# 编程基础 II - 可计算性

刘 钦

2017年春

# Outline

- 可计算函数
- 不可计算函数
- 图灵机
- Register Machine

# Outline

- 可计算函数
    - 自然数
    - 函数
    - 原始递归函数
    - 可计算函数模型
- 不可计算函数
- 图灵机
- Register Machine

为什么计算机可以计算?

哪些是计算机可以计算的?

可不可以用数学方法来证明?

目标：
建立一整套形式化的可计算模型

# 什么是自然数?

# 如何用形式化方法来定义自然数?

# 自然数的皮亚诺公理系统 - from wiki

- 皮亚诺的这五条公理用非形式化的方法叙述如下：

  i）1是自然数；

  ii）每一个确定的自然数a，都有一个确定的后继数a' ，a' 也是自然数（一个数的后继数就是紧接在这个数后面的数，例如，1的后继数是2，2的后继数是3等等）；

  iii）如果自然数b、c的后继数都是自然数a，那么b = c；

  iv）1不是任何自然数的后继数；

  v）任意关于自然数的命题，如果证明了它对自然数1是对的，又假定它对自然数n为真时，可以证明它对n' 也真，那么，命题对所有自然数都真。（这条公理保证了数学归纳法的正确性）

- 若将0也视作自然数，则公理中的1要换成0。

# 自然数的公理系统 - from wiki

- 更正式的定义如下：
- 一个戴德金-皮亚诺结构为一满足下列条件的三元组（X, x, f）：
  - X是一集合，x为X中一元素，f是X到自身的映射。
  - x不在f的值域内。（对应上面的公理4）
  - f为一单射。（对应上面的公理3）
  - 若A为X的子集并满足：
    - x属于A,且
    - 若a属于A,则f（a） 亦属于A
    - 则A = X。

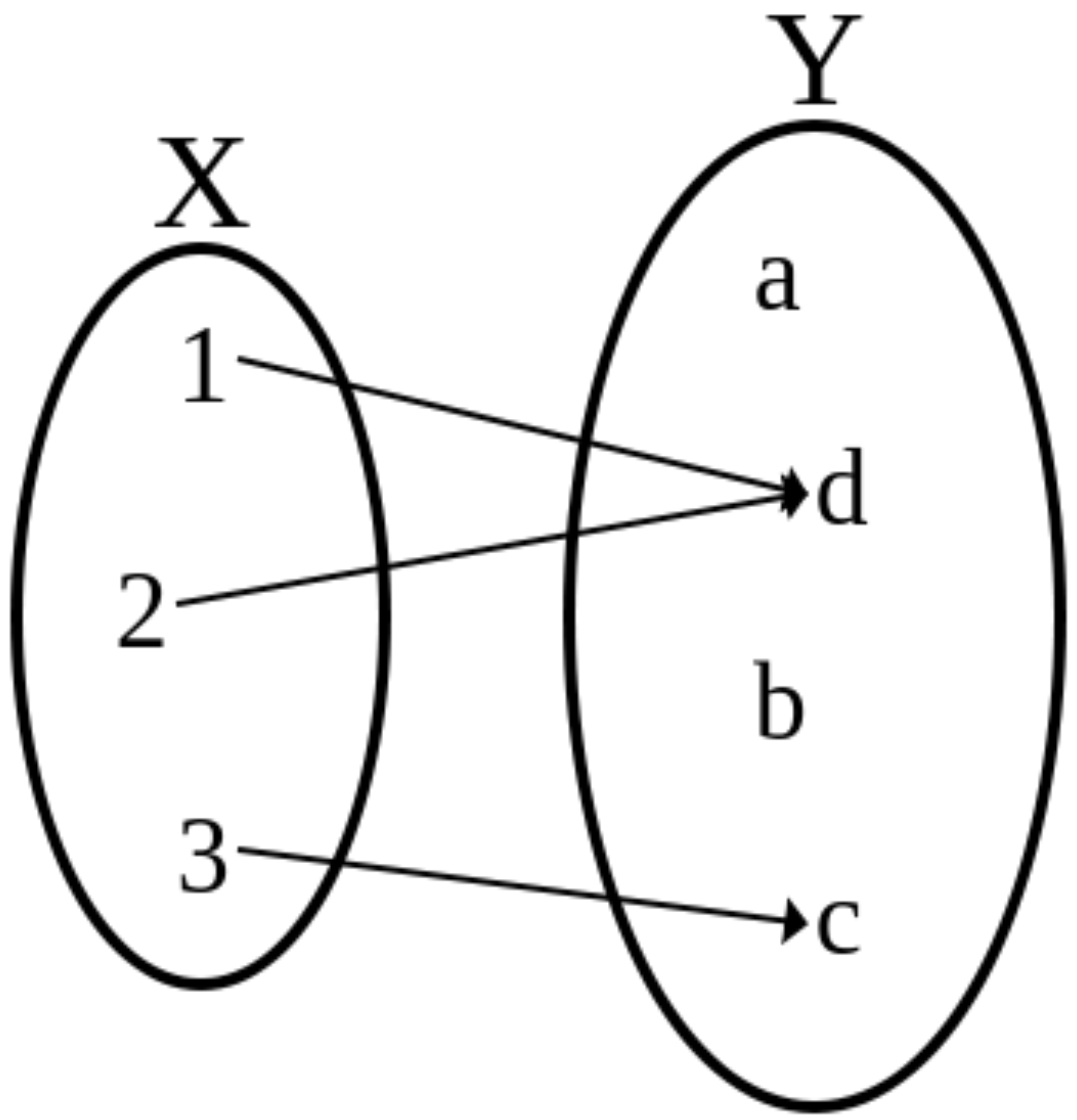
# 函数 - from wiki

从输入值集合$X$到可能的输出值集合$Y$的函数$f$（记作$f: X \to Y$）是$X$与$Y$的关系，满足如下条件：

1. $f$是**完全的**：对集合$X$中任一元素$x$都有集合$Y$中的元素$y$满足$x f y$（$x$与$y$是$f$相关的）。即，对每一个输入值，$y$中都有与之对应的输出值。
2. $f$是**多对一**的：若$f(x) = y$且$f(x) = z$，则$y = z$。即，多个输入可以映射到一个输出，但一个输入不能映射到多个输出。

定义域中任一$x$在到達域中唯一对应的$y$记为$f(x)$。

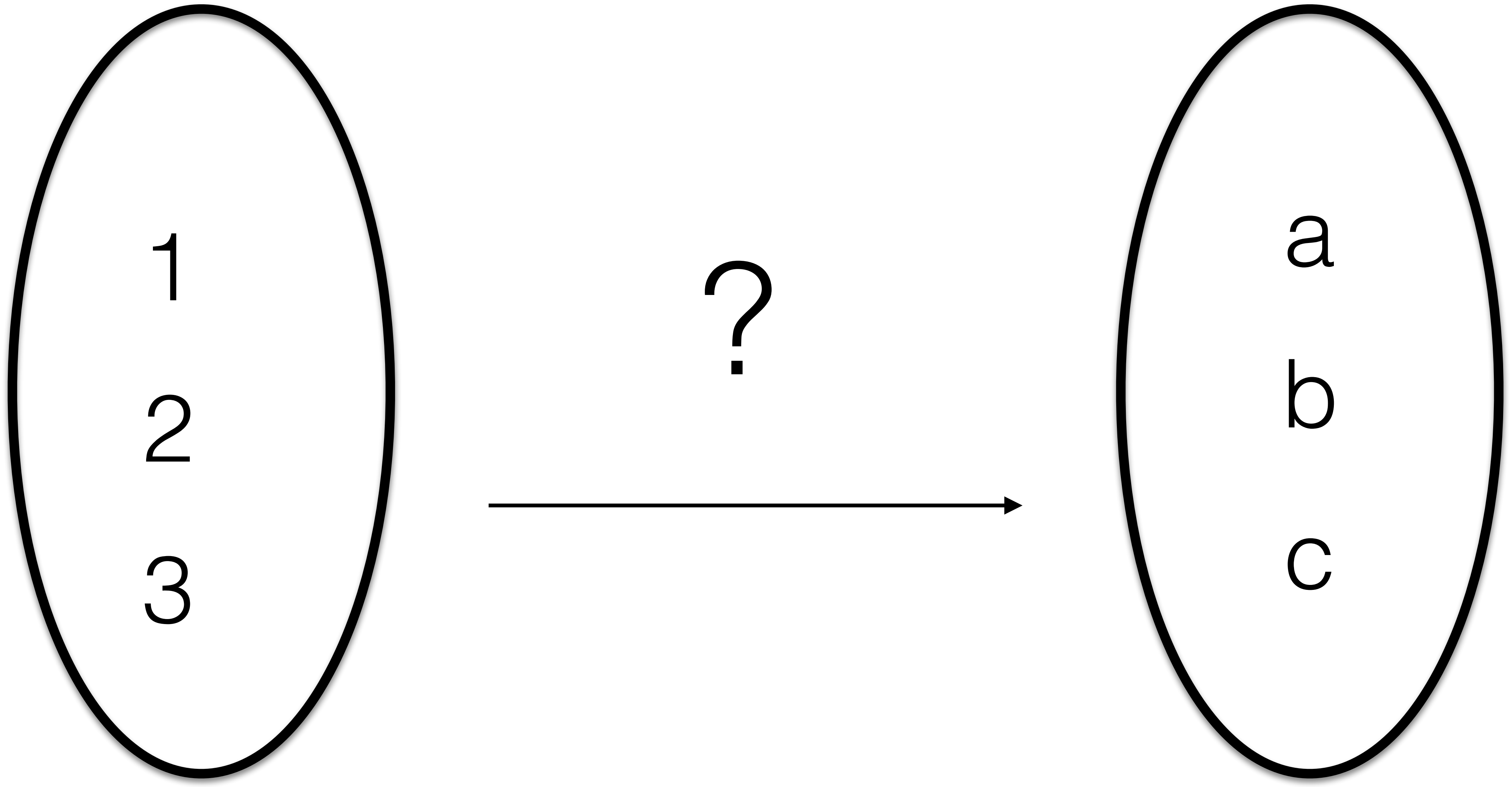
比上面定义更简明的表述如下：从$X$映射到$Y$的函数$f$是$X$与$Y$的直积$X \times Y$的子集。$X$中任一$x$都与$Y$中的$y$唯一对应，且有序对$(x, y)$属于$f$。



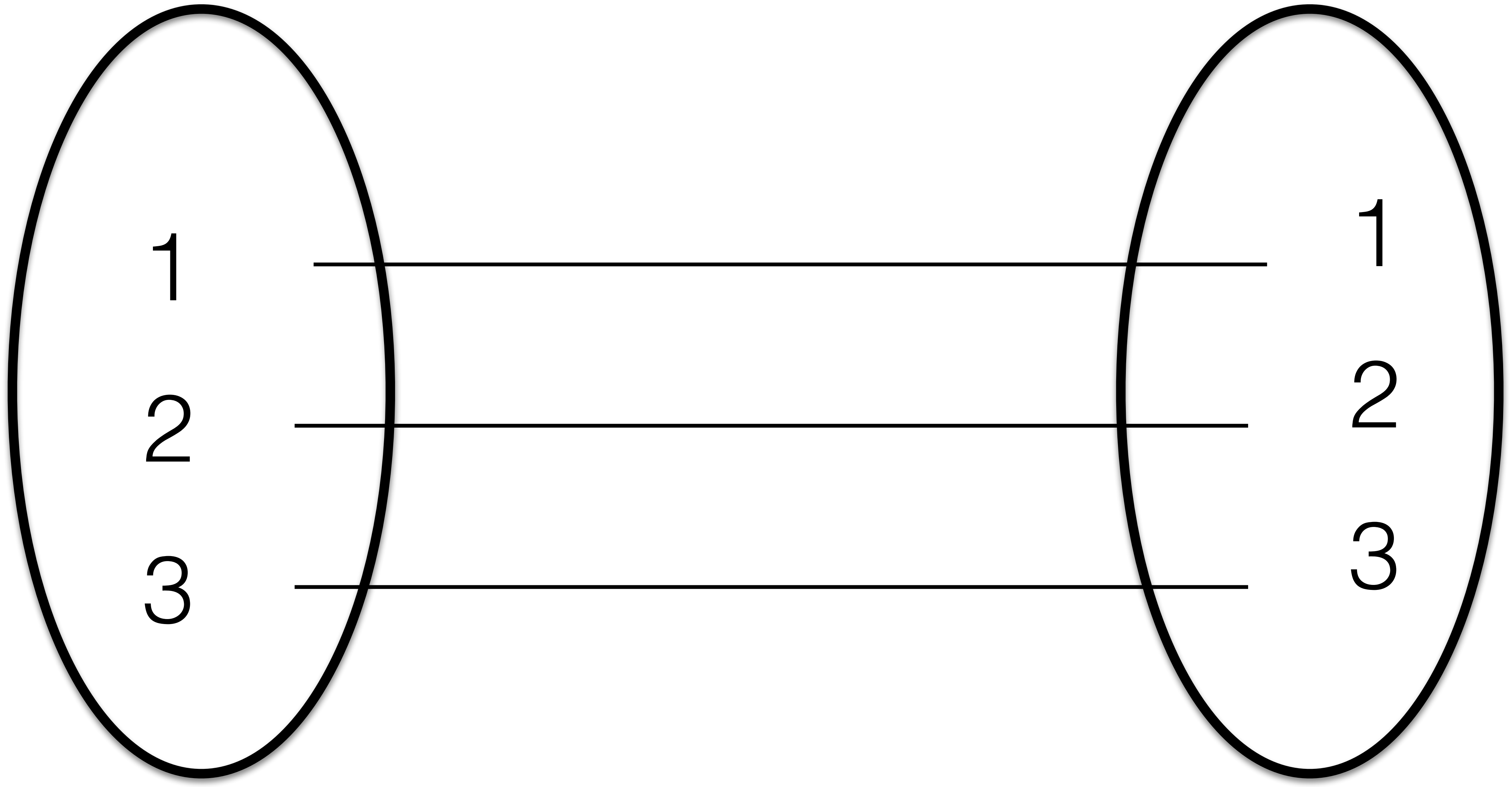
函数是一种映射关系

什么样的函数看上去可以计算?

# 我们能表达什么样的映射?

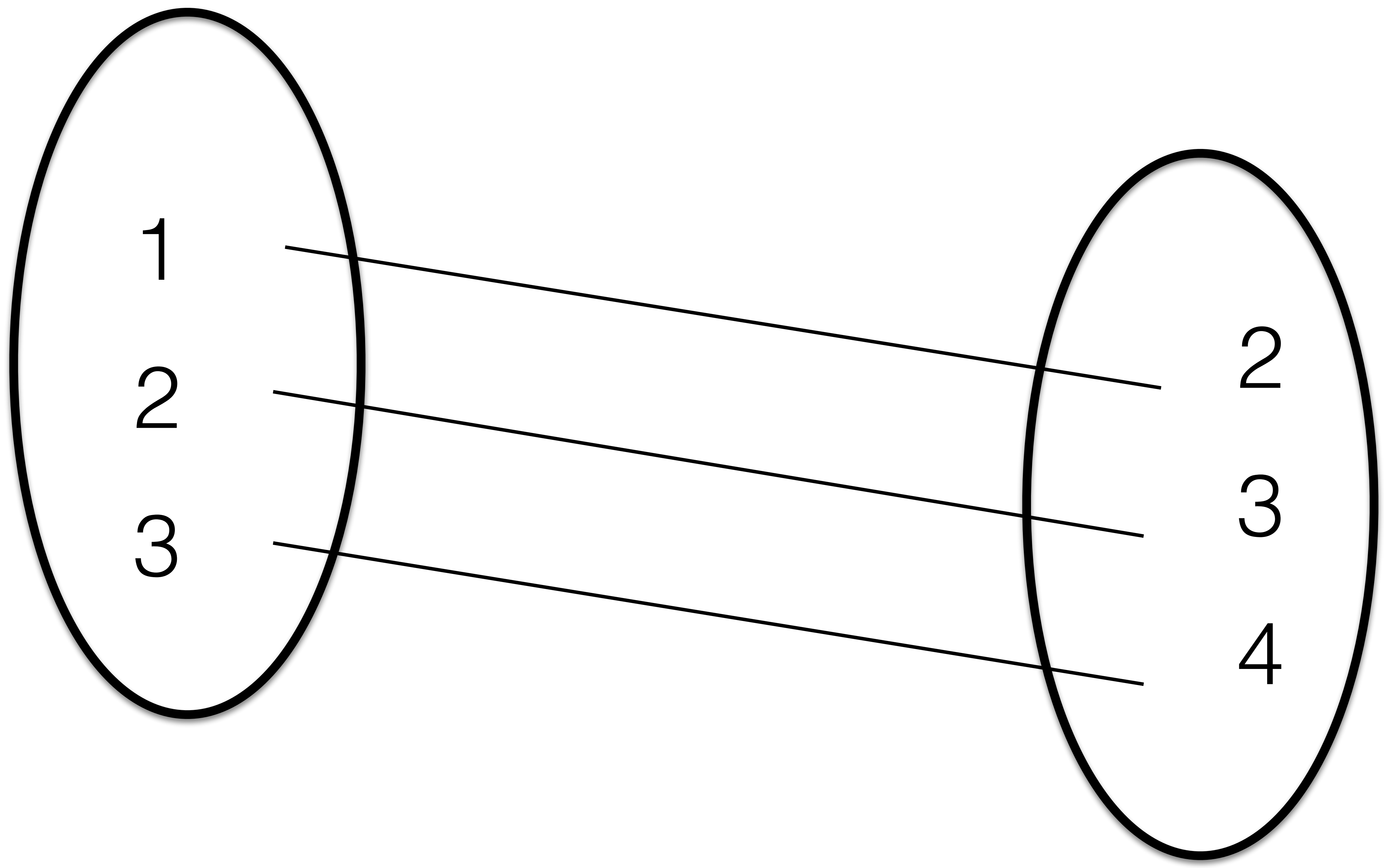


# 我们能表达什么样的映射?



$$f(x) = x$$

我们能表达什么样的映射?



$$f(x) = S(x)$$

//S(x)表示x的后继

$$f(x) = S(\ S(x)\ )$$

$$f(1) = 1$$
$$f(x) = S(f(P(x)))$$

//P(x)表示x的前继

递归函数感觉上是可计算的函数?

1923年，斯科朗提出并初步证明一切初等数论中的函数都可以由原始递归式作出，即都是原始递归函数。

# 原始递归函数 - from wiki

- 原始递归函数接受自然数或自然数的元组作为参数并生成自然数。接受 n 个参数的函数叫做 n-元函数。基本原始递归函数用如下公理给出:
  - **常数函数**: 0 元常数函数 0 是原始递归的。
  - **后继函数**: 1 元后继函数 S，它接受一个参数并返回皮亚诺公理给出的后继数，是原始递归的。
  - **投影函数**: 对于所有 n≥1 和每个 1≤i≤n 的 i，n 元投影函数 Pin，它接受 n 个参数并返回它们中的第 i 个参数，是原始递归的。
- 更加复杂的递归函数可以通过应用下列公理给出的运算来获得:
  - **复合**: 给定k 元原始递归函数 f，和 k 个 m 元原始递归函数 g1,...,gk，f 和 g1,...,gk 的复合，也就是 m 元函数 h(x1,...,xm) = f(g1(x1,...,xm),...,gk(x1,...,xm)), 是原始递归的。
  - **原始递归**: 给定 k 元原始递归函数 f，和 k+2 元原始递归函数 g，定义为 f 和 g 的原始递归的 k+1 元函数，也就是函数 h 这里的 h(0,x1,...,xk) = f(x1,...,xk) 并且 h(S(n),x1,...,xk) = g(h(n,x1,...,xk),n,x1,...,xk), 是原始递归的。
- 服从这些公理的函数是原始递归的，如果它是上述基本函数之一，或者它可以通过应用有限次数的运算获得自基本函数。

# 加法 - from wiki

直觉上我们会把加法递归的定义为:

$\text{add}(0,x)=x$

$\text{add}(n+1,x)=\text{add}(n,x)+1$

为了使它适合于严格的原始递归定义,我们定义:

$\text{add}(0,x)=P_1^{\,1}(x)$

$\text{add}(S(n),x)=S(P_1^{\,3}(\text{add}(n,x),n,x))$

(注意: 这里的 $P_1^{\,3}$ 是一个函数，它接受 3 个参数并返回第一个。)

$P_1^{\,1}$ 是简单的恒等函数；包含它是上述原始递归运算定义的要求；它扮演了 $f$ 的角色。$S$ 和 $P_1^{\,3}$ 的复合，它是原始递归的，它扮演了 $g$ 的角色。

# 减法

我们可以定义*有限减法*，就是说，截止到 0 的减法(因为我们还没有负数的概念呢)。首先我们必须定义"前驱" 函数，它担任后继函数的对立物。

直觉上我们会把前驱定义为:

$\text{pred}(0)=0$

$\text{pred}(n+1)=n$

为了使它适合正式的原始递归定义，我们写:

$\text{pred}(0)=0$

$\text{pred}(S(n))=P_2^2(\text{pred}(n),n)$

现在我们以类似加法的方式定义减法。

$\text{sub}(0,x)=P_1^1(x)$

$\text{sub}(S(n),x)=\text{pred}(P_1^3(\text{sub}(n,x),n,x))$

1931年，哥德尔在证明其著名的不完全性定理时，以原始递归式为主要工具把所有元数学的概念都算术化了。原始递归函数的重要性日益受到人们的重视，人们开始猜测，原始递归函数可能穷尽一切可计算的函数。

1928年，Wilhelm Ackermann (1896 - 1962，David Hilbert的学生) 发现x的y次幂的z-重积分 A(x,y,z)是递归的但不是原始递归的。Rosza Peter将A(x,y,z)简化到二元函数，初始条件由Raphael Robinson简化。

# 阿克曼函数

$$A(m,n)=\begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0. \end{cases}$$

- 非原始递归函数

*A*(*m*, *n*) 的值

| $m\backslash n$ | 0 | 1 | 2 | 3 | 4 | n |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | $n+1$ |
| 1 | 2 | 3 | 4 | 5 | 6 | $n+2$ |
| 2 | 3 | 5 | 7 | 9 | 11 | $2\cdot(n+3)-3$ |
| 3 | 5 | 13 | 29 | 61 | 125 | $2^{(n+3)}-3$ |
| 4 | 13 | 65533 | $2^{65536}-3$ | $A(3, 2^{65536}-3)$ | $A(3, A(4, 3))$ | $\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n+3 \text{ twos}}-3$ |
| 5 | 65533 | $A(4, 65533)$ | $A(4, A(5, 1))$ | $A(4, A(5, 2))$ | $A(4, A(5, 3))$ | |
| 6 | $A(5, 1)$ | $A(5, A(5, 1))$ | $A(5, A(6, 1))$ | $A(5, A(6, 2))$ | $A(5, A(6, 3))$ | |

# 证明思路简介

- Ackermann函数对两个变元都是单调增

- 对任意的原始递归函数$f(x_1,x_2,\ldots,x_n)$，存在一个仅仅依赖于f的常数M使得$f(x_1,x_2,\ldots,x_n)<A(M,\max\{x_1,x_2,\ldots,x_n\})$

- Ackermann函数不具有上述性质（即Ackermann函数不是原始递归函数）

前三个函数叫做"初始"或"基本"函数：（Kleene (1952) p. 219）：

- （1）**常数函数**：对于每个自然数*n*和所有的*k*:
  $f(x_1, \ldots, x_k) = n$。
  有时这个常数通过重复使用后继函数和叫做"初始对象0（零）"的对象来生成（Kleene (1952) p.?)
- （2）**后继函数**S: "从已经生成的对象到另一个对象n+1或n'（*n*的*后继者*）"（ibid）。
  S(x) $\equiv_{def}$ f(x) = x' = x +1
- （3）**投影函数**$P_i^k$（也叫做**恒等函数**$I_i^k$）:对于所有自然数*i*使得$1 \leq i \leq k$:
  $P_i^k(x_1, \ldots, x_k) =_{def} f(x_1, \ldots, x_k) = x_i$.

- （4）**复合算子**：复合也叫做**代换**，接受一个函数$h(x_1, \ldots, x_m)$和函数$g_i(x_1, \ldots, x_k)$对每个*i*有$1 \leq i \leq m$，并返回映射$x_1$, ... $x_k$到
  $f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$的一个函数。
- （5）**原始递归算子**：接受函数$g(x_1, \ldots, x_k)$和$h(y, z, x_1, \ldots, x_k)$并返回唯一的函数$f$使得
  $f(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$,
  $f(y+1, x_1, \ldots, x_k) = h(y, f(y, x_1, \ldots, x_k), x_1, \ldots, x_k)$。
- （6）**μ算子**：μ算子接受一个函数$f(y, x_1, \ldots, x_k)$并返回函数$\mu y f(y, x_1, \ldots, x_k)$，它的参数是$x_1$ , . . ., $x_k$。这个函数$f$要么是从自然数{ 0, 1, ... n }到自然数{ 0, 1, ... n }的数论函数，要么是运算于谓词(输出{ t, f })上生成{ 0, 1 }的表示函数。
  在任何一个情况下：这个函数μ*y f*返回最小的自然数$y$使得，如果这样的*y*存在，则$f(0,x_1,x_2,\ldots,x_k)$, $f(1,x_1,x_2,\ldots,x_k)$, ..., $f(y,x_1,x_2,\ldots,x_k)$都是有定义的，并且$f(y,x_1,x_2,\ldots,x_k) = 0$；如果这样的*y*不存在，则μ*y f*是对特定参数$x_1,\ldots,x_k$是未定义的。

# μ-递归函数

# 邱奇-图灵论题（Church–Turing thesis）

- is a combined hypothesis ("thesis") about the nature of functions whose values are effectively calculable;
- or, in more modern terms, functions whose values are algorithmically computable.
- In simple terms, the Church–Turing thesis states that a function is algorithmically computable if and only if it is computable by a Turing machine.

# Church–Turing thesis

- According to the Church–Turing thesis, computable functions are exactly the functions that can be calculated using a mechanical calculation device given unlimited amounts of time and storage space.

- Equivalently, this thesis states that any function which has an algorithm is computable.

- Note that an algorithm in this sense is understood to be a sequence of steps a person with unlimited time and an infinite supply of pen and paper could follow.

# 邱奇-图灵论题（Church–Turing thesis）

- Several independent attempts were made in the first half of the 20th century to formalize the notion of computability:
  - American mathematician Alonzo Church created a method for defining functions called the λ-calculus,
  - British mathematician Alan Turing created a theoretical model for machines, now called Turing machines, that could carry out calculations from inputs,
  - Austrian-American mathematician Kurt Gödel, with Jacques Herbrand, created a formal definition of a class of functions whose values could be calculated by recursion.
- All three computational processes (recursion, the λ-calculus, and the Turing machine) were shown to be equivalent

# Equivalent Models

- The class of computable functions can be defined in many equivalent models of computation, including
  - Turing machines
  - μ-recursive functions
  - Lambda calculus
  - Post machines (Post–Turing machines and tag machines).
  - Register machines

有没有注意到重要的假设

…, given unlimited amounts of time and storage space

可计算函数不一定实际可计算

# Outline

- 可计算函数
- 不可计算函数
  - 罗素悖论
  - 停机问题
  - 哥德尔不完备性定理
- 图灵机
- Register Machine

# 不可计算函数

# 理发师悖论

- 在某个城市中有一位理发师，他的广告词是这样写的：“本人的理发技艺十分高超，誉满全城。我将为本城所有不给自己刮脸的人刮脸，我也只给这些人刮脸。我对各位表示热诚欢迎！”来找他刮脸的人络绎不绝，自然都是那些不给自己刮脸的人。

- 可是，有一天，这位理发师从镜子里看见自己的胡子长了，他本能地抓起了剃刀，你们看他能不能给他自己刮脸呢?

- 如果他不给自己刮脸，他就属于“不给自己刮脸的人”，他就要给自己刮脸，而如果他给自己刮脸呢？他又属于“给自己刮脸的人”，他就不该给自己刮脸。于是产生矛盾。

# 罗素悖论

- 设命题函数P(x)表示“x∉x”，现假设由性质P确定了一个类A——也就是说“A={x|x ∉ x}”。
- 那么现在的问题是：
  - A∈A是否成立？
    - 首先，若A∈A，则A是A的元素，那么A不具有性质P，由命题函数P知A∉A；
    - 其次，若A∉A，也就是说A具有性质P，而A是由所有具有性质P的类组成的，所以A∈A。

# 停机问题

- 不存在这样一个程序（算法），它能够计算任何程序（算法）在给定输入上是否会结束（停机）。

# 证明停机问题 - 1

- 那么，如何来证明这个停机问题呢？反证。假设我们某一天真做出了这么一个极度聪明的万能算法（就叫God_algo吧），你只要给它一段程序（二进制描述），再给它这段程序的输入，它就能告诉你这段程序在这个输入上会不会结束（停机），我们来编写一下我们的这个算法吧：
- bool God_algo(char* program, char* input)
- {

  if(<program> halts on <input>)

  return true;

  return false;
- }
- 这里我们假设if的判断语句里面是你天才思考的结晶，它能够像上帝一样洞察一切程序的宿命。

# 证明停机问题 - 2

- 现在，我们从这个God_algo出发导出一个新的算法：
- bool Satan_algo(char* program)
- {

```
    if( God_algo(program, program) ){
        while(1); // loop forever!
        return false; // can never get here!
    }
    else
        return true;
```

- }
-

# 证明停机问题 - 3

- 正如它的名字所暗示的那样，这个算法便是一切邪恶的根源了。当我们把这个算法运用到它自身身上时，会发生什么呢?
  - Satan_algo(Satan_algo);
- 我们来分析一下这行简单的调用：
- 总之，我们有：
  - Satan_algo(Satan_algo)能够停机 => 它不能停机
  - Satan_algo(Satan_algo)不能停机 => 它能够停机
- 所以它停也不是，不停也不是。左右矛盾。
- 于是，我们的假设，即God_algo算法的存在性，便不成立了。正如拉格朗日所说：“陛下，我们不需要（上帝）这个假设”。

# 补充阅读：Lambdar演算的证明

Now we show a simple contradiction, which proves that the magical solver `Halting` cannot really exist. This question is: Does the following expression `E` returns `True` or `False`?

- E = Halting(λm.not(Halting(m,m)), λm.not(Halting(m,m)))

It turns out that this question cannot be answered. If `E` returns True, then we apply the function `λm.not(Halting(m,m))` to its argument `λm.not(Halting(m,m))`, and we get

```
not(Halting(λm.not(Halting(m,m)), λm.not(Halting(m,m))))
```

Alas, this is exactly the negation of the original expression `E`, which means `E` should be False. This is a contradiction (or call it a “paradox” if you like), which shows that the halting problem solver `Halting` cannot exist, which means that the halting problem cannot be solved.

# 哥德尔不完备性定理

- 任何相容的形式系统，只要蕴涵皮亚诺算术公理，就可以在其中构造在体系中既不能证明也不能否证的命题（即体系是不完备的）。

- 任何相容的形式系统，只要蕴涵皮亚诺算术公理，它就不能用于证明它本身的相容性。

# 哥德尔不完备性定理的证明 - 1

- 要证明哥德尔的不完备性定理，只需在假定的形式系统T内表达出一个为真但无法在T内推导出（证明）的命题。于是哥德尔构造了这样一个命题，用自然语言表达就是：
  - 命题P说的是“P不可在系统T内证明”（这里的系统T当然就是我们的命题P所处的形式系统了），也就是说“我不可以被证明”，跟著名的说谎者悖论非常相似，只是把“说谎”改成了“不可以被证明”。我们注意到，一旦这个命题能够在T内表达出来，我们就可以得出“P为真但无法在T内推导出来”的结论，从而证明T的不完备性。为什么呢？我们假设T可以证明出P，而因为P说的就是P不可在系统T内证明，于是我们又得到T无法证明出P，矛盾产生，说明我们的假设“T可以证明P”是错误的，根据排中律，我们得到T不可以证明P，而由于P说的正是“T不可证明P”，所以P就成了一个正确的命题，同时无法由T内证明！
- 如果你足够敏锐，你会发现上面这番推理本身不就是证明吗？其证明的结果不就是P是正确的？然而实际上这番证明是位于T系统之外的，它用到了一个关于T系统的假设“T是一致（无矛盾）的”，这个假设并非T系统里面的内容，所以我们刚才其实是在T系统之外推导出了P是正确的，这跟P不能在T之内推导出来并不矛盾。所以别担心，一切都正常。
- 那么，剩下来最关键的问题就是如何用形式语言在T内表达出这个P

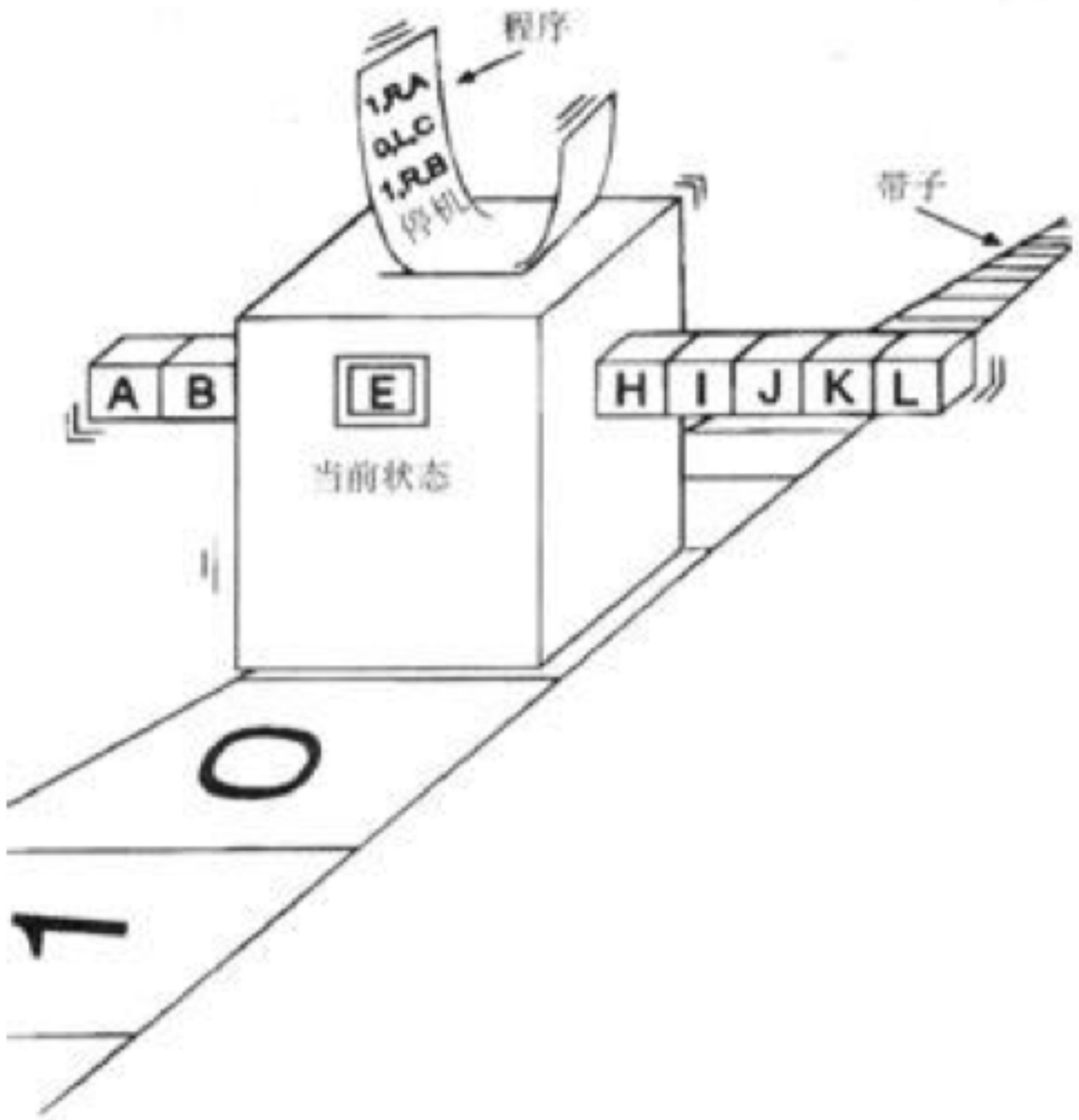
# 哥德尔不完备性定理的证明 - 2

- 哥德尔构造了这样一个公式：
  - N(n) is unprovable in T
- 我们用UnPr(X)来表达“X is unprovable in T”，于是哥德尔的公式变成了：
    - UnPr( N(n) )
- 现在，到了最关键的部分，首先我们把这个公式简记为G(n)——别忘了G内有一个自由变量n，所以G现在还不是一个命题，而只是一个公式，所以谈不上真假：
  - G(n): UnPr( N(n) )
- 又由于G也是个wff(well-formed formula)，所以它也有自己的编码g，当然g是一个自然数，现在我们把g作为G的参数，也就是说，把G里面的自由变量n替换为g，我们于是得到一个真正的命题：
  - G(g): UnPr( G(g) )
- 用自然语言来说，这个命题G(g)说的就是“我是不可在T内证明的”。看，我们在形式系统T内表达出了“我是不可在T内证明的”这个命题。而我们一开始已经讲过了如何用这个命题来推断出G(g)为真但无法在T内证明，于是这就证明了哥德尔的不完备性定理

# Outline

- 可计算函数
- 不可计算函数
- 图灵机
- Register Machine

# 图灵机




| 当前内部状态 s | 输入数值 i | 输出动作 o | 下一时刻的内部状态 s' |
|---|---|---|---|
| B | 1 | 前移 | C |
| A | 0 | 往纸带上写 1 | B |
| C | 0 | 后移 | A |
| … | … | … | … |

# 图灵机的定义

一台图灵机是一个七元组$(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$，其中$Q, \Sigma, \Gamma$都是有限集合，且满足

1. $Q$是状态集合；
2. $\sum$是输入字母表，其中不包含特殊的空白符□；
3. $b \in \Gamma$为*空白符*；
4. $\Gamma$是带字母表，其中$\square \in \Gamma$且$\Sigma \subset \Gamma$；
5. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$是转移函数，其中$L, R$表示读写头是向左移还是向右移；
6. $q_0 \in Q$是起始状态；
7. $q_{accept} \in Q$是接受状态。$q_{reject} \in Q$是拒绝状态，且$q_{reject} \neq q_{accept}$。

# 图灵机程序例子 - 3+2

设$M = (\{0,1,10,11\},\{0,1\},\{0,1,\Box\},\delta,0,,)$和$\delta:\{0,1,10,11\}\times\{0,1\}\rightarrow\{0,1,10,11\}\times\{0,1\}\times\{R,L,E,S\}$. 比如做一个以1的个数表示数值的加法运算，在磁带上的数据是0000001110110000，就是3+2的意思。程序$\delta$如下：

0,0 –> 0,0R

0,1 –> 1,1R

1,0 –> 10,1R

1,1 –> 1,1R

10,0 –> 11,0L

10,1 –> 10,1R

11,0 –> E

11,1 –> 0,0S

虽然这里给出与上面不同形式的定义，但两者是等价的，这里的定义能完成的工作并不比上面的定义多。

| 步数 | 状态 | 磁带 | 步数 | 状态 | 磁带 |
|---|---|---|---|---|---|
| 1 | 0 | 0000001110110000 | 9 | 1 | 0000001110110000 |
| 2 | 0 | 0000001110110000 | 10 | 1 | 0000001110110000 |
| 3 | 0 | 0000001110110000 | 11 | 10 | 0000001111110000 |
| 4 | 0 | 0000001110110000 | 12 | 10 | 0000001111110000 |
| 5 | 0 | 0000001110110000 | 13 | 10 | 0000001111110000 |
| 6 | 0 | 0000001110110000 | 14 | 11 | 0000001111110000 |
| 7 | 0 | 0000001110110000 | 15 | 0 | 0000001111100000（停机） |
| 8 | 1 | 0000001110110000 | -- 停机 -- | | |

# 图灵机程序例子 - f(x)=x+1

- q1,0,1,l,q2;
- q1,1,0,l,q3;
- q1,b,b,n,q4;
- q2,0,0,l,q2;
- q2,1,1,l,q2;
- q2,b,b,n,q4;
- q3,0,1,l,q2;
- q3,1,0,l,q3;
- q3,b,b,n,q4.

- 五元组（q1,,s1,s2,r,q2)分别表示：
- q1:当前状态
- s1:读写头从当前读入的数据（0或者1）
- s2:读写头即将写入当前方格的数据
- r/l/n:读写头向右移动一格/向左移动一格/保持不动
- q2:新状态
- 读写头一开始位于数据最右边一位，b表示空格，q1为初始状态，q4为结束状态。

# 图灵机程序例子 - f(x)=2^x

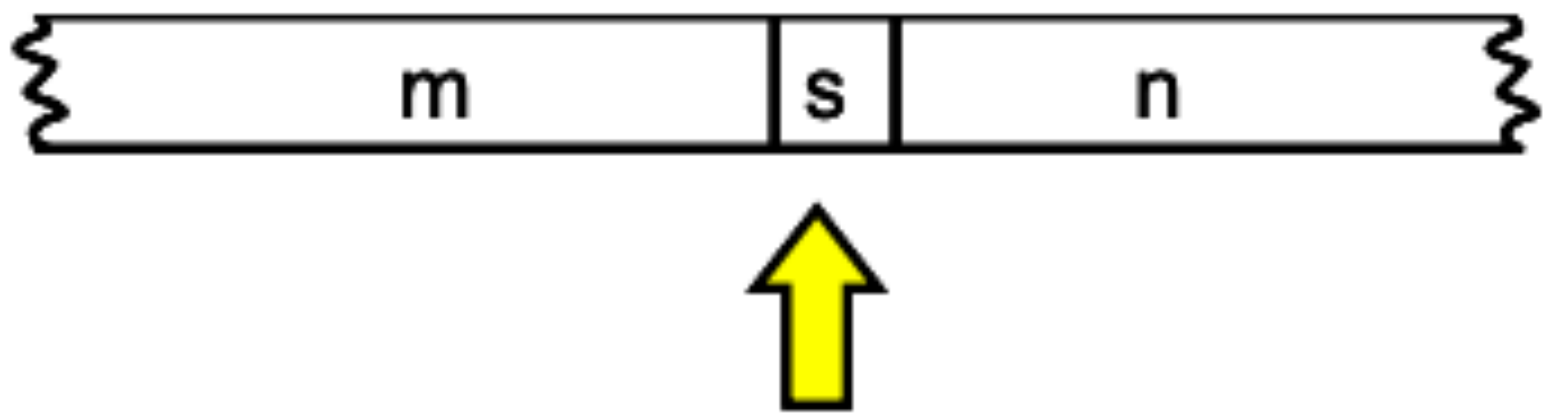
| 当前状态 \ 读入数据 | B | 0 | 1 |
|---|---|---|---|
| q1 | 1, L, q7 | 0, R, q1 | 1, R, q2 |
| q2 | B, R, q3 | 0, R, q2 | 1, R, q2 |
| q3 | 0, L, q4 | 0, R, q3 | Error |
| q4 | B, L, q5 | 0, L, q4 | Error |
| q5 | Error | 1, L, q5 | 0, L, q6 |
| q6 | B, R, q1 | 0, L, q6 | 1, L, q6 |
| q7 | Halt | B, L, q7 | Error |

# 另一个例子，是实现f(x)=2^x的

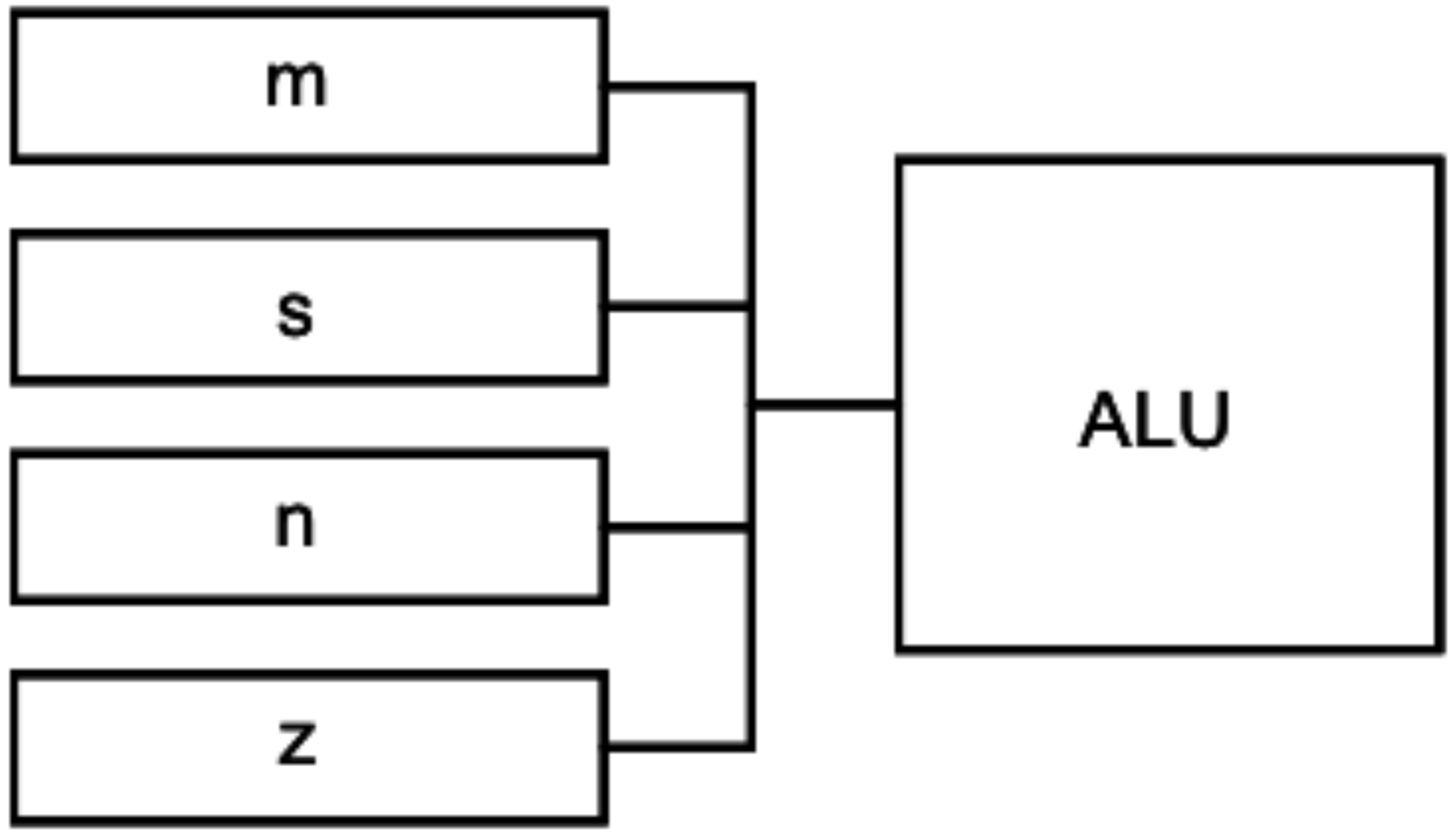
- 约定:
  - 1.开始时，纸带上只有一连续的方格串上放入相应于x的二进制值的符号，其余方格均为空白（用B表示）；
  - 2.读写头一开始位于表示x的方格的最左边一位所在方格；
  - 3.停机时，纸带上非空方格串所组成的二进制值即为所求结果。
- 程序的实现思想:
  - 我们知道，在二进制表示下，只要在原数后面添上一个0，就是原来的数乘以二。根据这个思想，我们每次在写一个0，同时原数减一，直到原数减为0，再在所写的0前面添加上一个1，就能得出所求函数的答案了。
- 状态含义:
  - q1:起始状态；忽略前导0，寻找该数的真正起始位置；如果出现_00……00_0…… 00的情况，则转入q7；
  - q2:读写头向右移动直到遇到空格；
  - q3: 此空格后0的数目加1；
  - q4:回到原来数的最右端；
  - q5: 该数减1；
  - q6: 回到数的开头，转入q1；
  - q7:结束状态。
  - 可以看到，q1到q7,7个状态，相当于一个个标志，而整个程序也相当于是一句句的goto语句。

# Outline

- 可计算函数
- 不可计算函数
- 图灵机
- Register Machine

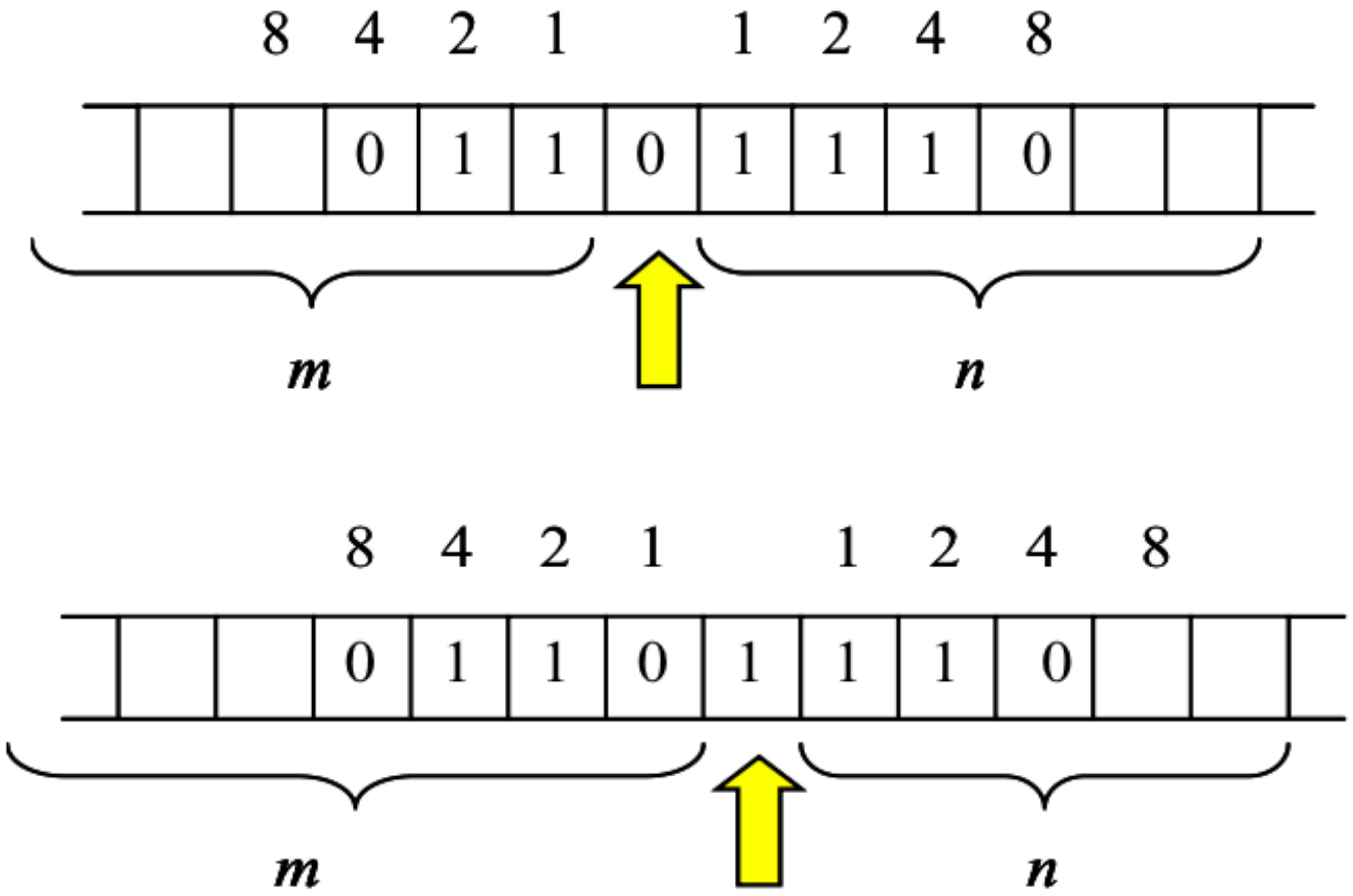



(a)




(b)

图灵机 对应 寄存器机

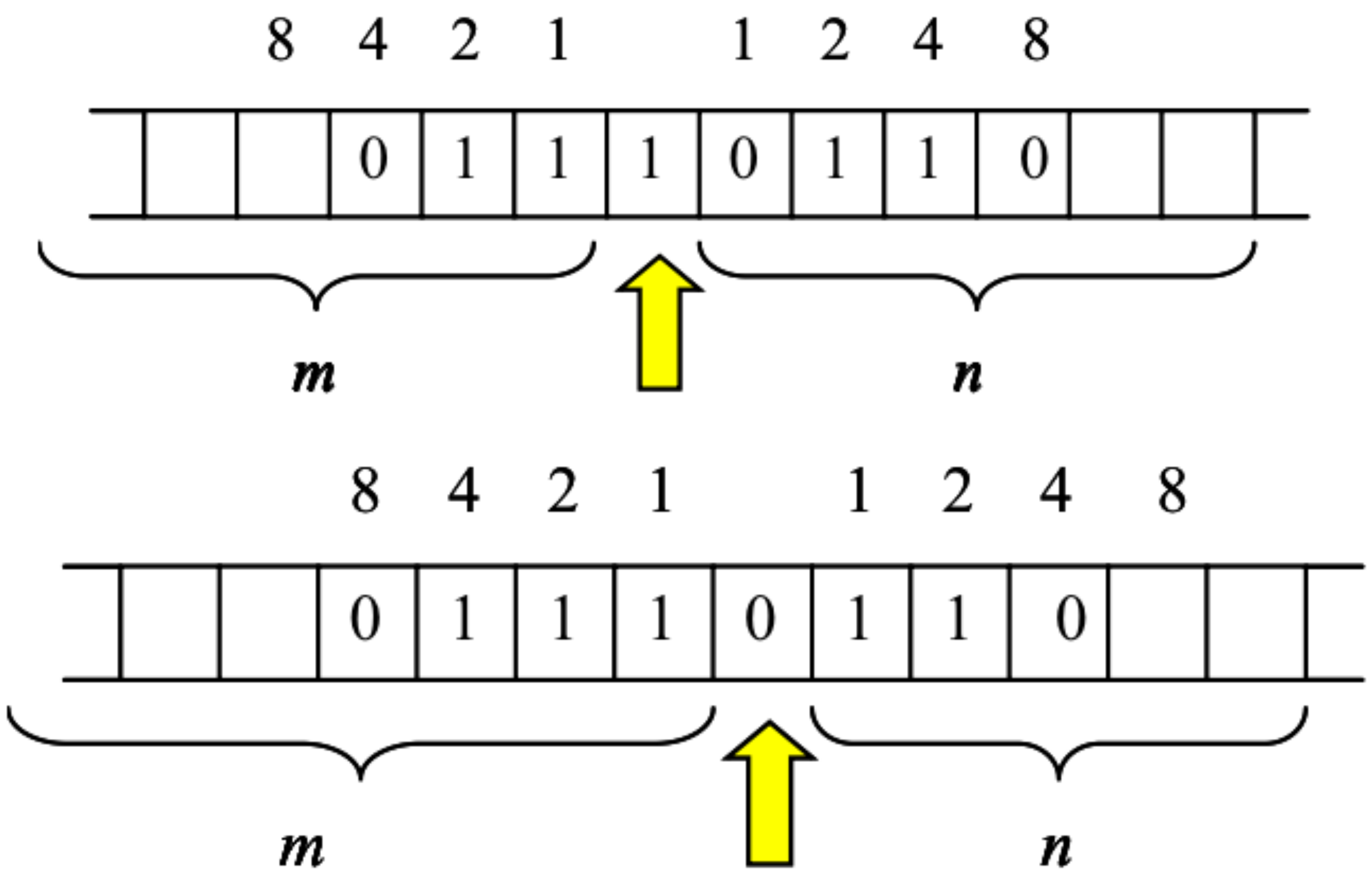



***m' = 2 x m*** 6 = 2 x 3

***n' = n/2 (whole number division)*** 3 = 7/2 (remainder 1)

***s' = remainder n/2*** remainder 7/2 = 1

读写头右移 s=0




***n' = n/2*** ***(whole number)*** 3 = 6/2 (remainder 0)

***s' = remainder n/2*** remainder 6/2 = 0

***m' = 2 x m + s*** 7 = 2 x 3 + 1

# 右移 s=1

# 右移

- We have discovered that the movement of the read/write head to the right on a Turing Machine tape is equivalent to the following three basic arithmetic calculations.
- $$s^* = rem(n/2)$$
$$m^* = s + 2m$$
$$n^* = n/2$$

| | |
|---|---|
| `mov reg,0` | Put 0 into register “reg” |
| `inc reg` | Add 1 to the contents of register “reg” |
| `decjmpreg reg,lab` | If register “reg” is zero, jump to the instruction labelled “lab” else subtract 1 from “reg” and do the next instruction |
| `jmp lab` | Jump to the instruction labelled “lab” |
| `hlt` | Halt |

# A Set of Minimal Instructions

| | | eax | ebx | ecx | edx | eax | ebx | ecx | edx | eax | ebx | ecx | edx |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | mov ecx,3 | | | 3 | | | | | | | | | |
| | mov ebx,0 | | 0 | | | | | | | | | | |
| L3: | decjmpreg ecx,L4 | | | 2 | | | | 1 | | | | 0 | |
| | inc ebx | | 1 | | | | 3 | | | | 5 | | |
| | inc ebx | | 2 | | | | 4 | | | | 6 | | |
| | jmp L3 | | | | | | | | | | | | |
| L4: | hlt | | | | | | | | | | | | |

$$m^* = s + 2m$$

```
        mov ecx,5
        mov eax,0
L3:     mov ebx,0
        decjmpreg ecx,L4
        inc ebx
        decjmpreg ecx,L4
        inc eax
        jmp L3
L4:     hlt
```

| ax | bx | cx | dx |
|---|---|---|---|
| | | 5 | |
| 0 | | | |
| 0 | | | |
| | | 4 | |
| | 1 | | |
| | | 3 | |
| 1 | | | |
| | | | |
| | | | |

| ax | bx | cx | dx |
|---|---|---|---|
| | | | |
| | | | |
| | 0 | | |
| | | 2 | |
| | 1 | | |
| | | 1 | |
| 2 | | | |
| | | | |
| | | | |

| ax | bx | cx | dx |
|---|---|---|---|
| | | | |
| | | | |
| | 0 | | |
| | | 0 | |
| | 1 | | |
| | | | |
| | | | |
| | | | |
| | | | |

$$s^* = \mathrm{rem}(n/2)$$

$$n^* = n/2$$

eax holds m
ebx holds s
ecx holds n
edx holds z

```
L2:     decjmpreg eax,L1
        inc edx
        inc edx
        jmp L2
```

*z = 2m*

```
L1:     decjmpreg ebx,L3
        inc edx
```

*If s is 1 z = z + 1*

```
L3:     decjmpreg edx,L4
        inc eax
        jmp L3
```

*m* = z*

```
L4:     mov ebx,0
        decjmpreg ecx,L6
        inc ebx
        decjmpreg ecx,L6
        inc edx
        jmp L4
```

*s* = rem(n/2)*
*z = n/2 whole*

```
L6:     decjmpreg edx,L7
        inc ecx
        jmp L6
```

*n* = z*

```
L7:     hlt
```

State
Even

```
read the symbol s
store s
if(s == 0), set s = 0 and
jmp to move-right code
if(s == 1), set s = 0 and
jmp to move-right code
```

| Current State *S* | Symbol read *s* | Symbol to write *s** | Direction to move *d* | New state *S** |
|---|---|---|---|---|
| *S0* | *s1* | *s*1* | *R* | *S1* |
| *S0* | *s2* | *s*2* | *L* | *S1* |
| *S0* | *s3* | *s*3* | *R* | *S2* |

Head move left code

Head move right code

| Current State *S* | Symbol read *s* | Symbol to write *s** | Direction to move *d* | New state *S** |
|---|---|---|---|---|
| *Even* | *0* | *0* | *R* | *Even* |
| *Even* | *1* | *0* | *R* | *Odd* |
| *Even* | *@* | *0* | *N* | *Halt* |

```
use the stored s
if(s == 0), jump to the
even state code
if(s == 1), jump to the
odd state code
```

# 状态的转移

# 只用两条指令

- L2: decjmpreg eax,L1
- inc edx
- inc edx
- ; jmp L2
- decjmpreg esi, L2
- ;
- L1: decjmpreg ebx,L3
- inc edx
- ;
- L3: decjmpreg edx,L4
- inc eax
- ; jmp L3
- decjmpreg esi, L3
- ;
- L4: mov ebx,0
- decjmpreg ecx,L6
- inc ebx
- decjmpreg ecx,L6
- inc edx
- ; jmp L4
- decjmpreg esi,L4
- ;
- L6: decjmpreg edx,L7
- inc ecx
- ; jmp L6
- decjmpreg esi, L6
- ;
- L7: hlt

# 只用两个寄存器

- register r
  - Then we Godelize these numbers (3D-space) as above to produce a single number r which we put into a single register r, like this:
  - $r = GS(a,b,c) = 2^a.3^b5^c$
- register z
  - the need to have a register with initial value zero. Let' call this register z

Theorem 1.

For every Turing Machine T there exists a register-based machine R which exhibits the same behaviour as T. This register-based machine consists of two registers and two instructions.