理论计算机科学导引

TCS - 毛宇尘老师班

shrike505

目录

泱泱猩热血，/汩若风袭泉，/起落樱唇际，/往来兰气间。

> Reference:
>
> - introtcs.org - 教材（电子书版）
> - https://fla.cuijiacai.com/ - 南京大学形式语言与自动机课程笔记

一、问题与编码

> 一言以蔽之，它（TCS）研究的是问题的上界与下界。

需要界定计算所需要解决的问题，以及计算所需要的设备（模型）。

这一节先规定前者。回顾一些经典的算法或数学上的问题：给定带权重的图 G，求其中的最短路/其的最小生成树；提供矩阵 A, B，求其乘积 AB。这些问题都可以看作一个函数：给定输入，求输出。

与程序设计中的函数强调 implementation（即 How to compute the answer）相比，这里的函数更多具有数学意义，强调 specification（即 What should the answer be）。

接下来聚焦这些函数的输入，计算机无法理解图、矩阵这些概念，只能理解二进制串（binary string），也就是一串又一串的 0 和 1——于是要通过某些编码方式将这些元素编码为 01 串。先定义一个字符表（Alphabet）：$\Sigma = \{0, 1\}$，于是长度为 n 的二进制串的集合可表示为 $\Sigma^n = \Sigma \times \Sigma \times \ldots \times \Sigma = \{(a_1, a_2, \ldots, a_n) \mid a_i \in \Sigma\}$

特别规定 Σ^0 是长度为 0 的串的集合，这个串用 e 表示，即 $\Sigma^0 = \{e\}$

$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$ 即为所有长度的二进制串集合。

> **前缀（Prefix）**
>
> $x = a_1 a_2 \ldots a_n, y = b_1 b_2 \ldots b_n$ 的拼接（Concatenation）为 $xy = a_1 a_2 \ldots a_n b_1 b_2 \ldots b_n$
> x 是 y 的一个前缀（Prefix），当对于某些 $z \in \Sigma^*, y = xz$
> 类似的可以定义后缀（Suffix），不再赘述

可以将 Σ 中的 0 和 1 换成任意字符，例如 26 字母，方框三角圆，以此组建你自己的 Alphabet！

1.1 编码（Encoding）

有了最基础的元素（字符），将图、矩阵、等等等等计算函数的输入转化为字符串的过程，称为编码，即一个映射 $E : A \to \{0, 1\}^*$。

> **编码性质**
>
> 显然，这个映射需要是单射（injective，在下文中会频繁表示为 one-to-one），即不同元素的映射结果（得到的字符串）必须是不同的。

例子

- 自然数 $n \in N$（parity(n) 是 n 对 2 取余的结果）：$NtS(n) = \begin{cases} 0 & \text{if } n{=}0 \\ 1 & \text{if } n{=}1 \\ NtS(\lfloor \frac{n}{2} \rfloor)\ \text{parity(n)} & \text{if } n{>}1 \end{cases}$
  - 亦即 n 的二进制表示
- 自然数对 $(a, b) \in N \times N$，自然的想法是 a 的编码拼接 b 的编码，但是会出现编码重复，并不是单射
  - 对于 1110，可以解释为 $(1, 6)$ 和 $(3, 2)$，这实质上是因为在计算机读取完前两个 1 时，并不知道它代表 3 还是一个其他数的前缀

1.1.1 prefix-free 编码

在第二个例子的教训下，我们需要找到的编码映射是 prefix-free 的，即对于任何的 $x \neq x'$，$E(x)$ 都不是 $E(x')$ 的前缀。

接下来 myc 老师突然就这个 prefix-free 证明了两个*寻找另一种编码*的引理，感觉很突兀。

引理 1.1.1

假设已经存在一个 prefix-free 的编码 $E : A \to \{0, 1\}^*$，那么对于编码 $\bar{E} : A^* \to \{0, 1\}^*$（$A^* = \bigcup_{n \geq 0} A^n$，我理解为一个*由任意长待映射元素*（$a_i \in A$）*序列组成的集合*，接下来要找到对这些元素序列的编码），命 $\bar{E}(a_1 a_2 ... a_n) = \begin{cases} E(a_1)E(a_2)...E(a_n) & \text{if } n{\geq}1 \\ e & \text{if } n{=}0 \end{cases}$，那么 $\bar{E}$ 是 one-to-one 的。

证明. 假设存在 $(a_1, a_2, ..., a_n) \neq (b_1, b_2, ..., b_m)$，使得 $\bar{E}(a_1 a_2 ... a_n) = \bar{E}(b_1 b_2 ... b_m)$，那么 $E(a_1)E(a_2)...E(a_n) = E(b_1)E(b_2)...E(b_m)$，且 $\exists i, s.t. \forall j < i, a_j = b_j$，且 $a_i \neq b_i$（即在第 i 个字符前两个元素序列的每个元素都相同）

那么 $E(a_1)E(a_2)...E(a_{i-1}) = E(b_1)E(b_2)...E(b_{i-1})$，则 $E(a_i)...E(a_n) = E(b_i)...E(b_m)$，那么对于 $E(a_i)$ 和 $E(b_i)$，要么前者是后者的前缀，要么后者是前者的前缀，又考虑到 $a_i \neq b_i$，则 E 不是 prefix-free 的，这与题设冲突。 □

引理 1.1.2

如果存在 one-to-one 的 $E : A \to \{0, 1\}^*$，那么存在 prefix-free 的 $E' : A \to \{0, 1\}^*$，且使得 $|E'(a)| \leq 2|E(a)| + 2, \forall a \in A$

证明. 将原编码中的 0 映射为 00，1 映射为 11，该元素再次编码结束后再添加一个 01，例如对于 $E(a) = 010$，$E'(a) = 00110001$

这种编码显然有性质 0：01 不会出现在任何编码的奇数-偶数位置，即 $\forall k \in N, E'(a_{2k+1})E'(a_{2k+2}) \neq 01$

试证 prefix-free 性：假设 $E'(a)$ 是 $E'(b)$ 的前缀，由于 01 标识了 $E'(a)$ 和 $E'(b)$ 的结束，且由于 性质 0，很明显有 $E'(a) = E'(b)$，那么 $E(a) = E(b)$，且 E 是单射，则 $a = b$，于是 prefix-free 得证。 □

结合两个引理可以得到一个结论：

定理

如果存在 one-to-one 的 $E : A \to \{0,1\}^*$，那么便存在 one-to-one 的 $E' : A^* \to \{0,1\}^*$

这是很重要的，对于数学元素，如果我们可以给数字做编码，那么就可以编码向量，再运用一次定理就能编码矩阵，然后是更高维的张量。

1.1.2 编码与可数的关系

下面四条等价

1. A 是可数的
2. A 是有限的，要么存在一个双射 $f : A \to N$
3. 存在单射 $g : A \to N$
4. 存在满射 $h : N \to A$

引理 1.1.3

$\{0,1\}^*$ 是可数的

证明. 对 $\{0,1\}^*$ 的元素进行这样的排序：e,0,1,00,01,10,11···

即先按长度排序，内部再按二进制数大小排序。

那么定义从字符串映射到排序后序号的函数 $f : \{0,1\}^* \to N$： $\forall x \in \{0,1\}^*, f(x) = \begin{cases} 0 & \text{if } x=e \\ 2^{|x|}+(x\text{对应的二进制数大小}) & \text{if } |x| \geq 1 \end{cases}$

可理解为组的序号+组内的序号，这是一个单射，于是可数。 □

这个引理也引导出一个定理，即可数性和单射编码的关系：

定理

A 是可数的当且仅当存在单射 $E : A \to \{0,1\}^*$

编码的设定完备后，我们面临的问题（Problem）就抽象为了一个从二进制串到二进制串的函数了（即对输入和输出都做编码）

二、计算模型

2.1 布尔电路（Boolean Circuit）

可以可以，问题（Problem）的输入输出已经被我们编码，可以投诸于计算了——那么，计算的具体步骤，或者说方法，是什么呢？

> 这一节里探讨的一类问题/函数统称为有限函数（Finite Function），其输入输出均为固定长度的一个二进制串，即 $f : \{0,1\}^n \to \{0,1\}^m$

一个很经典的有限函数计算模型即布尔电路（Boolean Circuit，我去，计算机系统 1），显然对于与门（AND）和或门（OR）而言，$n = 2, m = 1$，非门的 n 为 1，别的门电路可类比；另一个例子是 MAJ 函数，即对于 $n = 3$ 的输入的每一位，如果 1 占多数，则输出 1，否则输出 0

一个『电路』还是太具体了，不利于更本质的计算理解——将一个布尔电路 C 抽象为一个有向无环图 G，它包含如下节点（Nodes）:

- n input nodes：记为 $X[0], X[1], ..., X[n-1]$，均没有入度且出度至少为 1
- s gates：即逻辑门，根据种类有不同的度
  - 一个电路 C 的 size 定义为 $|C| = s$
- m output nodes：记为 $Y[0], Y[1], ..., Y[1]$

舒服了，可以利用门电路模拟从输入到输出的计算过程了：对于输入长为 n 的 $x \in \{0,1\}^n$，令其第 i 位即为 input nodes 中的 $X[i-1]$，输出答案 $y \in \{0,1\}^m$ 同理。

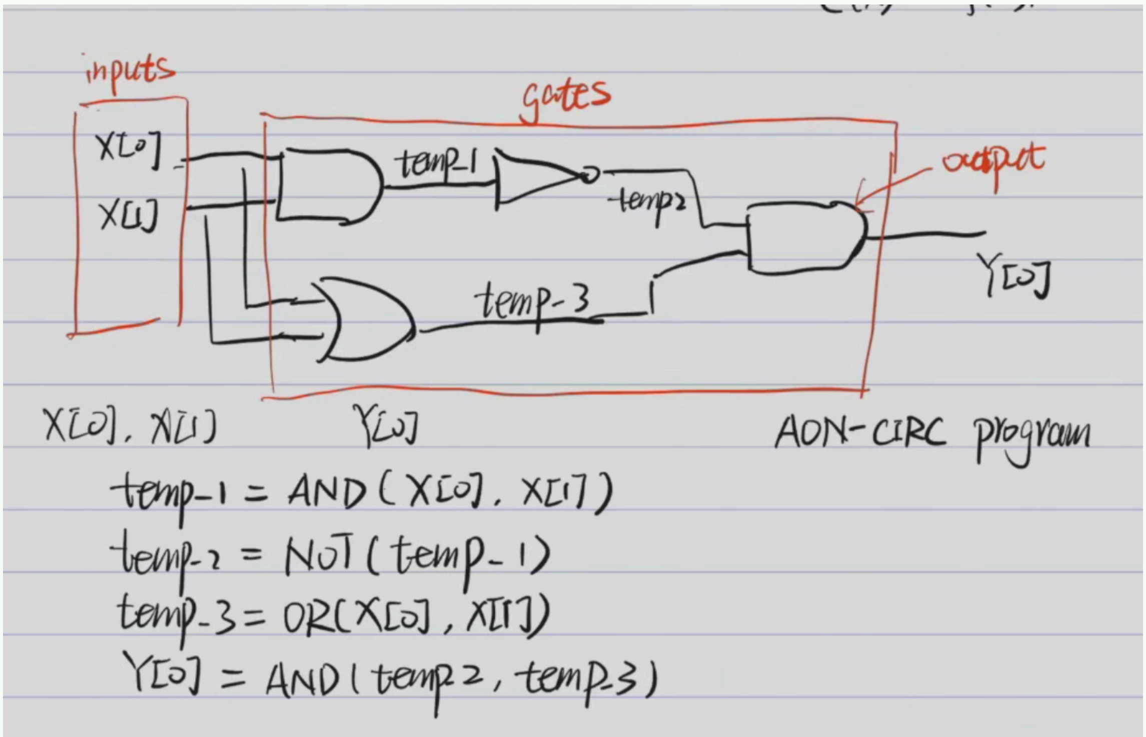
定义 2.1.1

记 $C(X) = (Y[0], Y[1], ..., Y[m-1])$，如果 $\forall x \in \{0,1\}^n, C(x) = f(x)$, 则称电路 C 计算了函数 f.

♣

现在还亟待一种对计算过程的书面化描述, 即, 这个电路的每一步, 信号(或者说数据)流过每一个门时得到了什么中间结果? 于是 ~~Anonjac~~ ~~Anon-Circ~~ AON-Circ（AND-OR-NOT Circuit）Program 登场了。

定义 2.1.2 (AND-OR-NOT Circuit Program)



对于图中的电路，将其每个中间逻辑门的输出保存为一个 temp 变量，便得到了下方的多行（Lines），这就是 AON-Circ Program

不难发现每一行对应一个逻辑门的计算过程，于是 Program 的行数与其对应电路中逻辑门的个数相同。此时 Program 的行数即为电路的 size

AON-C Program 计算了某个函数的定义与上方电路 C 计算函数类似，不再赘述。♣

实际上程序和电路这种对应关系就是等价的。

定理 2.1.3

一个函数可被一个有 s 个逻辑门的布尔电路计算，当且仅当它可以被 s 行的 AON-C Program 计算。♡

2.1.1 NAND 电路（NAND Circuit）

接下来介绍一种门：NAND，即 NOT(AND)，易知 AON 三者都可以只用 NAND 实现，于是可以搭建一个仅由 NAND 门构成的电路。

NAND Circuit	$\Leftrightarrow$	AON Circuit
s gates	$\rightarrow$	$\leq 2s$ gates, for NAND decomposes to NOT(AND)
$\leq 3s$ gates,for OR(a,b) $\equiv$ NAND(NAND(a,a),NAND(b,b))	$\leftarrow$	s gates

相类似的，有 NAND-Circ Program，其每一行都形如 `foo = NAND(bar,blah)`。

定理 2.1.4

Boolean Circuit, AON-Circ Program, NAND-Circ Program,NAND Circuit 四者的转化只需要 s 的常数倍（即 $\Theta(s)$）

定理 2.1.5

$\forall n, m, f : \{0,1\}^n \to \{0,1\}^m$，都存在一个布尔电路计算这个函数，其含有 $O(m \cdot n \cdot 2^n)$ 个逻辑门。

证明. 对于输出的某一位 $Y[j]$，其计算情况可枚举如下表：

$X[0]$	$\cdots$	$X[n-1]$	$Y[j]$
0	$\cdots$	0	some value
0	$\cdots$	1	some value
$\cdots$	$\cdots$	$\cdots$	$\cdots$
1	$\cdots$	1	some value

写出 $Y[j]$ 的具体计算式, 用析取范式表示: $Y[j] = (... \wedge ... \wedge ...) \vee (... \wedge ... \wedge ...) \vee ... \vee (... \wedge ... \wedge ...)$，其中共有 2^n 个括号，对应上表中的 2^n 行，每一行通过合取计算出一种情况下的 $Y[j]$，再析取得到 $Y[j]$ 的具体表达；每个括号中共有 n 项，对应 $X[0]$ 到 $X[n-1]$ 本身或取反再进行合取，于是得到 $n \cdot 2^n$

而 Y 的长度为 m，于是得到 $O(m \cdot n \cdot 2^n)$ □

满足上述定理，即可以计算任意函数的电路，称为*通用*（universal）的；要判断一个函数集合（化成的电路）是否是通用的，只需要判断其是否能计算 NAND。

2.2 计算规模

可以可以，已经可以用电路/程序计算某个函数/问题了，那么对于某个函数，我们需要多少个门的电路, 多少行的程序, 这是可以估量的吗? 由上面的定理似乎已经有了上界: $O(m \cdot n \cdot 2^n)$ (myc 剧透：其实可以把数量打到 $O(\frac{m \cdot 2^n}{n})$)。我们来从 NAND 电路入手。

ADD Function

试设计电路程序，计算 $\mathrm{ADD}:\{0,1\}^{2n}\to\{0,1\}^{n+1}$，$\mathrm{ADD}(x_0,...,x_{2n-1})=x_0...x_{n-1}+x_n...x_{2n-1}$

解.

```
def ADD(X[0], ..., X[2n-1]):
  Result = [0] * (n+1)
  Carry = [0] * (n+1)
  for i in range(n):
    Result[i] = XOR(Carry[i], XOR(X[i], X[i+n]))
    Carry[i+1] = MAJ(Carry[i], X[i], X[i+n])
  Result[n] = Carry[n]
  return Result
```

XOR 和 MAJ 函数只需要常数行的 NAND-Circ Program 实现，经过 n 次循环，于是该加法函数的规模（行数）即为 $O(n)$

这是很好的，计算规模与输入串的长度成正比。

MUL Function

乘法基于加法，摆了。

规模可以不断优化：$O(n^2)\to O\left(n^{\log_2 3}\right)\to$ even better.

LOOKUP Function

试设计电路程序，计算 $\mathrm{LOOKUP}:\{0,1\}^{2^k+k}\to\{0,1\}$，具体而言，输入分为两段：$2^k$ 位作为表，k 位视作一个最大可表达 2^k-1 的二进制数 i，输出为表中的第 i 位。

解. 利用归纳思想，$k=1$ 时：

```
def LOOKUP_1(X[0],X[1],i[0]):
if i[0] == 0:
  Y[0] = X[0]
else:
  Y[0] = X[1]
```

对于 k 时的情形

```
def LOOKUP_k(X[0],X[1],...,X[2^k-1],i[0],...,i[k-1]):
if i[0] == 0: # 根据剩下的 i 查 X 的前半段
  Y[0] = LOOKUP_(k-1)(X[0],...,X[2^(k-1)-1],i[1],...,i[k-1])
else:
  Y[0] = LOOKUP_(k-1)(X[2^(k-1)],...,X[2^k-1],i[1],...,i[k-1])
```

于是有规模 $\begin{cases}L(k)=C+2L(k-1)\\L(1)=O(1)\end{cases}$，解得 $O(2^k)$

2.2.1 优化效率

现在来探讨如何将某函数（n input, m output）计算的规模从 $O(m\cdot n\cdot 2^n)$ 优化至 $O(\frac{m\cdot 2^n}{n})$。

我们用上面介绍过的 LOOKUP 函数来理解，对于如下函数计算的过程真值表：

$X[0]$	$\cdots$	$X[n-1]$	$Y[j]$
0	$\cdots$	0	some value
0	$\cdots$	1	some value
$\cdots$	$\cdots$	$\cdots$	$\cdots$
1	$\cdots$	1	some value

进行计算时可以看作是将 $X[0]-X[n-1]$ 作为输入的二进制数索引 i，原函数输出 Y 的某一位 $Y[j]$ 在所有情况的 X 下的输出列作为输入的表（有 2^n 个取值可能性，即上表中的 2^n 行），从而计算 $Y[j]$ 确定取得的值）

例如对如下函数 g 的真值表：

$X[0]$	$X[1]$	$Y[0]$
0	0	1
0	1	0
1	0	1
1	1	1

命 $G_0=1, G_1=0, G_2=1, G_3=1$（即函数输出的 2^2 中情况），那么 $g(X[0],X[1]) = \text{LOOKUP}_2(G_0,G_1,G_2,G_3,X[0],X[1])$

接下来分析这种方法的效率：对于抽出输出的 2^n 种情况，每种 0 或 1 利用 ONE/ZERO 函数便可用常数行实现，于是共有 $O(2^n)$ 行；而 LOOKUP 函数的规模也为 $O(2^n)$，于是计算 $Y[j]$ 的规模为 $O(2^n)$；而 Y 有 m 位，于是总规模优化为 $O(m\cdot 2^n)$

> ⚠ Incomplete Section
>
> 接下来是如何将规模进一步优化到 $O(\frac{m\cdot 2^n}{n})$，但我没听咋懂 myc 的讲解，于是 skip 了

2.2.2 编码程序

现在考虑将 Circ-Program 编码为字符串形式：就一个有 s 行的 NAND-Circ Program 而言，其每一行都是 `foo = NAND(bar,blah)` 的形式，因此只需要考虑编码其中所有的变量（易知涉及的变量个数不超过 $3s$ 个），且每一行都可以看作是一个三元组（triple）：`(foo, bar, blah)`，于是共得到 s 个三元组。

（以 $3s$ 个变量为例）将这些变量记作 $0,1,2,3,...,n-1,n,...,n+m-1,...,3s-1$，其中前 n 个为对应函数的 input 变量（$X[0],X[1],...,X[n-1]$），中间 m 个为 output 变量（$Y[0],Y[1],...,Y[m-1]$，其余为中间变量。

于是每一个变量都可以编码为一个长度为 $\lceil\log(3s)\rceil$（hey where does this come from）的二进制串，只需要按照程序顺序拼接每个三元组中变量的编码，便得到了一个长度为 $3s\cdot\lceil\log(3s)\rceil$ 的二进制串。

编码完成了——但是，为什么要做这件事，难道说？没错，有了字符串编码的程序，我们甚至能将这个程序作为另一个程序的输入💀，例如下面这个例子。

EVAL Function

函数 $\mathrm{EVAL}_{s,m,n}:\{0,1\}^{3s\lceil\log(3s)\rceil+n}\to\{0,1\}^m$ 是对一个 NAND-Circ Program p（含 s 行，计算 $f:\{0,1\}^n\to\{0,1\}^m$）和一个输入 $x\in\{0,1\}^n$ 的计算（或者说，运行），其输入的前 $3s\cdot\lceil\log(3s)\rceil$ 位为 p 的编码，后 n 位为 input x，输出为 $f(x)$.

定理 2.2.1 (EVAL 的规模)

$\forall s,n,m,\exists$ NAND-Circ Program $U_{s,n,m}$ 计算函数 $\mathrm{EVAL}_{s,m,n}$，且其规模为 $O(s^2\cdot\log s)$

2.2.3 更长的输入：Infinite!

更一般的计算方法无疑是针对任意长度输入的函数 $f:\{0,1\}^*\to\{0,1\}^n$，然而这是无法利用已经讲述过的布尔电路/程序来计算的（回顾：我们的方法是，使用一个真值表，记录每组输入 $X[0]-X[n-1]$ 下 $Y[m]$ 的取值）

定理 2.2.2

For every $F:\{0,1\}^*\to\{0,1\}^*$, $BF=\begin{cases}F(x)_i & \text{if } i<|F(x)|,b=0\\ 1 & \text{if } i<|F(x)|,b=1\\ 0 & \text{if } i\geq|F(x)|\end{cases}$

使用 BF 计算 F 只需要遍历 0 到 $|F(x)|-1$ 即可，反过来只需要取每一位

三、语言

3.1 DFA 与正则语言

显然一个布尔函数 $f:\{0,1\}^*\to\{0,1\}$ 和这样一个集合对应：$A_f=\{x\in\{0,1\}^*\mid f(x)=1\}$，这个集合称为语言（Language）；此时显然有 $f(x)=1$ 当且仅当 $x\in A_f$

于是计算函数的值 $f(x)=?$ 与判断 $x\in?A_f$ 是等价的。

于是来研究语言的性质（自己觉得突兀不（流汗

考虑累加 x 的每一位再模二的 XOR 算法，可得到以下流程图：

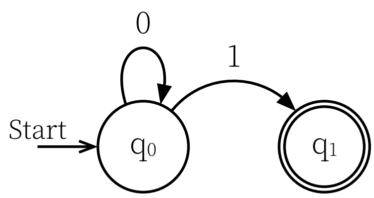


图 1 假设 q_1 为末态

定义 3.1.1

这被称作一个确定性有限自动机（Deterministic Finite Automaton, DFA），它可以看作一个四元组 (K, s, F, δ)，其中:

- K - 一组状态的有限集合
- $s \in K$ - 初始状态
- $F \subseteq K$ - 末态（接受状态）的集合
- $\delta : K \times \{0, 1\} \to K$ - 状态转移函数

假设有输入字符串 $x_0x_1...x_{n-1}$，DFA 的计算过程为：$s_0 = s, s_1 = \delta(s_0, x_0), s_2 = \delta(s_1, x_1), ..., s_n = \delta(s_{n-1}, x_{n-1})$, 然后判断 $s_n \in F$, 如果是则接受 (accept), 否则拒绝 (reject)。

定义 DFA 计算某个函数的方法：M computes f if M accepts x 当且仅当 $f_x = 1$

同时定义 DFA Decides 某个语言的方法：M decides A_f if M accepts x 当且仅当 $x \in A_f$

被 DFA 决定的语言称为正则语言（Regular Language）。

定理 3.1.2

某台 DFA M 可判断的语言有且仅有一个，即 $L(M) = \{x \in \{0, 1\}^* \mid M \text{ accepts } x\}$

练习. 证明空集，$\{0, 1\}^*$，$\{e\}$, $\{w \in \{0, 1\}^* \mid w$有一个字串是 101（何意味)$\}$ 是正则语言

解.

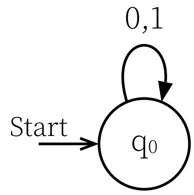


图 2　空集，使可接受集为空即可

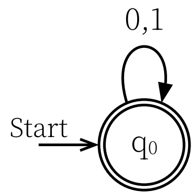


图 3　$\{0, 1\}^*$，使所有状态均为可接受状态即可

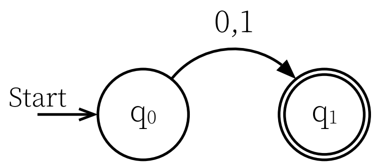


图 4　$\{e\}$，使初始状态为唯一可接受状态即可

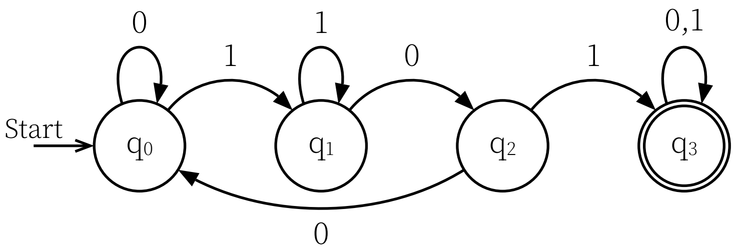


图 5 $\{w \in \{0,1\}^* \mid w$有一个字串是 $101\}$，设计状态转移使得读到连续的 101 时进入可接受状态即可

定理 3.1.3

如果 A 和 B 都是正则的，那么 $A \cup B$ 是正则的。(对新 DFA 的每一个成员进行探究即可)

3.2 NFA

定义 3.2.1

不确定性有限自动机（Nondeterministic Finite Automaton, NFA），如下是其与 DFA 的区别：

- 状态转移函数中，下一个状态可以有多个
- 读入空串也可导致状态转移

可见是在状态转移时有所不同；实际上这里的状态转移比“函数”要更一般，即关系。

$N = \{K, s, F, \Delta(\text{transition relation})\}, \Delta \subseteq K \times \{0, 1, e\} \times K$

NFA 做计算的过程也比 DFA 宽松一些：能读入字符串输入的全部并且“有一条路可以走通并走到可接收状态”（可理解为并行计算的）即可。

练习. 设计 NFA，计算 $W = \{w \in \{0,1\}^* \mid w$的倒数第二位是 $1\}$

解.

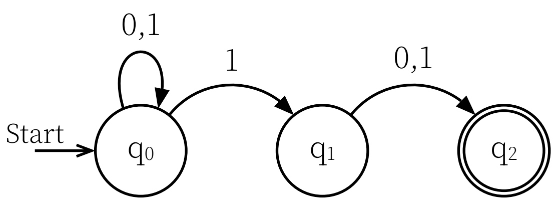


图 6 计算 $W = \{w \in \{0,1\}^* \mid w$的倒数第二位是 $1\}$

NFA 看起来可以猜测出一条正确的路径，似乎比 DFA 更强大一些，但实际上两者是等价的。

定理 3.2.2

DFA 能判断一个语言等价于 NFA 也能判断这个语言。

证明. DFA 本身就是一台特殊的 NFA，这个方向是显然的。

反过来的证明，想法是让 DFA 模拟 NFA 的 tree-like 计算过程。下面先研究一个 NFA 的情况

对于 $p \in K$，定义 $E(p) = \{p\} \cup \{q \mid (p, e, q) \in \Delta\}$（即 p 和不读入 symbol 即可到达的状态组成的集合）；假设该 NFA 从某层（tree-like arch）状态集合 Q 接收一个 0 到达下一层 Q'，那么 $Q' = \bigcup_{q \in Q} \bigcup_{p \in (q,0,p)} E(p)$

接下来用 DFA 模拟这个过程。很明显其每个元素有如下表示:

- 状态集合 $K' = \{Q \mid Q \subseteq K\}$
- 初始状态 $s' = E(s)$
- accepting states $F' = \{Q \mid Q \subseteq K, Q \cap F \neq \emptyset\}$
- transition function $\delta(Q, a) = \bigcup_{q \in Q} \bigcup_{p \in (q,a,p)} E(p)$

□

因此正则语言也可以使用 NFA 来判断。

定理 3.2.3

如果 A 和 B 都是正则的，那么 AB （拼接，Concatenation）是正则的。(构造一个新 DFA, 其状态为原两个 DFA 状态的笛卡尔积, 且仅当两个状态均为可接受状态时新状态才为可接受状态)

证明. 总有一个 A 和 B 的交界处，其前面的字串用 $\mathrm{NFA_A}$ 判断，后面的字串用 $\mathrm{NFA_B}$ 判断。交界处由 NFA “猜测”，通过 e 切换。 □

定理 3.2.4

如果 A 是正则的，那么 $A^* = \{a_1 a_2 ... a_k : k \geq 0 \text{ and } a_i \in A\}$ （即从 A 中选取若干串做拼接）是正则的。

证明. 对于 $\mathrm{NFA_A}$，每次跑完一个字串后利用“猜测”功能返回其初态。 □

3.3 正则表达式

每种正则语言可以用正则表达式（Regular Expression, RE）表示:

定义 3.3.1

- Base Case：$0, 1, \emptyset$ are REs，分别对应 $L(0) = \{0\}, L(1) = \{1\}, L(\emptyset) = \emptyset$
- 如果 R 和 S are REs，那么以下也是 REs：
  - $(R \cup S)$，对应 $L(R \cup S) = L(R) \cup L(S)$
  - (RS)，对应 $L(RS) = L(R)L(S)$（recall it is concatenation）
  - (R^*)，对应 $L(R^*) = (L(R))^*$
  - 括号书写时很麻烦，记录一个算数优先级 precedence：$*$ > concatenation > $\cup$

例子

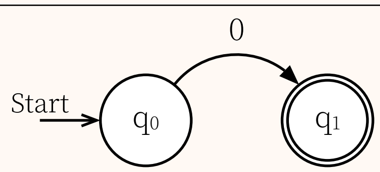
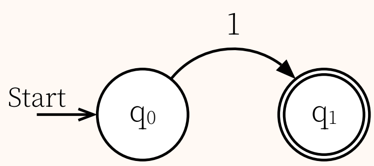
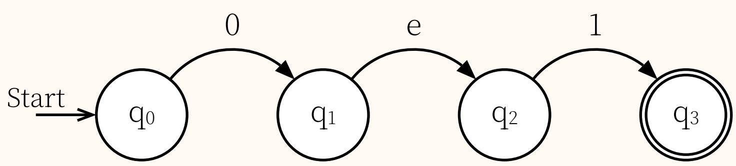
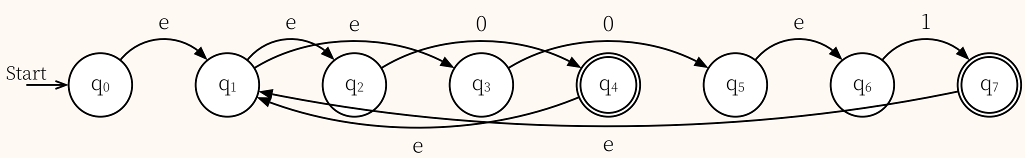
Language	RE
$\{e\}$	$\emptyset^*$
$\{w \in \{0,1\}^* : w \text{ starts with 0 and culminates with 1}\}$	$0(0 \cup 1)^*1$
$\{w \in \{0,1\}^* : w$至少包含两个0$\}$	$(0 \cup 1)^*0(0 \cup 1)^*0(0 \cup 1)^*$

定理 3.3.2

语言是正则的当且仅当其可以被某个正则表达式表示。

练习. 画出 $(01 \cup 0)^*$ 对应的 NFA。

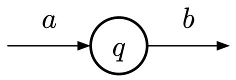
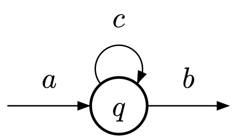
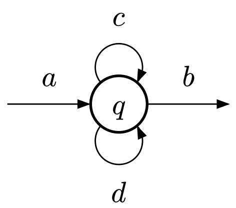
解.

RE	NFA
0	
1	
01	
$\{01 \cup 0\}^*$	

证明. 上面的练习可以总结出正则表达式向 NFA 的转化。从 NFA 向正则表达式的转化采用状态消除法（State Elimination Method）:

给定 NFA $N = (K, s, F, \Delta)$，由以下步骤得到正则表达式 R:

- 将 N 转化为 N'
  - N' 中没有状态指向其的 initial state（新建一个初态, 通过 e transition 指向原先的初态）
  - N' 的 accepting state 只有一个, 且该状态不指向任何状态（新建一个 accepting state, 使原先的若干 accepting states 通过 e transition 指向它）
- 删除 N' 中的非初态非末态状态，直到只剩下初态和末态；删除方法遵循下述规则（q 为将被删除的状态）：

NFA Component	RE (by State Elimination)
	ab
	ac^*b
	$a(c \cup d)^*b$

□

上述的状态删除法实质上是一个动态规划问题的求解。（我觉得不考）

3.3.1 Pumping Theorem

现在要证明某个语言是正则的，可以利用 DFA/NFA/RE 三种等价的方法证明，但要证明某个语言不是正则的，就比较困难了（需要说明所有的 FA/RE 都无法表示该语言?）。

于是介绍 Pumping Theorem，其说明了正则语言的一个性质：其中有些 Pattern 是可以重复的。

定义 3.3.3 (Pumping Theorem)

设 A 是一个正则语言，那么存在一个整数（Pumping length）$p \geq 1$，使得对于任意字符串 $s \in A$ 且 $|s| \geq p$，都可以将 s 分解为 $s = xyz$，满足：

1. $\forall i \geq 0, xy^i z \in A$（重复 Pattern y 任意次，字符串仍在语言中）
2. $|y| \geq 1$（可重复的 Pattern 非空）
3. $|xy| \leq p$（可以在前 p 个字符中找到该 Pattern）

证明. 由于 A 正则，于是存在 DFA M decides A，命 $p =$ number of states in M

对于 $w \in A, |w| \geq p$，考虑其被 DFA 计算的过程：$q_0 \to q_1 \to \dots \to q_{n-1} \to q_n$，$q_0, q_n$ 分别为初态和接受状态，由于要处理 w 的至少 p 位，必然有 $n \geq p$；与此同时，状态的总数只有 p 个，于是一定存在 i, j，使得 $0 \leq i < j \leq p$ 且 $q_i = q_j$（Pigeonhole Principle）

于是在状态 q_0 到 q_j 部分处理的字符串记为 x，在 q_i 到 q_j 部分处理的字符串记为 y，在 q_j 到 q_n 部分处理的字符串记为 z，则有 $s = xyz$.

由于 $q_i = q_j$，也就是说处理 y 的状态转换部分可以重复无数次；而 $|xy| = j \leq p$，且 $|y| = j - i \geq 1$，于是满足 Pumping Theorem 的要求。 □

♣

练习. 说明 $\{0^n 1^n : n \geq 0\}$ 不是正则的。

证明. 假设其是正则的，则存在 pumping length p。

考虑字符串 $w = 0^p 1^p$，它应当可以被分为 $w = xyz$ 且满足三条性质。

由于 $|xy| \leq p$，于是仅有可能 $xy = 0^k, k \leq p$，且 y 不为空串，那么 $y = 0^{k'}$

但这样一来，$xy^i z = 0^{k-k'+ik'} 1^p$，显然对于某些 i，新得到的字符串不在该语言中，与第一条性质矛盾。 □

3.4 Pushdown Automaton

我们已经介绍过 DFA, NFA, Regex 三种等价的计算模型，现在做出的考虑是：向其中添加一些新的 feature，以计算更多的函数，决定更多的语言。

比方说，在 DFA/NFA 中，状态切换时只是单纯地转移到下一个状态，但并没有“记忆”已经处理过的信息；如果除了 FA 的四元组外，能有一个数据结构来存储一些信息，或许能计算更多的函数。（这篇文章介绍了一个 PDA 在实际游戏开发中的例子：https://zhuanlan.zhihu.com/p/575664217）

定义 3.4.1 (PDA)

Pushdown Automaton(PDA) = NFA + stack

PDA 是一个四元组 $P = (K, s, F, \Delta)$其中 K, s, F 和 NFA 中的定义一样，$\Delta \subseteq (K \times \{0, 1, e\} \times \{0, 1\}^*) \times (K \times \{0, 1\}^*)$（其中第一部分的 $\{0, 1\}^*$ 表示栈顶将被 pop 出来的字符串，第二部分的 $\{0, 1\}^*$ 表示将被 push 到栈内的字符串）（何意味?）

e.g. 对于 $(q, a, \gamma, p, \beta) \in \Delta$，表示 PDA 在状态 q 下读入字符 a，且栈顶为 γ 时，可以转移到状态 p，并将栈顶的 γ pop 出，push 入 β。

似乎这么看来，每次进行 transition 时，都需要从栈里 pop 出一些东西，再 push 入一些东西，但实则可以用形如 $(q, a, \gamma, p, \beta\gamma)$ 的 relation 来表示“只 push 入而不 pop 出”的操作。

♣

由此可以看出，PDA 中状态的转换通常还伴随着栈的操作，而不仅仅只是先前两种 FA 那样的状态转换；因此方便起见，定义一个新的“配置”（Configuration），PDA 通过 relation 在不同的配置之间进行转换，这个配置也称为**即时描述（Instantaneous Description，ID）**。

这个配置相当于一种新的“状态”，思考一下，其与什么有关?

- 当前 PDA 所处的状态 $q \in K$
- 剩余未处理的输入字符串 $w \in \{0, 1\}^*$
- 栈内当前的字符串 $\alpha \in \{0, 1\}^*$，栈顶在左侧

定义 3.4.2 (Configuration of PDA)

A Configuration ID $C \in K \times \{0, 1\}^* \times \{0, 1\}^*$, i.e. $C = (q, w, \alpha)$

♣

如果一个 PDA P 从配置 C_1 只通过一步转移到配置 C_2，则记作 $C_1 \underset{P}{\vdash} C_2$；对于不限制次数的转移，记作 $C_1 \overset{*}{\underset{P}{\vdash}} C_2$。

定义 3.4.3 (Decision of PDA)

PDA P accepts string $w \in \{0, 1\}^*$ if $\exists q \in F$ such that $(s, w, e) \overset{*}{\underset{P}{\vdash}} (q, e, e)$，即走到可接受态 & 栈空 & 输入处理完

$L(P) = \{w \in \{0, 1\}^* \mid P \text{ accepts } w\}$, P decides $L(P)$

$L(P)$ 称为**上下文无关语言（Context-Free Language，CFL）**。

♣

找出下面 CFL 的 PDA

1. $L = \{w \in \{0,1\}^* \mid \text{in w, number of 0s is equal to number of 1s}\}
   - $K = F = \{q\}$
   - $s = q$（只有一个状态作为初态和可接受态）
   - $\Delta = \{$

     $((q,0,e),(q,0)),$ # push 0 onto empty stack when reading

     $((q,0,0),(q,00)),$ # push another 0 onto stack when reading 0 and stack top is 0

     $((q,0,1),(q,e)),$ # pop 1 from stack when reading 0 and stack top is 1

     $((q,1,e),(q,1)),$ # push 1 onto empty stack when reading 1

     $((q,1,1),(q,11)),$ # push another 1 onto stack when reading 1 and stack top is 1

     $((q,1,0),(q,e)),$ # pop 0 from stack when reading 1 and stack top is 0

     $\}$
   - 易知，每次在栈顶为 1/0 的情况下读到 0/1 就会抵消掉栈顶的 1/0，于是要达成最终栈空的条件，必然需要 0 和 1 数量相等
2. $L = \{ww_{\text{Reversal}} \mid w \in \{0,1\}^*\}$
   - $K = \{l, r\}$，l 表示正在读取左半部分，r 表示正在读右半部分
   - $s = l$
   - $F = \{r\}$
   - $\Delta = \{$

     $((l,0,e),(l,0)),$

     $((l,1,e),(l,1)),$ # Push whatever read while we are on the former half

     $((l,e,e),(r,e)),$ # NFA guess when we reach the middle and transit to latter

     $((r,0,0),(r,e)),$

     $((r,1,1),(r,e)),$ # Pop out what's there, if we read the same on the latter half

     $\}$

3.5 语法（Grammar）

接下来学习一个全新的概念：**语法（Grammar）**。先前的三种 FA 都是用于判断某个字符串是否属于某个语言，而语法则是用于**生成（generate）**某个语言中的字符串；主要介绍**上下文无关语法（Context-Free Grammar, CFG）**。

定义 3.5.1 (CFG)

A CFG $G = \{V, S, R\}$, where:

- V is a finite set of symbols, including 0,1
- $S \in V - \{0,1\}$: start symbol
- $R \in (V - \{0,1\}) \times V^*$: rules (must be finite)
  - 为表示方便，一般将规则写作 $A \to \alpha$，其中 $A \in V - \{0,1\}, \alpha \in V^*$

♣

> 引述
>
> 为了推导出一个 CFG 所描述的语言中的字符串，我们从起始符号开始，不断地将变量 A 用它的某一个产生式的体来替代，从而得到所有的字符串。这里，A 的产生式指的是以 A 为头的产生式。
>
> — https://fla.cuijiacai.com/04-cfg/#_4-1-2-%E5%BD%A2%E5%BC%8F%E5%8C%96%E5%AE%9A%E4%B9%89

为了解析 rules 的推导过程，定义推导（Derivation）：

定义 3.5.2 (Derive)

if $A \to \alpha \in R$，then $\beta A\gamma \Rightarrow \beta\alpha\gamma$，其中 $\beta, \gamma \in V^*$

对于若干次的推导，类似的，表示为 $\overset{*}{\Rightarrow}$

接下来就可以定义 CFG 生成语言的方法：G generates $w \in \{0,1\}^*$ if $S \overset{*}{\Rightarrow} w$ & $L(G) = \{w \in \{0,1\}^* \mid G \text{ generates } w\}$

> Find CFG for these languages
>
> 1. $\{w \in \{0,1\}^* \mid w = w_{\text{Reversal}}\}$，即回文句
>   - 考虑到 $w = 0u0$ or $1u1$ where $u = u_{\text{Reversal}}$
>   - R: $S \to e|1|0|1S1|0S0$

3.6 上下文无关语言（Context-Free Language, CFL）

大的来了：上面两小部分讲述的 PDA 和 CFG 其实是等价的计算模型！即一个语言被某台 PDA 决定当且仅当它可以被某个 CFG 生成。

定理 3.6.1

CFG $\Leftrightarrow$ PDA

证明.

1. 试证 CFG $\Rightarrow$ PDA
   - Given CFG $G = (V, S, R)$, 构造 PDA $P = (K, p, F, \Delta)$, where:
     - $K = \{p, q\}$
     - $F = \{q\}$
     - $\Delta = \{$

       $((p, e, e), (q, S))$, # Push start symbol onto empty stack

       $((q, e, A), (q, \alpha))$ for each $A \to \alpha \in R$, # Replace non-terminal A on top of stack with alpha

       $((q, a, a), (q, e))$ for each $a \in \{0, 1\}$, # Pop out what's on stack if it matches the read input

       $\}$
2. 试证 PDA $\Rightarrow$ CFG
   - Given PDA $P = (K, s, F, \Delta)$，先将其转化为 simple 的。

定义 3.6.2 (Simple PDA)

A PDA $P = (K, \Delta, s, F)$ is simple, if

- $|F| = 1$（可接受态只有 1 个）
- $\forall((p, a, \alpha), (q, \beta)) \in \Delta$，either $\alpha = e$ & $|\beta| = 1$，or $|\alpha| = 1$ & $\beta = e$（每次 transition 要么只 push 一个 symbol，要么只 pop 一个 symbol）

定理 3.6.3 (PDA => simple PDA)

这里介绍通过以下几个步骤如何将 PDA 转化为 simple PDA:

1) if $|F| > 1$，回忆 小节 3.3 部分从 NFA 转为 RE 的 proof，创造一个新的可接受态 f，$\forall q \in F$，创造一个新的 transition $((q, e, e), (f, e))$，然后令 $F = \{f\}$ 即可。
2) for $((p, a, \alpha), (q, \beta)) \in \Delta$,
   - if $|\alpha| \geq 1$& $|\beta| \geq 1$, split the transition into two via adding a intermediate state r: $((p, a, \alpha), (r, \beta))$ and $((r, e, e), (q, e))$, then we go to next cases
   - if $|\alpha| > 1$&$\beta = e$, split the pop operation into $|\alpha| - 1$ steps via adding intermediate states $r_1, r_2, \ldots, r_{|\alpha|-1}$: $((p, a, \alpha[0]), (r_1, e))$, $((r_1, e, \alpha[1]), (r_2, e))$, $\cdots$, $\left(\left(r_{|\alpha|-1}, e, \alpha[|\alpha| - 1]\right), (q, e)\right)$
   - if $\alpha = e$& $|\beta| > 1$, split the push operation just like above
   - if $\alpha = \beta = e$, also split into 2 via adding an intermediate state r: $((p, a, 0), (r, e))$ and $((r, e, e), (q, 0))$（push 0 再 pop 出来）

- 接下来把 simple PDA 转化为 CFG: Given simple PDA $P = (K, s, \{f\}, \Delta)$, 构造 CFG $G = (V, S, R)$
  - 先定义 V : symbol set, 由于期望中, G 要生成的语言就是 PDA 所决定的, 而 P 决定一个语言又是对其中所有的 $w \in \{0,1\}^*$ 进行从初态到可接受态的转换，因此 V 中需要包含所有可能的 state pairs : $(p, q) \in K \times K$，由于 CFG 处理的是 symbol，需要将这个 pair 表示为 symbol，即 $V = \{0,1\} \cup \{A_{pq} \mid (p, q) \in K \times K\}$
  - 接下来先不着急继续寻找其他两个元素的定义, 想一想我们的目标: 将这些符号 A_{pq} produce 出某些 $w \in \{0,1\}^*$, 并且令其与“w 也可通过 P 的判断得到”这个陈述是等价的。严谨阐述就是 $A_{pq} \overset{*}{\Rightarrow} w$ for some $w \in \{0,1\}^* \Leftrightarrow w \in \left\{u \in \{0,1\}^* \mid (p, u, e) \vdash_P^* (q, e, e)\right\}$
  - 显然我们会发现一个很特殊的 symbol A_{sf}，它对应的语言正是 PDA P 决定的语言，考虑到 CFG 做的本职工作就是把 S 阐释为一种语言，因此使这两种语言相同，令 $S = A_{sf}$ 即可。

□

四、图灵机（Turing Machine）

> **合味道**
>
> 我要提一个以著名计算机科学家名字命名的，一个屈居于以初代校长为名的扫码学院下的神秘班级了。

尽管 PDA/CFG 已经比 FA 强大了不少，但它们仍然无法计算所有的函数/决定所有的语言，例如，$\{0^n1^n0^n \mid n \geq 0\}$，于是，隆重端出图灵机（Turing Machine, TM）。

> 引述
>
> The “granddaddy” of all models of computation is the Turing machine.
>
> — introtcs.org

想象一根从一端开始无限长的纸带（Tape），上面划分为一个个方格（Cell），每个方格中可以写入一个 symbol；同时有一个读写头（Head）可以在纸带上左右移动，并且可以读出/写入当前所在方格的 symbol：Symbol 有四种：$1, 0, \triangleright, \emptyset$，最后两种分别是纸带的开头和表示该 Cell 为空的字符

该纸带还附带一个状态控制器（State Controller），读写头每到一个 Cell 时，根据当前状态和该 Cell 中的 symbol，可以完成下面两个操作：

- Write a symbol to the current Cell
- Move the head one cell to the left or right/Stay/Halt

定义 4.1 (Turing Machine)

A Turing Machine $M = (K, \Sigma, s, \delta)$, where:

- K is a finite set of states
- $s \in K$ is the initial state
- Σ is a finite set of symbols, $\{0, 1, \triangleright, \emptyset\} \subseteq \Sigma$
- δ is “Transition Function”
  - $\delta : K \times \Sigma \to K \times \Sigma \times \{\text{MoveLeft}, \text{MoveRight}, \text{Stay}, \text{Halt}\}$，即根据当前状态和当前 Cell 中的 symbol，输出下一状态、写入的 symbol、读写头的操作。

♣

直接来看该模型是怎么接收字符串的：对于 input $x = x_0x_1...x_{n-1}$，TM 先将纸带的前 n 个 Cell 写为 $\triangleright\, x_0x_1...x_{n-1}$（$T[0] = \triangleright, T[1] = x_0, T[2] = x_1, ..., T[n] = x_{n-1}$），其余 Cell 均为 $\emptyset$，接下来记初态 $s = 0$，且开始从 $\triangleright$ 后的 $i = 0$ 位置开始读取字符串，重复以下几步：

- 根据当前 Cell 的内容和当前状态 q，通过 $\delta(q, T[i]) = (q', a, \text{Direction})$ 得到下一状态 q'、写入 symbol a 和读写头的操作。（对于开头的 $\triangleright$，要求其唯一且不可覆盖，即 $a == \triangleright$ 当且仅当 $T[i] == \triangleright$）
- 写入 Symbol，转移状态：$T[i] = a, q = q'$
- switch Direction
  - if Direction = MoveRight, $i = i + 1$
  - if Direction = MoveLeft, $i = \max(0, i - 1)$
  - if Direction = Stay, $i = i$
  - if Direction = Halt, stop

停机后，TM 会输出一个东西：$M(x) = T[1]T[2]...T[i]$ where $T[i+1]$ 是第一个 $\emptyset$。

如果接受完了 input 还没有停机，则记 $M(x) = \bot$

定义 4.2 (Computation via TM)

TM M computes function $f : \{0,1\}^* \to \{0,1\}^*$ if $\forall x \in \{0,1\}^*, M(x) = f(x)$

注意这要求了 TM 必须对所有输入停机；可被 TM 计算的函数称为**可计算函数**（Computable Function）。

4.1 图灵完备

使用一些常见的编程语言都可以实现上面介绍的 TM 计算步骤，这些语言被称为**图灵完备**（Turing Complete）的语言，即其可以完成图灵机做的事情。

现在从另一个方向开展研究：图灵机可以做到这些个编程语言能完成的任务吗?

4.1.1 NAND-TM

就一种名叫 NAND-TM 的编程语言，其可以看作先前介绍的 NAND-CIRC 的扩展：允许使用数组（Array）存储任意长度的字符串，并且允许使用循环（循环次数与输入长度有关，不一定为常数的 Loop）来处理这些数组。

- NAND-TM 中有两种数据类型：
  - i: integer value
  - other: boolean value
- 有两种变量类型：
  - scalar: boolean variable
  - array: array of boolean variables (can be infinite long)
    - All arrays are accessed only and simultaneously by i
- 其输入 X 和 输出 Y 都是 array 类型的变量
- 程序最后一行是 `MODANDJUMP(a,b)`: modify i and jump back to the first line.
  - if $a = 1$ and $b = 1, i = i + 1$
  - if $a = 0$ and $b = 1, i = i - 1$
  - if $a = 1$ and $b = 0, i = i$
  - if $a = 0$ and $b = 0$, halt
- 所有变量（除输入 X 外）初始均为 0

于是在程序每轮执行中，对 $A[i], B[i]$... 等进行 NAND 操作，并且通过 `MODANDJUMP` 来控制 i 的变化，进行循环处理。

定理 4.1.1 (TM & NAND-TM)

TM ⇔ NAND-TM，并且其元素有如下对应关系

TM Element	NAND-TM Element
finite states	scalar variables
tape	array variables
head position	integer variable i

证明.

- 左推右：
  - 假设有 TM $M = (K, \Sigma, s, \delta)$，构造 NAND-TM 程序 P:
    - 使用 $\lceil \log_2(|K|) \rceil$ 个 scalar 变量来表示 TM 的状态；使用 $\lceil \log_2(|\Sigma|) \rceil$ 个 array 变量来表示 TM 的 tape；两个 scalar 变量表示读写头的四个操作（可编码为 00，01，10，11）
    - 那么 relation $\delta : K \times \Sigma \to K \times \Sigma \times \{\text{MoveLeft}, \text{MoveRight}, \text{Stay}, \text{Halt}\}$ 转化为了 function $\delta' : \{0,1\}^{\lceil \log_2(|K|) \rceil + \lceil \log_2(|\Sigma|) \rceil + 2} \to \{0,1\}^{\lceil \log_2(|K|) \rceil + \lceil \log_2(|\Sigma|) \rceil + 2}$，这是一个有限函数，也就是说，可以被一个 NAND-CIRC program 计算。
    - 记其计算得到的结果最后两位为 a, b，则在程序最后加上 `MODANDJUMP(a,b)`，即得到与原 TM 对应的 NAND-TM。
- 右推左：
  - 假设有 NAND-TM 程序 P，构造 TM
  - k 个 scalar 变量，对应 TM 的 2^k 个状态
  - l 个 array 变量，那么字符集数量 $|\Sigma| = 2^l$
  - 对于 $f : \{0,1\}^{k+l} \to \{0,1\}^{k+l}$ 的这个 NAND-TM Program，可转化为 $g : \{0,1\}^{k+l} \to \{0,1\}^{k+l} \times \{\text{MoveLeft}, \text{MoveRight}, \text{Stay}, \text{Halt}\}$ 的 TM Transition Function

□

再加一点语法糖：

- GOTO 语句：`GOTO L`，表示跳转到 `line` 为 L 的行继续执行。
- Multi-Indexing：允许使用多个 index 来访问不同 array 的不同位置
- Multi-Dimensional Arrays：允许使用多维数组

4.1.2 NAND-RAM

TM 可以再次进化为 RAM，实际上现代编程语言一般都基于 RAM 模型。

纸带记为 Memory，每个 Cell 可以存储一个整数而非一个字符，并且有 k 个寄存器（Register），每个寄存器可以存储一个 integer value。

1. 可以 Write/Read Memory Cell，并且是指哪就能访问哪，不需要读写头
2. 可以对寄存器进行加减乘除等算术运算
3. 可以根据判断结果进行下一步操作

同样，RAM 也对应一种 NAND-RAM 编程语言，其有 integer 变量（有最大值）和 index 访问的 array；还有许多算术运算指令。

可以证明 NAND-TM ⇔ NAND-RAM

定理 4.1.2 (Church-Turing Thesis)

图灵机是终极的计算模型。

4.2 通用图灵机（Universal Turing Machine）

> 引述
>
> The universal machine/program - “one program to rule them all”.
>
> — introtcs.org

现在已经有了最强最厉害的计算模型了，我们希望找到一台能够完成所有任务的图灵机（这被称为 Universality），而非对每个函数都寻找一个专用的 TM。

命这样的一台图灵机为 $U(M,x) = \begin{cases} M(x) & \text{if M is encoding of a TM} \\ M,x & \text{Otherwise} \end{cases}$

为了证明 U 的存在性，需要先说明如何对图灵机（$M = (K, \Sigma, s, \delta)$）进行编码：

- 状态集 $K \to 0, 1, ..., |K| - 1$，命编码为 0 的为初态 s
- 字符集 $\Sigma \to 0, 1, ..., |\Sigma| - 1$
- 转移函数 $\delta : K \times \Sigma \to K \times \Sigma \times \{\text{MoveLeft}, \text{MoveRight}, \text{Stay}, \text{Halt}\}$，表示为五元组的集合 $\{(\text{CurrentState}, \text{CurrentSymbol}, \text{NextState}, \text{WrittenSymbol}, \text{Direction})\}$，这集合里的元组数量为 $|K| \times |\Sigma|$，对每个元组中的每个元素进行编码即可。

em，然后对于任意的输入 x，只需要就上面的编码方式进行 relation 的查询，即可利用输入的图灵机进行计算；也就是说，U 是存在的。

4.3 函数的可计算性（Computability）

现在，不止有最强的计算模型，我们甚至有 rule them all 的通用图灵机了——那么，世界上所有的问题，都可以被图灵机解决吗？

定理 4.3.1 (Uncomputability of some functions)

存在一个布尔函数 $F : \{0, 1\}^* \to \{0, 1\}$，其无法被任何图灵机计算。

证明.

- 理论上由如下两点即可证明
  - set of boolean functions: uncountable
  - set of Turing machines: countable (because TMs can be coded)
- 尝试构造该 F: $F(x) = \begin{cases} 0 \text{ if } x \text{ is encoding of a TM } M \text{ and } M(x)=1 \\ 1 \text{ otherwise} \end{cases}$
- 即 $F(x) = 0$ 时，有一 TM 将自己的编码作为自己的输入且输出 1
- 则 $\forall$ TM M，记其编码为 m
  - 如果 $M(m) = 1$,则 $F(m) = 0$
  - 如果 $M(m) \neq 1$,则 $F(m) = 1$
- 由于 M 并不是在所有相同的输入下都和 F 有相同的输出，那么 F 无法被 M 计算，即无法被任何图灵机计算。

□

呃呃，这会不会有种钦定的感觉？感觉这个函数是专门找出来驳倒图灵机似的——然而，确实有很多有价值的、但是被发现是不可计算的函数。

4.3.1 停机问题

定义 4.3.2 (Halting Problem)

Given a TM M and input $x \in \{0,1\}^*$, does M halt on x?

该问题对应函数为 $\text{Halt}(M,x) = \begin{cases} 1 \text{ if M is a TM that halts on x} \\ 0 \text{ otherwise} \end{cases}$

定理 4.3.3 (Uncomputability of Halt)

$\text{Halt}(M,x)$ is uncomputable.

证明. 在小节 4.3 部分已经构造出了一个不可计算的函数 F，因此利用反证法，试证下面这个陈述成立："如果 $\text{Halt}(M,x)$ 可被某个 TM M_{Halt} 计算，那么 F 也可计算"。

即：利用已知的神秘 Oracle TM M_{Halt} 来计算 F。

这台 TM M_F 的工作流程如下：

1. 对输入 x，先利用 M_{Halt} 计算 $\text{Halt}(x,x)$
2. if $\text{Halt}(M_x,x)=0$，说明 TM M_x 在输入 x 时不会停机/x 根本就不是一台 TM 的编码，于是 $F(x)=1$
3. if $\text{Halt}(M_x,x)=1$，说明 TM M_x 在输入 x 时会停机，于是
    1) run x on TM M_x until it halts
    2) if $M_x(x)==1$，由 F 的定义可知 $F(x)=0$；else $F(x)=1$

综上，TM M_F 可以计算 F，与 F 不可计算矛盾，因此 $\text{Halt}(M,x)$ 也是不可计算的。 □

证明停机问题无法计算后，可以通过其证明更多不可计算的函数/问题了。

Halt zeto

Given a TM M, does M halt on input 0?

该问题对应函数 $\text{HALTONZERO}(M) = \begin{cases} 1 \text{ if } M \text{ is a TM that halts on } 0 \\ 0 \text{ otherwise} \end{cases}$

证明. 类似上一个定理证明中采用的反证法, 只需要证明: "如果这个函数可计算, 那么停机问题也可计算"(由于已知 Halt 是不可计算的, 那么可以直接得到这个函数的不可计算性)。

类似的，假设神秘 Oracle TM $M_{\text{HALTONZERO}}$ 可以计算 $\text{HALTONZERO}(M)$，那么可以构造 TM M_{HALT} 来计算停机问题，具体而言，通过下面几步证明:

1. 构造 TM N 使得 M 在 x 上停机当且仅当 N 在输入 0 上停机，其中 M, x 是 M_{HALT} 的输入
    1) N 是这样的：忽略输入，直接 run M on input x；易验证 M 在 x 上停机当且仅当 N 在 0 上停机
2. 运行 $M_{\text{HALTONZERO}}(N)$ 就可以判断 N 是否在输入 0 上停机；且 $M_{\text{HALT}}(M, x) = M_{\text{HALTONZERO}}(N)$，于是可以计算停机问题。
3. 但由于停机问题不可计算，因此 HALTONZERO 也是不可计算的。

□

Zero Function

Given a TM M, is $M(x) = 0 \forall x \in \{0,1\}^*$?

该问题对应函数 $\text{ZEROFUNC}(M) = \begin{cases} 1 \text{ if } M \text{ is a TM that computes the zero function} \\ 0 \text{ otherwise} \end{cases}$

证明. 依旧，假设 M_{ZERO} 可以计算 $\text{ZEROFUNC}(M)$，那么试构造 TM M_{HALT} 来计算停机问题:

1. 构造 TM N 使得 M 在 x 上停机当且仅当 N 计算 ZERO 函数
    1) N: run M on input x; return 0.
2. Run $M_{\text{ZEROFUNC}}(N)$ 来判断 N 是否计算 ZERO 函数；且 $M_{\text{HALT}}(M, x) = M_{\text{ZEROFUNC}}(N)$，于是可以计算停机问题。
3. 但由于停机问题不可计算，因此 ZEROFUNC 也是不可计算的。

□

上面两个例子中采用的证明过程是类似的：构造新问题涉及的中间件 TM N 使得 $\text{HALT}(M, x) = \text{TargetTM}(N)$，即从停机问题转换为目标问题的图灵机，这被称为*归约*（reduction）。

定义 4.3.4 (Reduction)

A Reduction from F to G is a `computable` function $R : \{0,1\}^* \to \{0,1\}^*$ such that $\forall x \in \{0,1\}^*, F(x) = G(R(x))$.

引理 4.3.4

If G is computable, then F is computable.

$$\Leftrightarrow$$

If F is uncomputable, then G is uncomputable.

♡

♣

Rice's Theorem

Given a TM M, does M have property P?（P 是一个布尔函数）

该问题对应函数 $\mathrm{P}(M) = \begin{cases} 1 \text{ if } M \text{ is a TM that has property P} \\ 0 \text{ otherwise} \end{cases}$

Rice’ s theorem: 如果 P 是 semantic & non-trivial 的，那么 P 是不可计算的。

两台图灵机 M_1, M_2 是 functionally equivalent 的，如果 $\forall x \in \{0,1\}^*, M_1(x) = M_2(x)$。

一个性质 $P : \{0,1\}^* \to \{0,1\}$ 是 semantic 的，如果对于任意两台 functionally equivalent 的图灵机 M_1, M_2，有 $\mathrm{P}(M_1) = \mathrm{P}(M_2)$

一个性质 P 是 trivial 的，如果 P 是 constant function

证明. TBD □