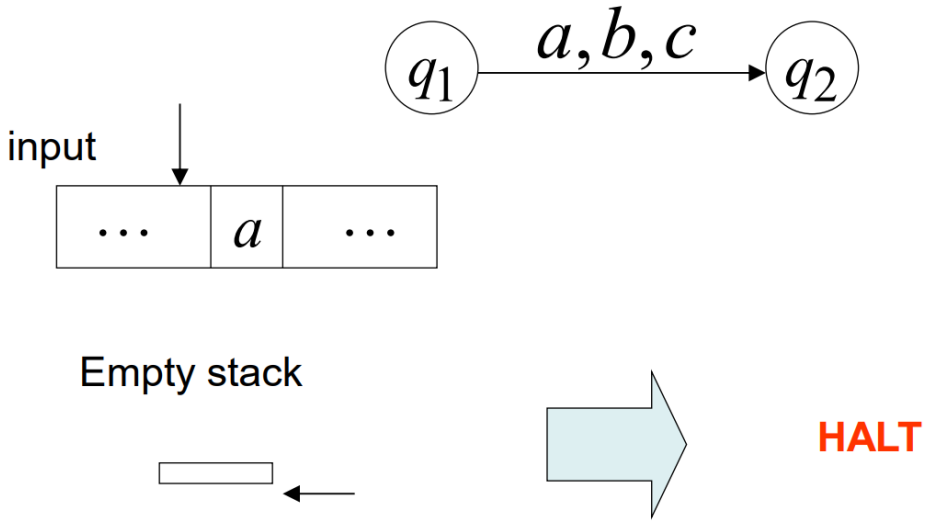
再次强调：**不允许一次 pop 多个字符**（但是可以 push 多个）

初始栈符号 z 允许弹出：

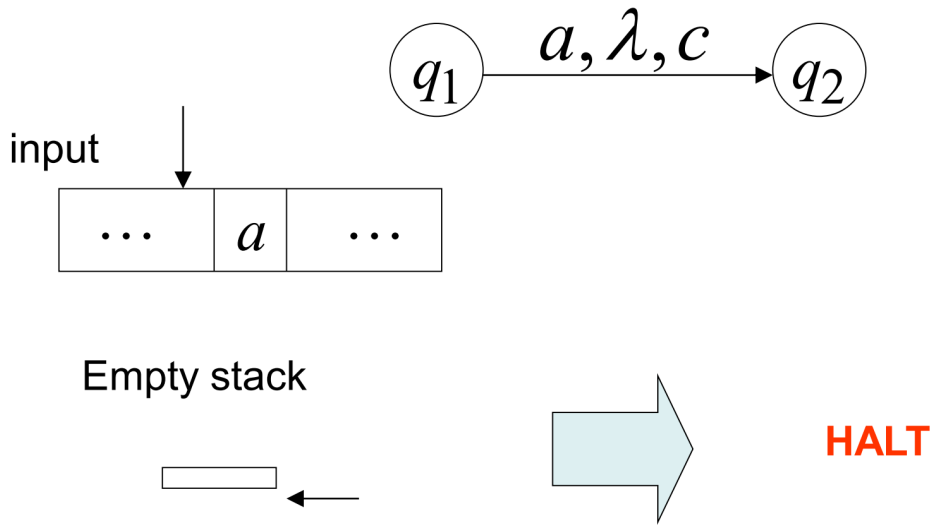




空栈不允许转移：



The automaton **Halts** in state q_1
and **Rejects** the input string



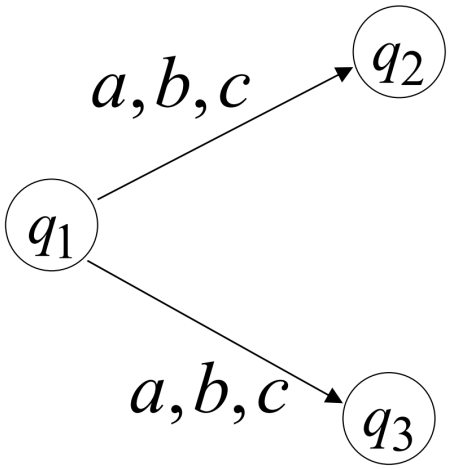
The automaton **Halts** in state q_1
and **Rejects** the input string

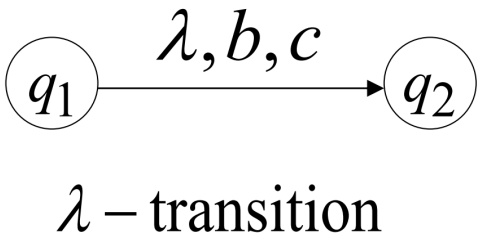
No transition is allowed to be followed When the stack is empty —— 只要看到空栈，直接判定不能继续转移，不需要看 pop 和 push 什么.

> 如果不能继续转移，但是还有未处理的 input symbol，则该路径拒绝. 若所有可能路径都拒绝，则拒绝该字符串.

③ 非确定性

”非确定性“ 指以下行为被允许：





λ – transition

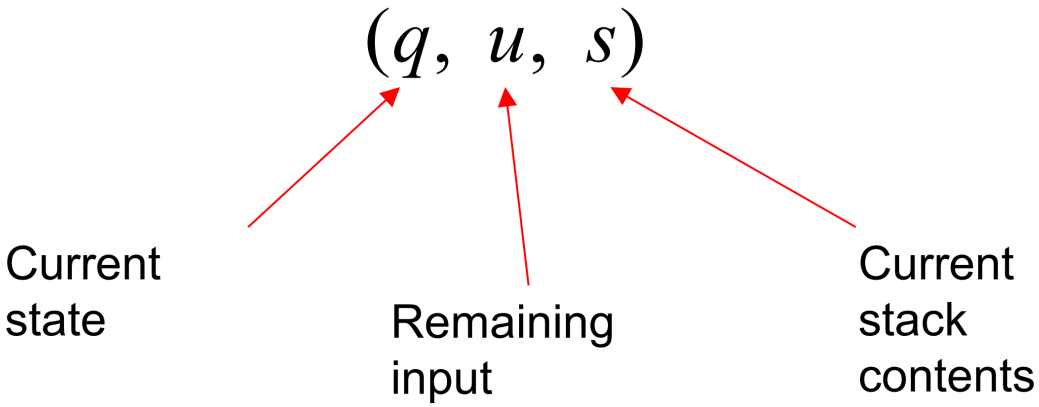
> 以及允许转移函数对某些输入没有作定义.

④ 瞬时描述

> Instantaneous Description (ID)

瞬时描述用于精确表示 PDA 在计算过程中的某一时刻的完整状态.

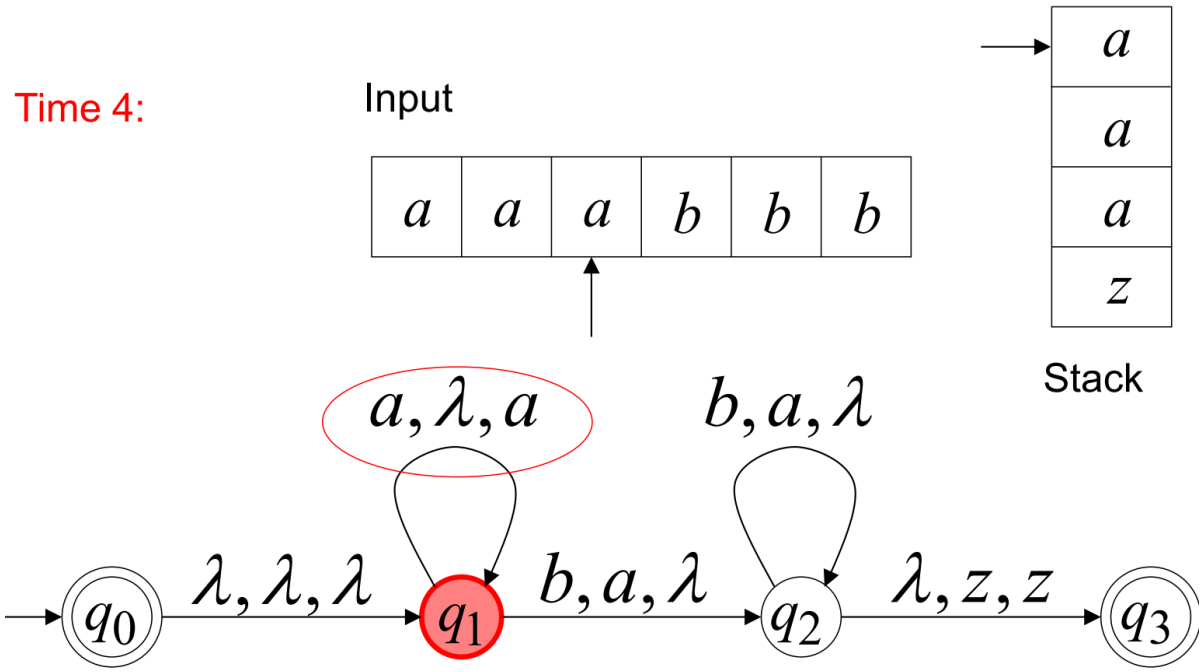
一个 ID 即一个三元组：



Example:

$$(q_1, bbb, aaaz)$$

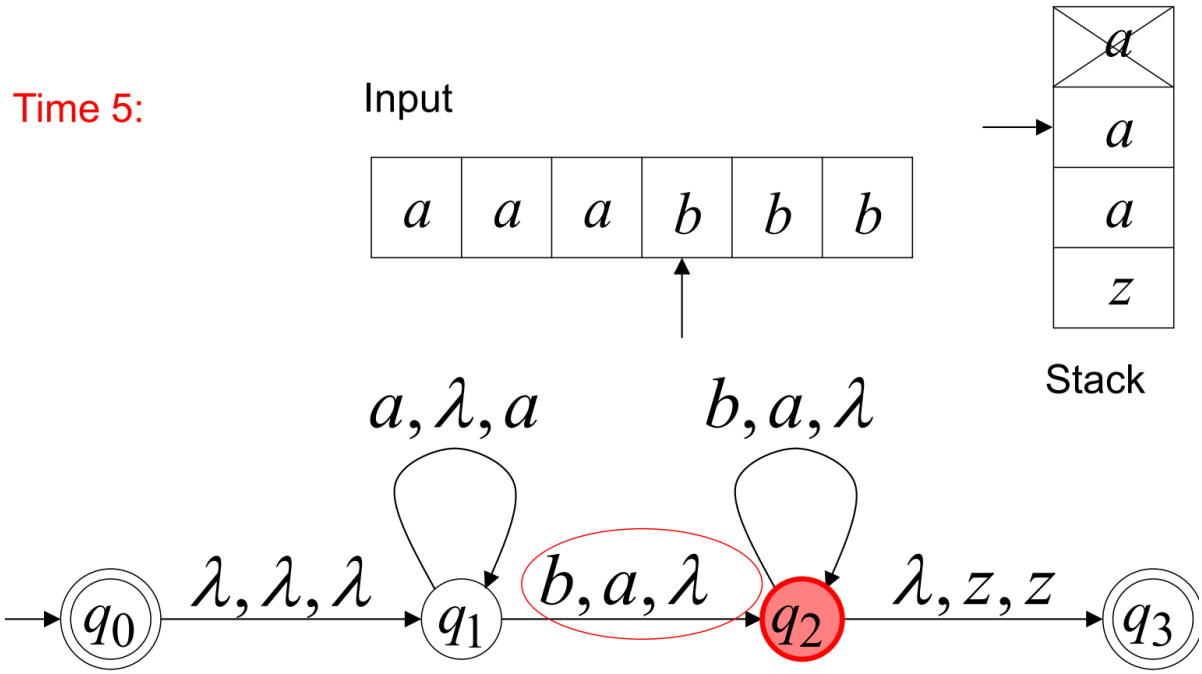
Time 4:



Example: **Instantaneous Description**

$$(q_2, bb, aaz)$$

Time 5:

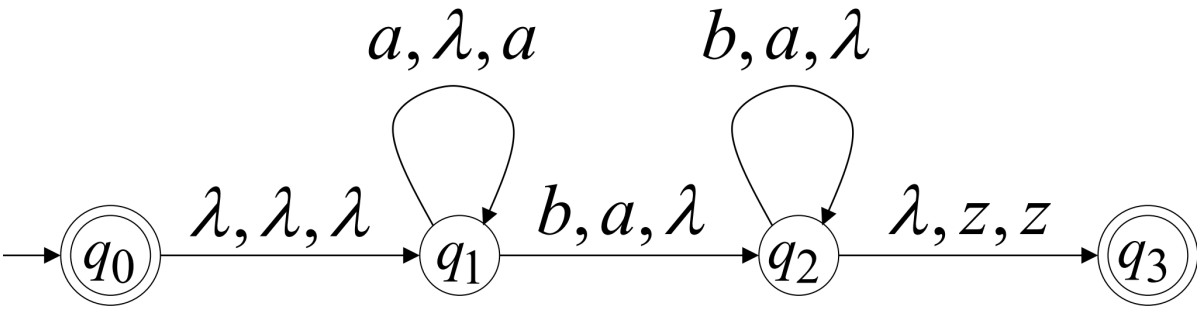


The symbol "⊢" denotes a move from one ID to another.

We write:

$$(q_1, bbb, aaaz) \vdash (q_2, bb, aaz)$$

例：



A computation:

$$\begin{aligned}(q_0, aaabbb, z) &\vdash (q_1, aaabbb, z)\\ \vdash (q_1, aabbb, az) &\vdash (q_1, abbb, aaz)\\ \vdash (q_1, bbb, aaaz) &\vdash (q_2, bb, aaz)\\ \vdash (q_2, b, az) &\vdash (q_2, \lambda, z) \vdash (q_3, \lambda, z)\end{aligned}$$

For convenience we write:

$$(q_0, aaabbb, z) \overset{*}{\vdash} (q_3, \lambda, z)$$

⑤ 被 NPDA 接受的语言

A string is accepted if there is a computation such that:

All the input is consumed **and** the last state is a final state

At the end of the computation, we do not care about the stack contents.

如果不被接受，即被拒绝.

拒绝准则：一个字符串被 NPDA 拒绝，当且仅当所有计算路径都不能接受该字符串，即在每条计算中：

- 输入无法被消耗，或
- 输入被消耗完，但最后的状态不是 Final state，或
- 栈头移到栈底以下（尝试对空栈进行 pop 操作）

被 NPDA 接受的语言：所有被接受字符串的集合.

形式化定义：

Language $L(M)$ of NPDA M:

$$L(M) = \{w : (q_0, w, z) \overset{*}{\vdash} (q_f, \lambda, s)\}$$

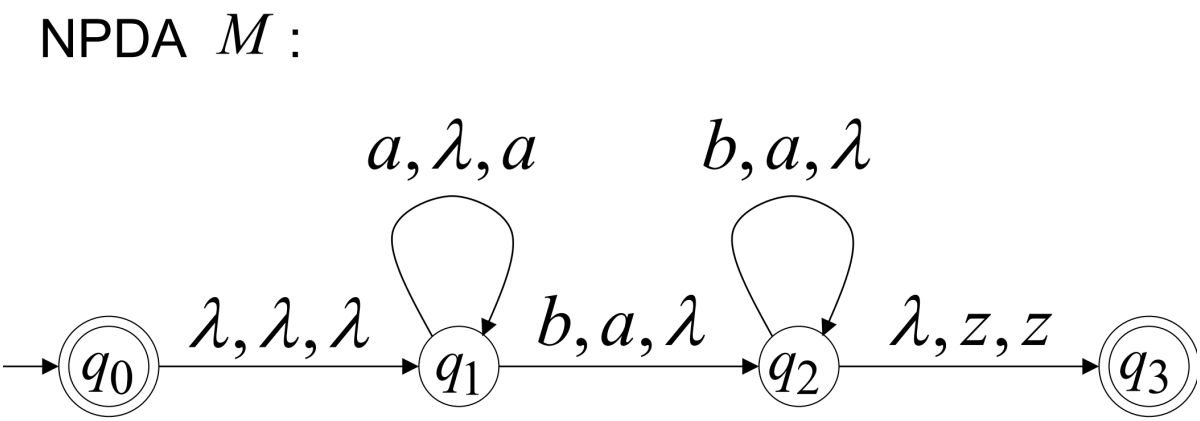
其中，(q_0, w, z) 是起始瞬时描述，(q_f, λ, s) 是最终瞬时描述. 字符串被 NPDA 接受，即存在一条计算路径，使得机器从起始配置经过某些转移后，到达最终配置.

注意，最终瞬时描述要满足：

- 状态为 final state；
- Remaining input 为 λ；
- 栈中内容 $s \in \Gamma^*$ 可以为合法的任意内容，不关心.

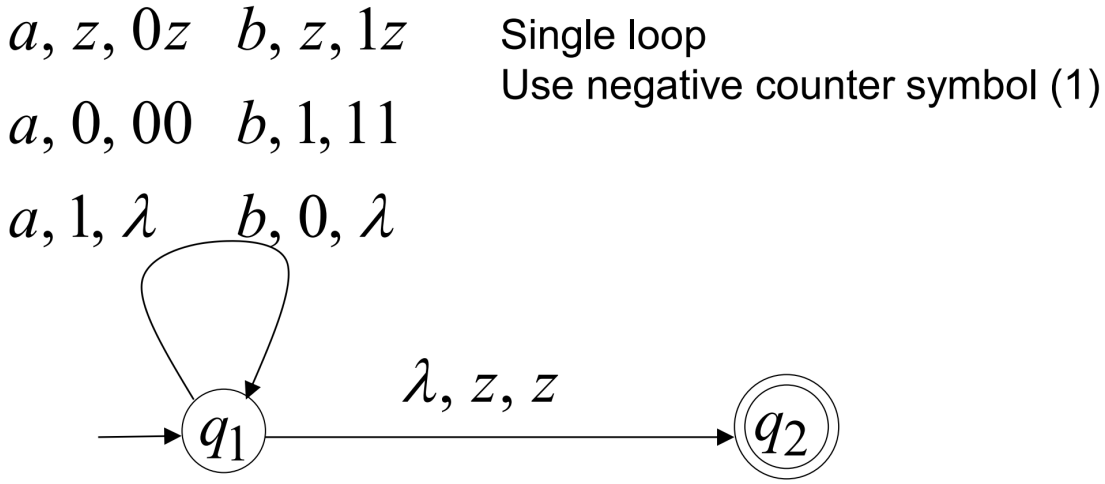
⑥ 例子

Example 1 $L(M) = \{a^n b^n : n \geq 0\}$



这里所有的 a 必须在 b 之前，因此先对输入 a 进行累积（push），然后用输入 b 进行抵消（pop），最后恰好抵消才能进入 final state.

Example 2 $L(M) = \{w : n_a = n_b\}$

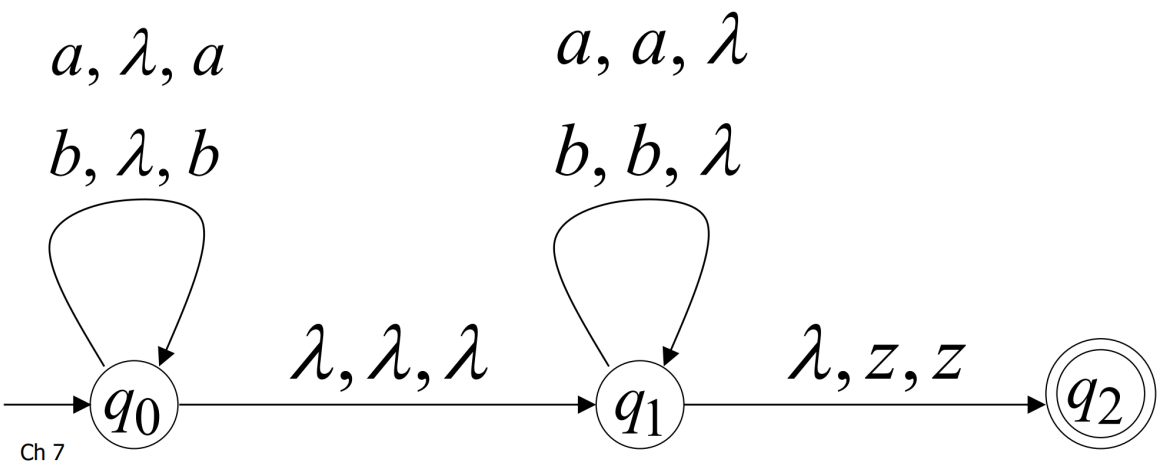


因为不关心 a, b 的顺序，只关心数量，因此可以给 a, b 输入分别引入计数符号. 每个瞬时，0 和 1 的数量可以反映 a 和 b 的相对数量. 其中，0 的数量表示 a 比 b 多的数量，1 的数量表示 b 比 a 多的数量. 每接受一个 a，可以选择 push 一个 0 或 pop 一个 1（根据当前栈顶来选择操作，总有一个操作是可实现的），二者对相对数量的影响一致（使 a 的相对数量增加 1 / 使 b 的相对数量减少 1）；每接受一个 b，

可以选择 push 一个 1 或 pop 一个 0，二者对相对数量的影响也一致（使 b 的相对数量增加 1 / 使 a 的相对数量减少 1）. 注意，如果选择的是 push，一定能保证栈下方没有任何表示另一个 input symbol 的栈字符. 如，接受 a 时，如果栈顶是 0，则一定能保证整个栈中没有任何的 1，如果栈顶是 1，则能保证栈中没有任何的 0（从栈顶往栈底递推）. 对于任意一个瞬时，如果栈中只有 z，则已接受的字符串中 $n_a = n_b$，此时可以空转移到 q_2（但是空转移到 q_2 后如果仍有未接受的字符，也不算 accept）.

Example 3 $L(M) = \{ww^R\}$

栈有后入先出的特性，配合适当的转移规则能很好匹配回文（palindrome）.

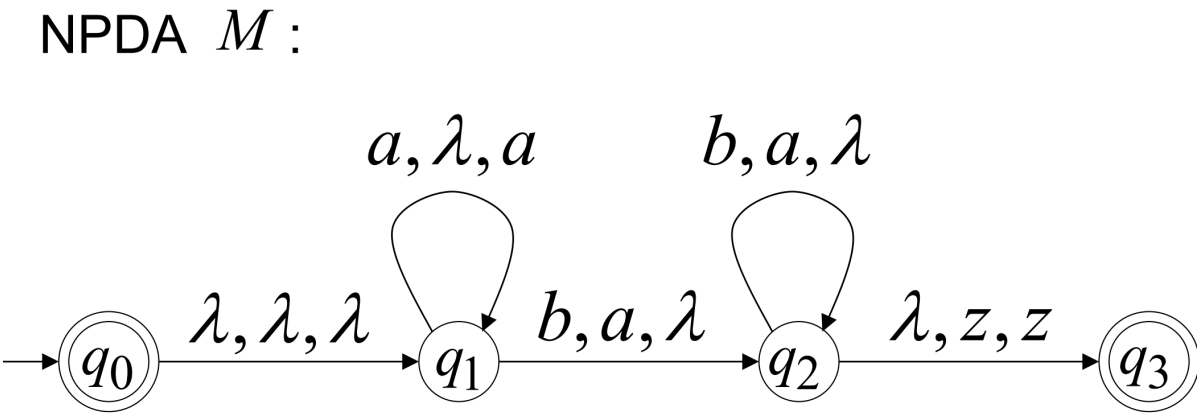


q_0 的 loop 用 push 处理前半段，q_1 的 loop 用 pop 处理后半段. 只有 push 顺序和 pop 顺序恰好相反，栈才能在处理结束后只剩 z. 满足这一条件的字符串即回文（palindrome）.

Example 4 $L(M) = \{a^n b^m : n \geq m - 1\}$

> 注意，课件的答案有误.

Recall Example 1 $L(M) = \{a^n b^n : n \geq 0\}$:



只需在此基础上，在最后一条转移中添加两个：b, z, z（允许 b 多一个）和 b, a, λ（允许 a 多任意个）.

7.2 NPDA 与上下文无关语言

> Pushdown Automata and Context-Free Languages

定理：$\{\text{Context-free Languages}\} = \{\text{Languages Accepted by NPDAs}\}$

注意逻辑：研究正则语言时，我们用 DFA 定义正则语言，然后在分别证明另外三种表示（NFA / 正则文法 / 正则表达式）和 DFA 的等效性.

研究上下文无关语言时，我们用上下文无关文法来定义上下文无关语言，然后再证明 NPDA 的等效性.

定理证明分为两步：

$\{\text{Context-free Languages}\} \subseteq \{\text{Languages Accepted by NPDAs}\}$

和

$\{\text{Context-free Languages}\} \supseteq \{\text{Languages Accepted by NPDAs}\}$

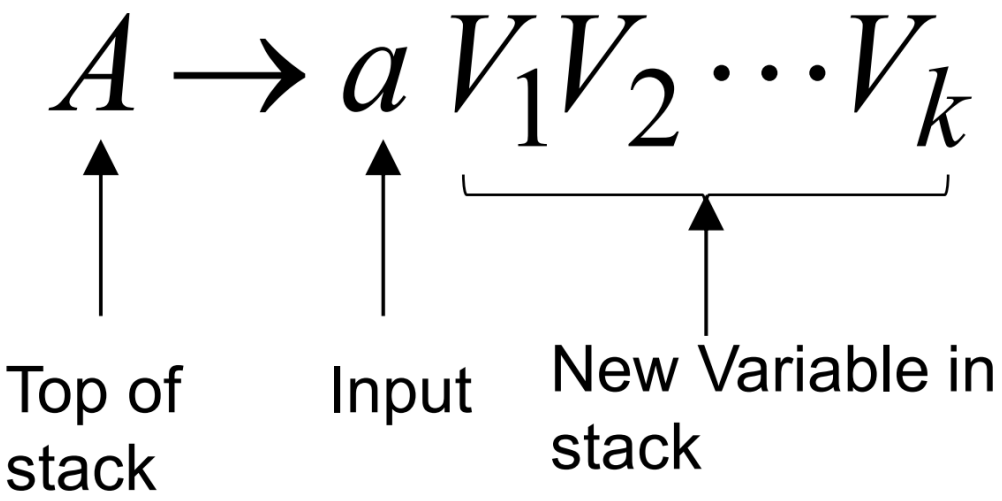
7.2.1 CFG $\rightarrow$ NPDA

Convert any context-free grammar G to an NPDA M with $L(G) = L(M)$

法一：Greibach 范式-最左推导

对不生成 λ 的 CFG，首先，将 G 转为 Greibach 范式.

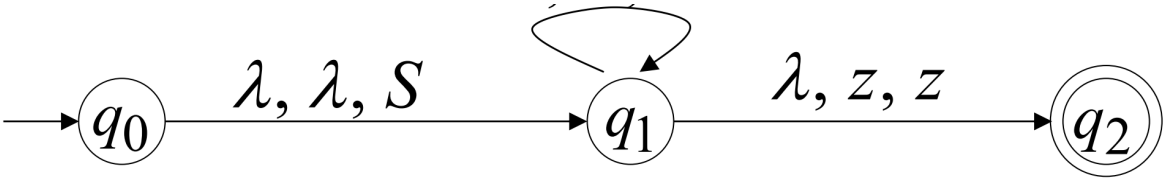
> 不生成 λ 的 CFG 一定存在等价的 Greibach Normal Form.
>
> 对于能生成 λ 的 CFG 要特殊处理. 见步骤 ⑤.



然后构建一个 NPDA M 来模拟 G 的最左推导.

> 注意，模拟最左推导和任意顺序推导在能力上是等价的，即由任意顺序推导出的字符串也一定能由最左推导得到.
>
> 选择模拟最左推导而不是任意顺序推导，因为栈每次只能弹出顶部元素，如果栈顶到栈底对应剩余变量串从左到右，那么不断弹出栈顶相当于不断对最左变量进行推导.

机制：使用栈来存储推导过程中未被消解的变量串，每次转移表示一次推导. 通常维护三个状态：



应用 ①~④ 构建 NPDA

① 首先，构建一条转移：

$$\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$$

> 即 push 初始变量 S 到栈顶，然后进入 q_1.
>
> 这里还未使用 $S \to aV_1V_2 \dots V_k$，只是先把 S push 入.

即 $q_0 \xrightarrow{\lambda, z, Sz} q_1$（课件不规范写法：$q_0 \xrightarrow{\lambda, \lambda, S} q_1$）

② 对每个产生式 $A \to aV_1V_2 \dots V_k$，构建一条转移：

$$(q_1, V_1V_2 \dots V_k) \in \delta(q_1, a, A)$$

> 这里包括 $S \to aV_1V_2 \dots V_k$，即左侧为初始变量的产生式.

即 $q_1 \xrightarrow{a, A, V_1V_2 \dots V_k} q_1$，在图中体现为 loop（可以有多个）.

> 因为每次只能 pop 栈顶，即最左推导，那么生成的 symbol 一定是 input 的**未读取部分**中的最左 symbol. 因为在推导式中，这个 symbol 左侧不会有变量，且左侧如果有 symbol 也是被读过的（左侧 symbol 由左侧变量生成，这个生成在之前就完成了）.

③ 对每个产生式 $A \to a$（右侧无变量），构建一条转移：

$$(q_1, \lambda) \in \delta(q_1, a, A)$$

> 这里直接 pop 栈顶变量，然后读取对应的 symbol，不 push 新的变量.

即 $q_1 \xrightarrow{a, A, \lambda} q_1$，在图中体现为 loop（可以有多个）.

④ 当所有变量都通过上述转移被消解，栈中剩余 z. 构建转移：

$$\delta(q_1, \lambda, z) = \{(q_2, z)\}$$

> 栈中只剩 z 表明推导结束（没有剩余变量），通过空转移到最终状态.

即 $q_1 \xrightarrow{\lambda,z,z} q_2$. 其中 q_2 是 final state.

⑤ 特殊情况：若原 CFG 能生成 λ，则不能转 GNF. 但注意到，无论该 CFG 一开始是否有 $S \to \lambda$，如果我们通过 `Lec 6 1.3.1 移除空产生式` 的等价变换方法，移除掉 $S \to \lambda$ 之外的所有空产生式，则等价的 CFG 一定有 $S \to \lambda$.

> 原先有的自然有，原先没有的也会在移除过程中添加这一条，因为是等价变换，变换后也要能够生成 λ. 而移除掉 $S \to \lambda$ 之外的所有空产生式后只剩这一条能让该 CFG 生成 λ，所以它一定存在.

因此，操作如下：

用 `Lec 6 1.3.1 移除空产生式` 移除其他空产生式 $A \to \lambda$（若有）. 此时保留 $S \to \lambda$（一定有），然后把除 $S \to \lambda$（一定有）之外的产生式看成一个新的 CFG，并转 GNF 进行 ①~④.

> 一定能转 GNF，因为唯一一条能生成 λ 的产生式被排除在外
>
> 这里新的 CFG 就是生成 $L - \{\lambda\}$ 的 CFG.

最后，添加一条转移单独处理 $\lambda \in L$：

$$(q_2, z) \in \delta(q_0, \lambda, z)$$

即 $q_0 \xrightarrow{\lambda,z,z} q_2$.

> 从起始状态直接空转移到 final state.

这个特殊情况结合 `7.2` 的定理，可以得出结论：

L 是上下文无关语言，当且仅当 $L - \{\lambda\}$ 是上下文无关.

> 若 $\lambda \notin L$，则 $L - \{\lambda\} = L$，显然成立；
>
> 若 $\lambda \in L$，则由特殊情况 ⑤ 可知：
>
> - 从 L 的 CFG 可构造出 $L - \{\lambda\}$ 的 CFG（移除 $S \to \lambda$ 之外的空产生式，保留 $S \to \lambda$，然后除 $S \to \lambda$ 之外的产生式就是 $L - \{\lambda\}$ 的 CFG）；
> - 从 $L - \{\lambda\}$ 的 CFG 也可以构造出 L 的 CFG（添加 $S \to \lambda$） 即可；
> - 结合 `7.2` 的定理也可以说，上下文无关必有 NPDA，从 $L - \{\lambda\}$ 的 NPDA 可以构造出 L 的 NPDA（添加一条起点到终点的空转移即可），有 NPDA 就是上下文无关.
> - 综上，L 和 $L - \{\lambda\}$ 同为上下文无关/非上下文无关.

法二：读取推导分离

> 更通用，不需要特殊考虑 $\lambda \in L$.

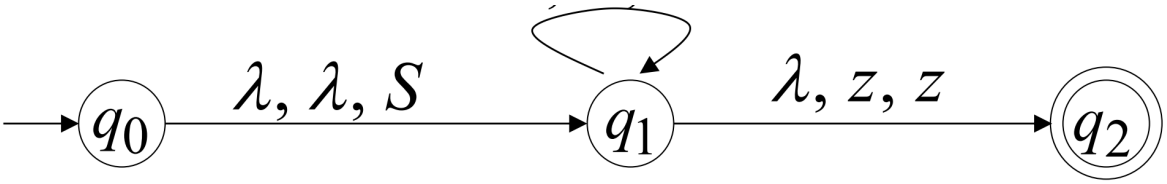
法一的特点是先转 Greibach 范式，然后读取和消变量同时进行.

法二不需要转 Greibach，但是读取 input 和消变量分离进行.

机制：使用栈来存储推导过程中未被消解的**变量、终结符混合串**，每次转移表示一次推导**或读取**.

> 混合串是推导的句型（中间状态）去掉已读取的终结符串前缀，得到的结果.

通常维护三个状态：



① 首先，构建一条转移：

$$\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$$

> 即 push 初始变量 S 到栈顶，然后进入 q_1.
>
> 这里还未使用 $S \to x$，只是先把 S push 入.

即 $q_0 \xrightarrow{\lambda,z,Sz} q_1$（课件不规范写法：$q_0 \xrightarrow{\lambda,\lambda,S} q_1$）

② 对每个产生式 $A \to x$，构建一条转移：

$$(q_1, x) \in \delta(q_1, \lambda, A)$$

> 这里包括 $S \to x$，即左侧为初始变量的产生式.

即 $q_1 \xrightarrow{\lambda,A,x} q_1$，在图中体现为 loop（可以有多个）.

③ 对每个终结符 a，构建一条转移：

$$(q_1, \lambda) \in \delta(q_1, a, a)$$

> 这里直接 pop 栈顶终结符，然后读取它，不 push 新的内容. 因为每次只能 pop 栈顶，那么读取的 symbol 一定是 input 的**未读取部分**中的最左 symbol. 因为在推导式中，这个 symbol 左侧不会有变量，且左侧如果有 symbol 也是被读过的（作为栈顶，它左侧的 symbol 先前已被 pop 并读取了）.

即 $q_1 \xrightarrow{a,a,\lambda} q_1$，在图中体现为 loop（可以有多个）.

④ 当所有变量和终结符都通过上述转移被消解，栈中剩余 z.

构建转移：

$$\delta(q_1, \lambda, z) = \{(q_2, z)\}$$

> 栈中只剩 z 表明推导、读取结束，通过空转移到最终状态.

即 $q_1 \xrightarrow{\lambda,z,z} q_2$. 其中 q_2 是 final state.

7.2.2 NPDA $\rightarrow$ CFG

Convert any NPDA M to a context-free grammar G with $L(G) = L(M)$

步骤一：NPDA 等价变换

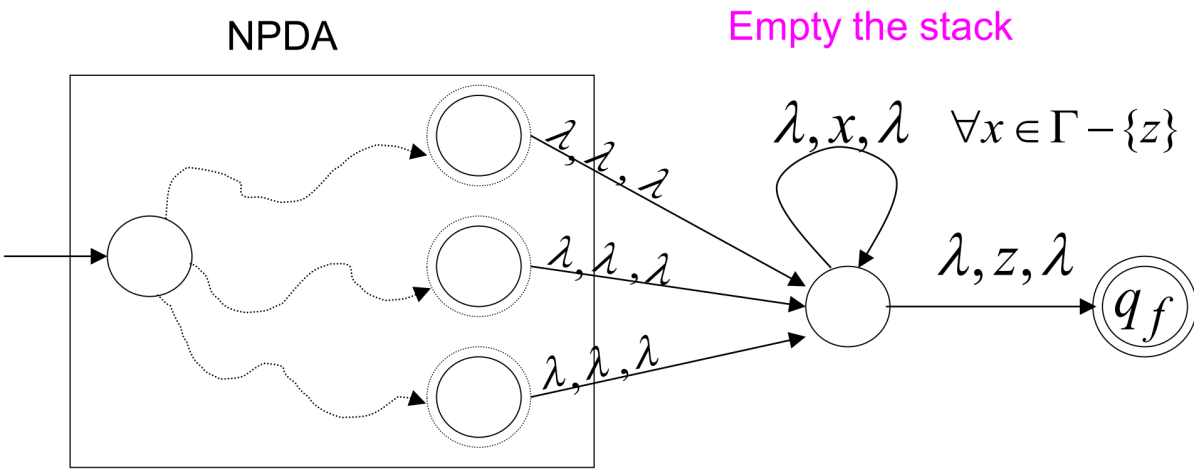
为了使转换步骤更容易形式化，我们先对已知的 NPDA M 进行等价变换.

目标：Modify (if necessary) the NPDA so that

- It has a **single final state** and **empties the stack when it accepts a string**
- All transitions are in a special form: 弹 1 推 0 或弹 1 推 2.

① 单 final，空 stack

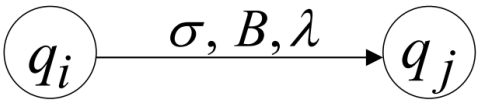
修改一：Modify the NPDA so that it empties the stack and has a unique final state



> 注意，空 stack 要求把 z 也 pop 掉.

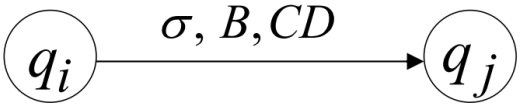
② 弹 1 推 0，弹 1 推 2

修改二：modify the NPDA so that transitions have the following forms



OR

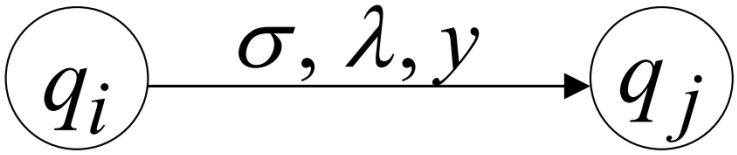
B, C, D : stack symbols

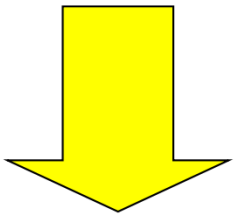


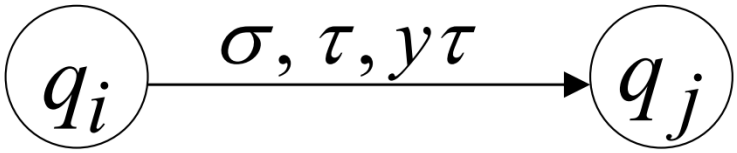
Each move either increases or decreases the stack content by a single symbol

分类讨论，可以把原转移分别进行如下的等价变形：

首先，不规范写法写成规范：







$$\forall \tau \in \Gamma - \{z\}$$

> 这里 y 的长度可以任意.

规范的转移分为弹 1 推 1、弹 1 推多、只弹不推. 我们要把它们转为弹 1 推 0 或弹 1 推 2.

① 弹 1 推 1

pop 1 push 1：形如 $q_i \xrightarrow{a,A,B} q_j$

引入 1 个中间状态：$q_i \xrightarrow{a,A,\lambda} q_{\text{中间}} \xrightarrow{\lambda,\tau,B\tau} q_j$

其中 $\tau \in \Gamma$ 是任意能入栈的字符. 实际上只会弹出栈顶字符，然后马上补回去再 push 个 B.

② 弹 1 推多

pop 1 push n（$n \geq 2$）：形如 $q_i \xrightarrow{a,A,V_1'V_2'\ldots V_n'} q_j$

引入 n 个中间状态：

$$q_i \xrightarrow{a,A,\lambda} q_1 \xrightarrow{\lambda,\tau,V_n'\tau} q_2 \xrightarrow{\lambda,\tau,V_{n-1}'\tau} q_3 \to \cdots \to q_n \xrightarrow{\lambda,\tau,V_1'\tau} q_j$$

也可以理解为 recursively 引入中间状态：

$$q_i \xrightarrow{a,A,V_2'\ldots V_n'} q_{\text{中间}} \xrightarrow{\lambda,\tau,V_1'\tau} q_j$$

其中 $\tau \in \Gamma$ 是任意能入栈的字符.

③ 只弹不推：就是弹 1 推 0，保留即可.

步骤二：构建 CFG

> The Grammar Construction

每个变量以三元组形式书写：(q_iBq_j)

其中，q_i 和 q_j 是 NPDA 的状态，$B \in \Gamma$ 可以为任意栈符号（包括 z）.

注意，这里和 NFA 转 DFA 时的状态类似，变量的书写只是标签，不是变量本身. 在构建完 CFG 后可以把变量标签任意修改.

终结符同 NPDA.

三元组 (q_iBq_j) 表示 NPDA 从 q_i 经过**若干** pop/push 后到达 q_j，并在不访问 B 以下任何符号的前提下弹出了 B. 如果这样的事情不可能发生，那么对应三元组就不参与 CFG 的产生式构建.

> 三元组 (q_iBq_j) 表示从 q_i 到 q_j 弹出 B 要满足 “不访问 B 以下任何符号” 的大前提. 因为 CFG 转换成功的关键在于计算可以被递归分解，即 $(q_0zq_f) \overset{*}{\Rightarrow} w$ 这个任务可以被分解为多个子任务分步完成，每个子任务 (q_iBq_j) 的栈可视范围（处理的内容）为 q_i 的栈顶 B 及可能有的新插入的内容，不考虑任何 B 以下的栈内容.

① 弹 1 推 0：产生终结符

For each transition $q_i \xrightarrow{a,B,\lambda} q_j$

We add production $(q_iBq_j) \to a$

注意：允许空字符 $(q_iBq_j) \to \lambda$.

② 弹 1 推 2：产生 Greibach

For each transition $q_i \xrightarrow{a,B,CD} q_j$

We add productions $(q_iBq_k) \to a(q_jCq_l)(q_lDq_k)$

For all possible states q_k, q_l in the automaton.

意思是，对于任意变量 (q_iBq_k)，该 transition 提供了一条可能的中间路径（先转移到 j，然后再转移到 k），前提是把该 transition 带来的效果一并考虑，即使用这个路径的代价是，接受一个 a，然后分别 pop C 和 D. 由于 pop C 和 pop D 要分开进行（一次只能 pop 一个），因此需要引入中间变量 q_l，先从 q_j 弹出 C 到 q_l，再从 q_l 弹出 D 到 q_k. 当然，pop C 和 D 时可能进一步插入新的中间变量.

③ 起始变量

我们模拟的 NPDA 接受一个字符串 w，即 $(q_0zq_f) \overset{*}{\Rightarrow} w$，因此起始变量就是 (q_0zq_f).

In general, $(q_iAq_j) \overset{*}{\Rightarrow} w$ if and only if the NPDA goes from q_i to q_j by reading string w and the stack doesn't change below A and then A is removed from stack

Therefore, $(q_0zq_f) \overset{*}{\Rightarrow} w$ if and only if w is accepted by the NPDA.

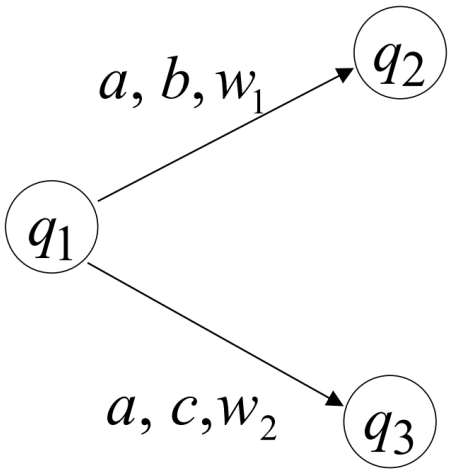
Therefore, for any NPDA, there is a context-free grammar that accepts the same language.

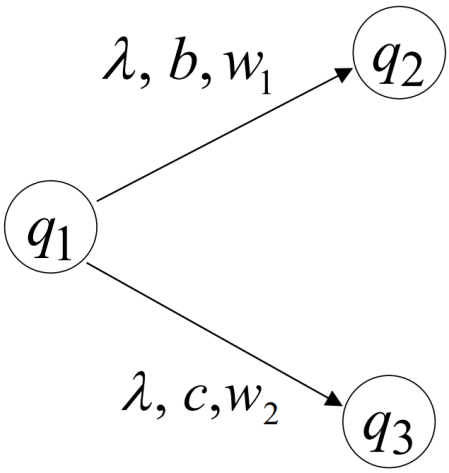
7.3 DPDA 与确定性上下文无关语言

> Deterministic Pushdown Automata and Deterministic CFLs

DPDA：基于 NPDA，不允许在同一状态和栈顶符号下，同时允许多个转移.

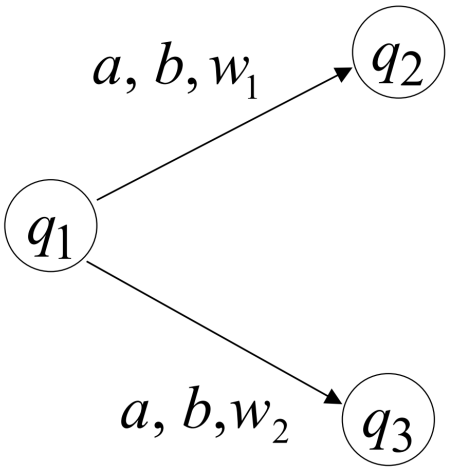
Allowed transitions:

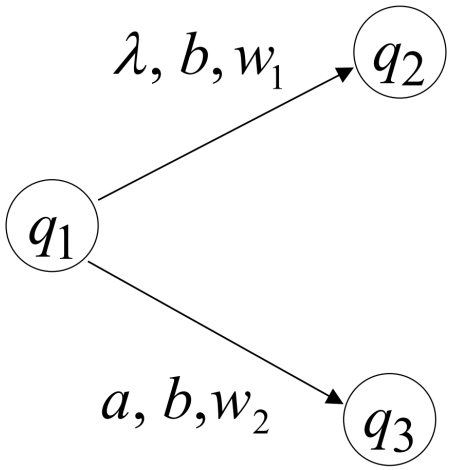




> 因为栈顶元素也可以用来决定方向，所以 DPDA 允许同一状态朝不同方向转移，只要弹出的栈顶不同（只需要根据当前栈顶元素来决定性地判断走哪个方向）.

Not allowed:





根据当前状态及当前栈顶元素无法确认转移方向：不允许.

7.3.1 DPDA 确定性下推自动机

$\delta(q, a, b)$ contains at most one element.

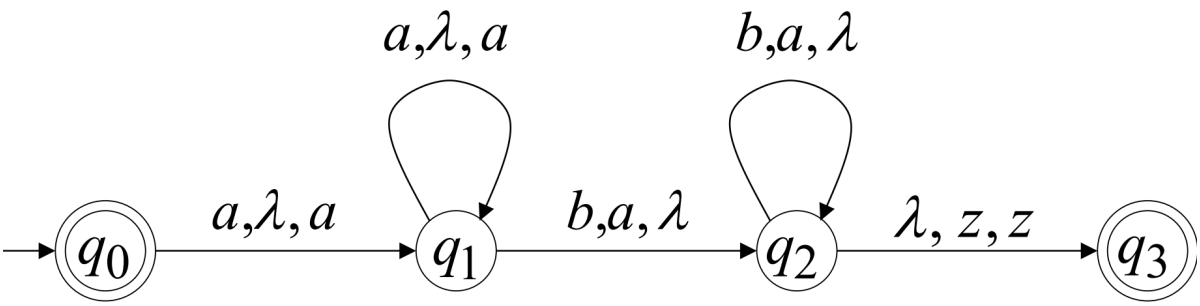
If $\delta(q, \lambda, b)$ is not empty, then $\delta(q, c, b)$ must be empty for every $c \in \Sigma$

> 即不允许在同一状态和栈顶符号下，同时允许 λ 转移和读入符号 c 的转移.
>
> 这里 empty 是 empty set，即未对该 transition 进行定义，这是允许的，不同于 DFA.

At all times at most one possible move

dpda 允许同一状态、同一栈顶接受不同字符进行不同转移，但是不能同一状态、同一栈顶接受相同字符进行不同转移，也不允许同一状态、同一栈顶同时有字符转移和空转移（因为空转移和字符转移同时有的话，走到这里就可以有不确定的多种选择）

例：$L(M) = \{a^n b^n : n \geq 0\}$



7.3.2 确定性上下文无关语言

A language L is deterministic context-free if there exists some DPDA that accepts it

7.4 NPDA 比 DPDA 更强

NPDAs have more power than DPDAs

There are context-free languages that are not deterministic

由于每个 DPDA 都是一个 NPDA，所以确定性上下文无关语言是上下文无关语言的一个子集.

$$\{\text{Deterministic Context-Free Languages}\} \subseteq \{\text{Context-Free Languages}\}$$

存在上下文无关语言是非确定性的，即它们不能被任何 DPDA 接受.

> 这意味着 NPDA 和 DPDA 并不等价，即 NPDA 并不总是能转为等价的 DPDA. 这和 NFA 总能转为 DFA 不同.

We will show that there exists a context-free language L which is not accepted by any DPDA

The language is

$$L = \{a^n b^n\} \cup \{a^n b^{2n}\} \qquad n \geq 0$$

We will show:

- L is context-free
- L is not deterministic context-free

① Language L is context-free

只需要给出一个生成它的 CFG：

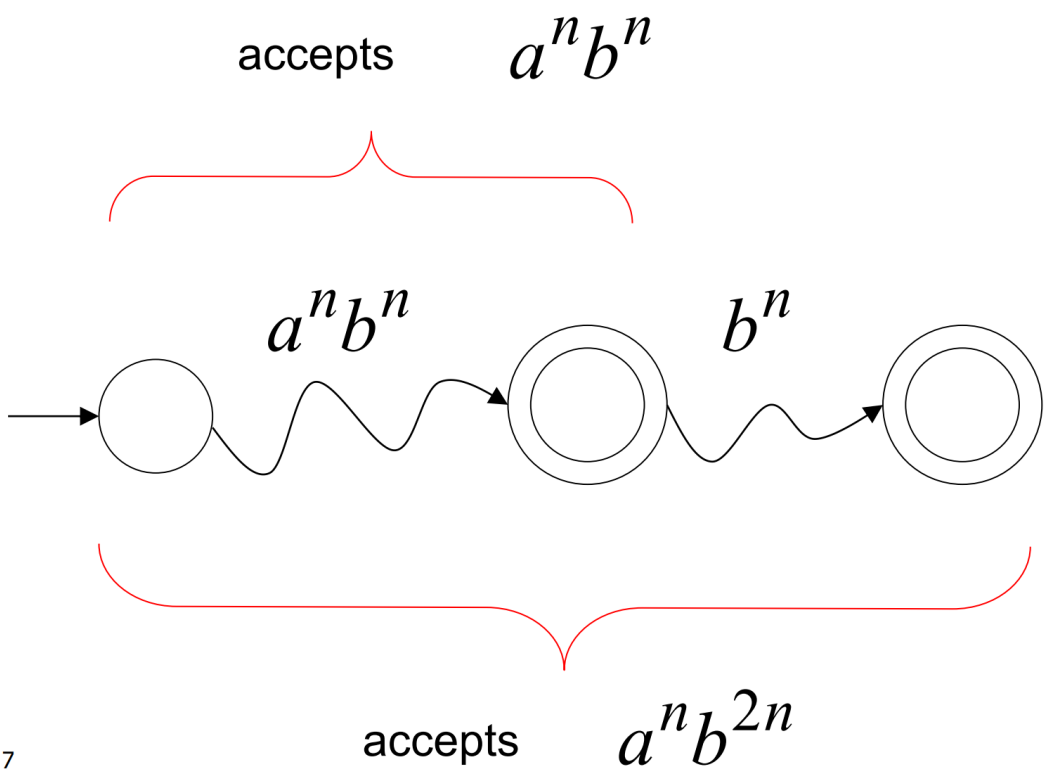
$$
\begin{aligned}
S &\to S_1 | S_2 \\
S_1 &\to aS_1b | \lambda \\
S_2 &\to aS_2bb | \lambda
\end{aligned}
$$

② L is not deterministic context-free

即证明 there is no DPDA that accepts L

反证法：Assume for contradiction that L is deterministic context-free. Therefore, there is a DPDA M that accepts L

则该 DPDA 在接受 a^nb^n 时可以到达最终状态，继续接受 b^n 又能到达另一个最终状态.



Ch 7

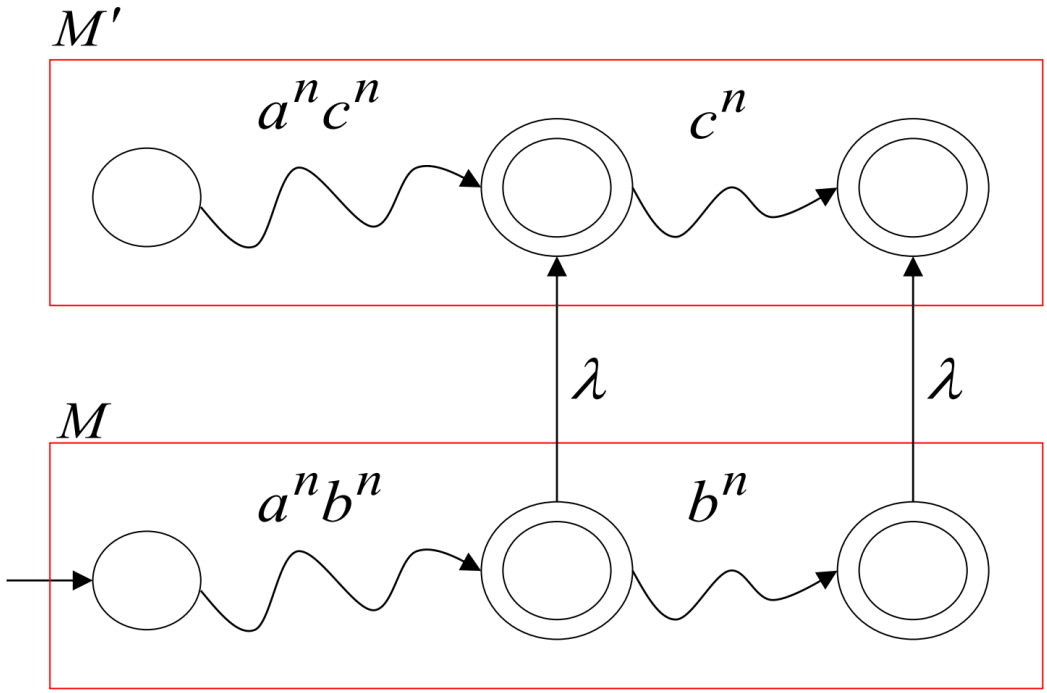
已知 $\{a^nb^nc^n : n \geq 0\}$ is not context-free.

> 见 `Lec 8`

If L were a deterministic context-free language, then

$$\hat{L} = L \cup \{a^nb^nc^n : n \geq 0\}$$

would be a context-free language. We show this by constructing an npda $\hat{M}$ for $\hat{L}$, given a dpda M for L.



7

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ with

$$Q = \{q_0, q_1, \ldots, q_n\}$$

Then consider $\hat{M} = \left(\hat{Q}, \Sigma, \Gamma, \delta \cup \hat{\delta}, z, \hat{F}\right)$ with

$$\hat{Q} = Q \cup \{\hat{q_0}, \hat{q_1}, \ldots, \hat{q_n}\}$$
$$\hat{F} = F \cup \{\hat{q_i} : q_i \in F\}$$

and $\hat{\delta}$ constructed from δ by including

$$\hat{\delta}(q_f, \lambda, s) = \{(\hat{q_f}, s)\}$$

for all $q_f \in F$, $s \in \Gamma$, and

$$\hat{\delta}(\hat{q_i}, c, s) = \{(\hat{q_j}, u)\}$$

for all

$$\delta(q_i, b, s) = \{(q_j, u)\}$$

$q_i \in Q, s \in \Gamma, u \in \Gamma^*$. For M to accept $a^n b^n$ we must have

$$(q_0, a^n b^n, z) \overset{*}{\vdash}_M (q_i, \lambda, u)$$

with $q_i \in F$. Because M is **deterministic**, it must also be true that

$$(q_0, a^n b^{2n}, z) \overset{*}{\vdash}_M (q_i, b^n, u)$$

> 意思是，接受 $a^n b^{2n}$ 必然会经过 (q_i, b^n, u)，NPDA 做不到这一点，它可以绕路.

so that for it to accept $a^n b^{2n}$ we must further have

$$(q_i, b^n, u) \overset{*}{\vdash}_M (q_j, \lambda, u_1)$$

for some $q_j \in F$. But then, by construction

$$(\hat{q}_i, c^n, u) \overset{*}{\vdash}_{\hat{M}} (\hat{q}_j, \lambda, u_1)$$

so that $\hat{M}$ will accept $a^n b^n c^n$. It remains to be shown that no strings other than those in $\hat{L}$ are accepted by $\hat{M}$. The conclusion is that $\hat{L} = L(\hat{M})$, so that $\hat{L}$ is context-free. But we will show in `Lec 8` that $\hat{L}$ is not context-free. Therefore, our assumption that L is a deterministic context-free language must be false.

Lec 8 上下文无关语言的封闭性 泵引理

8.1 泵引理

> Two Pumping Lemmas

`Lec 4` 中介绍了判定一个语言不是正则语言的泵引理. 本节介绍两个新的泵引理.

8.1.1 上下文无关语言的泵引理

> The Pumping Lemma for Context-Free Languages

用于证明某些语言不是上下文无关语言.

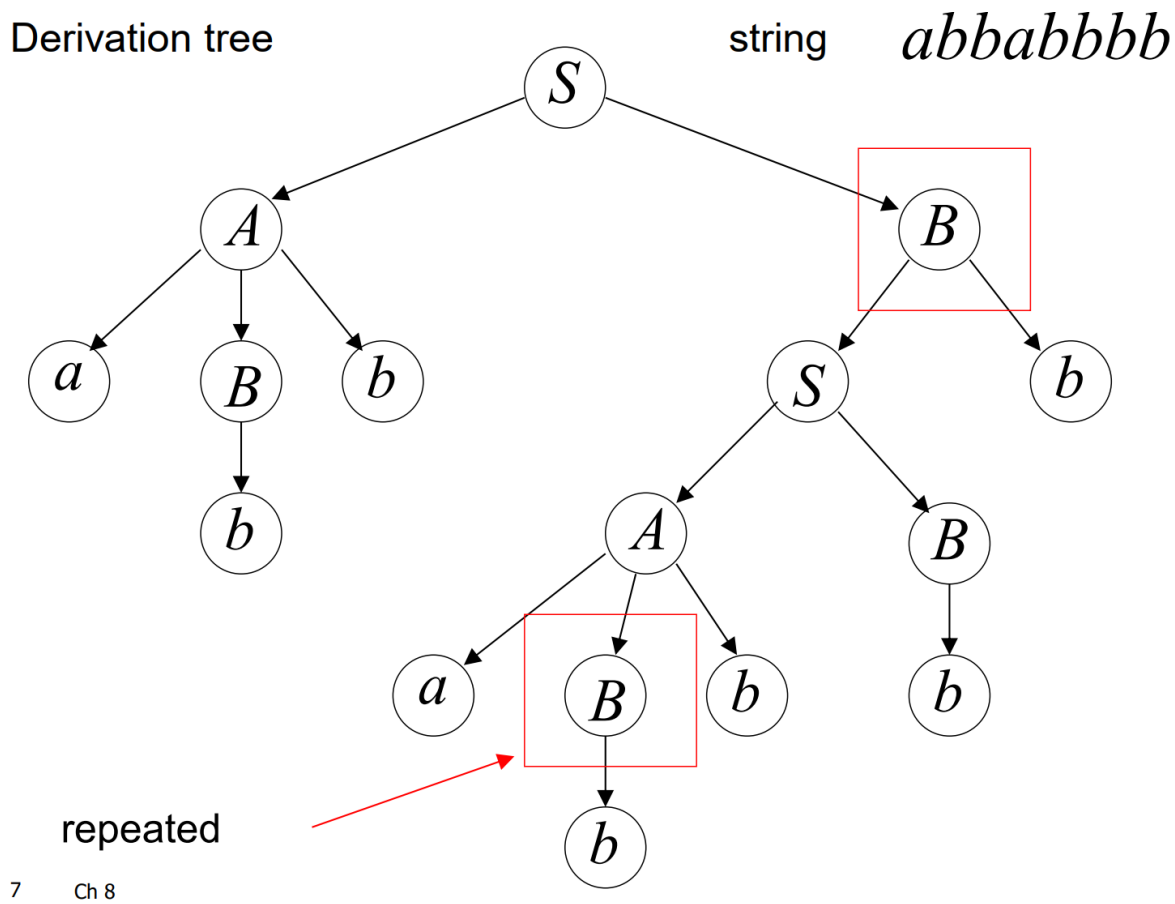
描述：对于一个**无限**（字符串数量无限，而不是长度无限）的上下文无关语言 L，存在一个整数 m，对于任何长度 $\geq m$ 的字符串 $w \in L$，都可以分解为 $w = uvxyz$，并满足：

- $|vxy| \leq m$
- $|v| + |y| \geq 1$
- $uv^i xy^i z \in L$ for all $i = 0, 1, 2, \ldots$

注意，对于有限的语言，它一定是上下文无关语言（甚至一定是正则语言），无法应用泵定理.

有效性证明：

CFG 推导树有一个基本特性：对于一个足够长的字符串，在它的推导过程中必然会出现变量的重复.



当一个变量 A 在推导树的同一路径上重复出现时，它就在推导树中形成了一个循环结构. 此时，对于一个足够长的字符串 w（$|w| \geq m$），它可以被分解为五个部分：

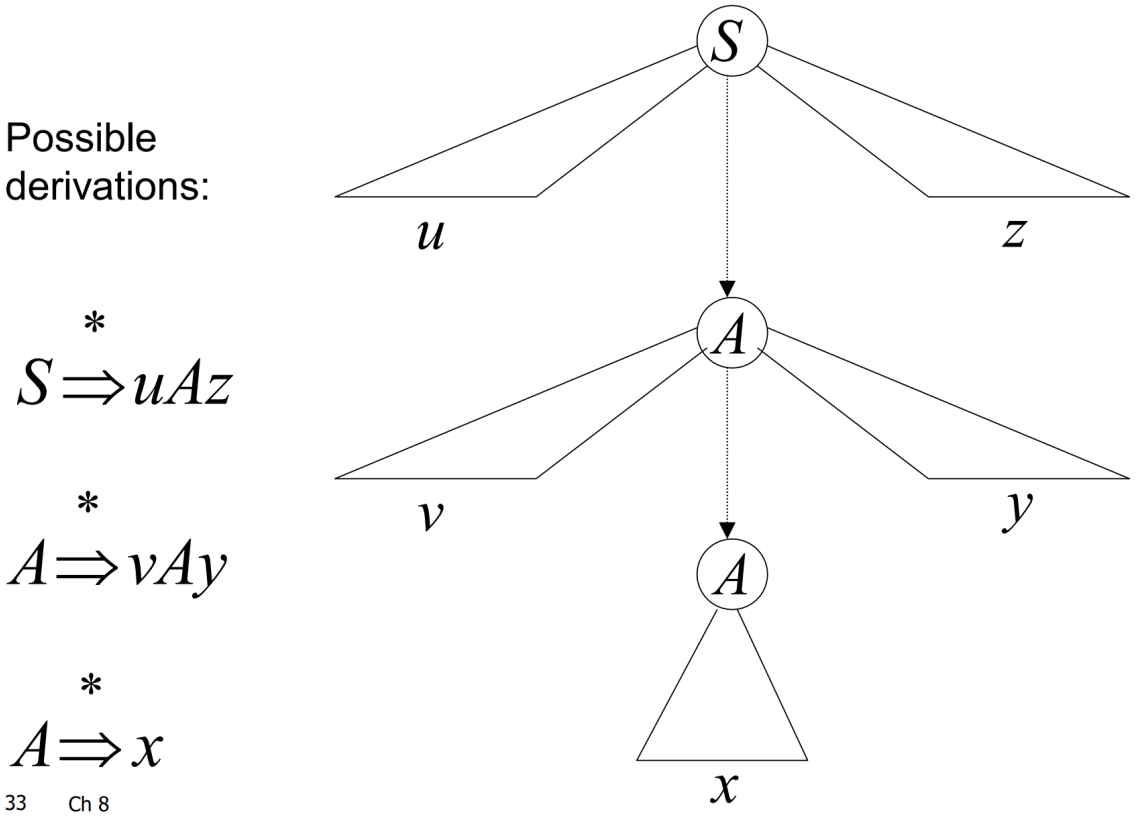
$$w = uvxyz$$

其中，

- u 是 A 开始循环时的前缀；
- z 是 A 开始循环时的后缀；
- vxy 是上层 A（循环起点的 A）推导出的终结符串.
- v 和 y 是一次循环 $A \overset{*}{\Rightarrow} vAy$ 产生的两端字符串.
- x 是下层 A（循环末尾的 A）推导出的终结符串.

注意，由于是无限语言（无限数量的不同字符串），必然存在 $|w| \geq m$ 的字符串.

> 反证法：假设 L 不存在任何满足 $|w| \geq m$ 的字符串 w，则所有 $w \in L$ 有 $|w| < m$. 则 $|L| \leq \sum_{i=0}^{m-1} |\Sigma|^i$（把所有组合都用上才能取等，取等仍然是有限的）. 因为不存在会导出有限语言，所以无限语言是必然存在 $|w| \geq m$ 的字符串 w 的.



分解需要满足长度约束：$|vxy| \leq m, |vy| \geq 1$

解释：

$|vxy| \leq m$：这是给对手的约束. 注意，泵引理的核心是用它来反证某个语言不是 CFL，所以只要能满足泵引理，对对手的限制越多越好. 例如，这里限制分解要满足 $|vxy| \leq m$，对手就只能根据玩家提供的 w，给出一个长度很受限的局部 vxy，这样对于玩家来说下一步分类讨论就简单多了.

> 记重复变量为 A，推导为 $S \overset{*}{\Rightarrow} uAz \overset{*}{\Rightarrow} uvAyz \overset{*}{\Rightarrow} uvxyz$. 若可泵片段本身 $|vxy| > m$，则可以进一步拆出更小的可泵片段：$A \overset{*}{\Rightarrow} u'Bz' \overset{*}{\Rightarrow} u'v'By'z' \overset{*}{\Rightarrow} u'v'x'y'z'$，其中 $u'v'x'y'z' = vxy$ 且 $v'x'y' \leq m$. 所以我们取满足泵引理的最严格限制，让对手提供.
>
> 正则语言的泵引理中，xy 是从起点出发，到第一个被重复的状态，再到完成第一次状态重复的路径；类似地，这里 $A \overset{*}{\Rightarrow} vxy$ 是从最后一个（离起点最远）被重复的变量出发，到完成最后一次变量重复，再到推导出结果的过程. 一个从前限制，一个从后限制，逻辑类似.

$|vy| \geq 1$：一次循环泵出的两侧字符不全为空字符. 通过把 CFG 转为等价的无单位产生式和无空产生式的文法可以实现这一点.

在 `7.2.1 法一` 结尾，有结论：

L 是上下文无关语言，当且仅当 $L - \{\lambda\}$ 是上下文无关.

因此所有操作在 $L - \{\lambda\}$ 上运行即可.

> 例如，若 L 是上下文无关语言，则 $L - \{\lambda\}$ 也是上下文无关，进而满足泵引理；若玩家想用泵引理挑战 L，则用泵引理挑战 $L - \{\lambda\}$ 即可，因为挑战 $L - \{\lambda\}$ 成功即挑战 L 成功.

基于此，可以得到 m 的下界：

Let G be the grammar of L, G' be the grammar of $L - \{\lambda\}$.

> 这里 G 移除了单位产生式以及除 $S \to \lambda$ 外的空产生式，G' 在此基础上把可能的 $S \to \lambda$ 也移除.

待补充（要转 CNF，结论是 $\approx b^{|V'|}$）

当 $|w| \geq m$，树高超过上限，必有一条路径出现重复变量. 此时 G' 没有空产生式，因此这个重复可以泵出终结符（即 $|vy| \geq 1$）. 由于 G 只比 G' 多了一条可能的 $S \to \lambda$，实际上对于 $|w| \geq m$ 不会用到，所以 G 的推导树一致，也可泵.

例：证明 $L = \{a^n b^n c^n : n \geq 0\}$ 不是上下文无关语言.

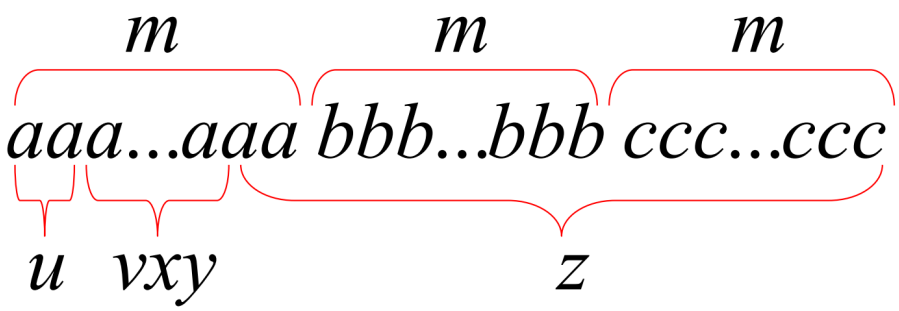
使用泵引理：

对手：声称 L 是 CFL，给出一个 m

玩家：Pick $w = a^m b^m c^m$ with $|w| \geq m$，请对手分解

对手：给出一种分解 $w = uvxyz$ with $|vxy| \leq m$ and $|vy| \geq 1$

玩家：分类讨论 vxy 的位置，逐个击破

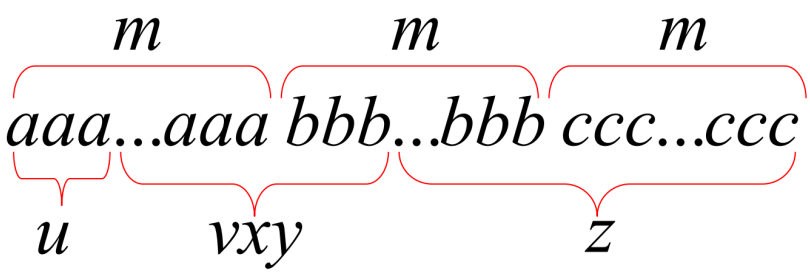


Pick $i = 2$，$uv^2xy^2z = a^{m+|vy|}b^m c^m \notin L$

Contradiction!

Case 2 (within b^m) and Case 3 (within c^m)：同 Case 1

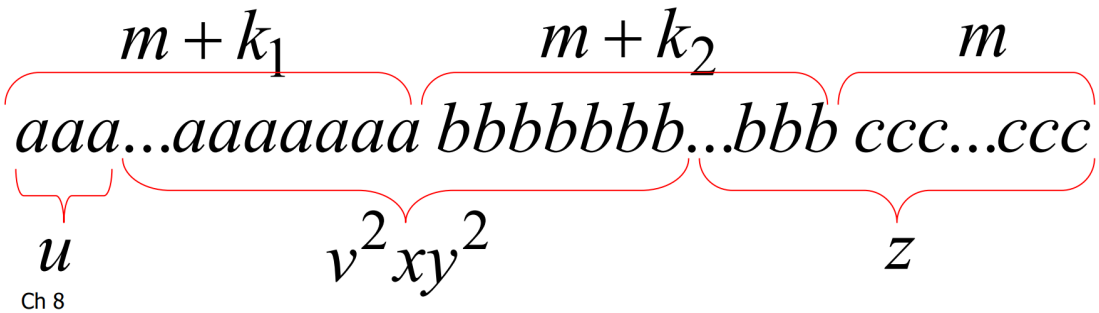
Case 4: vxy overlaps a^m and b^m



分三种 Possibility 讨论：

Case 4: Possibility 1: v contains only a

$k_1 + k_2 \geq 1$ y contains only b



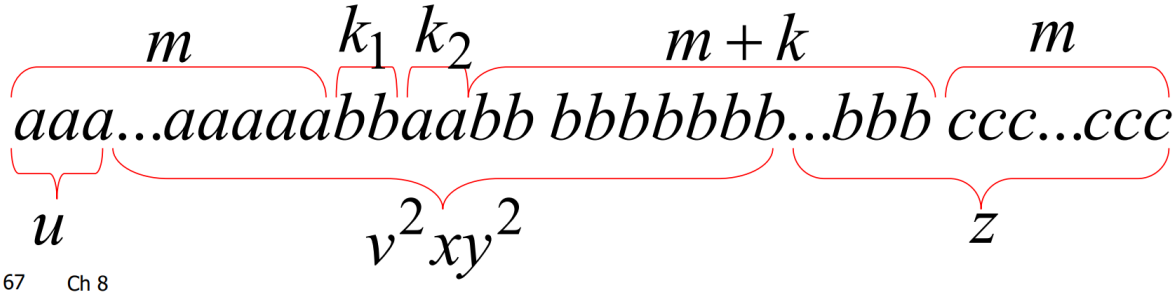
63 Ch 8

Pick $i = 2$, $uv^2xy^2z = a^{m+|v|}b^{m+|y|}c^m \notin L$

Contradiction!

Case 4: Possibility 2: v contains a and b

$k_1 + k_2 + k \geq 1$ y contains only b



67 Ch 8

Pick $i = 2$, $uv^2xy^2z = a^m b^{k_1} a^{k_2} b^{m+k} c^m \notin L$

> 原因：k_1, k_2, k 不能全为 0.

Contradiction!

Possibility 3 同 Possibility 2.

Case 5 (vxy overlaps b^m 和 c^m)：同 Case 4.

Since $|vxy| \leq m$（泵引理的严格要求），string vxy cannot overlaps a^m, b^m and c^m at the same time, there are no other cases to consider.

In all cases we obtained a contradiction. Therefore, the original assumption that $L = \{a^n b^n c^n : n \geq 0\}$ is context-free must be wrong.

> 因为对手找不出符合条件的分解.

例 2：证明 $L = \{vv : v \in \{a, b\}^*\}$ is not context free.

待补充.

例 3：证明 $L = \{a^{n!} : n \geq 0\}$ is not context free.

待补充.

例 4：证明 $L = \{a^{n^2} b^n : n \geq 0\}$ is not context free.

待补充.

8.1.2 线性语言的泵引理

> Pumping Lemma II for Linear Languages

Recall: `Lec 3 3.1` 介绍了线性文法.

> 产生式右侧至多只有一个变量的**上下文无关**文法
>
> 当时还没教上下文无关，现在更新一下概念：通常来说，线性文法指的都是线性上下文无关文法，即左侧必须是单个变量，右侧至多一个变量. 因此，线性文法是一种特殊的 CFG.

如果一个上下文无关语言 L 可以由一个线性上下文无关文法 G 生成，则 L 是线性语言.

注意：

- 线性文法生成的语言是线性语言，但是线性语言也可能由非线性文法生成，即一个线性语言可能同时由一个线性文法和一个非线性文法生成. 但是只要存在生成它的线性文法，就认为它是线性语言.

> 非线性文法生成线性语言的例子

- 线性语言一定是上下文无关的. 线性文法是一种特殊的 CFG.

线性语言的泵引理：

For infinite linear language L

there exists an integer m such that

for any string $w \in L, |w| \geq m$

we can write $w = uvxyz$

with lengths $|uvyz| \leq m$ and $|vy| \geq 1$

and it must be $uv^ixy^iz \in L$, for all $i = 0, 1, 2, \ldots$

待补充.

8.2 上下文无关语言的封闭性

> Closure Properties

`Lec 4` 介绍了正则语言的封闭性，本节介绍上下文无关语言的封闭性.

8.2.1 Union

Context-free languages are closed under Union

证明：可以构造新的 CFG.

For context-free languages L_1, L_2 with context-free grammars G_1, G_2 and start variables S_1, S_2

The grammar of the union $L_1 \cup L_2$ has new start variable S and additional production
$S \to S_1|S_2$

8.2.2 Concatenation

Context-free languages are closed under: Concatenation

证明：可以构造新的 CFG.

For context-free languages L_1, L_2 with context-free grammars G_1, G_2 and start variables S_1, S_2

The grammar of the concatenation L_1L_2 has new start variable S and additional production
$S \to S_1S_2$

8.2.3 Star Operation

Context-free languages are closed under: Star-operation

L is context free $\Rightarrow L^*$ is context-free.

证明：可以构造新的 CFG.

For context-free languages L with context-free grammars G and start variables S

The grammar of the star operation L^* has new start variable S_1 and additional production $S_1 \to SS_1|\lambda$

8.2.4 非封闭属性

不封闭：intersection、Complement.

证明封闭性要给出通用构造，但证明没有封闭性只要举反例：

$L_1 = \{a^n b^n c^m\}$ is context-free

$L_2 = \{a^n b^m c^m\}$ is context-free

but their intersection $L_1 \cap L_2 = \{a^n b^n c^n\}$ is not context-free.

> 泵引理.

假设 Context-free languages are closed under Complement，则 $\overline{\overline{L_1} \cup \overline{L_2}}$ 也是 CFL. 然而，由德摩根律，$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$ 不是 CFL. 假设不成立.

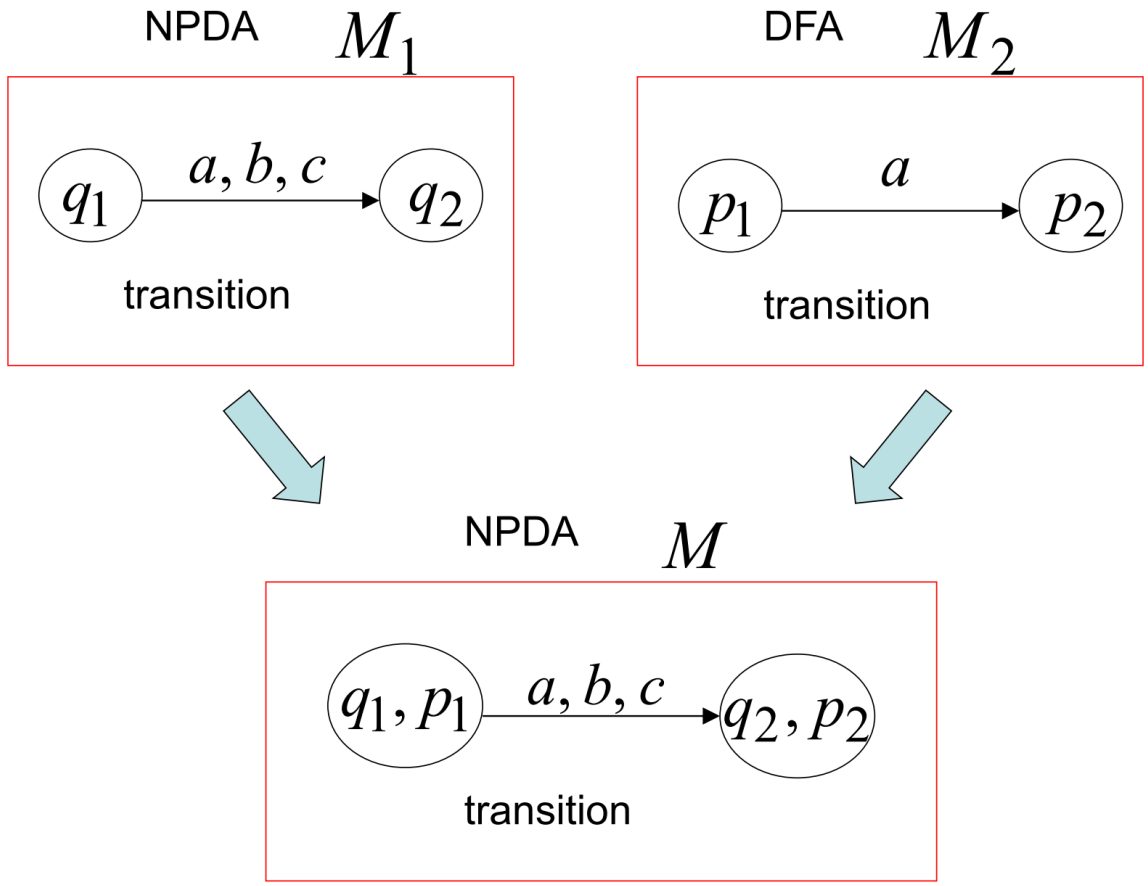
8.2.5 特殊：CFL 和 RL 的交集

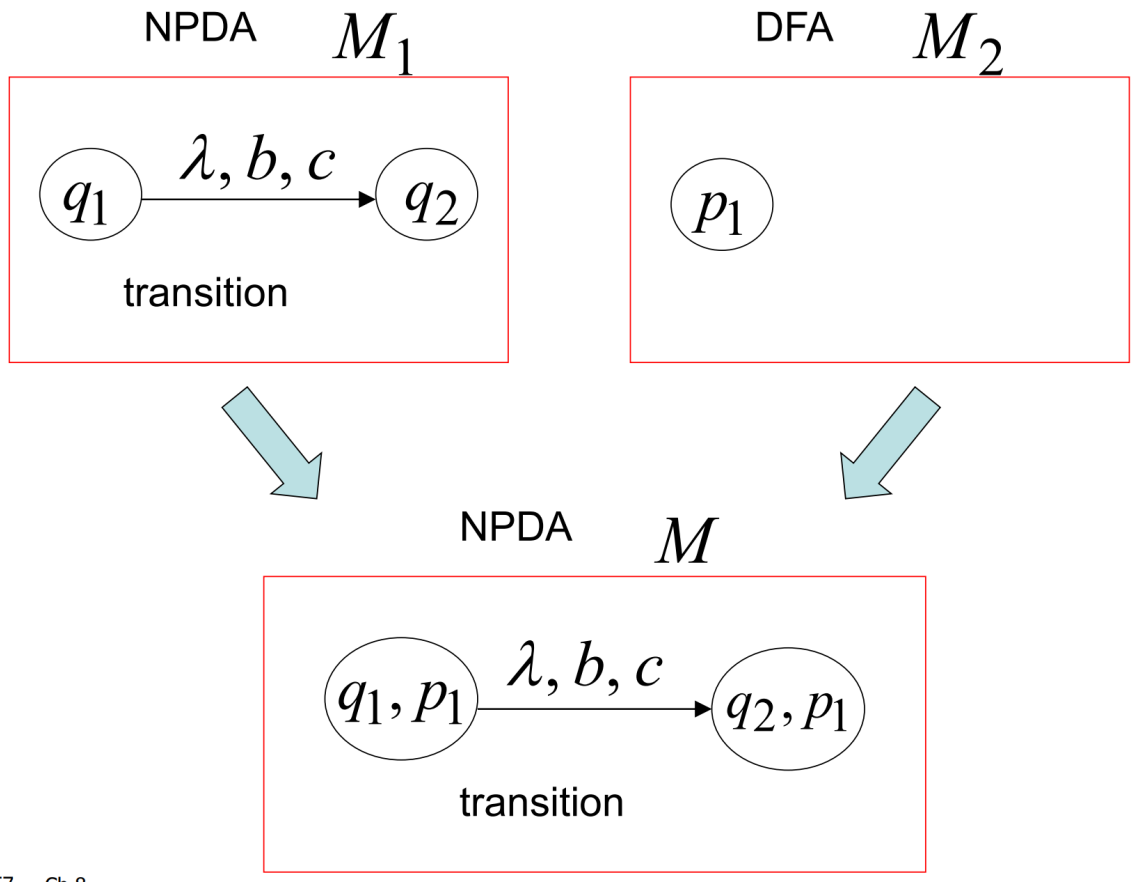
The intersection of a context-free language and a regular language is a context-free language

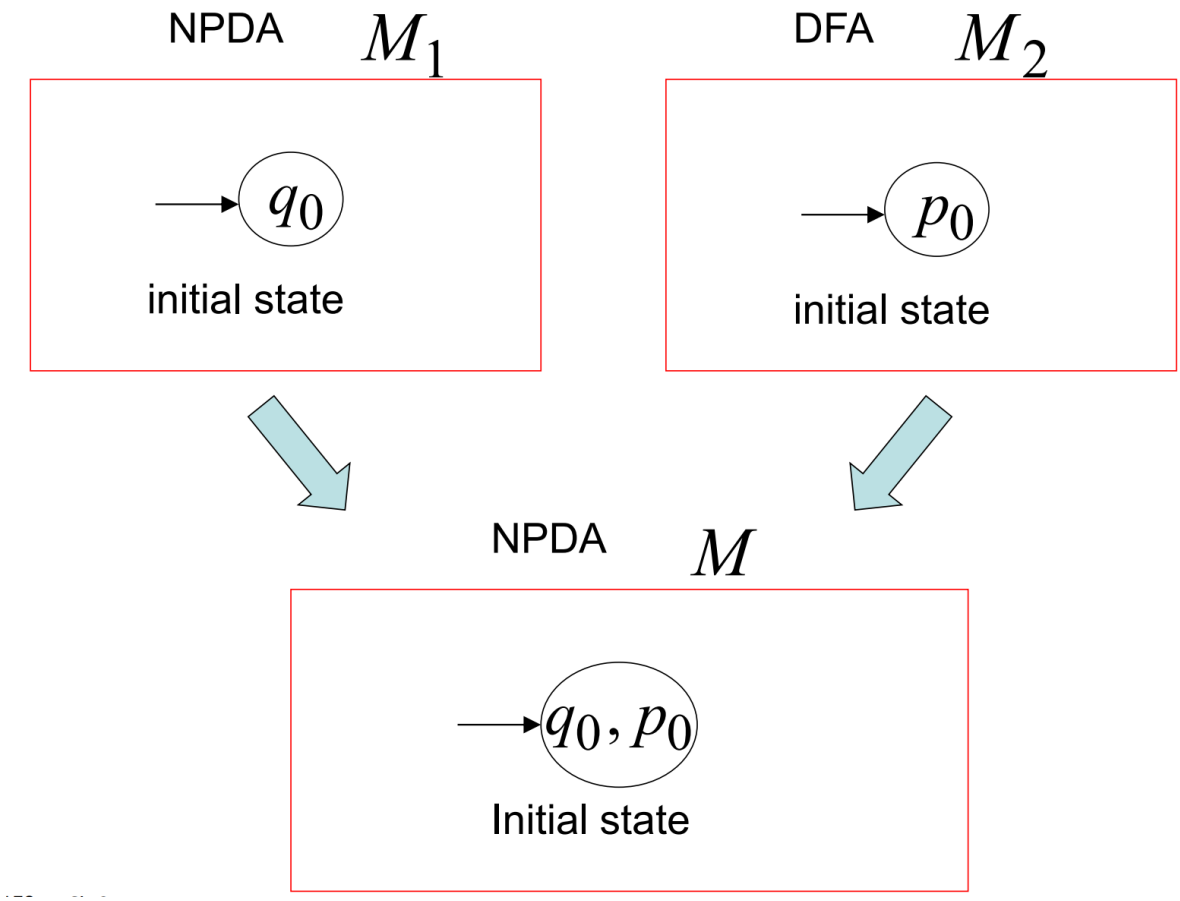
证明：设 L_1 是上下文无关语言，M_1 是接受它的 NPDA. L_2 是正则语言，M_2 是接受它的 DFA.

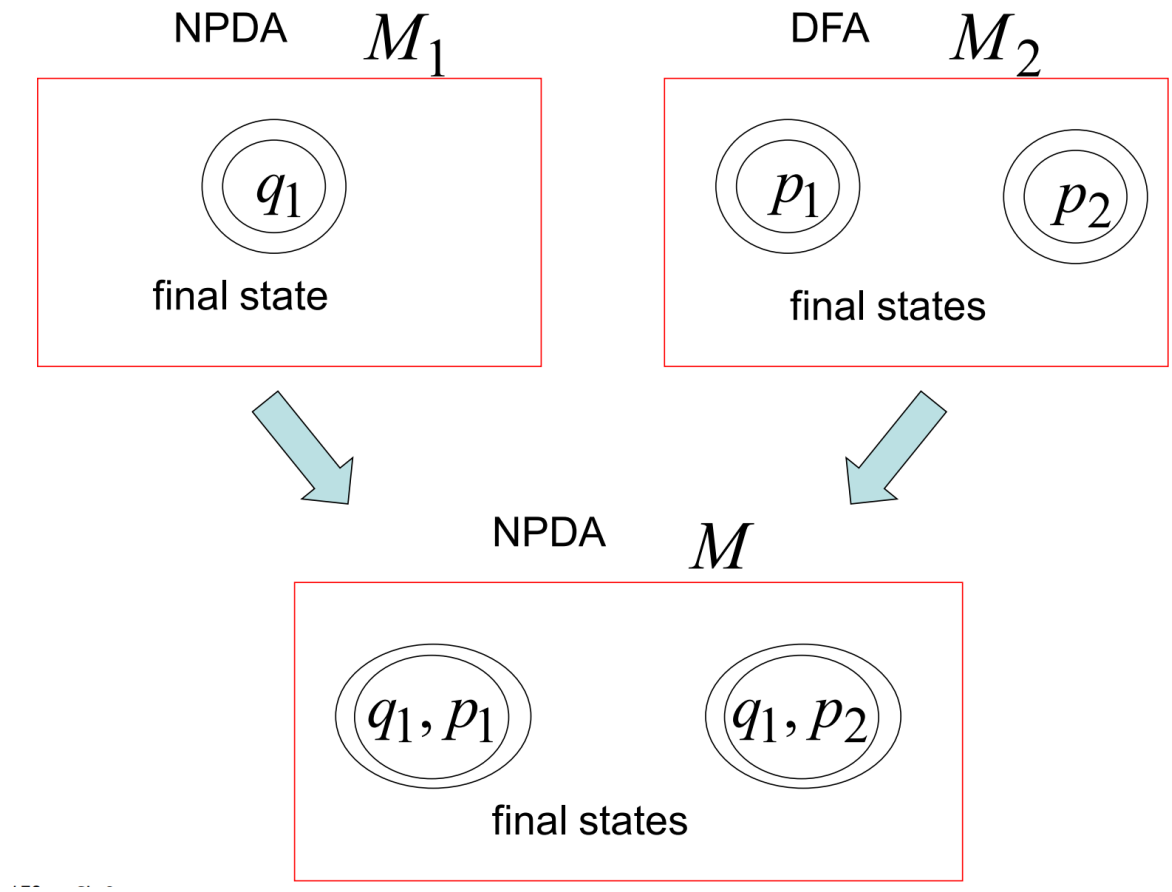
构建一个新的 NPDA M，能够接受 $L_1 \cap L_2$

构建方法：使 M 的运行能够同时模拟 M_1 和 M_2 读取相同字符时的表现. 如果 M_1 和 M_2 阅读到不同字符，它们的转移组合不会体现在新的 M 上.


NPDA
M_1
q_1
a, b, c
q_2
transition
DFA
M_2
p_1
a
p_2
transition
NPDA
M
q_1, p_1
a, b, c
q_2, p_2
transition


NPDA
M_1
q_1
λ, b, c
q_2
transition
DFA
M_2
p_1
NPDA
M
q_1, p_1
λ, b, c
q_2, p_1
transition


NPDA
M_1
q_0
initial state
DFA
M_2
p_0
initial state
NPDA
M
q_0, p_0
Initial state


NPDA
M_1
q_1
final state
DFA
M_2
p_1
p_2
final states
NPDA
M
q_1, p_1
q_1, p_2
final states

综上，上下文无关语言和正则语言的交集是上下文无关语言.

例：证明 $L = \{a^n b^n : n \neq 100, n \geq 0\}$ 是 context-free

证明：

已知 $L_1 = \{a^{100} b^{100}\}$ is regular, $L_2 = \{a^n b^n\}$ is context-free.

正则语言对补集封闭，即 $\overline{L_1}$ is regular.

$L = L_2 \cap \overline{L_1}$ is context-free.

> 因为是上下文无关语言和正则语言的交集.

例：证明 $L = \{w : n_a = n_b = n_c\}$ is not context-free

证明：反证法

假设 L 上下文无关，则 $L \cap \{a\}^* \{b\}^* \{c\}^* = \{a^n b^n c^n\}$

其中 $\{a\}^* \{b\}^* \{c\}^*$ 是正则语言，因为正则语言对 star operation 和 concatenation 是封闭的. 因此推出 $\{a^n b^n c^n\}$ 是上下文无关，这和我们用泵引理推出的结论相悖，因此假设是错的.

> 注意，这里可以出考题，即证明一个语言不是上下文无关，除了使用泵引理，还可以结合封闭性的综合知识. 对于这种描述的语言，没有清晰结构的，尤其要考虑这一点.

8.3 上下文无关语言的判定算法

> Decidable Properties of Context-Free Languages

对于上下文无关语言，以下问题是可判定的：

8.3.1 成员资格问题

> Membership Question

判定一个给定字符串 w 是否属于语言 $L(G)$

Membership Algorithms: Parsers

- Exhaustive search parser: $O(P^{2|w|+1})$

> 见 `Lec 5 2.1.1 暴力解析`

- CYK parsing algorithm: $O(|w|^3)$

8.3.2 空语言问题

> Empty Language Question

For CFG G，判定 $L(G)$ 是否为空.

Algorithm:

Remove useless variables

Check if start variable S is useless

8.3.3 无限语言问题

> Infinite Language Question

判定 $L(G)$ 是否无限

Algorithm:

Remove λ-productions, unit productions, useless variables

Create dependency graph for variables

> 这里的 dependency graph 和 `Lec 6 1.3.2` 的不完全一样. 这里只要左边变量产生右边变量，就能看作一条转移，即使产生了终结符或其他变量.

If there is a loop in the dependency graph then the language is infinite

Example:

$$S \to AB$$

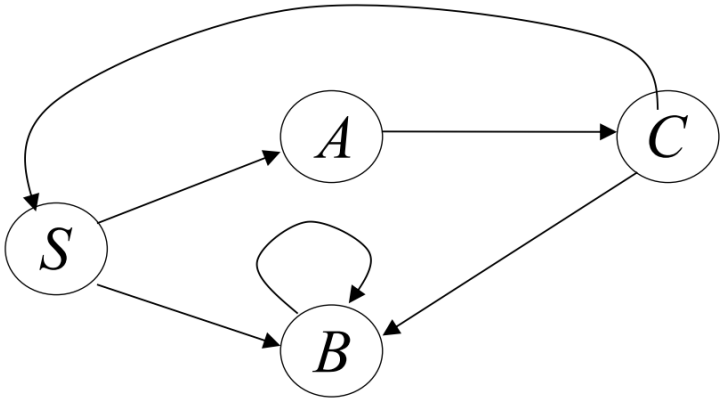
$$A \to aCb \mid a$$

$$B \to bB \mid bb$$

$$C \to cBS$$

Infinite language

Dependency graph



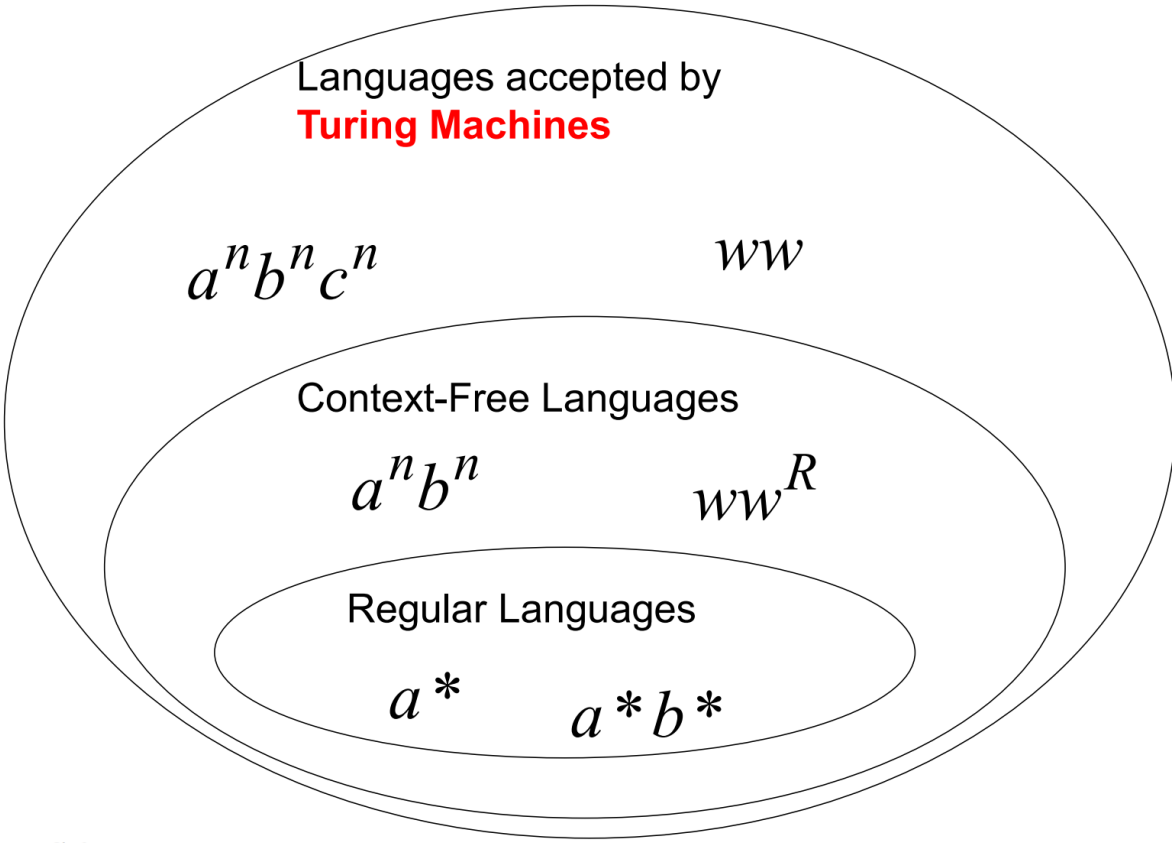
177 Ch 8

Lec 9 图灵机

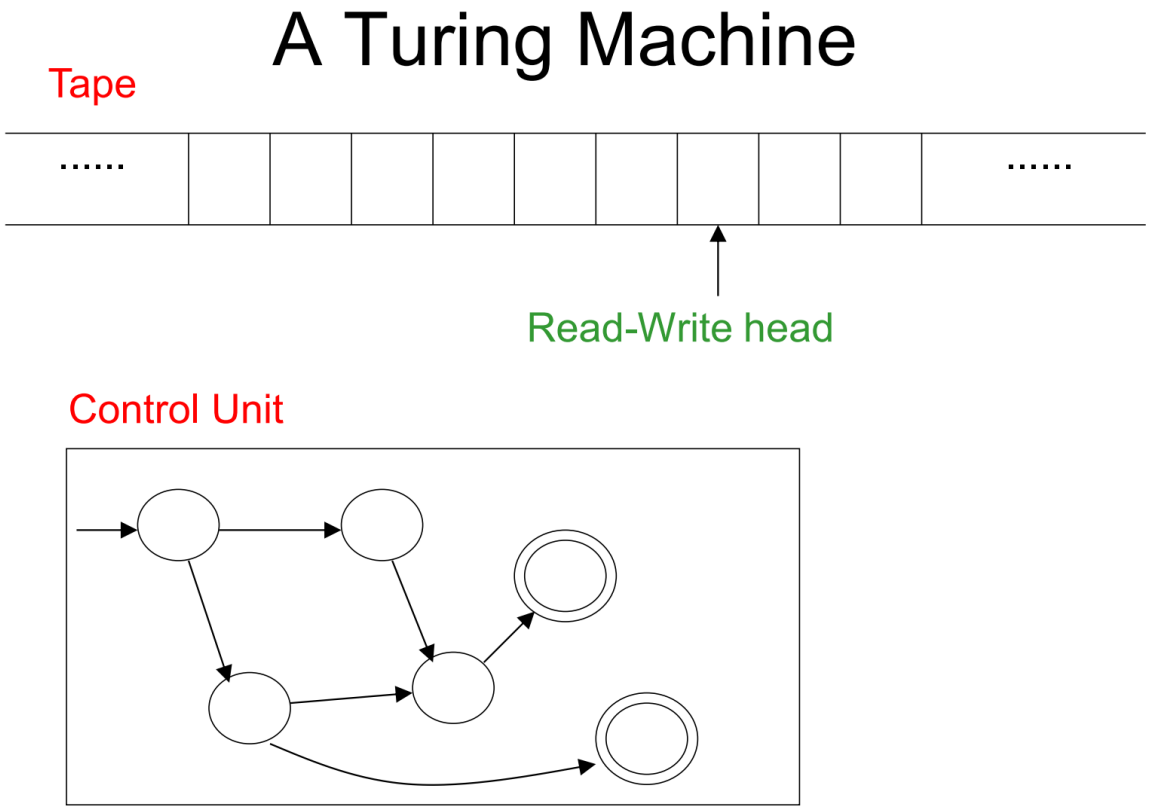
学习完上下文无关语言的泵定理，我们发现仍然存在一些语言是上下文无关文法无法识别的. 例如 $a^nb^nc^n$ 和 ww

本章将引入图灵机，它能够识别所有上下文无关语言，还能够识别一部分上下文无关语言之外的语言.

> 注意，仍然有一些语言无法被图灵机识别.


Languages accepted by
Turing Machines
$a^n b^n c^n$
ww
Context-Free Languages
$a^n b^n$
ww^R
Regular Languages
$a*$
$a*b*$

1. 图灵机


A Turing Machine
Tape
.....
.....
Read-Write head
Control Unit

组成：

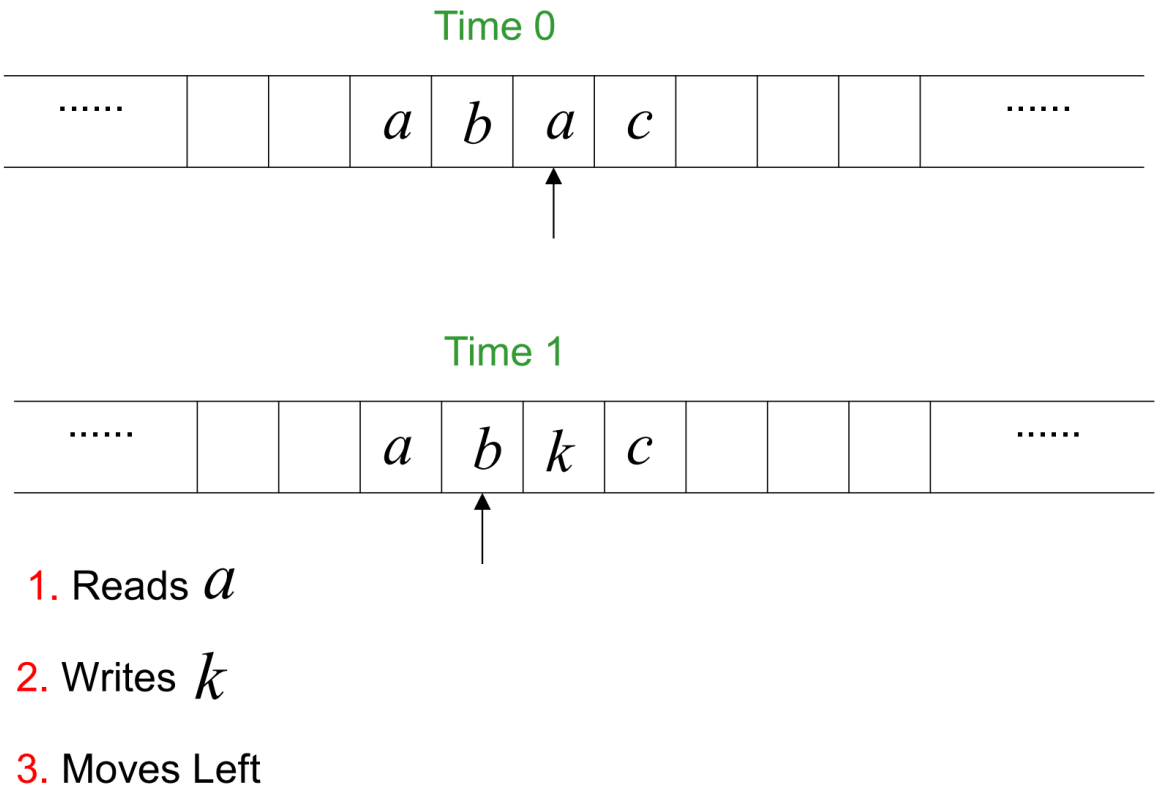
- 控制单元（Control Unit）
- 读写头（Read-Write head）
- 纸带（Tape）：无限长，没有边界.

1.1 读写头

在每个时间步，读写头执行三个动作：

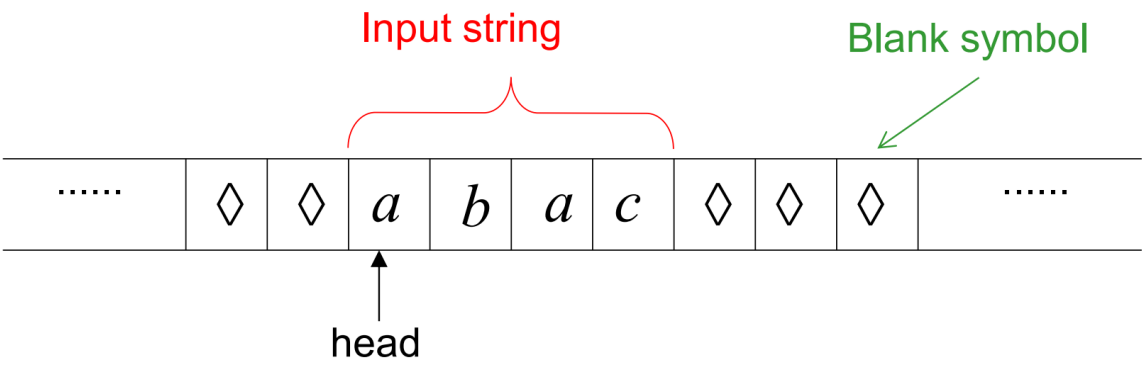
- Reads a symbol
- Writes a symbol
- Moves Left or Right

例：



1.2 纸带与输入字符串

无限纸带被分成单元格，每个单元格可以存放一个符号.



空白符号（Blank symbol）：用一个菱形符号（◊）表示. 这些空白符号填充了纸带上所有没有输入的地方，纸带在两端无限沿伸.

输入字符串（Input string）：

- 要处理的字符串，例如图中 $abac$
- 输入字符串被放置在纸带的连续单元格上.

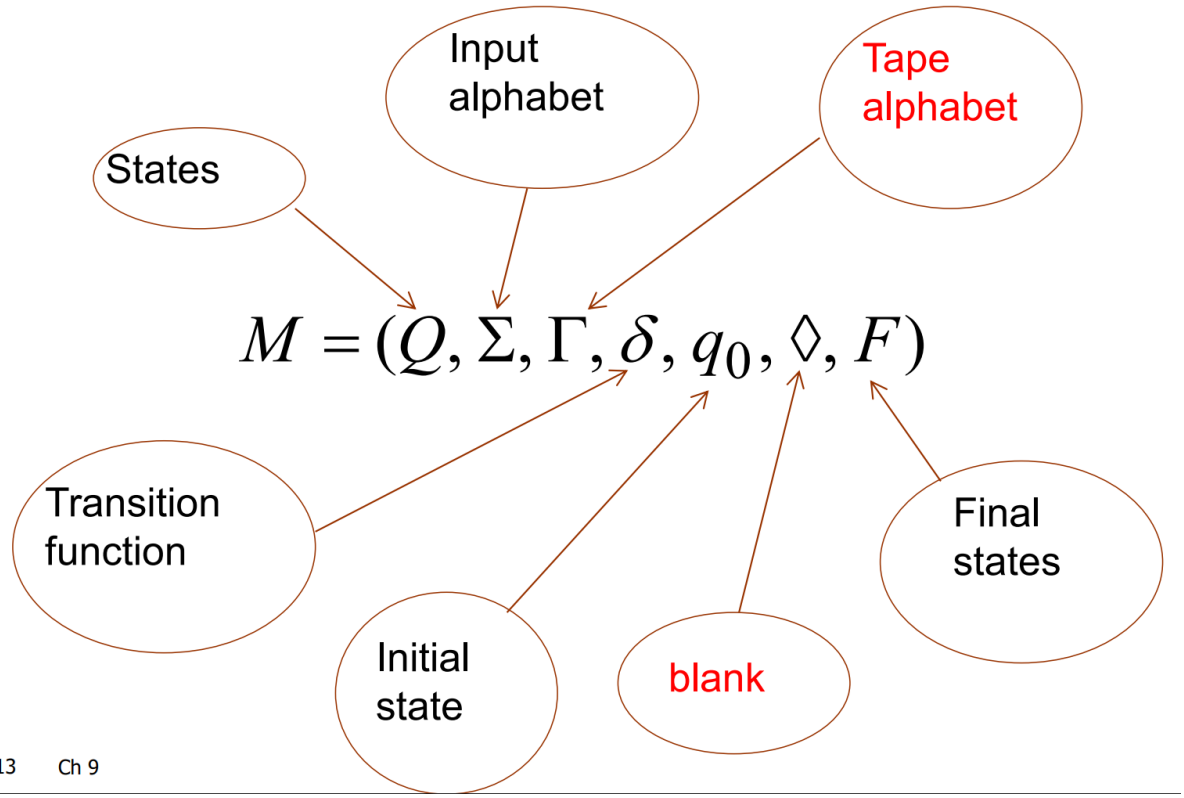
  > 初始连续，后面可以在中间写入空白符号打断.

- **输入字符串永远不会为空**.

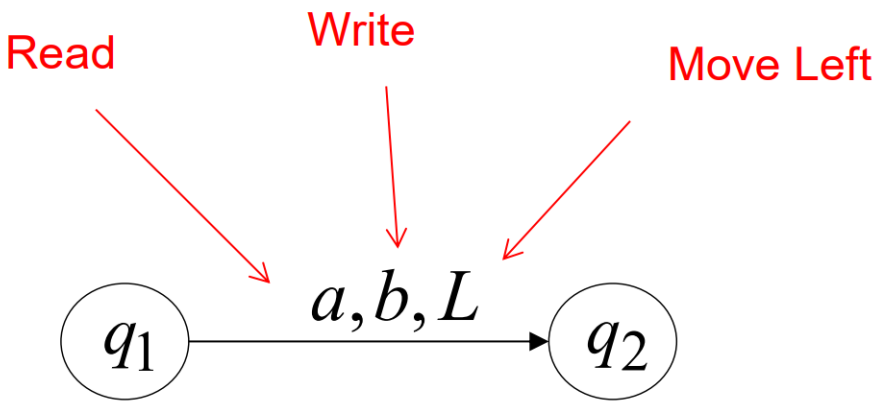
初始配置（Initial Setup）

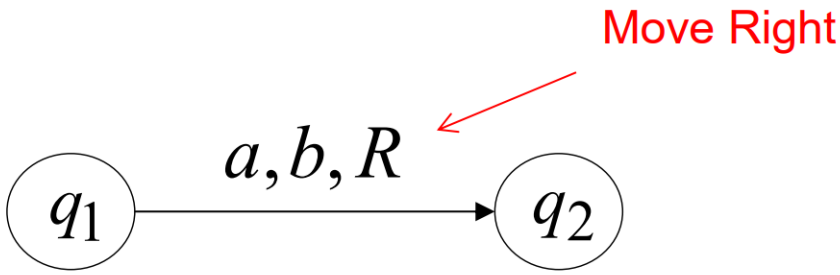
- 读写头（Head）：用一个向上的箭头表示.
- 起始位置：读写头从输入字符串的**最左侧位置**开始.

1.3 图灵机：形式化定义

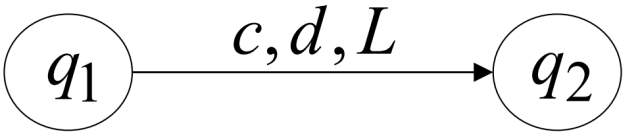

Input alphabet
Tape alphabet
States
$M = (Q, \Sigma, \Gamma, \delta, q_0, \Diamond, F)$
Transition function
Final states
Initial state
blank

1.3.1 状态与转移


Read
Write
Move Left
a, b, L
q_1
q_2


Move Right
a, b, R
q_1
q_2

1.3.2 转移函数

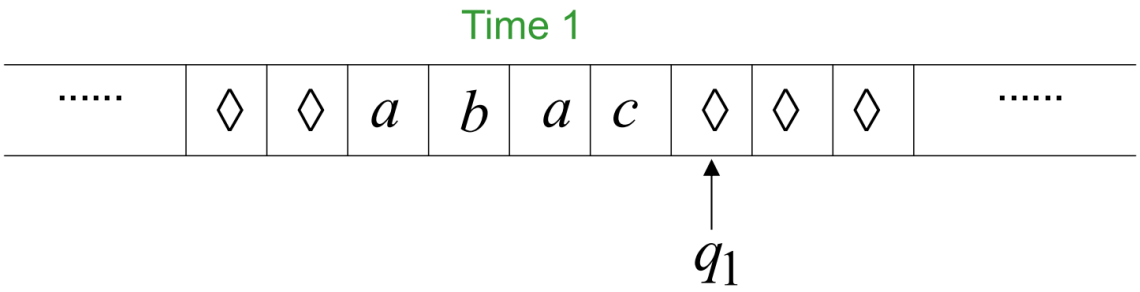


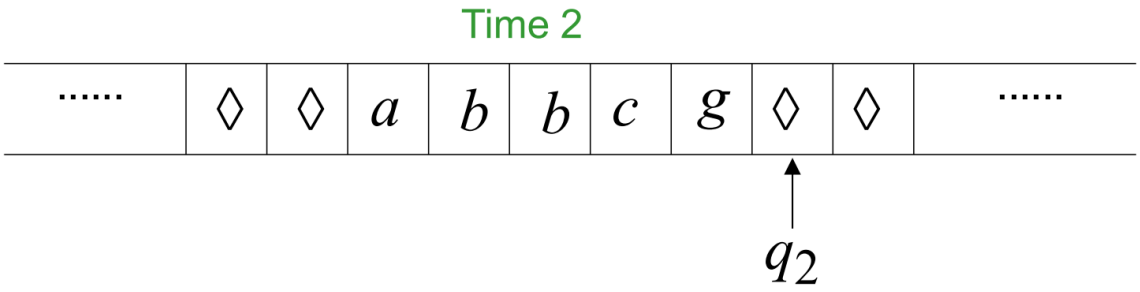
$$\delta(q_1,c)=(q_2,d,L)$$

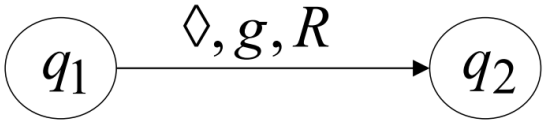
和 DFA 不同，图灵机的转移函数 δ 是一个偏函数，即对于某个状态 q_i 和纸带符号 a，$\delta(q_i,a)$ 可能没有定义.

1.3.3 确定性

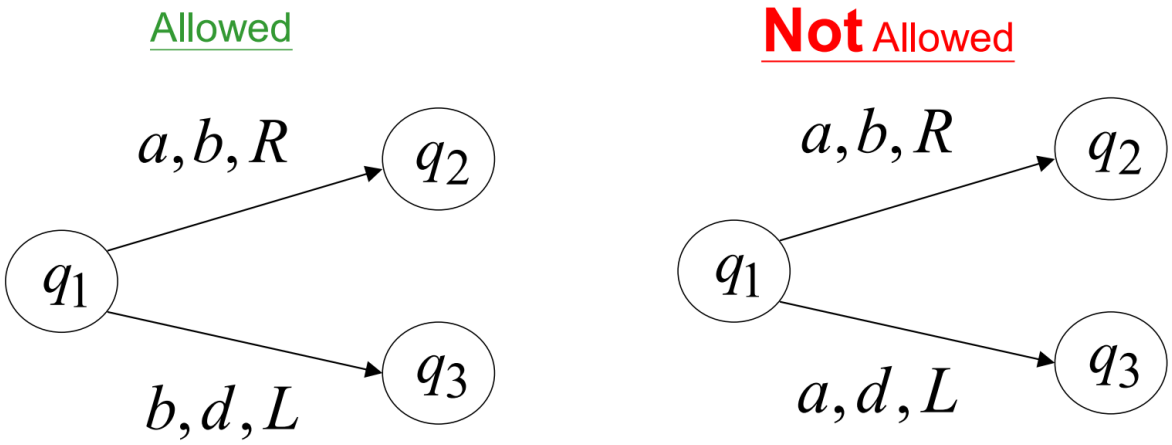
图灵机允许读取或写入空白符号：







这里空白符号被视为一个正常符号，而非空转移. 实际上，在图灵机中，空转移是不允许的. 此外，不确定性的操作也不允许：



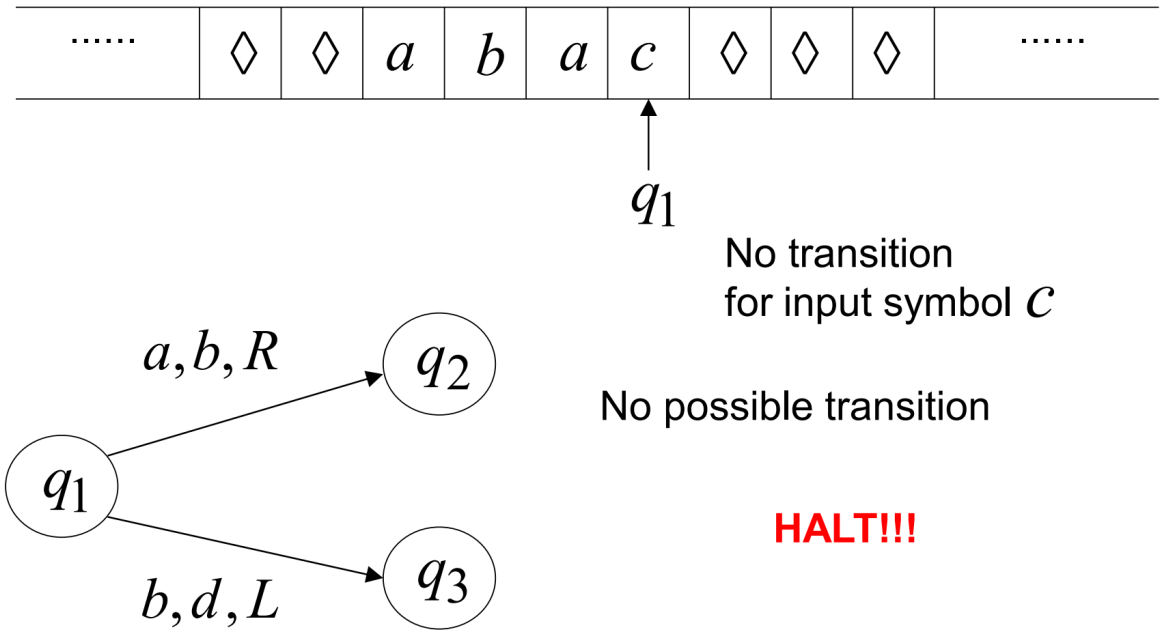
也就是说，Turing Machines are deterministic.

1.3.4 停机

> Halting

停机条件：如果机器处于某个状态 q_i 并读取到符号 a，但没有可能的转移 $\delta(q_i, a)$ 可以遵循，则机器会立即停机.

例：



如果机器在这种情况下停机，并且当前状态是非终止状态，则输入被**拒绝**.

1.3.5 终止状态

图灵机运行时，除了纸带、读写头，还维护一个控制单元（转移图）. 读写头在磁带上运行的同时，状态也在转移图中转移.

在机器定义中，已经预先制定了最终状态集. 注意，最终状态不允许有出转移. 也就是说，状态进入最终状态时，直接停机.

注意：

最终状态一定没有出转移，但没有出转移的不一定是最终状态.

没有出转移一定停机，有出转移可能停机，也可能不停机. 判断停不停机，分两步：

- 先看当前状态 q_i，若为最终状态，直接停机；
- 若 q_i 不是最终状态，看纸带内容. 假设当前读写头指 a，看机器有没有对转移 $\delta(q_i, a)$ 作定义. 若有，不停机. 若没有，停机.

> 注意，对于当前状态和特定输入，转移函数最多只能定义一个转移（确定性）. 只要看这一个有没有定义即可.

Allowed

Not Allowed

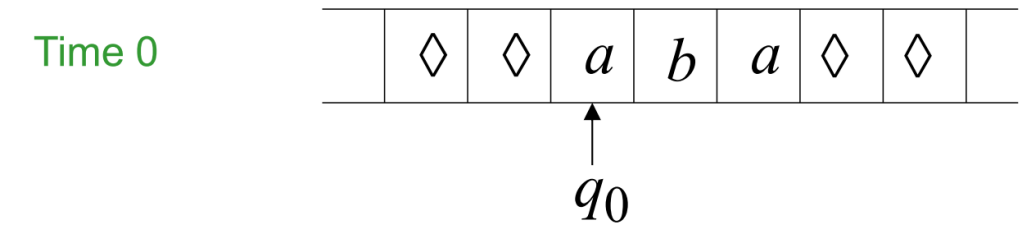
进入终止状态的停止，其实就是缺少定义转移的停止. 只是如果停止在终止状态会被接受，停在其他状态不会被接受.

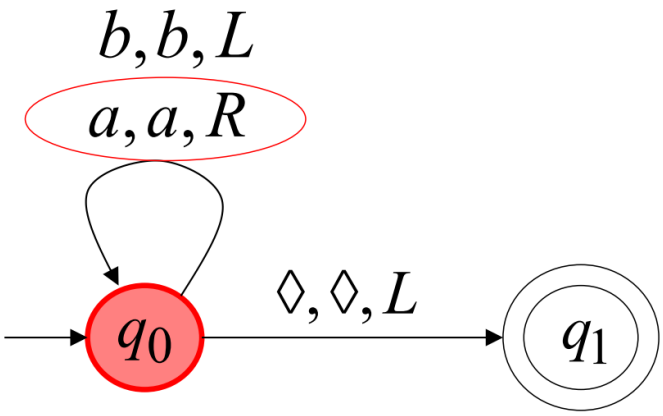
1.3.6 接受 & 拒绝

If machine halts in a final state, accept input

If machine halts in a non-final state or if machine enters an infinite loop, reject input

无限循环例子：





1.3.7 例子

待补充.

1.4 配置 & 瞬时描述

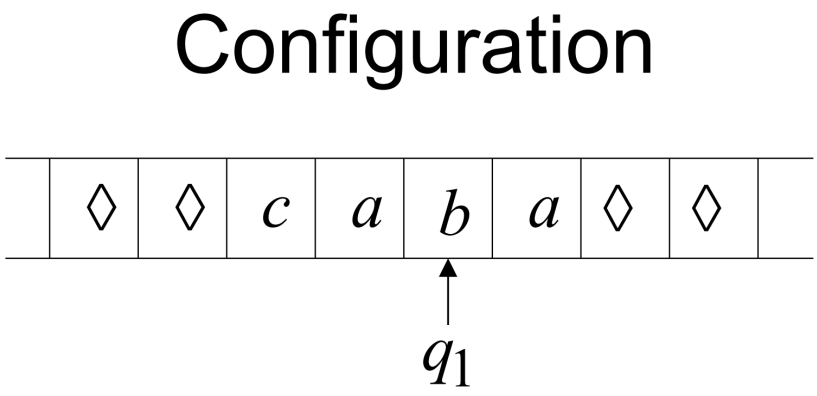
> Configuration & Instantaneous description

“配置” 指的是图灵机在运行过程中某一瞬间的**完整状态快照**. 为了完全描述机器的当前状况，配置需要包含三个关键信息：

- 纸带内容：纸带上当前所有符号（忽略两侧无限空白符号）
- 读写头位置：读写头当前指向哪个单元格.
- 当前状态：控制单元所处的当前状态.

瞬时描述符号：为了方便书写和分析图灵机每一步运行（即方便地表示配置），我们使用一种紧凑的符号表示法，即瞬时描述.

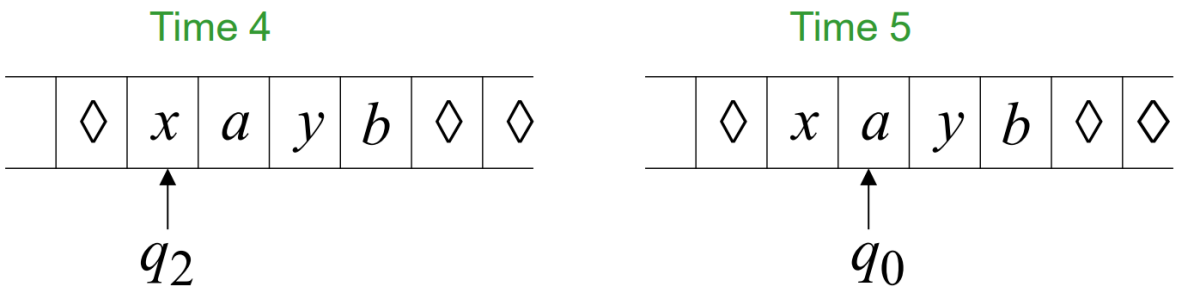
配置用图形可以如下表示：



用瞬时描述，则为 $ca\ q_1\ ba$

> 当前状态写在读写头当前指向符号（b）的左侧.

待补充（move）



A Move: $q_2\ xayb \vdash x\ q_0\ ayb$

2. 递归可枚举语言

> Recursively Enumerable Languages
>
> 有时简称为图灵可识别语言（Turing-Recognizable Languages）

For any Turing Machine M

$$L(M) = \{w : q_0 w \overset{*}{\vdash} x_1 q_f x_2\}$$

The sequence of configurations leading to a halt state will be called a **computation**.

3. 标准图灵机

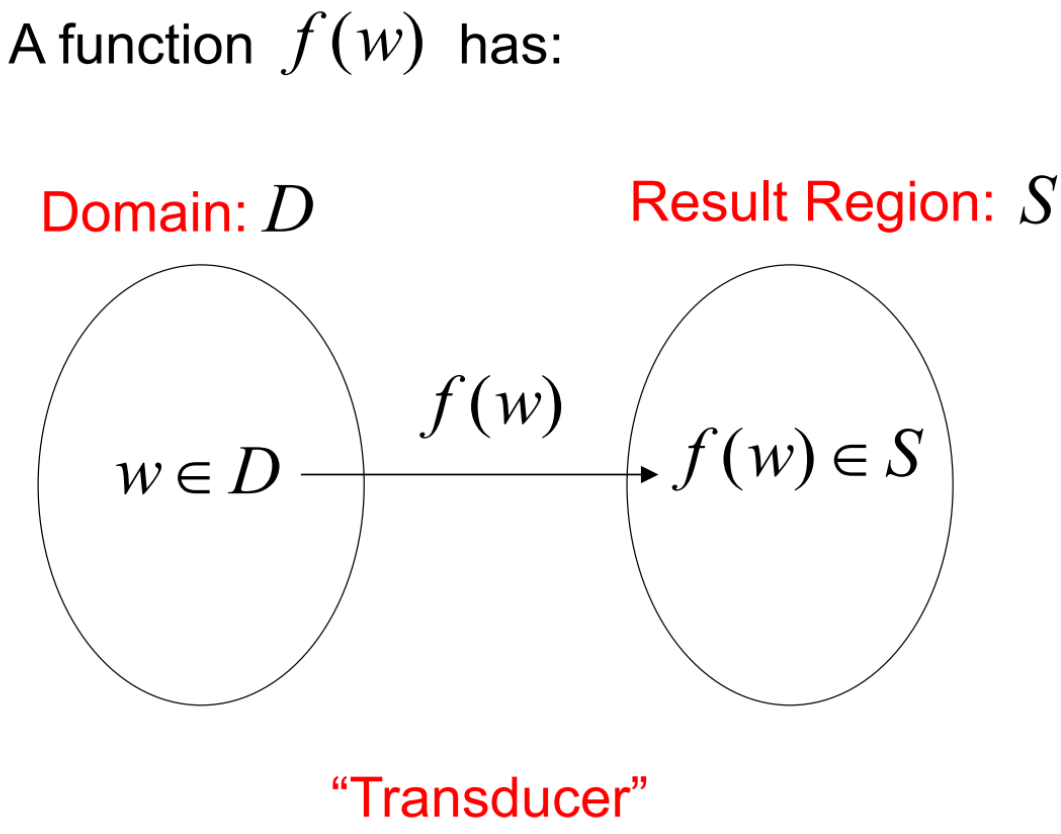
Main features of the standard Turing machine:

- Deterministic
- Infinite tape in both directions
- Tape is the input/output file

4. 使用图灵机计算函数

> Computing Functions with Turing Machines

图灵机不仅可以用来识别语言，还可以用来计算函数.



定义域（Domain）D

- 所有有效输入 w 的集合
- 图灵机从定义域 D 中接受输入字符串 w

结果区域（Result Region）S

- 所有可能的输出 $f(w)$ 的集合
- 图灵机运行结束后，纸带上留下的结果 $f(w)$ 属于这个集合

函数 $f(w)$

- 表示将定义域 D 中的输入 w 映射到结果区域 S 中的输出 $f(w)$ 的规则或过程.

4.1 换能器

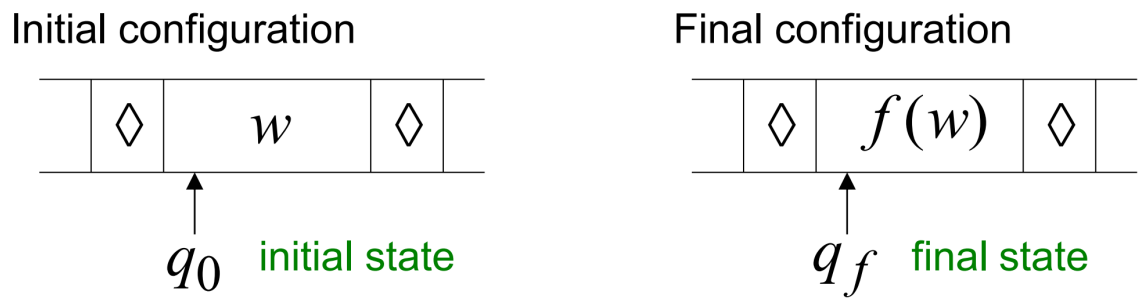
> Transducer

自动机理论中，一个换能器是一种能将一个输入（或输入序列）转换成一个输出（或输出序列）的自动机.

图灵机作为换能器：当图灵机用来计算函数时，它扮演的就是一个换能器的角色.

4.2 图灵可计算

A function f is computable (Turing Computable) if there is a Turing Machine M such that



For all $w \in D$ Domain

A function f is computable (Turing Computable) if there is a Turing Machine M such that

$$q_0 w \overset{*}{\vdash} q_f f(w)$$

For all $w \in D$ domain

A function may have many parameters

Example: Addition function

$$f(x, y) = x + y$$

We prefer unary representation:

Decimal: 5

Binary: 101

Unary: 11111

4.2.1 加法

例：The function $f(x, y) = x + y$ is computable. x, y are integers

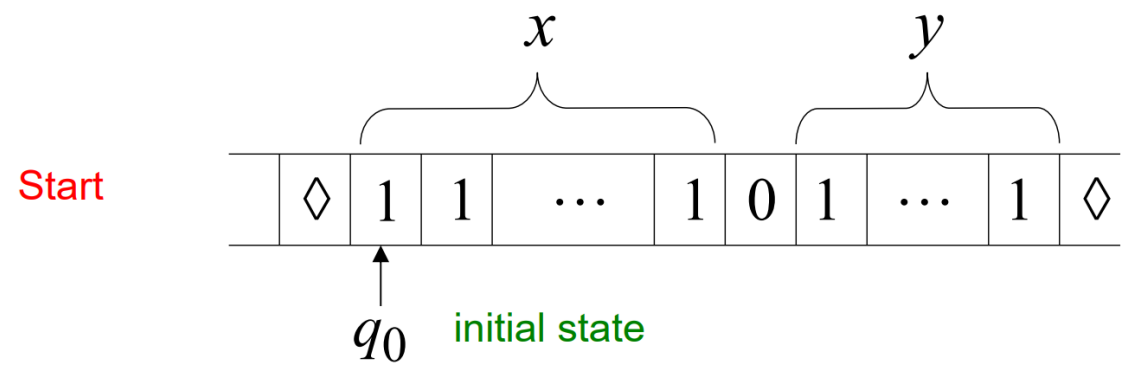
Turing Machine:

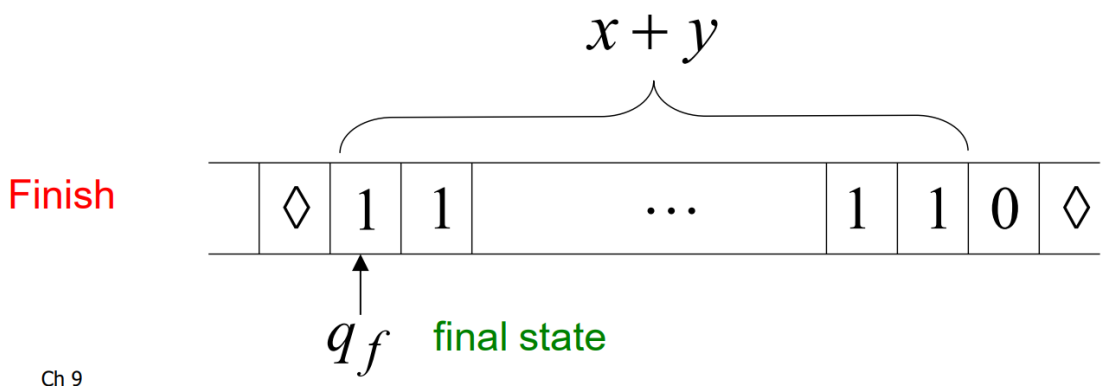
Input string: $x0y$ unary

> 这里 0 是 delimiter that separates the two numbers

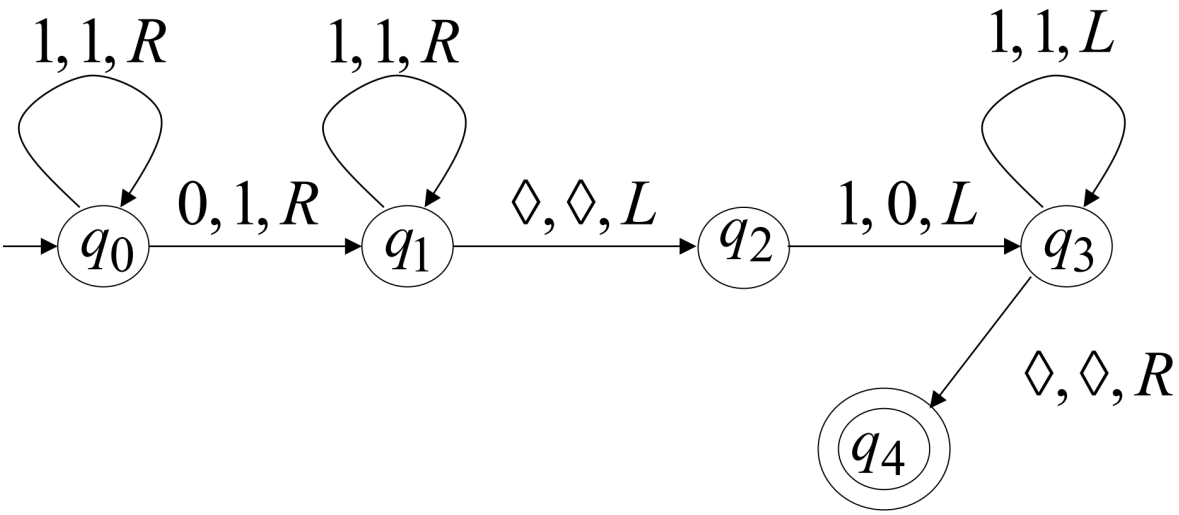
Output string: $xy0$ unary

> The 0 helps when we use the result for other operations





图灵机：



4.2.2 乘法

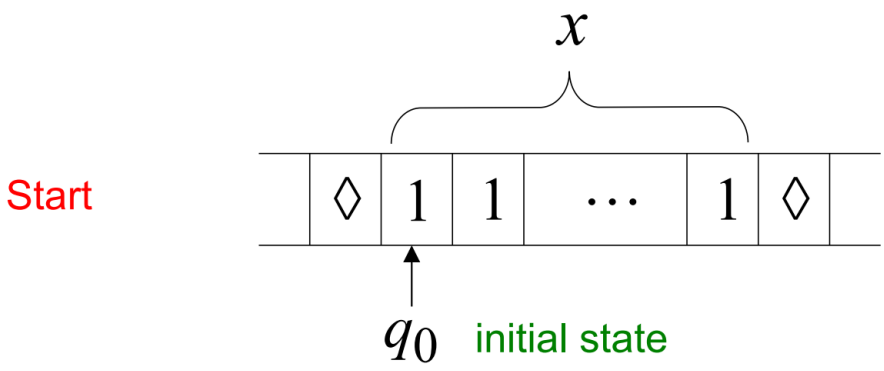
The function $f(x) = 2x$ is computable

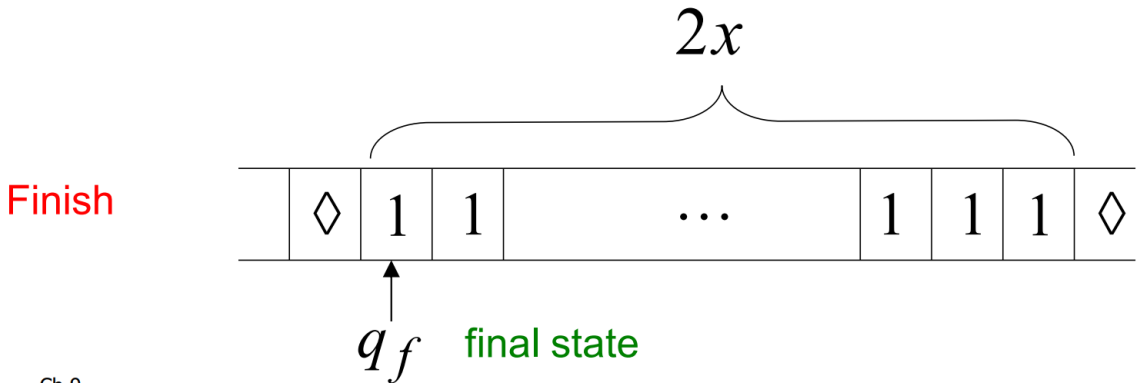
x is integer

Turing machine:

Input string: x unary

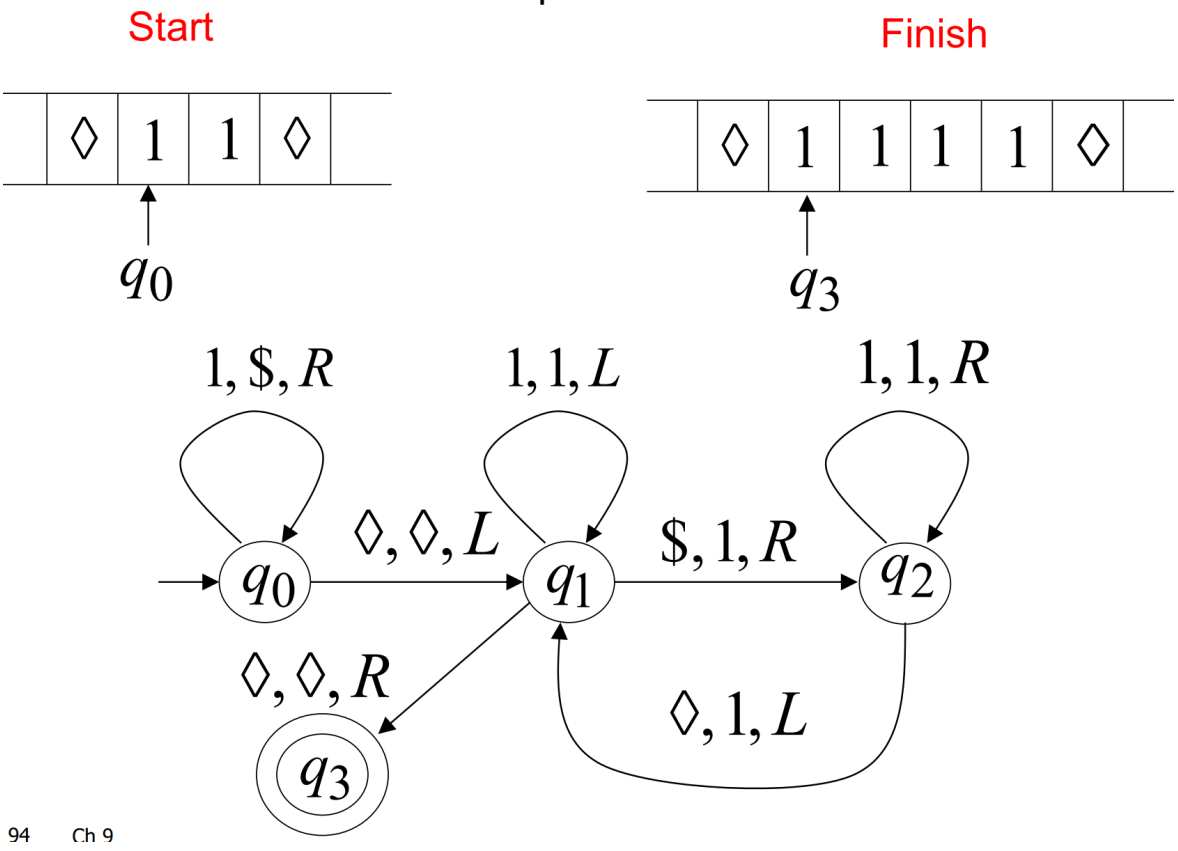
Output string: xx unary


x
Start
◊ 1 1 ··· 1 ◊
q_0 initial state


2x
Finish
◊ 1 1 ··· 1 1 1 ◊
q_f final state

图灵机：

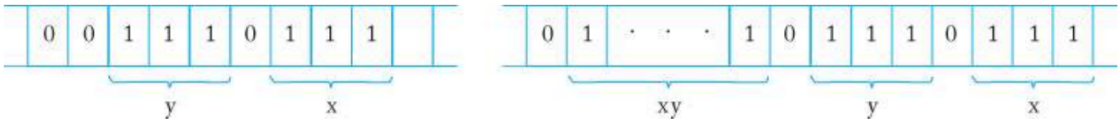
Example


Start
Finish
◊ 1 1 ◊
q_0
◊ 1 1 1 1 ◊
q_3
1, $, R
1, 1, L
1, 1, R
◊, ◊, L
$, 1, R
◊, ◊, R
◊, 1, L
q_0
q_1
q_2
q_3

例:

- Design a Turing machine that multiplies two positive integers in unary notation

1. Repeat the following steps until x contains no more 1's
   Find a 1 in x and replace it with another symbol a
   Replace the leftmost 0 by 0y
2. Replace all a's with 1's



4.2.3 比较

$$f(x,y)=\begin{cases}1 & \text{if } x>y\\ 0 & \text{if } x\le y\end{cases}$$

This function is computable.

Input: $x0y$

Output: 1 or 0

伪代码:

Repeat

    match a 1 from x with a 1 from y

Until all of x or y is matched

If a 1 from x is not matched

    erase tape, write 1

else

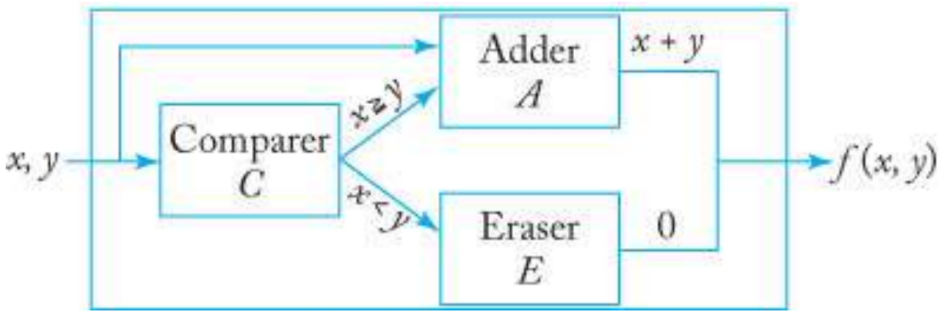
    erase tape, write 0

4.3 组合图灵机

> Combining Turing Machines

Example 9.12:

$$f(x,y)=\begin{cases} x+y & \text{if} \quad x \geq y \\ 0 & \text{if} \quad x<y \end{cases}$$



$$q_{C,0}w(x)0w(y) \overset{*}{\vdash} q_{A,0}w(x)0w(y) \quad \text{If x} \geqq \text{y} \quad \Longrightarrow \quad q_{A,0}w(x)0w(y) \overset{*}{\vdash} q_{A,f}w(x+y)0$$

$$q_{C,0}w(x)0w(y) \overset{*}{\vdash} q_{E,0}w(x)0w(y) \quad \text{If x} < \text{y} \quad \Longrightarrow \quad q_{E,0}w(x)0w(y) \overset{*}{\vdash} q_{E,f}0$$

这里涉及宏指令（macroinstruction）

```
if a then q_j else q_k
```

微观上在图灵机是这么实现的：

转移函数	描述
$\delta(q_i,a)=(q_{j0},a,R)$	**匹配 a 的情况**：从任何状态 q_i 读取到 a，进入新的中间状态 q_{j0}，磁带符号保持不变 a，并将读写头向右移动一位 (R). 这开始了一个过渡到状态 q_j 的序列.
$\delta(q_i,b)=(q_{k0},b,R)$	**不匹配 a 的情况**：从任何状态 q_i 读取到任何不是 a 的符号 $b \in \Gamma-\{a\}$，进入新的中间状态 q_{k0}，磁带符号保持不变 b，并将读写头向右移动一位 (R). 这开始了一个过渡到状态 q_k 的序列.
$\delta(q_{j0},c)=(q_j,c,L)$	**完成 q_j 过渡**：从状态 q_{j0} 读取到任何符号 $c \in \Gamma$，进入目标状态 q_j，磁带符号 c 保持不变，并将读写头向左移动一位 (L).

转移函数	描述
$\delta(q_{k0}, c) = (q_k, c, L)$	**完成 q_k 过渡：**从状态 q_{k0} 读取到任何符号 $c \in \Gamma$，进入目标状态 q_k，磁带符号 c 保持不变，并将读写头向左移动一位 (L).

4.4 图灵论题

Any computation carried out by mechanical means can be performed by a Turing Machine

Computer Science Law: A computation is mechanical if and only if it can be performed by a Turing Machine

There is no known model of computation more powerful than Turing Machines

4.4.1 算法的定义

An algorithm for function $f(w)$ is a Turing Machine which computes $f(w)$

When we say: There exists an algorithm

We mean: There exists a Turing Machine that executes the algorithm

4.4.2 图灵论题

Anything that can be done on any existing digital computer can also be done by a Turing machine

No one has yet been able to suggest a problem, solvable by what we intuitively consider an algorithm, for which a Turing machine program cannot be written

Alternative models have been proposed for mechanical computation, but none of them is more powerful than the Turing machine model

Lec 10 图灵机变体

本章讨论图灵机的各种变体，并证明它们与标准图灵机模型具有相同的计算能力.

关键在于证明每种变体都可以被标准图灵机模拟，反之亦然.

Same Power of two classes means:

- Both classes of Turing machines accept the same languages
- For any machine M_1 of first class there is a machine M_2 of second class such that $L(M_1) = L(M_2)$

模拟（Simulation）：a technique to prove same power

Simulate the machine of one class with a machine of the other class

1. Stay-Option 图灵机

读写头除了左移（L）和右移（R）外，还可以停留在原地（S）

证明：Stay-Option Machines have the same power with Standard Turing machines

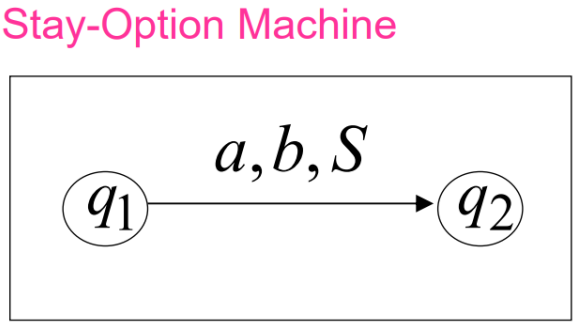
① Stay-Option Machines are at least as powerful as Standard machines

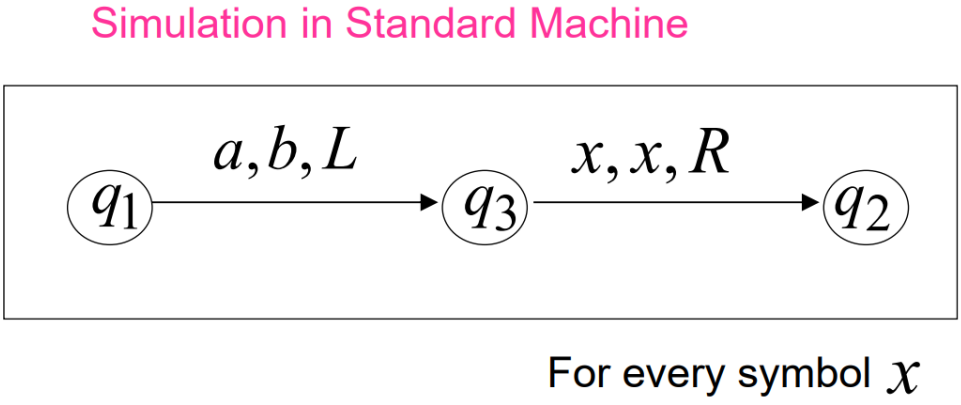
> 因为标准图灵机就是特殊的 Stay-Option 图灵机，只是从来不使用 Stay. 所以 Stay-Option 图灵机可以模拟标准图灵机，因此 Stay-Option 图灵机能力大于等于标准图灵机.

② Standard Machines are at least as powerful as Stay-Option machines

标准图灵机也可以模拟 Stay-Option 图灵机

模拟方式：对于左右移动的转移，直接复制；对于 Stay 的转移：





2. Semi-Infinite Tape 图灵机

磁带仅向一个方向（例如右侧）无限延伸.

2.1 多轨磁带

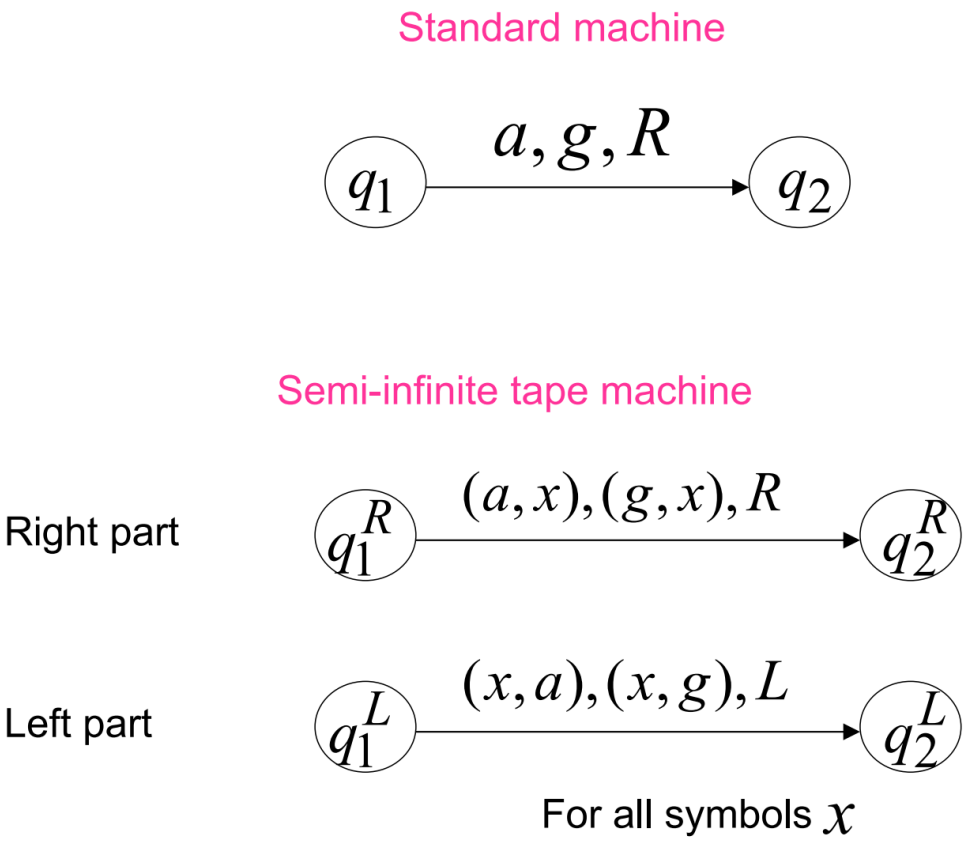
> Multiple Track Tape

多轨磁带 $\neq$ 多带图灵机（Multitape Turing Machines）

2.2 Semi-Infinite Tape

标准图灵机模拟半无穷磁带：trivial.

半无穷磁带模拟标准图灵机：



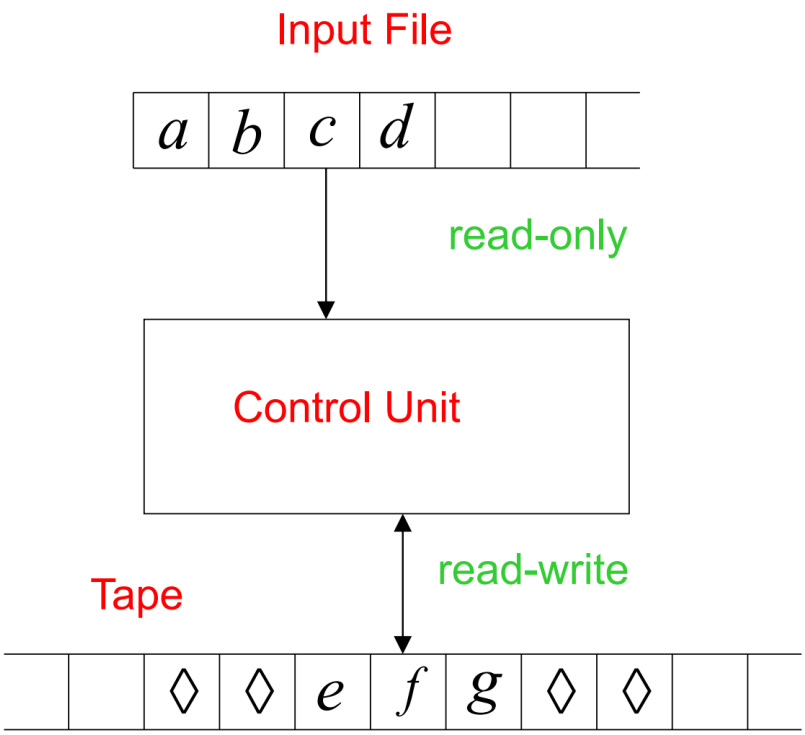
理解：

- 把标准图灵机磁带从参考点对折，分别作为半无限磁带的两轨.

- 标准图灵机在处理参考点左侧字符时，半无限带机在处理 Left part 轨. 但是因为多轨本质是一条物理磁带每个位置存放多元组，所以双轨必须同时处理（和多带不同）. 此时只需要对 Right part 轨写入和读取相同的字符即可（就等效于没有对 Right part 进行操作）.
- 当标准图灵机处理参考点右侧时类似.
- 由于标准图灵机同一状态接受参考点左侧和右侧的字符对半无限磁带机来说是不同的，因此半无限磁带机的控制单元需要把标准图灵机所有状态复制为 L 和 R 两份.

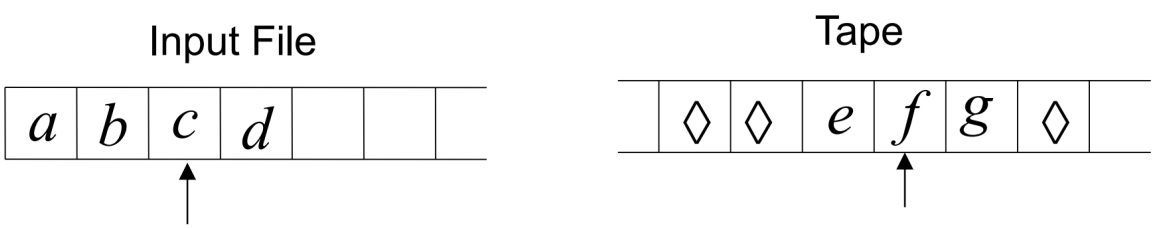
3. Off-Line 图灵机

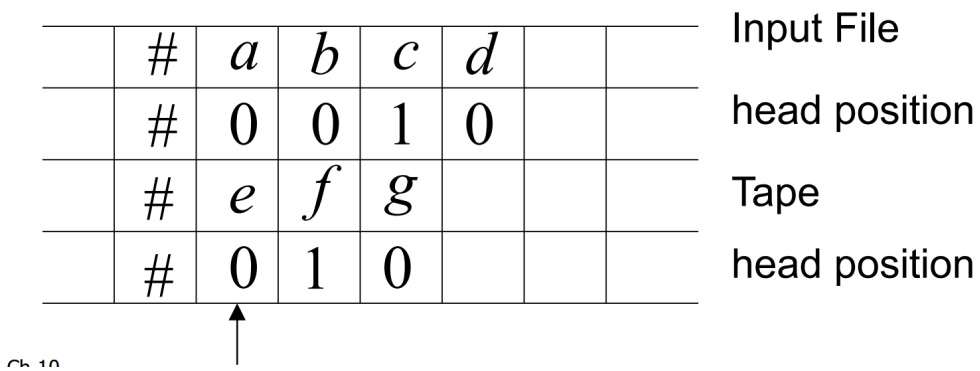
离线图灵机：除了读写磁带外，还有一个只读的输入文件.



离线机模拟标准图灵机：把 Input copy 到磁带，后续操作同标准图灵机.

标准图灵机模拟离线机：使用四轨磁带.





对于离线图灵机的每一次状态转换，标准图灵机需要执行一系列复杂动作来模拟：

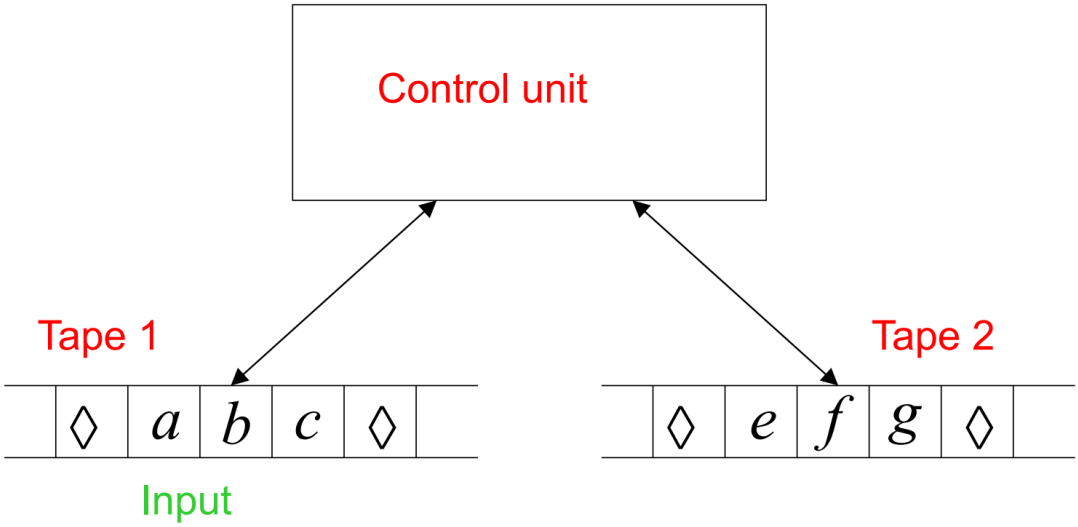
1. 返回参考点 `#`.
2. 查找当前输入文件符号（关注前两行）
3. 返回参考点 `#`.
4. 查找当前磁带符号（关注后两行）.
5. 转换：根据离线机的转换规则，更新磁带内容和两个磁头的位置标记.

结论：Off-line machines have the same power with Standard machines

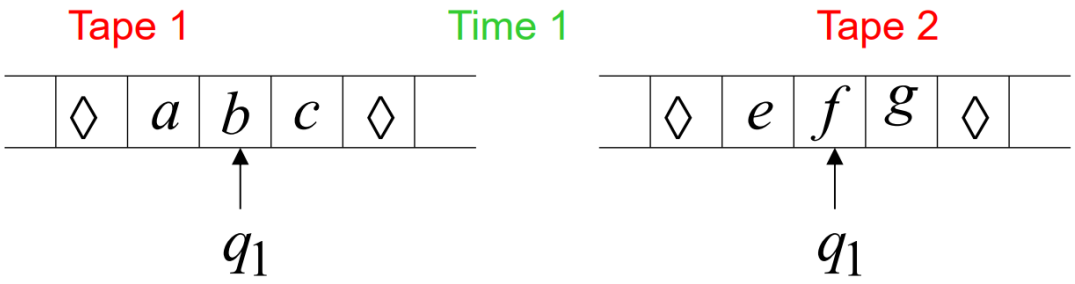
4. Multitape 图灵机

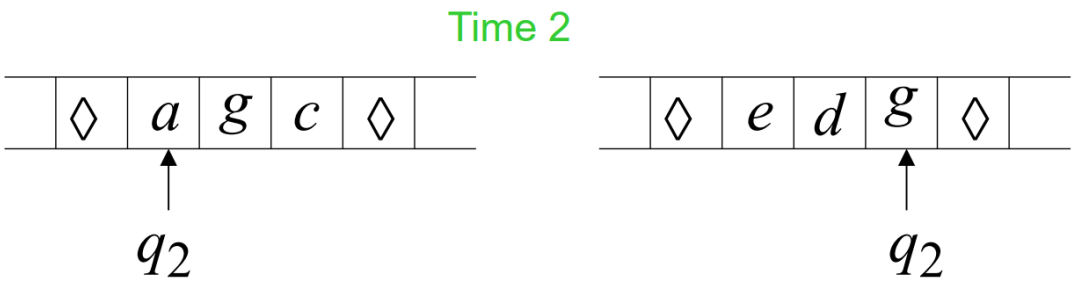
具有多个独立的磁带和磁头，所有磁头同步移动.

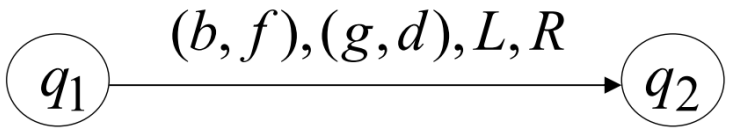
Multitape Turing Machines



多条磁带同时移动，但是可以朝不同方向.







结论：Multi-tape machines have the same power with Standard Turing Machines

注意：Same power doesn't imply same speed.

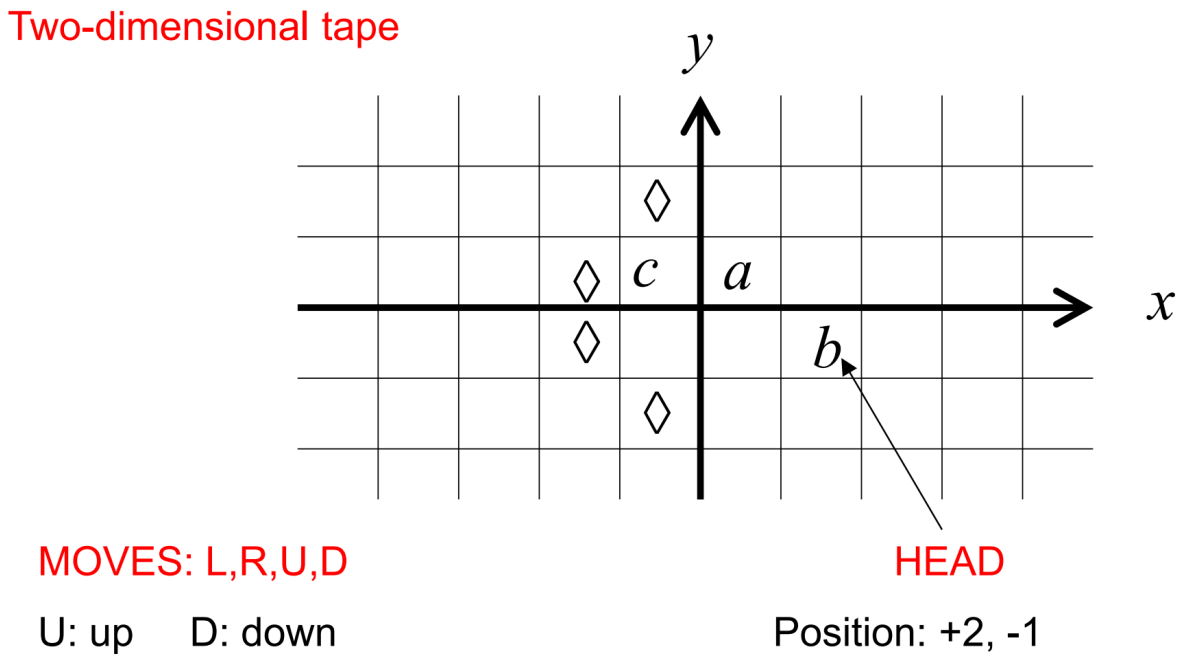
对于语言 $L=\{a^nb^n\}$,

标准机的接受时间是 $O(n^2)$：Go back and forth n^2 times

双带机的接受时间是 $O(n)$

- Copy b^n to tape 2, $O(n)$
- Leave a^n on tape 1, $O(n)$
- Compare tape 1 and tape 2, $O(n)$

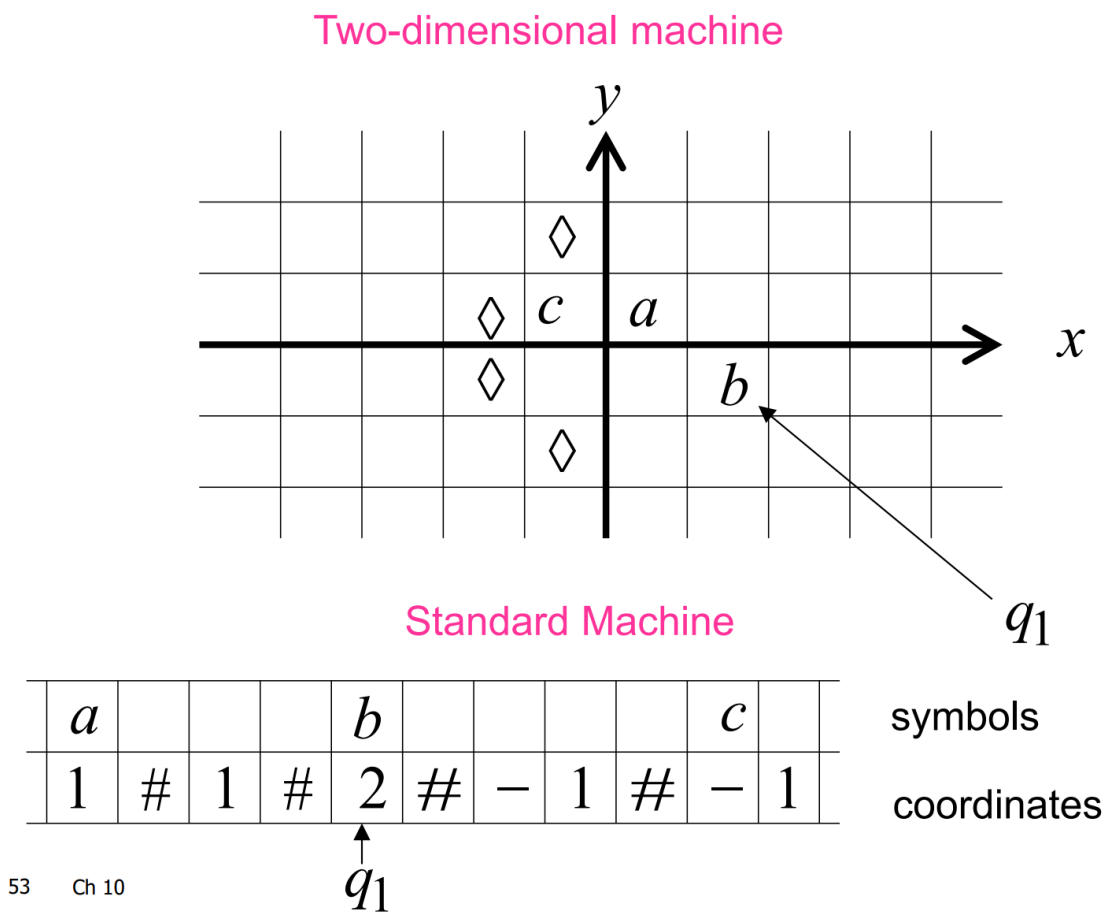
5. MultiDimensional 图灵机



Multidimensional machines simulate Standard machines: Use one dimension

Standard machines simulate Multidimensional machines:

- Use a two track tape
- Store symbols in track 1
- Store coordinates in track 2



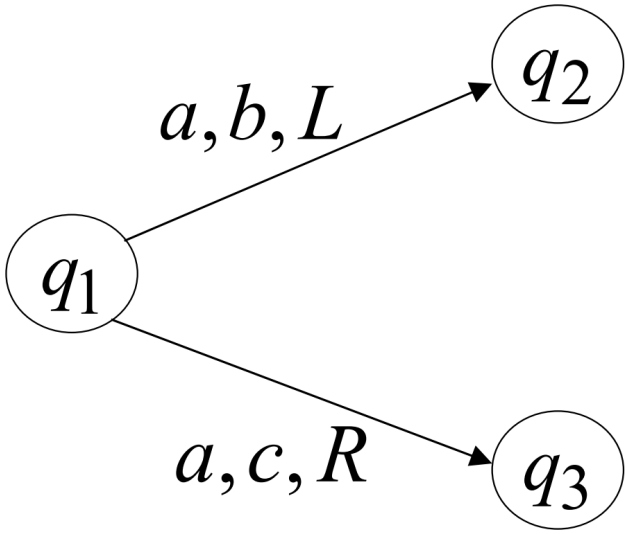
53 Ch 10

标准图灵机模拟二维图灵机，Repeat for each transition

- Update current symbol
- Compute coordinates of next position
- Go to new position

结论：MultiDimensional Machines have the same power with Standard Turing Machines

6. NonDeterministic 图灵机



Input string w is accepted if this is a possible computation:

$$q_0 w \overset{*}{\vdash} x q_f y$$

其中

$q_0 w$ 是 Initial configuration

q_f 是 Final state

$x q_f y$ 是 Final Configuration，机器到这个配置时停机并接受.

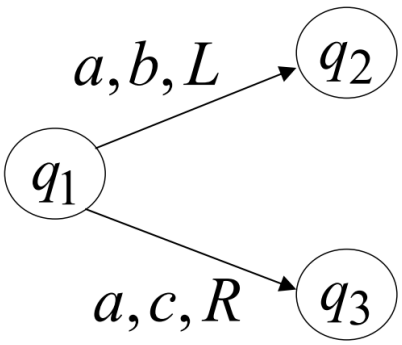
只要存在一条计算路径即接受，不需要所有路径都接受.

Nondeterministic Machines simulate Standard (deterministic) Machines: Every deterministic machine is also a nondeterministic machine

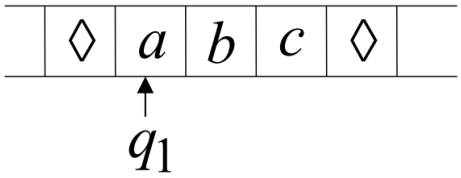
Deterministic machines simulate NonDeterministic machines:

- Keeps track of all possible computations
- Stores computations in a two-dimensional tape

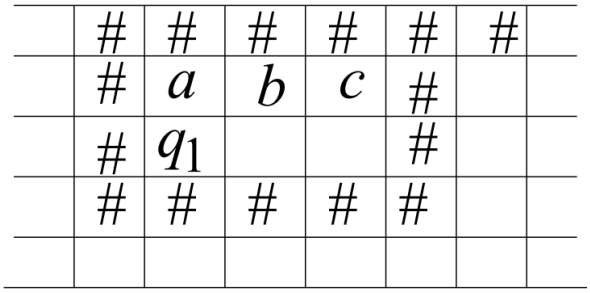
NonDeterministic machine



Time 0



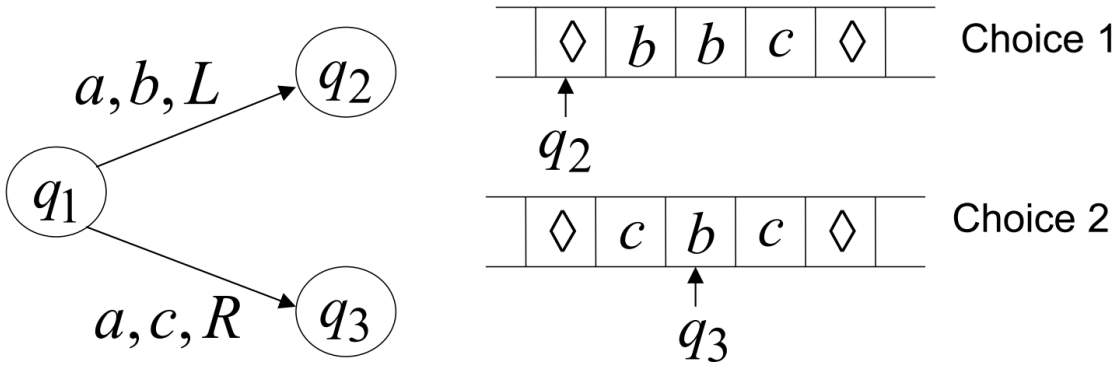
Deterministic machine



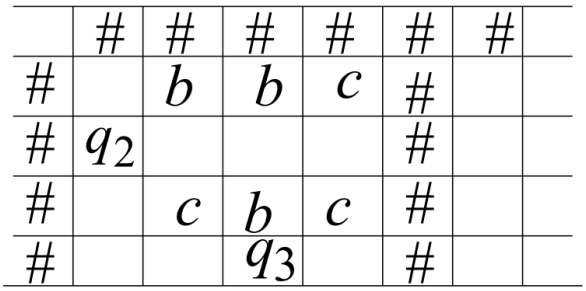
Computation 1

NonDeterministic machine

Time 1



Deterministic machine



Computation 1

Computation 2

确定性（二维）图灵机模拟非确定性图灵机：

> 类似使用广度优先搜索来探索非确定性图灵机的计算树.

Repeat

- 执行每条计算路径的一步：DTM 会依次查看它当前磁带上存储的所有正在进行的 NTM 计算路径（配置），并对每一条路径执行 NTM 的一次状态转换.
- 如果当某条计算路径执行 NTM 状态转换时，它有 ≥ 2 个可能的下一步选择，则在空白处创建分支.

结论：NonDeterministic Machines have the same power with Deterministic machines

注意：The simulation in the Deterministic machine takes time exponential time compared to the NonDeterministic machine

7. 通用图灵机

标准图灵机是硬编码的，只能执行一个程序. 通用图灵机是可重新编程的机器.

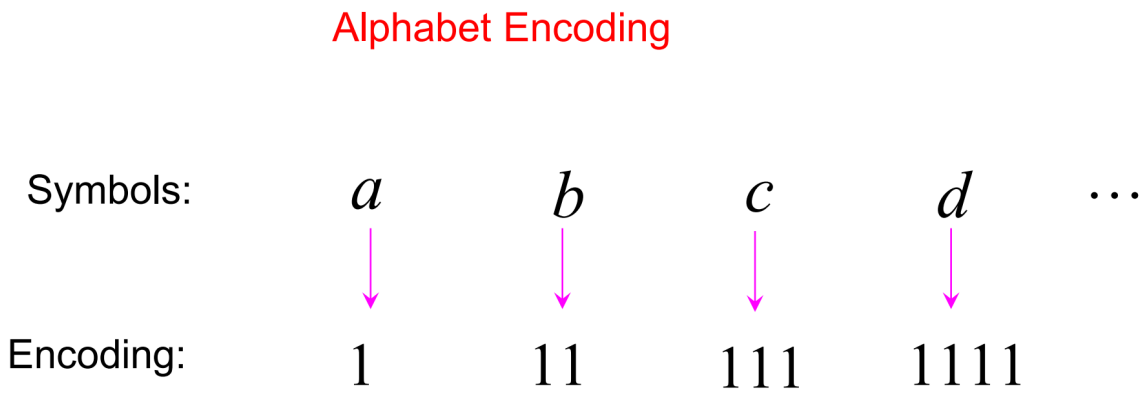
通用图灵机可以模拟任何其他图灵机. 它的输入包括被模拟图灵机的转换描述和初始磁带内容.

它通常有三个磁带：

- 磁带 1：存储被模拟图灵机的编码（二进制字符串）
- 磁带 2：存储被模拟图灵机的磁带内容
- 磁带 3：存储被模拟图灵机的状态.

7.1 图灵机二进制编码

We encode Turing machine M as a string of symbols



State Encoding

States: q_1 q_2 q_3 q_4 $\cdots$

Encoding: 1 11 111 1111

Head Move Encoding

Move: L R

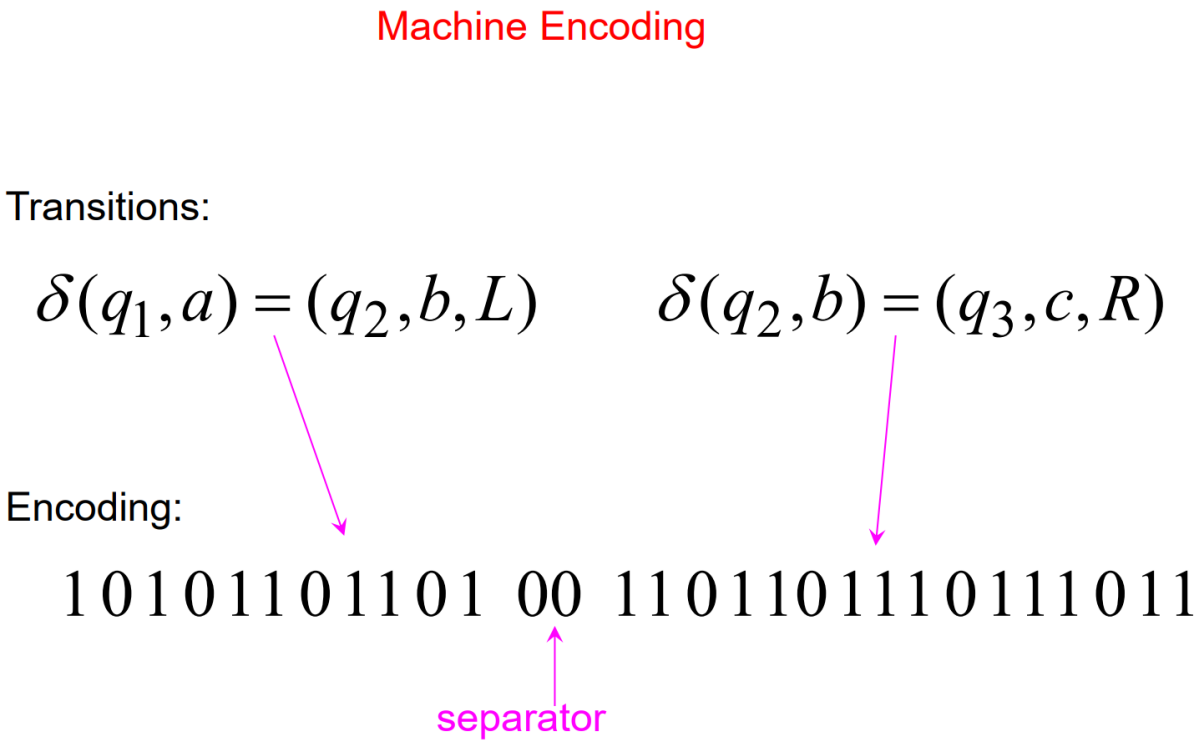
Encoding: 1 11

Transition Encoding

Transition: $\delta(q_1, a) = (q_2, b, L)$

Encoding: $1\,0\,1\,0\,11\,0\,11\,0\,1$

separator



Tape 1 contents of Universal Turing Machine:

encoding of the simulated machine M as a binary string of 0's and 1's

因为每个图灵机可以被一个二进制编码描述，因此图灵机的集合即一个语言. 这个语言的每个字符串都对应一台图灵机的二进制编码.

8. 可数 & 不可数

Infinite sets are either Countable or Uncountable.

8.1 可数集

> Countable set

Any **finite set** or

> 注意：有限集一定可数.

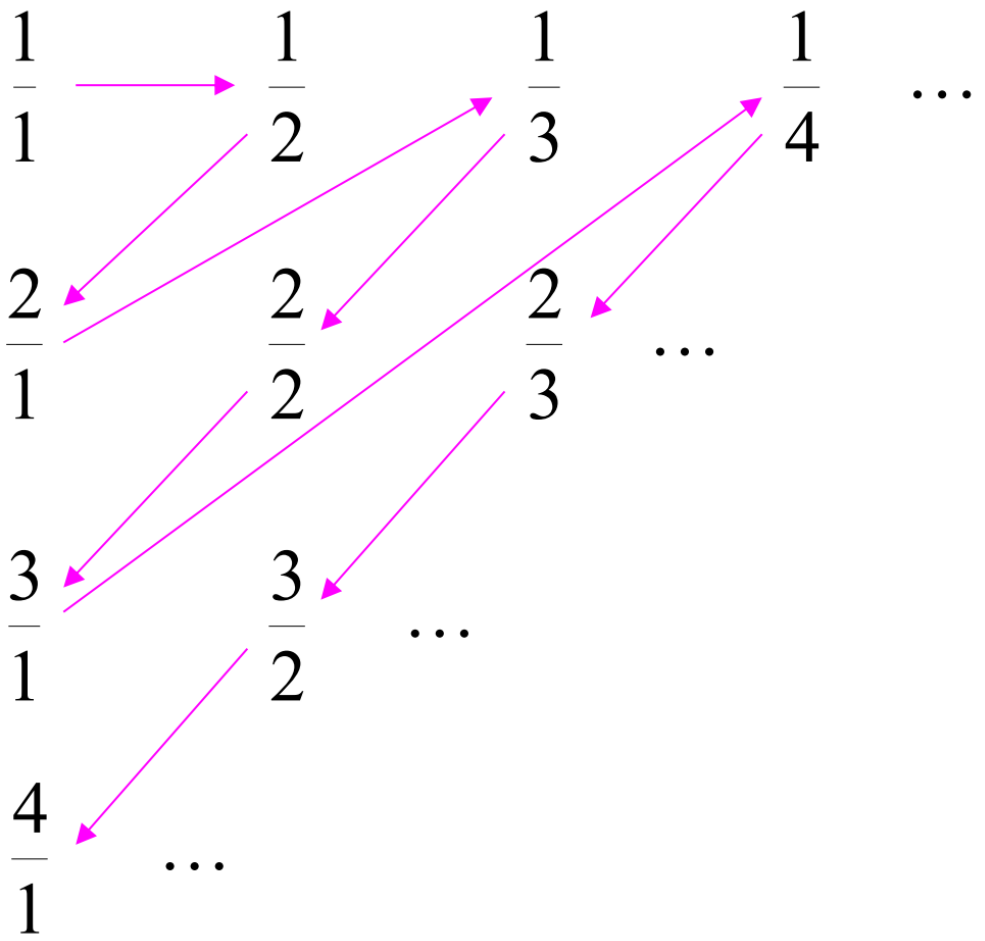
any countably infinite set:

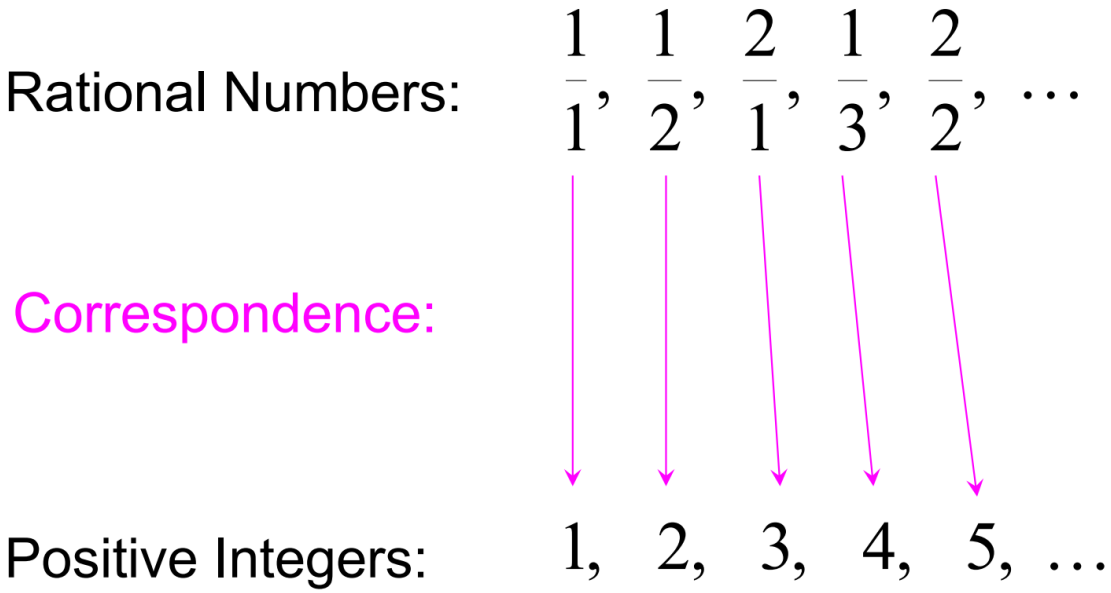
- There is one to one correspondence between elements of the set and natural numbers

可数集的子集一定可数.

例：偶数集是可数的. 因为偶数 $2n$ 和正整数 $n+1$ 一一对应.

例：有理数集是可数的.


1/1 1/2 1/3 1/4 …
2/1 2/2 2/3 …
3/1 3/2 …
4/1 …


Rational Numbers:
1/1, 1/2, 2/1, 1/3, 2/2, …
Correspondence:
Positive Integers:
1, 2, 3, 4, 5, …

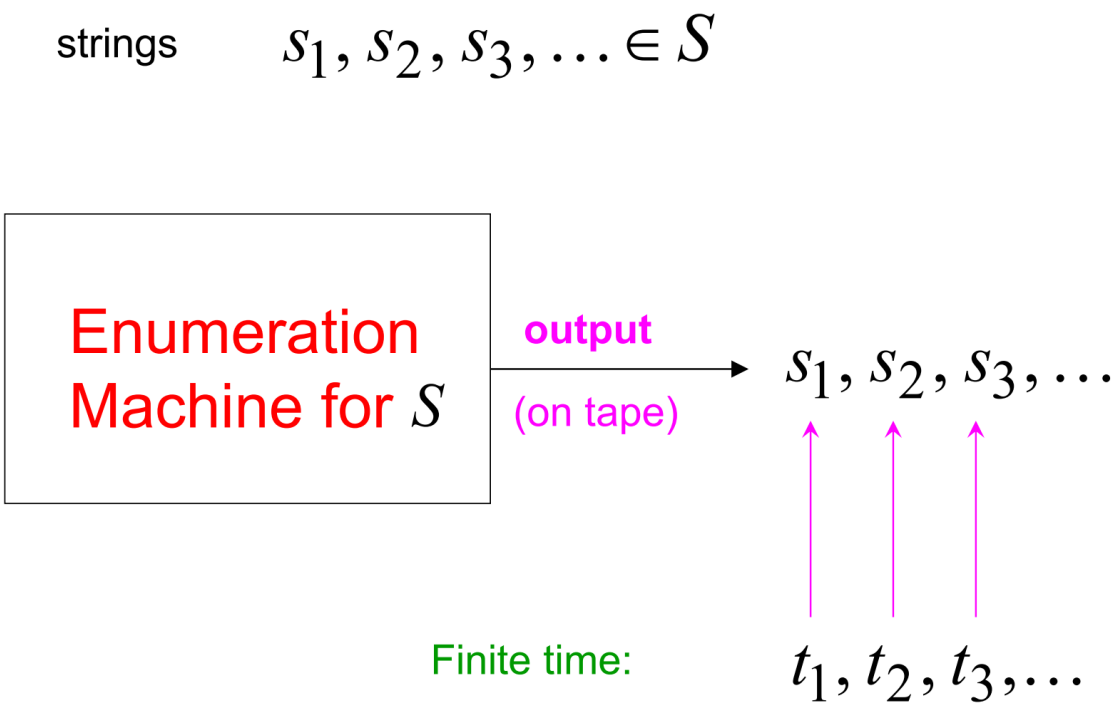
8.2 枚举过程

Let S be a set of strings.

An **enumeration procedure** for S is a Turing Machine that generates all strings of S one by one.

and each string is generated in **finite time**.

> 注意，这里不要求 S 要在有限时间内全部计算完，而是要求每个字符串在有限时间内计算完.



If for a set there is an enumeration procedure, then the set is countable

定理：**所有图灵机的集合是可数的.**

证明：任意图灵机可以编码为 0's 和 1's 的二进制字符串. 因此可以给出图灵机集合的枚举过程：

- 按一定顺序枚举 $0, 1$ 构成的字符串
- 每个枚举，判断是否是图灵机. 若是，则输出，不是则忽略.

注意：一个二进制字符串是否是图灵机的有效编码，这是可判定的. 因为图灵机的编码是按照语法规则进行，图灵机是有限元组，组成部分有限，转移函数是有限规则集，编码规则是有限、形式化的. 对一个有限长度的字符串进行有限次的扫描和匹配，就可以判定该字符串是否是一台图灵机，整个判定过程是保证停机的.

8.3 不可数集

A set is uncountable if it is not countable

定理：Let S be an **infinite** countable set, the powerset 2^S of S is uncountable.

> 注意：无限可数集的幂集是不可数集，但有限可数集的幂集是可数集. 因为有限可数集的幂集仍然有限，有限的集合一定是可数的.

证明：

Since S is countable, we can write

$S = \{s_1, s_2, s_3, \ldots\}$

S 的子集构成的集合即幂集 2^S，幂集的每个元素都是 S 的子集：

Powerset element	Encoding				
	s_1	s_2	s_3	s_4	$\cdots$
$\{s_1\}$	1	0	0	0	$\cdots$
$\{s_2, s_3\}$	0	1	1	0	$\cdots$
$\{s_1, s_3, s_4\}$	1	0	1	1	$\cdots$

把幂集元素排成一列放左侧，右侧记录每个幂集元素， S 中的元素是否在内，1 表示在.

假设幂集可数，则左侧幂集元素可以用枚举过程枚举：

Powerset element	Encoding					
t_1	1	0	0	0	0	$\cdots$
t_2	1	1	0	0	0	$\cdots$
t_3	1	1	0	1	0	$\cdots$
t_4	1	1	0	0	1	$\cdots$

取对角线数字，逐个取反，生成一个新幂集元素 t_i.

按照定义，t_i 的第 i 个元素是取反自己第 i 个元素得到的，即又等于 0 又等于 1，显然不可能. 因此，幂集可数（幂集元素可以枚举）的假设是错的.

注意：康托尔对角线法只适用于无限集（原集合和幂集均无限）

有限集的幂集直接判断为可数，因为有限集的幂集也是有限的.

实数集也是不可数的，可以用康托尔对角线法证明.

证明：假设可数，则 $[0,1]$ 之间的实数也可数.

按枚举顺序排成一列：

$$
\begin{aligned}
r_1 &= 0.d_{11}d_{12}\dots \\
r_2 &= 0.d_{21}d_{22}\dots \\
r_3 &= 0.d_{31}d_{32}\dots \\
&\vdots
\end{aligned}
$$

d_{ij} 是第 i 个数 r_i 的第 j 位小数.

构造矛盾数 $r^* \in [0,1]$，它的第 i 位计为 d_i^*:

- 若 $d_{ii} = 1$，则 $d_i^* = 2$
- 若 $d_{ii} \neq 1$，则 $d_i^* = 1$

这个数是实数，应该存在于列表中，假设它是 r_i.

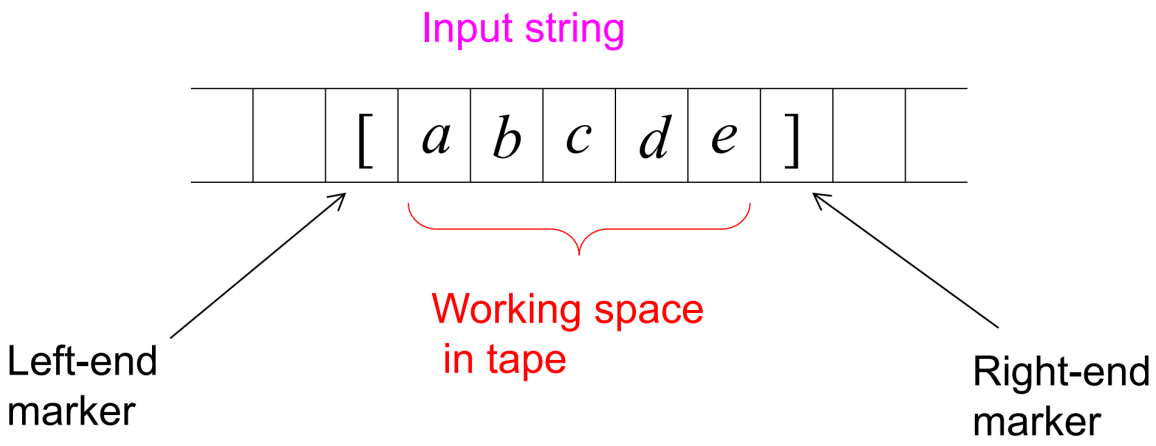
然而，r_i 的第 i 位由自己构造而来，如果自己等于 1 则为 2，如果自己不等于 1 则为 1. 矛盾.

9. 线性有界自动机

> Linear Bounded Automata, LBA

LBA 与图灵机相同，但其可用的磁带空间仅限于输入字符串所占用的空间

能力：LBA 能力强于非确定性下推自动机，但弱于标准图灵机.



All computation is done between end markers

线性有界自动机接受的语言是上下文敏感语言.

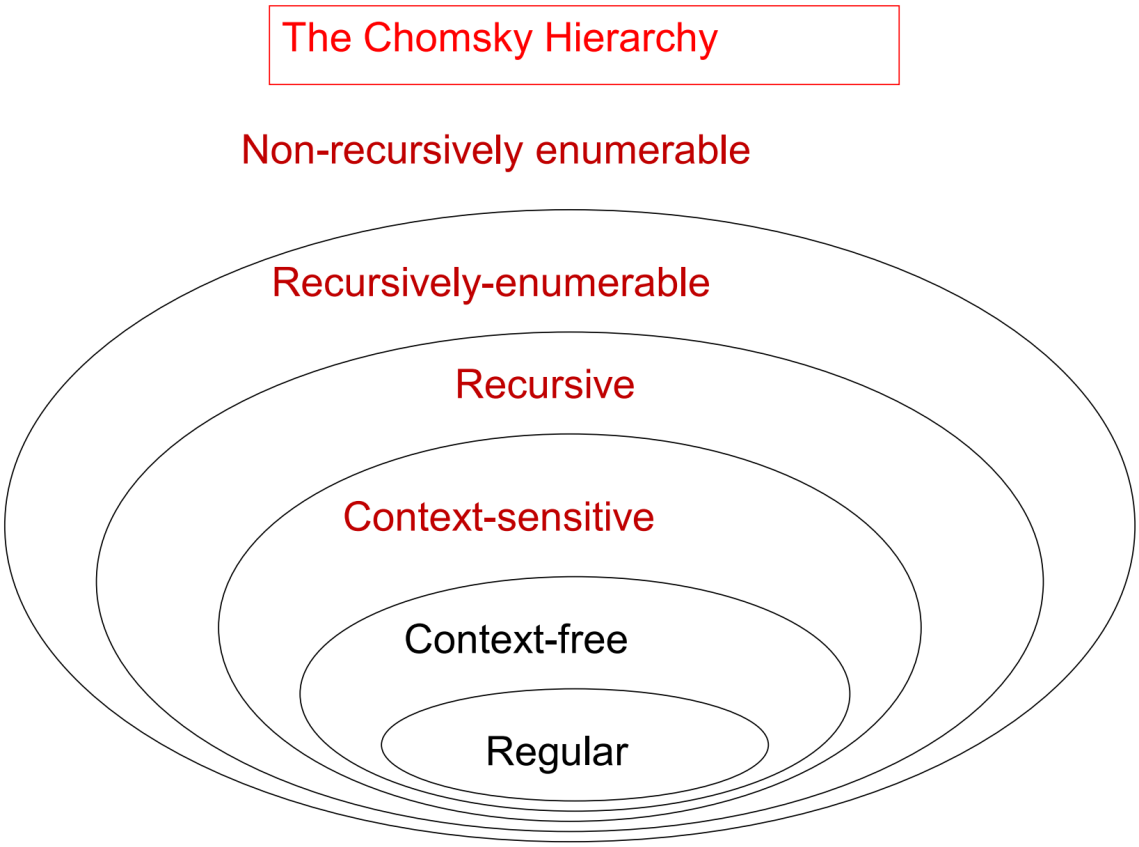
> 见 `Lec 11-12 8. 上下文敏感文法`

Lec 11-12 递归语言 递归可枚举语言

1. Chomsky 谱系

> The Chomsky Hierarchy

`Lec 10` 中提到，存在图灵机无法识别的语言.



2. 递归可枚举语言

> recursively enumerable

A language is **recursively enumerable** (RE) if some Turing machine accepts it

Let L be a recursively enumerable language and M the Turing Machine that accepts it

For string w:

If $w \in L$, then M halts in a final state

If $w \notin L$, then M halts in a non-final state **or loops forever**

> 无限循环也视为不接受.
>
> 注意，字符串长度有限，但仍可能使某些 M 陷入死循环.

递归可枚举语言的子集不一定是递归可枚举.

3. 递归语言

A language is **recursive** if some Turing machine accepts it and **halts on any input string**（对任何输入都不会陷入死循环）.

换句话说，递归语言存在一个**成员判定算法** (membership algorithm).

Let L be a recursive language and M the Turing Machine that accepts it

For string w:

If $w \in L$, then M halts in a final state

If $w \notin L$, then M halts in a non-final state

递归语言是递归可枚举语言的一部分，即递归语言一定是递归可枚举语言，递归可枚举语言不一定是递归语言.

4. 枚举过程

> Enumeration Procedures

枚举过程（Enumeration Procedure）是指一种**算法或过程**，它能够系统地、一个接一个地列出（或"打印"）一个语言中所有的字符串.

如果一个语言是无限的，枚举过程也必须能够无限地运行，并列出语言中每一个字符串.

注意，枚举过程强调的是完整性而非顺序性，只要保证语言中每一个字符串最终都在某个有限时间点被打印输出，这个过程就是有效的枚举过程，打印的顺序并不重要.

递归可枚举 $\Leftrightarrow$ 枚举过程

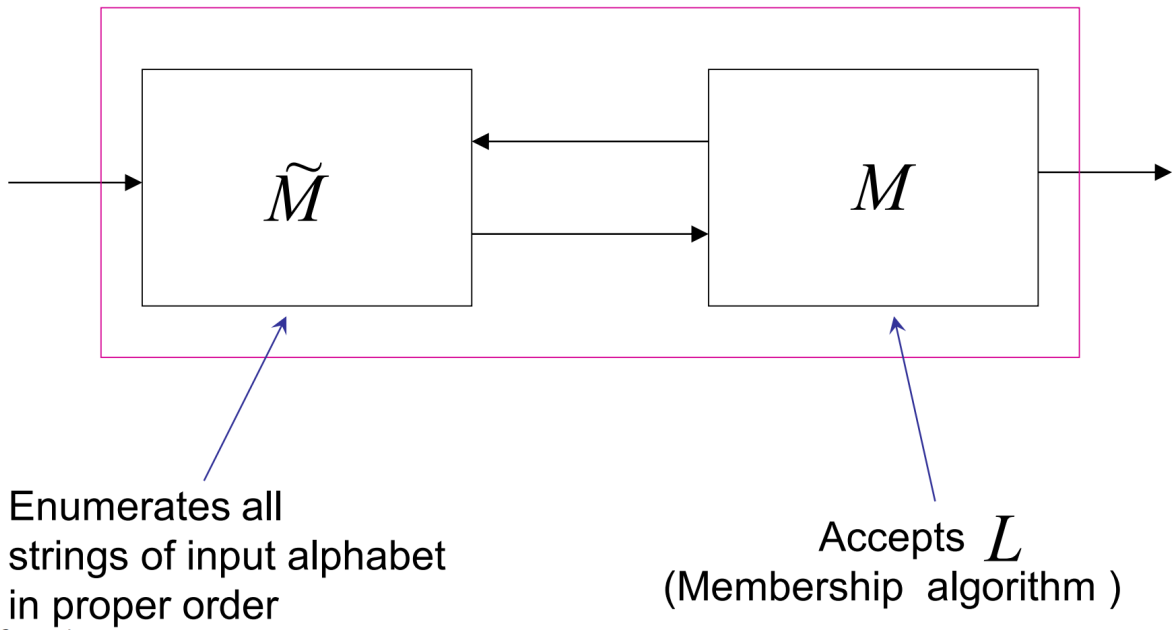
一个语言是递归可枚举的，当且仅当存在一个枚举过程可以列出该语言中的所有字符串.

4.1 递归语言 $\Rightarrow$ 枚举过程

If a language is recursive then there is an enumeration procedure for it

证明核心：将图灵机 M（成员判定器）转化为一个枚举机.

假设字母表为 $\{a, b\}$（以二元作示例，实际可任意）.



枚举机由两部分构成，$\tilde{M}$ 按词典顺序（$\lambda, a, b, aa, ab, ba, bb, \ldots$）依次产生所有可能的字符串，并将这些字符串传给 M. M 是一台接受 L 的图灵机，检查 $\tilde{M}$ 传来的字符串，若属于 L 则接受并打印，不属于 L 则拒绝并忽略. 因为 L 是递归语言，即 M 接受属于它的所有字符串，并且对于不属于它的字符串，M 总会停止在一个非最终状态来表示拒绝，而不会陷入死循环. 整台枚举机就是一个把 L 中字符串一一枚举出来的机器. 枚举的过程即 L 对应的枚举过程.

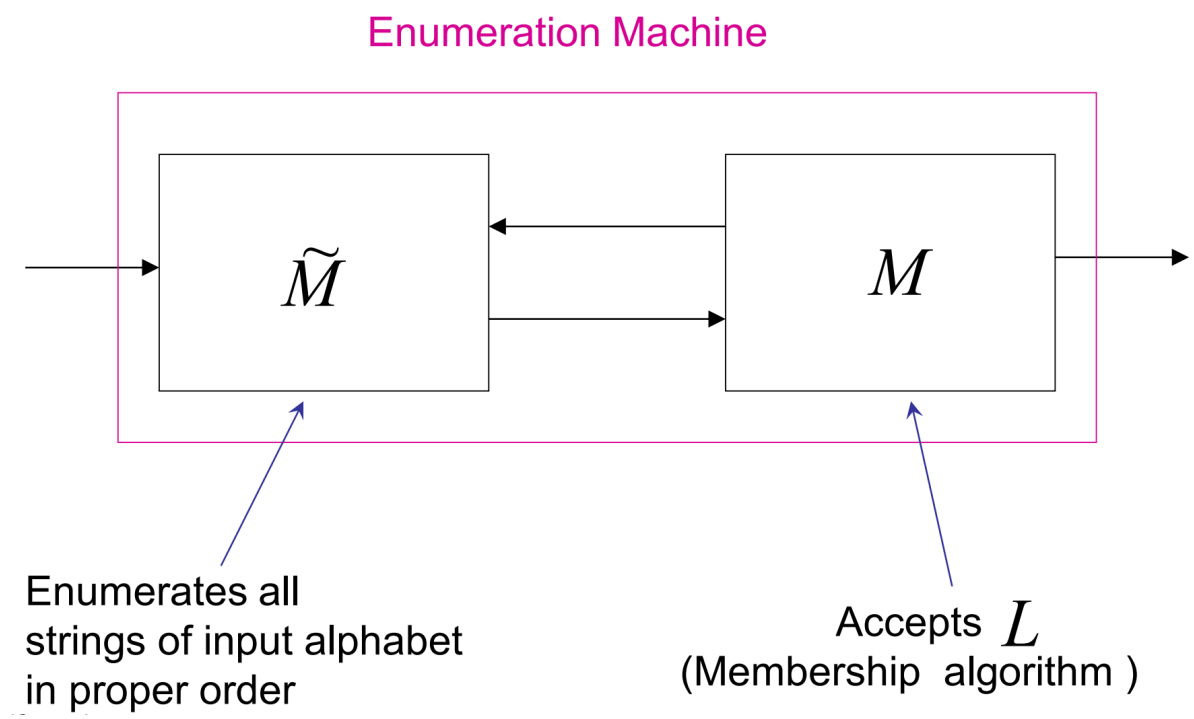
$\tilde{M}$ 一定存在，因为字母表的任意组合是可枚举的（和自然数集一一对应，即可数）.

4.2 递归可枚举 $\Rightarrow$ 枚举过程

If a language L is recursively enumerable then there is an enumeration procedure for it

> 注意，这个定理比 `4.1` 更强. 因为它可以直接导出 `4.1`

同 `4.1`，构造枚举机:



但是，这里不保证递归语言，只保证递归可枚举，即不属于 L 的字符串可能使 M 陷入死循环，无法判定.

4.2.1 交替检查

更好的方法：交替检查

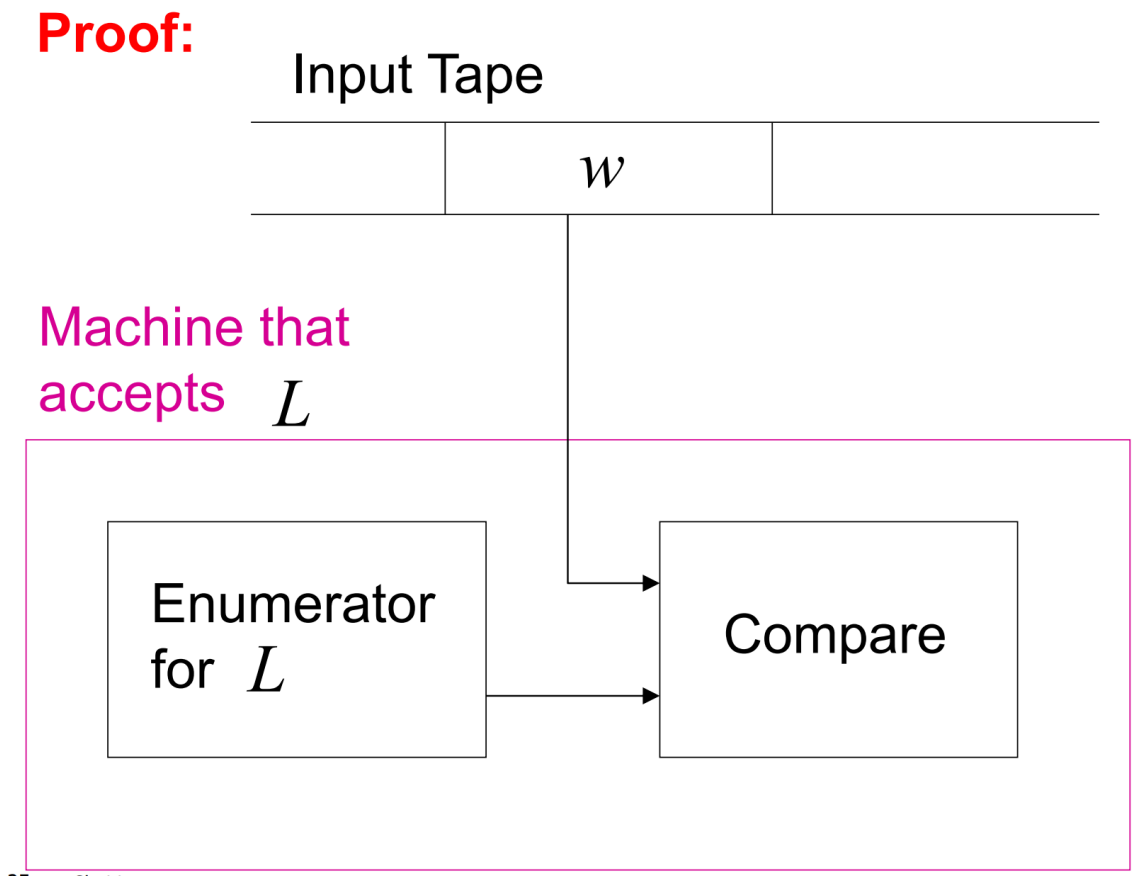
> 有点类似 JS 的异步处理.

假设 $\tilde{M}$ 生成字符串 $w_1, w_2, \ldots$

在 $\tilde{M}$ 生成第一个字符串 w_1 并传给 M 时，M 不运行整个 w_1，而是运行 w_1 的第一步，然后等待 $\tilde{M}$ 生成第二个字符串；在 $\tilde{M}$ 生成第二个字符串 w_2 时，M 不运行整个 w_2，而是运行 w_2 的第一步和 w_1 的第二步，然后等待 $\tilde{M}$ 生成第三个字符串...以此类推，任意一个字符串只要能在有限时间内被接受，就会在某个有限大的时刻被打印出来，不需要等待之前生成的可能导致 M 死循环的非 L 字符串处理完成.

4.3 枚举过程 $\Rightarrow$ 递归可枚举

If for language L there is an enumeration procedure then L is recursively enumerable.



Enumerator for L 是我们假设的语言 L 存在的一个枚举程序. 要证明 L 是递归可枚举的，我们需要构造一个图灵机 M，证明它可以接受 L.

该图灵机包含一个枚举程序和一个比较器，二者都可以由图灵机实现（可计算）. 若 $w \in L$，则对比到一致字符串时，接受 w；若 $w \notin L$，会一直枚举新的字符串来对比，进入死循环（拒绝）.

由 `4.2` 和 `4.3` 可知，A language is recursively enumerable if and only if there is an enumeration procedure for it

5. 非递归可枚举语言

We want to find a language that is not Recursively Enumerable. This language is not accepted by any Turing Machine

考虑字母表 $\{a\}$.

因为所有的图灵机集合是可数的. 可数集的子集也是可数集. Consider Turing Machines that accept languages over alphabet $\{a\}$. 它是图灵机集合的子集，因此是可数的，按顺序记为 $M_1, M_2, \ldots$

每个 $\{a\}$ 上的图灵机接受的语言都是 $\{a\}^*$ 的一个子集，即部分接受部分不接受. 接受的用 1 表示，不接受用 0 表示，可以作出下图：

	a^1	a^2	a^3	a^4	$\cdots$
$L(M_1)$	0	1	0	1	$\cdots$
$L(M_2)$	1	0	0	1	$\cdots$
$L(M_3)$	0	1	1	1	$\cdots$
$L(M_4)$	0	0	0	1	$\cdots$

这张图类似证明无限可数集的幂集是不可数的.

> 见 `Lec 10 8.3 不可数集`

这里 $\{a\}^*$ 是一个无限的可数集，而它所有子集构成的幂集即语言集合. Consider the language

$$L = \{a^i : a^i \in L(M_i)\}$$

即对角线元素构成的集合.

对它取补集，即

$$\overline{L} = \{a^i : a^i \notin L(M_i)\}$$

假设它是递归可枚举，则有某个 M_k 能够接受它. 但是对于这个 M_k，有

$a^k \notin L(M_k)$ 且 $a^k \in L$，或

$a^k \in L(M_k)$ 且 $a^k \notin L$，

矛盾.

> 本质上是 $L(M_k), a^k$ 这个位置的元素不能等于自己的取反，即这个新产生的语言 $\overline{L}$ 在整张图之外.

Therefore, the machine M_k cannot exist.

Therefore, the language $\overline{L}$ is not recursively enumerable.

6. 存在递归可枚举但非递归语言

即存在某些语言 L 能被图灵机接受，但接受它的图灵机对某些不属于 L 的输入会进入死循环（视为拒绝）.

仍然讨论该图：

	a^1	a^2	a^3	a^4	$\cdots$
$L(M_1)$	0	1	0	1	$\cdots$
$L(M_2)$	1	0	0	1	$\cdots$
$L(M_3)$	0	1	1	1	$\cdots$
$L(M_4)$	0	0	0	1	$\cdots$

语言 $L = \{a^i : a^i \in L(M_i)\}$ 是递归可枚举，但非递归的.

证明：

首先证明它是递归可枚举. 设计一个接受它的图灵机：

For any input string w

Compute i, for which $w = a^i$（使用枚举程序一个一个比较）

Simulate M_i on input a^i

If M_i accepts, then accept w

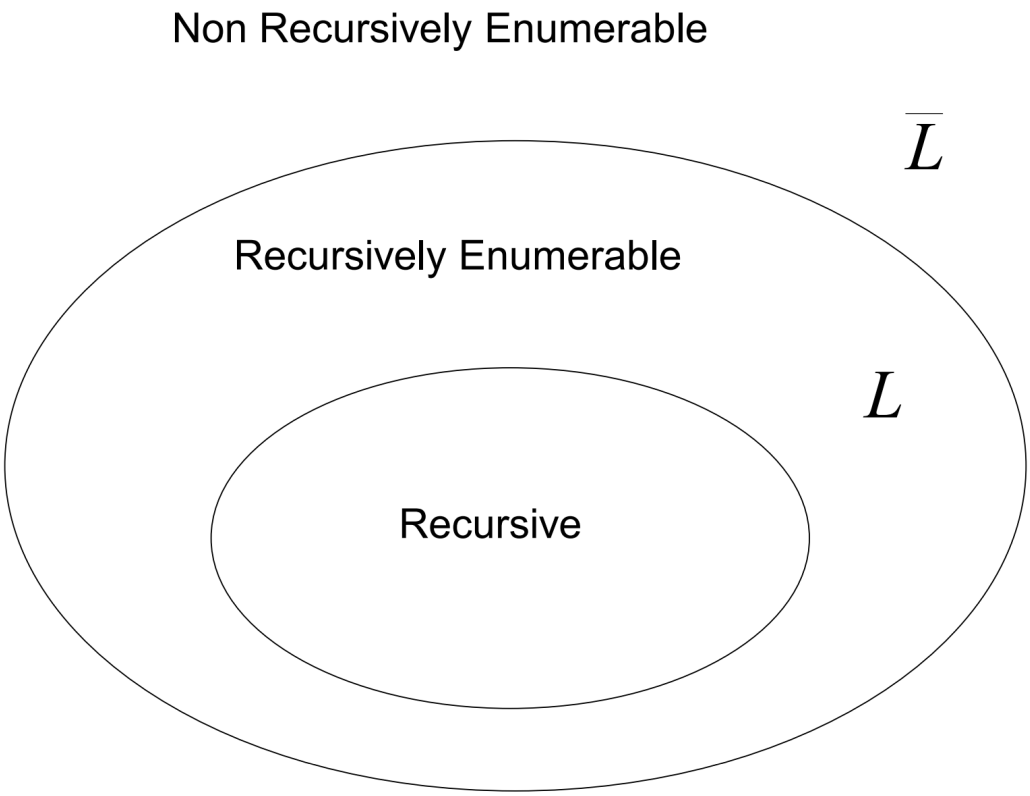
然后证明非递归：

我们在 `5. 非递归可枚举语言` 中已知 $\overline{L}$ 非递归可枚举语言，因此也不是递归语言.

假设 L 是递归语言，则 $\overline{L}$ 也是递归语言，矛盾.

> 若 L 是递归语言，则存在 M 接受 L，且对于不接受的字符串不会死循环. 则使用另一个 M'，把 M 的最终状态和非最终状态对调，即 M 接受的 M' 不接受，M 不接受的 M' 接受. 因为 M 不会死循环，M' 也不会死循环. M' 接受的语言就是 $\overline{L}$，即 $\overline{L}$ 是递归语言.

Therefore, L is recursively enumerable but not recursive.



定理：如果一个语言 L 及其补集 $\overline{L}$ 都是递归可枚举的，那么 L 和 $\overline{L}$ 都是递归的.

证明：构造一个 M'，同时模拟 M_L 和 $M_{\overline{L}}$ 并交替检查.

> 类似 `4.2.1 交替检查`，但细节有所不同.

对于输入 w，先运行 M_L 第一步，再运行 $M_{\overline{L}}$ 第一步；接着运行 M_L 第二步，再运行 $M_{\overline{L}}$ 第二步...以此类推，无论 w 是否属于 L，都能在有限时间里被 M_L 接受（判定为属于）或被 $M_{\overline{L}}$ 接受（判定为拒绝）. 因此 L 是递归的.

L 是递归的则 $\overline{L}$ 也是递归的，证明：

把 M' 的最终状态和非最终状态对调构造 M''，即 M' 接受的 M'' 不接受，M' 不接受的 M'' 接受. 因为 M' 不会死循环，M'' 也不会死循环. M'' 接受的语言就是 $\overline{L}$，即 $\overline{L}$ 是递归语言.

7. 无限制文法

> Unrestricted Grammars

产生式 $u \to v$ 的箭头两侧都是 String of varibles and terminals

例：

$$S \to aBc$$
$$aB \to cA$$
$$Ac \to d$$

定理：A language L is recursively enumerable if and only if L is generated by an unrestricted grammar

图灵机接受的语言都可以由无限制文法生成，反之亦然

8. 上下文敏感文法

> Context-Sensitive Grammars

产生式 $u \to v$ 的箭头两侧都是 String of varibles and terminals，且满足非压缩性（Noncontracting）：$|u| \leq |v|$

另一种写法：$\alpha A \beta \to \alpha \gamma \beta$

其中，

A 是一个变量，

α 和 β 是 String of varibles and terminals（可为空）

γ 是 String of varibles and terminals（不可为空）

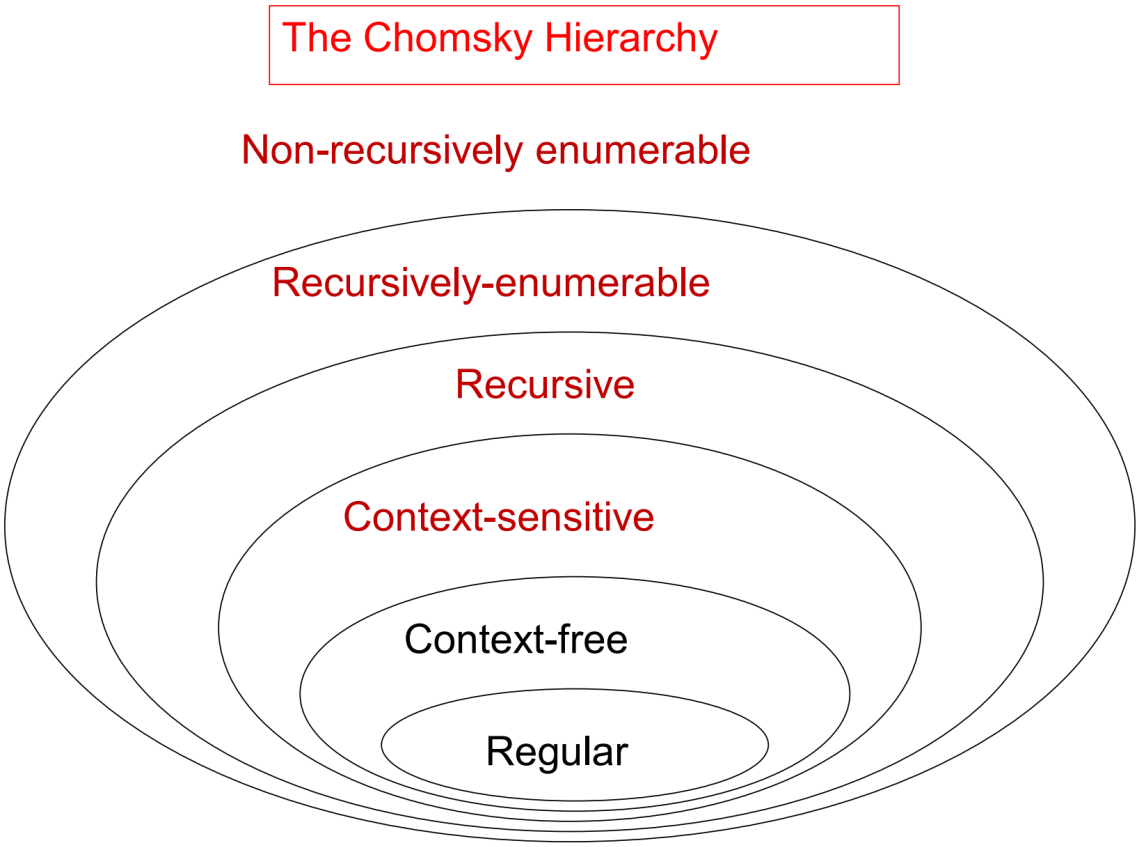
定理：A language L is context sensitive if and only if L is accepted by a Linear-Bounded automaton

> 证明不在本课范围.

注意：上下文敏感语言一定是递归语言，但递归语言不一定是上下文敏感语言.

因为上下文敏感语言被线性有界自动机接受，而线性有界自动机是一类特殊的图灵机. 由于它有界，能进入的不同格局总数是有限的（格局即状态、读写头所指的符号、读写头在整个纸带相对位置的组合）. 因此可以设计一个通用图灵机 T 来模拟线性有界自动机的操作，并确保 T 永远能停机：

在模拟 LBA 的基础上，维护一个计数器，或记录所有访问过的格局，如果重复进入同一格局，表明该 LBA 进入死循环，直接停机并拒绝. 因此，上下文敏感语言是递归语言. 而递归语言不一定是上下文敏感语言，因为不是所有图灵机都是线性有界自动机.



9. 可判定性

> Decidability

Consider problems with answer YES or NO

例:

Does Machine M have three states?

Is string w a binary number?

Does DFA M accept any input?

A problem is **decidable** if some Turing machine decides (solves) the problem

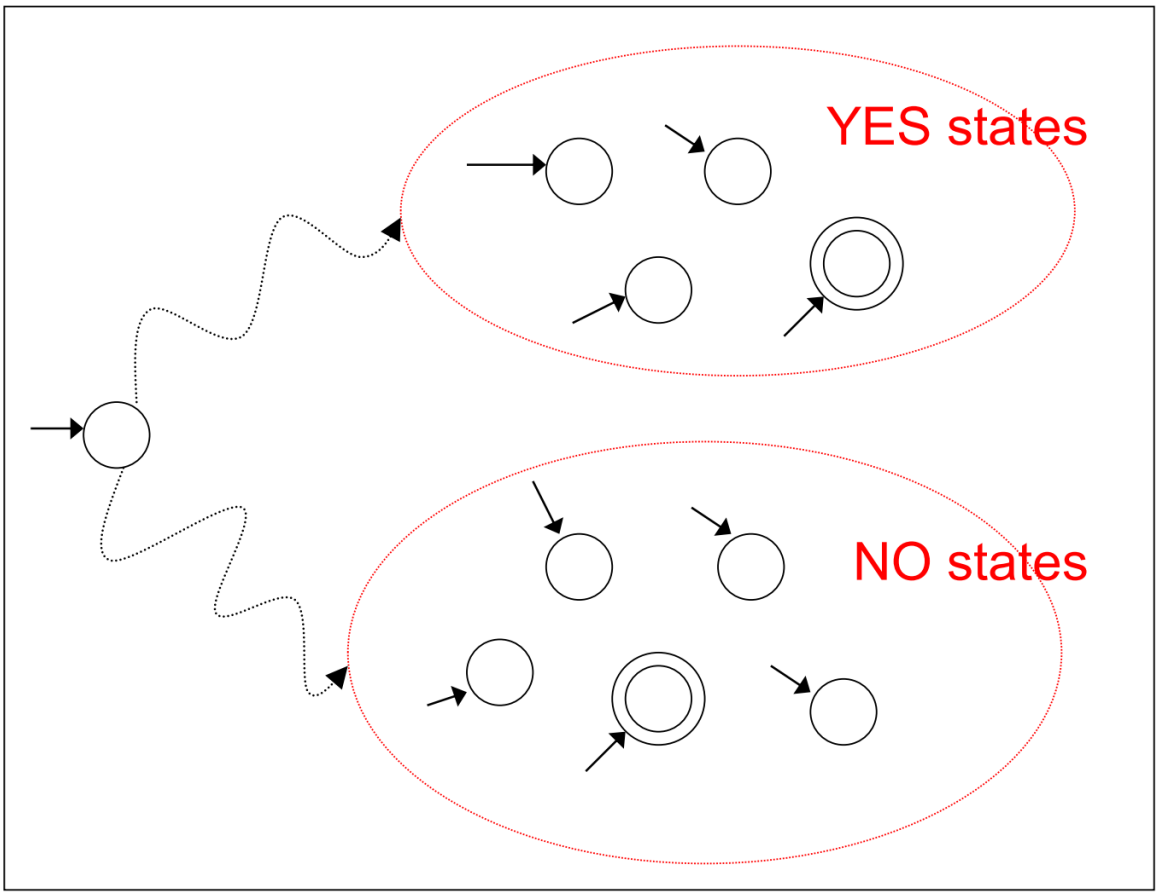
上面三个问题都是可判定问题.

The Turing machine that decides (solves) a problem answers YES or NO for each instance of the problem

The machine that decides (solves) a problem:

- If the answer is YES then halts in a yes state
- If the answer is NO then halts in a no state

These states may not be final states



注意：如果对于某个实例输入，图灵机会进入死循环，在有限时间内无法给出判定结果，则这个问题就被认为是不可判定的.

Difference between Recursive Languages and Decidable problems: for decidable problems, the YES states may not be final states

10. 不可判定问题

Some problems are undecidable: there is no Turing Machine that solves all instances of the problem

10.1 成员资格问题

A simple undecidable problem: The membership problem

Input:

- Turing Machine M
- String w

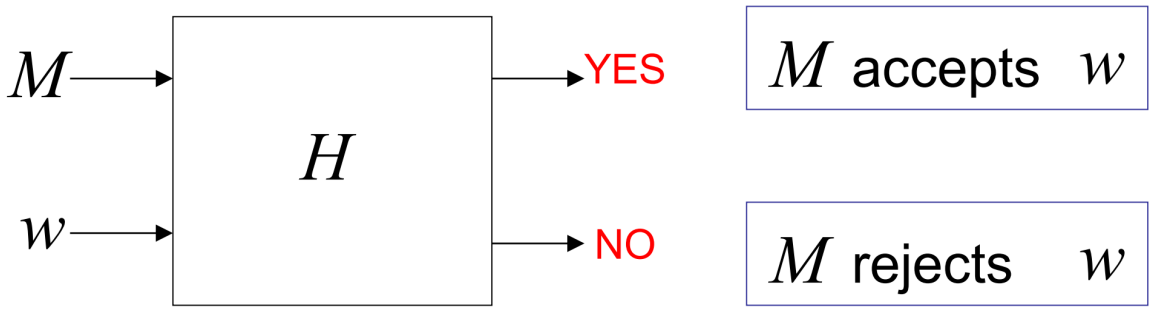
Question: Does M accept w? ($w \in L(M)$?)

结论：The membership problem is undecidable

证明：反证法

Assume for contradiction that the membership problem is decidable

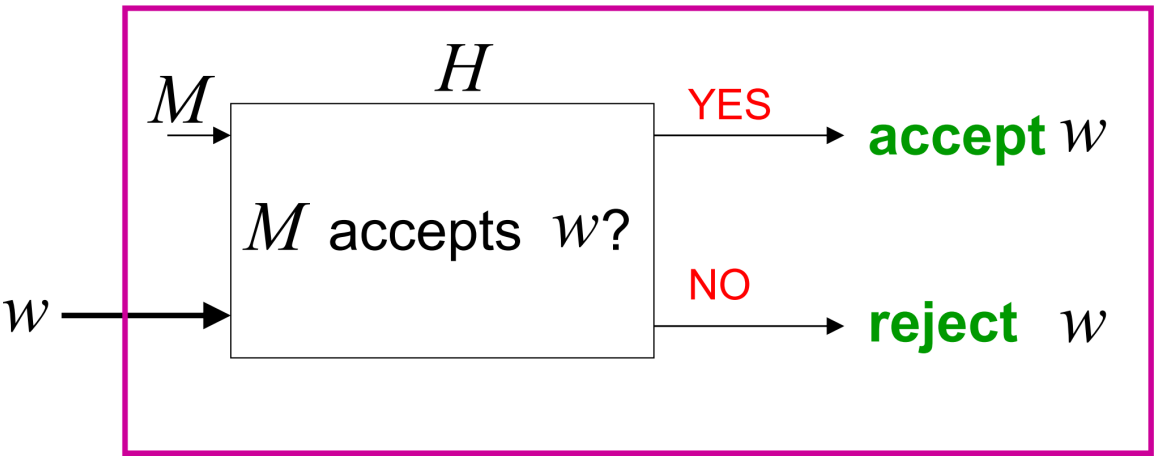
Thus, there exists a Turing Machine H that solves the membership problem



Let L be a recursively enumerable language

Let M be the Turing Machine that accepts L

We will prove that L is also recursive: we will describe a Turing machine that accepts L and halts on any input



因为 H 能够在有限时间内作出判定，则用它改装的图灵机也能在有限时间内接受或拒绝字符串（而不会死机）.

Therefore, L is recursive

然而，M 是任选的，它可能接受仅递归可枚举，但非递归的语言. 对于这类 M，我们的假设会导出矛盾的结论. 因此，无法保证成员资格问题是可判定的.

> 当然，如果已知输入的 M 接受的是递归语言（即不考虑会死循环的图灵机），则成员资格问题可判定.

10.2 停机问题

Input:

- Turing Machine M
- String w

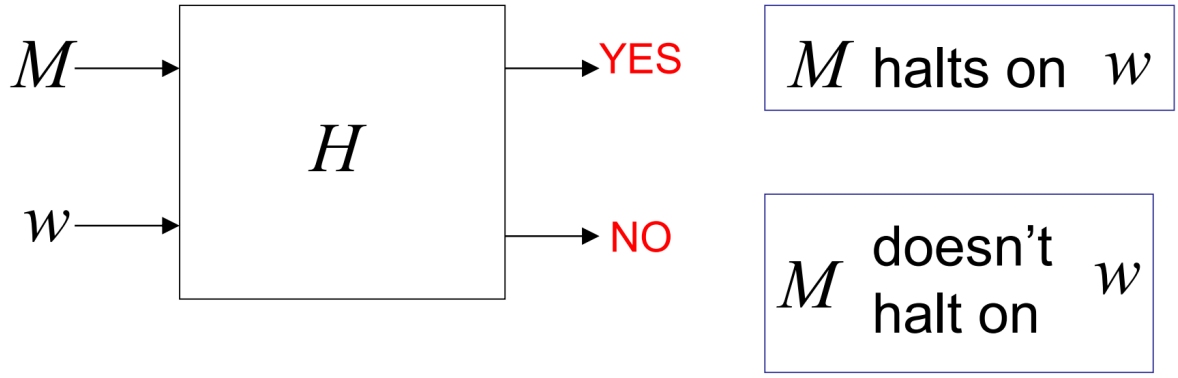
Question:

Does M halt on input w?

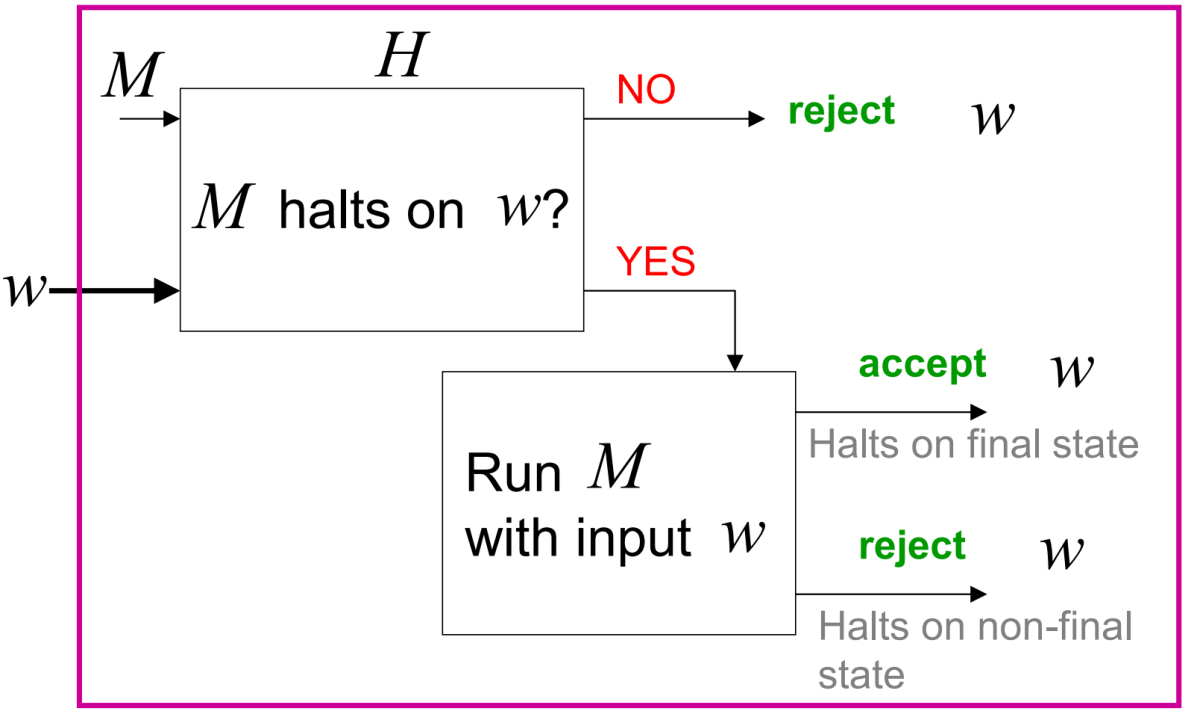
结论：The halting problem is undecidable

证明：假设 the halting problem is decidable

则存在图灵机 H 能够在有限时间内判定每个实例：

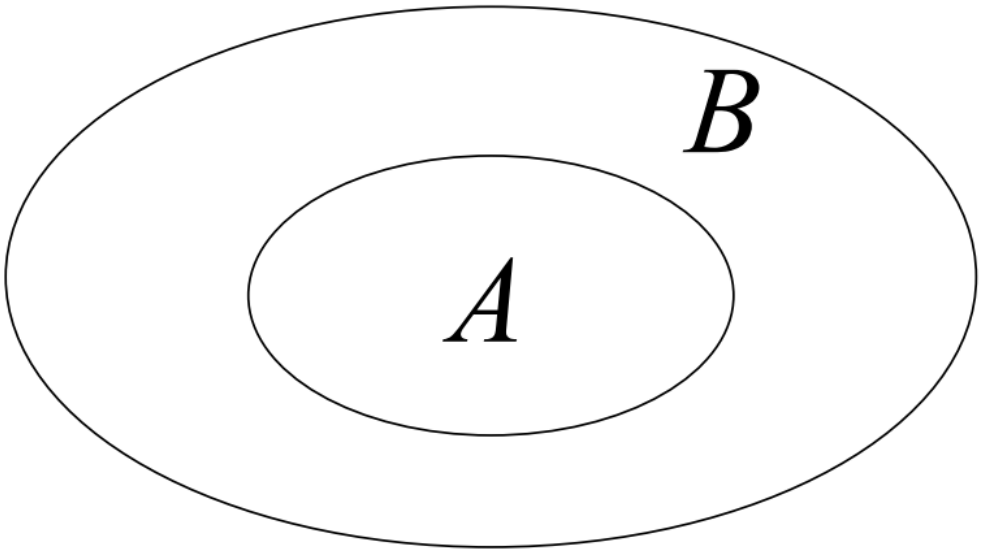


证明类似 `10.1`，基于 H 构造一个图灵机，它先判定 M 是否停机，若不停机直接拒绝，若知道会停机则传给 M 接受或拒绝. 这样，会导出所有递归可枚举语言都是递归语言，而这个结论是错误的.



11. 可归约性

> Reducibility



A 归约到 B：这里不是表示集合的包含关系，而是表示问题的难度关系. 即 A 的难度 $\leq B$ 的难度，如果 B 可判定，则 A 也可判定；如果 B 不可判定，则 A 可能可判定可能不可判定；如果 A 不可判定，则 B 也不可判定.

如果想证明一个问题是不可判定的，可以通过把一个已知的不可判定问题归约到这个问题（即证明一个更简单的问题是不可判定的，并且这个更简单的问题能够归约到我们需要探讨是否可判定的问题）

11.1 停机问题 $\leq$ 状态进入问题

the halting problem is reduced to the state-entry problem

状态进入问题

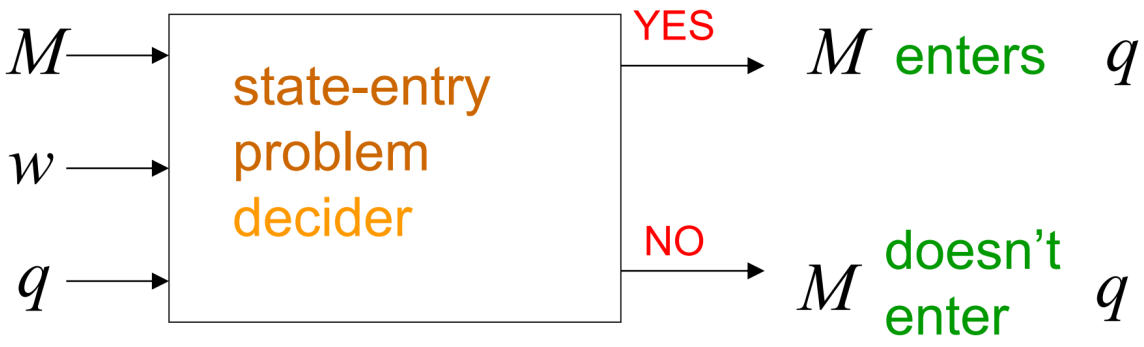
Input

- Turing Machine M
- State q
- String w

Question: Does M enter state q on input w?

结论：状态进入问题是不可判定的.

证明：假设可判定，则有一个 Decider：



可构造出一个停机问题的 Decider. 但我们知道停机问题是不可判定的，所以这样的 Decider 不存在.

11.2 停机问题 $\leq$ 空带停机问题

the halting problem is reduced to the blank-tape halting problem

Input: Turing Machine M

Question: Does M halt when started with a blank tape?

结论：空带停机问题是不可判定问题.

12. 不可计算函数

> Uncomputable Functions

A function is uncomputable if it cannot be computed for all of its domain

例：$f(n) =$ maximum number of moves until any 最终会停机的 Turing machine with n states halts when started with the blank tape

> Busy Beaver function，忙碌的海狸函数的变种

证明：假设 $f(n)$ 可计算，构造一个 Decider for 空带停机问题

Input: machine M

1. Count states of M: m
2. Compute $f(m)$
3. Simulate M for $f(m)$ steps starting with empty tape

If M halts then return YES

otherwise return NO（因为最终会停机的 M 必然在 $f(m)$ 步之内停机）

这样，空带停机问题可判定，这和我们之前的结论矛盾.

因此，$f(n)$ 不可计算.

期末考试

> 范围：Lec 6 - Lec 10

Homework

Hw 1

Hw 2

Hw 3

Hw 4

Q1

Find an npda on $\Sigma = \{a, b, c\}$ that accepts the language

$$L = \{w_1 c w_2 : w_1, w_2 \in \{a, b\}^*, w_1 \neq w_2^R\}$$

Solution:

$$
\begin{aligned}
M &= (Q, \Sigma, \Gamma, \delta, q_0, z, F) \\
Q &= \{q_0, q_1, q_2, q_f\} \\
\Sigma &= \{a, b, c\} \\
\Gamma &= \{a, b, z\} \\
F &= \{q_2, q_f\} \\
\delta(q_0, a, \lambda) &= \{(q_0, a)\} \\
\delta(q_0, b, \lambda) &= \{(q_0, b)\} \\
\delta(q_0, c, \lambda) &= \{(q_1, \lambda)\} \\
\delta(q_1, a, a) &= \{(q_1, \lambda)\} \\
\delta(q_1, b, b) &= \{(q_1, \lambda)\} \\
\delta(q_1, \lambda, a) &= \{(q_2, \lambda)\} \\
\delta(q_1, \lambda, b) &= \{(q_2, \lambda)\} \\
\delta(q_1, a, b) &= \{(q_f, \lambda)\} \\
\delta(q_1, b, a) &= \{(q_f, \lambda)\} \\
\delta(q_1, b, z) &= \{(q_f, z)\} \\
\delta(q_1, a, z) &= \{(q_f, z)\} \\
\delta(q_f, a, z) &= \{(q_f, z)\} \\
\delta(q_f, b, z) &= \{(q_f, z)\} \\
\delta(q_f, a, a) &= \{(q_f, \lambda)\} \\
\delta(q_f, b, b) &= \{(q_f, \lambda)\} \\
\delta(q_f, a, b) &= \{(q_f, \lambda)\} \\
\delta(q_f, b, a) &= \{(q_f, \lambda)\}
\end{aligned}
$$

Explanation:

If $w_1 = w_2^R$, the npda stay in q_1 and refuse it.

Once the palindrome structure is broken, it will goes to q_f.

If $|w_2| > |w_1|$, npda will go to q_f.

If $|w_2| < |w_1|$, and palindrome is not broken until finishing reading, it will go to q_2.

Q2

Construct a pda that accepts the language defined by the grammar

$$S \to aSbS|bSaS|\lambda$$

Solution:

$$
\begin{aligned}
M &= (Q, \Sigma, \Gamma, \delta, q_0, z, F) \\
Q &= \{q_0, q_1, q_2\} \\
\Sigma &= \{a, b\} \\
\Gamma &= \{a, b, S, z\} \\
F &= \{q_2\} \\
\delta(q_0, \lambda, \lambda) &= \{(q_1, S)\} \\
\delta(q_1, a, a) &= \{(q_1, \lambda)\} \\
\delta(q_1, b, b) &= \{(q_1, \lambda)\} \\
\delta(q_1, \lambda, S) &= \{(q_1, bSaS)\} \\
\delta(q_1, \lambda, S) &= \{(q_1, aSbS)\} \\
\delta(q_1, \lambda, S) &= \{(q_1, \lambda)\} \\
\delta(q_1, \lambda, z) &= \{(q_2, z)\}
\end{aligned}
$$

Q3

Show that the following language is deterministic

$$L = \{a^n b^m : m \leq n + 2\}$$

Solution:

Prove by constructing a dpda

$$
\begin{aligned}
M &= (Q, \Sigma, \Gamma, \delta, q_0, z, F) \\
Q &= \{q_0, q_1, q_2\} \\
\Sigma &= \{a, b\} \\
\Gamma &= \{A, z\} \\
F &= \{q_0, q_1, q_2\} \\
\delta(q_0, a, z) &= \{(q_1, AAAz)\} \\
\delta(q_0, b, z) &= \{(q_2, Az)\} \\
\delta(q_1, a, A) &= \{(q_1, AA)\} \\
\delta(q_1, b, A) &= \{(q_2, \lambda)\} \\
\delta(q_2, b, A) &= \{(q_2, \lambda)\}
\end{aligned}
$$

Explanation:

① If $n = 0$

$m = 0$: q_0 is final state, accept

$m = 1$: $(q_0, b, z) \vdash (q_2, \lambda, Az)$, accept

$m = 2$: $(q_0, bb, z) \vdash (q_2, b, Az) \vdash (q_2, \lambda, z)$, accept

$m \geq 3$: halt and refuse

② If $n \geq 1$

$m = 0$:

$$(q_0, \overbrace{a \dots a}^{n}, z) \vdash (q_1, \overbrace{a \dots a}^{n-1}, AAAz) \overset{*}{\vdash} (q_1, \lambda, \overbrace{A \dots A}^{n+2}z)$$

accept.

$1 \leq m \leq n+2$:

$$\begin{aligned}(q_0, \overbrace{a \dots a}^{n}\overbrace{b \dots b}^{m}, z) &\vdash (q_1, \overbrace{a \dots a}^{n-1}\overbrace{b \dots b}^{m}, AAAz) \\ &\overset{*}{\vdash} (q_1, \overbrace{b \dots b}^{m}, \overbrace{A \dots A}^{n+2}z) \\ &\vdash (q_2, \overbrace{b \dots b}^{m-1}, \overbrace{A \dots A}^{n+1}z) \\ &\overset{*}{\vdash} (q_2, \lambda, \overbrace{A \dots A}^{n-m+2}z)\end{aligned}$$

accept.

$m > n+2$: halt and refuse.

Q4

Show that the language is not context-free.

$L = \{w \in \{a, b, c\}^* : n_a(w) = n_b(w) \geq n_c(w)\}$

Prove: Using the Pumping Lemma for context-free languages

Assume L is context-free

Opposite: give m

Player: Pick $w = a^m c^m b^m$ with $w \in L$ and $|w| \geq m$

Opposite: give $w = uvxyz$ with $|vxy| \leq m$ and $|vy| \geq 1$

Player:

Case 1: vxy is within a^m $\underbrace{\overbrace{aaa}^{m}\overbrace{a \dots aaa}^{m}\overbrace{ccc \dots ccc}^{m}\overbrace{bbb \dots bbb}^{m}}_{u \quad vxy \quad z}$

Pick $i = 2$, $uv^2xy^2z = a^{m+|vy|}c^m b^m \notin L$ (because $m + |vy| \neq m$)

Contradiction!

Case 2: vxy is within c^m $\underbrace{\overbrace{aaa \dots aaa}^{m}\overbrace{ccc \dots ccc}^{m}\overbrace{bbb \dots bbb}^{m}}_{u \quad vxy \quad z}$

Pick $i = 2$, $uv^2xy^2z = a^m c^{m+|vy|}b^m \notin L$ (because $m + |vy| > m$)

Contradiction!

Case 3: vxy is within b^m $\overbrace{aaa...aaa}^{m}\ \overbrace{ccc...ccc}^{m}\ \overbrace{bbb...bbbbb}^{m}$ (u, vxy, z)

Pick $i = 2$, $uv^2xy^2z = a^m c^m b^{m+|vy|} \notin L$ (Similar to Case 1)

Contradiction!

Case 4: vxy overlaps a^m and c^m $\overbrace{aaa...aaa}^{m}\ \overbrace{ccc...ccc}^{m}\ \overbrace{bbb...bbb}^{m}$ (u, vxy, z)

Possibility 1: v contains only a, y contains only c

Pick $i = 2$, $uv^2xy^2z = a^{m+|v|}c^{m+|y|}b^m \notin L$

(Because $|v| + |y| \geq 1$, $|v|$ and $|y|$ can't be 0 at the same time. If $|v| \neq 0$ then $m + |y| \neq m$; If $|y| \neq 0$ then $m + |y| > m$.)

Contradiction!

Possibility 2: v contains a and c, y contains only c

Pick $i = 2$, $uv^2xy^2z = a^m c^{k_1} a^{k_2} c^{m+k} b^m \notin L$, where $v = \overbrace{a...a}^{k_2}\overbrace{c...c}^{k_1}$, $y = \overbrace{c...c}^{k}$

(Because $k_1 + k_2 = |v|$, $k_1 + k_2 + k \geq 1$. k_2 and $k_1 + k$ can't be 0 at the same time. If $k_2 \neq 0$, $n_a(w) = m + k_2 \neq n_b(w)$; If $k_1 + k \neq 0$, $n_c(w) = m + k_1 + k > n_b(w)$)

Contradiction!

Possibility 3: v contains only a, y contains a and c

Pick $i = 2$, $uv^2xy^2z = a^{m+k} c^{k_1} a^{k_2} c^m b^m \notin L$, where $v = \overbrace{a...a}^{k}$, $y = \overbrace{a...a}^{k_2}\overbrace{c...c}^{k_1}$ (Similar to Possibility 2)

Contradiction!

Case 5: vxy overlaps c^m and b^m

Similar to Case 4, there are 3 possibilities and all of them pick $i = 2$ will lead to contradiction.

There are no other cases to consider. In all cases we obtained a contradiction $\Rightarrow$ the original assumption is wrong.

Q5

Find an npda on $\Sigma = \{a, b\}$ that accepts the following language

$$L = \{a^n b^m : n \leq m + 1\}$$

Solution:

$$\begin{aligned}
M &= (Q, \Sigma, \Gamma, \delta, q_0, z, F) \\
Q &= \{q_0, q_1, q_2\} \\
\Sigma &= \{a, b\} \\
\Gamma &= \{A, Y, z\} \\
F &= \{q_2\} \\
\delta(q_0, a, z) &= \{(q_0, Yz)\} \\
\delta(q_0, a, Y) &= \{(q_0, A)\} \\
\delta(q_0, a, A) &= \{(q_0, AA)\} \\
\delta(q_0, a, z) &= \{(q_2, z)\} \\
\delta(q_0, \lambda, z) &= \{(q_2, z)\} \\
\delta(q_0, \lambda, \lambda) &= \{(q_1, \lambda)\} \\
\delta(q_1, b, A) &= \{(q_1, \lambda)\} \\
\delta(q_1, \lambda, z) &= \{(q_2, z)\} \\
\delta(q_2, b, z) &= \{(q_2, z)\}
\end{aligned}$$

Explanation: $n \leq m + 1 \Leftrightarrow m \geq n - 1$

① If $n = 0$, m can take any non-negative integer.

② If $n = 1$, m can take any non-negative integer.

Possible path: $(q_0, a\overbrace{b \dots b}^{m}, z) \vdash (q_2, \overbrace{b \dots b}^{m}, z) \overset{*}{\vdash} (q_2, \lambda, z)$

③ If $n \geq 2$, $m \geq n - 1$

Possible path:

$$\begin{aligned}
(q_0, \overbrace{a \dots a}^{n}\overbrace{b \dots b}^{m}, z) &\vdash (q_0, \overbrace{a \dots a}^{n-1}\overbrace{b \dots b}^{m}, Yz) \\
&\vdash (q_0, \overbrace{a \dots a}^{n-2}\overbrace{b \dots b}^{m}, Az) \\
&\overset{*}{\vdash} (q_0, \overbrace{b \dots b}^{m}, \overbrace{A \dots A}^{n-1}z) \\
&\vdash (q_1, \overbrace{b \dots b}^{m}, \overbrace{A \dots A}^{n-1}z) \\
&\vdash (q_1, \overbrace{b \dots b}^{m-1}, \overbrace{A \dots A}^{n-2}z) \\
&\overset{*}{\vdash} (q_1, \overbrace{b \dots b}^{m-n+1}, z) \\
&\vdash (q_2, \overbrace{b \dots b}^{m-n+1}, z) \\
&\overset{*}{\vdash} (q_2, \lambda, z)
\end{aligned}$$

(If $m < n - 1$, npda can't use $\delta(q_1, \lambda, z) = \{(q_2, z)\}$ because there are still a few "A" on top of stack)

Hw 5

Q1

Consider a variant of a Turing machine in which the read–write head may move several cells left or right in a single transition. Formally define such a machine, and explain how it can be simulated by a standard Turing machine that moves its head only one cell per step.

Q2

Give a formal definition of a two-tape Turing machine; then write programs (Transition Graph or Transition Function) that accept the languages $L = \{ww^R : w \in \{a, b\}^+\}$. Assume that $\Sigma = \{a, b\}$ and that the input is initially all on tape 1.

Q3

Construct a Turing machine (Transition Graph or Transition Function) to compute the function:

$$f(w) = w^R$$

where $w \in \{0, 1\}^+$

Q4

Let Σ be a finite alphabet.

i) Prove that Σ^* is countable.

ii) Prove that the set of all languages over Σ, i.e., $P(\Sigma^*)$, is uncountable.

iii) Conclude that some languages connot be recognized by any Turing machine.

Appendix

1. 停机问题

CS 著名的不可判定问题，由 Turing 首次提出.

问题：给定一个程序 P 和输入 x，能否写一个 “万能程序” H，来判断

- 程序 P 在输入 x 上最终会停机（结束运行），
- 还是会一直运行下去（死循环）.

结论：停机问题是不可判定的.

> 不存在这样的万能程序 H，可以对所有程序和输入都给出正确答案.

证明：

> 待补充.

2. 随机游走

> 见 `2.3 句型 & 句子 例 3`.

小结论：从 0 出发经过若干步回到 0，每步随机前进（$+1$）或后退（-1）. 若已知第一步和最后一步相同（同为前进或后退），则从第一步走完到最后一步走出之前，一定至少经过一次 0.