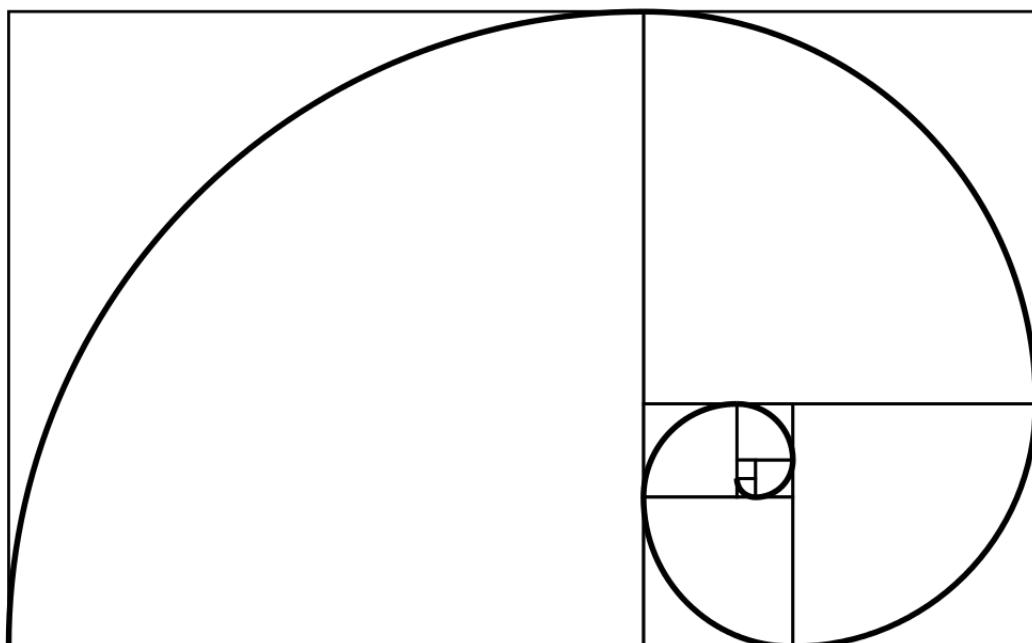


# 编程中的数学



刘新宇<sup>1</sup>

May 3, 2019

<sup>1</sup>刘新宇  
Version: 0.6180339887498949  
Email: liuxinyu95@gmail.com



# 目录

<b>1 自然数</b>	<b>7</b>
1.1 数的诞生 . . . . .	7
1.2 皮亚诺自然数公理 . . . . .	8
1.3 自然数和计算机程序 . . . . .	9
1.4 自然数的结构 . . . . .	11
1.5 自然数的同构 . . . . .	15
1.6 形式与结构 . . . . .	18
<b>2 递归</b>	<b>21</b>
2.1 万物皆数 . . . . .	21
2.2 欧几里得算法 . . . . .	23
2.2.1 欧几里得和《几何原本》 . . . . .	24
2.2.2 欧几里得算法 . . . . .	24
2.2.3 扩展欧几里得算法 . . . . .	26
2.2.4 欧几里得算法的意义 . . . . .	30
2.3 $\lambda$ 演算 . . . . .	32
2.3.1 表达式化简 . . . . .	33
2.3.2 $\lambda$ 抽象 . . . . .	34
2.3.3 $\lambda$ 变换规则 . . . . .	34
2.4 递归的定义 . . . . .	37
2.4.1 Y组合子 . . . . .	38
2.5 $\lambda$ 演算的意义 . . . . .	39
2.6 更多的递归结构 . . . . .	40
2.7 递归的形式与结构 . . . . .	41
2.8 扩展阅读 . . . . .	42
<b>3 群、环、域</b>	<b>45</b>
3.1 群 . . . . .	47
3.1.1 群的定义 . . . . .	50
3.1.2 半群与半群 . . . . .	52
3.1.3 群的性质 . . . . .	54
3.1.4 置换群 . . . . .	58
3.1.5 循环群 . . . . .	60
3.1.6 子群 . . . . .	62
3.1.7 拉格朗日定理 . . . . .	66
3.2 环与域 . . . . .	75
3.2.1 环的定义 . . . . .	76
3.2.2 除环和域 . . . . .	77

3.3 伽罗瓦理论 . . . . .	78
3.3.1 扩域 . . . . .	79
3.3.2 自同构和伽罗瓦群 . . . . .	80
3.3.3 伽罗瓦基本定理 . . . . .	81
3.3.4 可解性 . . . . .	83
3.4 扩展阅读 . . . . .	86
<b>4 范畴论</b>	<b>87</b>
4.1 范畴 . . . . .	89
4.1.1 范畴的例子 . . . . .	91
4.1.2 箭头≠函数 . . . . .	94
4.2 函子 . . . . .	95
4.2.1 函子的定义 . . . . .	95
4.2.2 函子的例子 . . . . .	96
4.3 积和余积 . . . . .	101
4.3.1 积和余积的定义 . . . . .	103
4.3.2 积和余积的性质 . . . . .	105
4.3.3 积和余积作为函子 . . . . .	107
4.4 自然变换 . . . . .	109
4.4.1 自然变换的例子 . . . . .	110
4.4.2 自然同构 . . . . .	113
4.5 数据类型 . . . . .	114
4.5.1 起始对象和终止对象 . . . . .	114
4.5.2 幂 . . . . .	118
4.5.3 笛卡尔闭和对象算术 . . . . .	122
4.5.3.1 0次幂 . . . . .	122
4.5.3.2 1的幂 . . . . .	123
4.5.3.3 1次幂 . . . . .	123
4.5.3.4 幂的和 . . . . .	123
4.5.3.5 幂的幂 . . . . .	124
4.5.3.6 积的幂 . . . . .	124
4.5.4 多项式函子 . . . . .	124
4.5.5 F-代数 . . . . .	125
4.5.5.1 递归和不动点 . . . . .	129
4.5.5.2 初始代数和向下态射 . . . . .	130
4.5.5.3 代数数据类型 . . . . .	134
4.6 小节 . . . . .	136
4.7 扩展阅读 . . . . .	138
4.8 附录：例子代码 . . . . .	138
<b>5 推导</b>	<b>141</b>
5.1 叠加——构建的融合 . . . . .	142
5.1.1 列表的叠加操作 . . . . .	143
5.1.2 叠加——构建融合律 . . . . .	144
5.1.3 列表的构建形式 . . . . .	145
5.1.4 使用融合律化简 . . . . .	146
5.1.5 类型限制 . . . . .	148
5.1.6 用范畴论推导融合律 . . . . .	148
5.2 巧算100 . . . . .	150

目录	5
5.2.1 穷举法 . . . . .	151
5.2.2 改进 . . . . .	152
5.3 小节和扩展阅读 . . . . .	154
5.4 附录代码 . . . . .	154
6 无穷 . . . . .	157
6.1 无穷概念的提出 . . . . .	159
6.1.1 无穷的哲学 . . . . .	161
6.1.2 穷竭法与微积分 . . . . .	162
6.2 潜无穷与编程 . . . . .	165
6.2.1 余代数和无穷流★ . . . . .	166
6.3 实无穷的思考 . . . . .	169
6.3.1 无穷王国的花园 . . . . .	170
6.3.2 一一对应与无穷集合 . . . . .	172
6.3.2.1 康托尔与戴德金 . . . . .	174
6.3.2.2 利用（可数）无穷定义斐波那契数列和哈明数列 . . . . .	176
6.3.3 可数无穷与不可数无穷 . . . . .	177
6.3.4 戴得金分割 . . . . .	179
6.3.5 超限数和连续统假设 . . . . .	181
6.3.5.1 超限数 . . . . .	181
6.3.5.2 连续统假设 . . . . .	183
6.4 无穷与艺术 . . . . .	184
6.5 附录：例子代码 . . . . .	188
6.6 附录：康托尔定理的证明 . . . . .	189
6.7 附录：巴赫《音乐的奉献》无限上升的卡农 . . . . .	189
7 悖论 . . . . .	191
7.1 计算的边界 . . . . .	192
7.2 罗素悖论 . . . . .	194
7.2.1 罗素悖论的影响 . . . . .	196
7.3 数学基础的分歧 . . . . .	197
7.3.1 逻辑主义 . . . . .	197
7.3.2 直觉主义 . . . . .	198
7.3.3 形式主义 . . . . .	199
7.3.4 公理集合论 . . . . .	200
7.4 哥德尔不完全性定理 . . . . .	202
7.5 不完全性定理的证明 . . . . .	204
7.5.1 构建形式系统 . . . . .	204
7.5.1.1 公理和推理规则 . . . . .	205
7.5.1.2 印符系统的不完全性 . . . . .	206
7.5.2 哥德尔配数 . . . . .	207
7.5.3 构造自我指涉 . . . . .	208
7.6 万能的程序与对角线证明 . . . . .	208
7.7 尾声 . . . . .	209

## I 附录 211

## Appendices

.1 加法交换律的证明 . . . . .	213
.2 倒水趣题完整程序 . . . . .	214
.3 积和余积的唯一性 . . . . .	214
.4 集合的笛卡尔积和不相交并集构成积和余积的证明 . . . . .	215

# 第1章 自然数

数能引导我们走向真理

——柏拉图写给老师苏格拉底的信

## 1.1 数的诞生

自从人类进化开始，数的概念就伴随着我们。有人认为，数直接催生了人类的语言和文字。我们的祖先在长期的狩猎——采集活动中，逐渐掌握了数的概念。最早可能是简单的数数，例如数清果实的数量。随着文明的发展，人们逐渐开始进行物品交易。随着交易物品数量增长，很快需要采用助记工具来处理更大的数。考古发现在今天的伊朗一带，人们在公元前4000年左右开始使用陶制的小球来辅助计数。比如用两个刻有十字的小球代表两只羊，同时还有代表十只羊，二十只羊的不同小球。为了防止忘记或者篡改数过的数目，人们还把这些小球放在陶罐中用泥土封存起来。图1.1是乌鲁克时期的陶制计数罐和小球<sup>[1]</sup>。交易过程中，人们可以通过这些工具掌握货物的数量<sup>[2]</sup>。

然而随着交易的增加和数目的增大，这样的陶罐和小球就不够方便了。大约公元前3500年，美索不达米亚的苏美尔人开始在泥板上刻划符号来记录交易。将泥板烤硬后就可以方便的保存记录。人们在这一时期用坚硬的笔在泥板上刻出不同的符号，同时表示交易的物品和数量。比如用一个象形符号表示五头牛，而用另一个象形符号表示十只羊。

数的最大进步发生在公元前3100年左右，从出土的泥板中，我们发现苏美尔人开始将数字从它代表的物品中抽象出来。人们不再使用一个符号同时表示物品和数量，而是用一个符号表示交易的数量，接下来用另一个符号表示交易的物品。例如先用一个符号表示五，然后跟上一个牛的象形符号表示五头牛；而表示五只羊的时候，人们用同样的符号表示五，然后再跟上一个羊的符号。这些泥板上的符号逐渐演变成了古巴比伦的楔形文字。

产生抽象的数是智慧生命思维的结果。人们发现三个鸡蛋、三棵树、三个陶罐都可以用数字三来表示。这是一种强大的工具。从此，我们可以对抽象的数字进行操作，然后再把结果应用到各种具体事物上。例如我们可以把抽象的数字三加上一得到四，从而知道捡拾三个鸡蛋后再捡拾到一个鸡蛋会得到四个鸡蛋。同时我们也知道烧制三个陶罐后再烧制一个陶罐会得到四个陶罐。人们逐渐从解决单一问题发展到解决一类问题。

生产劳动中从数数开始，我们的老祖先逐渐发展出了操作抽象数字的方法，包括数字的加法、减法，更为强大的乘法，以及用于分配事物的除法。在丈量分割土地，计算谷物容量时，又逐渐

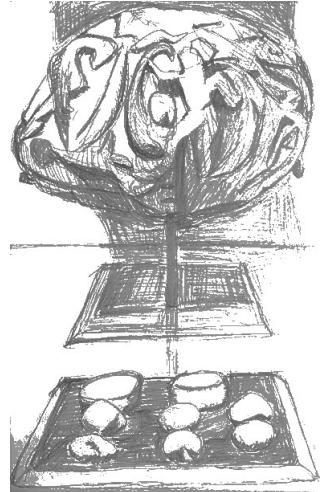
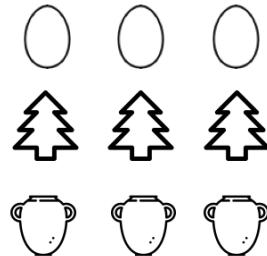


图 1.1：卢浮宫陈列的乌鲁尔时期的计数陶罐和一组计数陶球。

1	Y	11	<Y	21	<<Y	31	<<<Y	41	<<<<Y	51	<<<<<Y
2	W	12	<W	22	<<W	32	<<<W	42	<<<<W	52	<<<<<W
3	WV	13	<WV	23	<<WV	33	<<<WV	43	<<<<WV	53	<<<<<WV
4	V	14	<V	24	<<V	34	<<<V	44	<<<<V	54	<<<<<V
5	WV	15	<WV	25	<<WV	35	<<<WV	45	<<<<WV	55	<<<<<WV
6	WV	16	<WV	26	<<WV	36	<<<WV	46	<<<<WV	56	<<<<<WV
7	V	17	<V	27	<<V	37	<<<V	47	<<<<V	57	<<<<<V
8	V	18	<V	28	<<V	38	<<<V	48	<<<<V	58	<<<<<V
9	WV	19	<WV	29	<<WV	39	<<<WV	49	<<<<WV	59	<<<<<WV
10	<	20	<	30	<<	40	<<<	50	<<<<		

(a) 巴比伦楔形文字中的数字[3]

3



(b) 抽象的数字三

将抽象的数和几何量联系起来。各个文明几乎分别独立地发现了数与形的内在规律。我们发现古埃及、古希腊、古中国都各自发现了毕达哥拉斯定理（勾股定理），古埃及人把它用于金字塔建造这样的伟大工程实践。从现代文明追根溯源，我们可以说自然数是数学和自然科学这条长河的源头。德国数学家克罗内克说“上帝创造了自然数，其余都是人的工作。”<sup>1</sup>

## 1.2 皮亚诺自然数公理

古希腊的欧几里得在他的伟大著作《几何原本》中开创了公理化的方法。他用五条公理和五条公设作为基石，精心构建一条一条定理的证明。每一个结论都仅仅使用公理和此前已经证明的定理。最终构建出了叹为观止的几何大厦。然而对于自然数，长期以来人们却没有建立起它的公理化形式系统。也许人们一直认为自然数的结论是直观和显而易见的。直到1889年，意大利数学家皮亚诺才为自然数建立起了严格的公理化系统。这就是著名的皮亚诺公理。也许是上帝的巧合，欧几里得几何公理有五条，皮亚诺公理也有五条：

1. 0是自然数。用符号表示为  $\exists 0 \in N$ ;
2. 每个自然数都有它的下一个自然数，称为它的后继。用符号表示为  $\forall n \in N, \exists n' = succ(n) \in N$ ;

似乎仅仅有这两条公理，我们已经能够定义出无穷无尽的自然数了，从0开始，下一个是1，下一个是2，接下来是3，……，接下来是某个  $n$ ，下一个是  $n + 1$ ，……以至无穷。但是好挑剔的数学家给出了一个反例：考虑只有两个元素  $\{0, 1\}$  组成的数字系统，定义0的后继为0，0的后继为1。这样也满足上面的两条公理，却不是我们想像中的自然数。

为此我们还需要第三条皮亚诺公理来排除这种情况。

3. 0不是任何自然数的后继。用符号表示为  $\forall n \in N : n' \neq 0$ ;

仅仅有这三条公理就够了么？我们还可以给出一个反例：考虑有限元素  $\{0, 1, 2\}$  组成的数字系统，定义0的后继是1，1的后继是2，2的后继还是2。这样也能满足上述三条公理。为此我们还需要第四条皮亚诺公理。

4. 不同的自然数有不同的后继数。或者说，如果两个自然数的后继数相同，那么这两个自然数相等。用符号表示为  $\forall n, m \in N : n' = m' \Rightarrow n = m$ ;

<sup>1</sup> 一说为整数，我们会在后继关于康托尔和无穷的章节再次提到克罗内克。

但是，仅仅用这四条公理仍然不够，因为可以存在这样的反例：考虑集合 $\{0, 0.5, 1, 1.5, 2, 2.5, \dots\}$ ，定义0的后继是1、1的后继是2……，0.5的后继是1.5、1.5的后继是2.5……但0.5不是任何元素的后继。为了排除这样的含有“不可达”元素的反例，还需要最后一条皮亚诺公理。

5. 如果自然数的某个子集包含0，并且其中每个元素都有后继元素。那么这个子集就是全体自然数。用符号表示为 $\forall S \subset N : (0 \in S \wedge \forall n \in S \Rightarrow n' \in S) \Rightarrow S = N$ .

为什么公理5可以排除掉上述的反例呢？我们考虑集合 $\{0, 0.5, 1, 1.5, 2, 2.5, \dots\}$ 的一个子集 $\{0, 1, 2, \dots\}$ 。它包含0，并且每个元素都有后继元素，但是它不等于原集合。因为1.5、2.5……都不在这个子集中。所以它不满足第五条公理。公理5还有另外一个响亮的名字——归纳公理，它可以这样等价地描述：

5. 任意关于自然数的命题，如果证明了它对自然数0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）

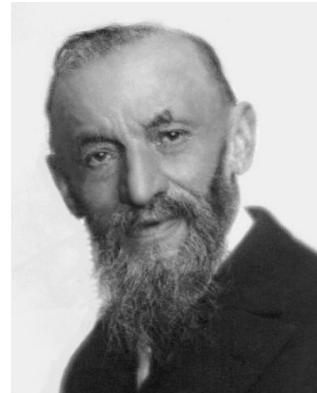
以上就是完整的五条皮亚诺公理，用它们可以构建出一阶算术系统，也称为皮亚诺算术系统<sup>2</sup>。

朱塞佩·皮亚诺1858年8月27日生于意大利库内奥（Cuneo）附近的斯宾尼塔（Spinetta）村。这是位于意大利北部都灵附近的农村。他出生的时候，正值意大利统一。1876年皮亚诺考入都灵大学学习，1880年毕业后就留校任教。皮亚诺最开始讲授微积分课程，1887年他和克罗西奥（Carola Crosio）结婚。1886年起，皮亚诺还同时担任都灵军事学院的教授。从19世纪80年代起，皮亚诺开始研究数理逻辑，并致力于数学基础的构建工作。他撰写了《数学公式汇编》这本巨著，力图把所有的数学成果都用形式化的方法汇集起来。这本书可以说为数学的严密化奠定了基础。1900年在第二届国际数学家大会上，罗素遇到了皮亚诺，他在自传（1951）中说[63]：

“这次大会是我的精神生活的一个转折点，因为在那我遇到了皮亚诺。在此之前，我已经听说过他的名字，也知道他的一些工作。我突然明白了，他的符号提供了我多年来一直试图寻找的分析的工具，而且从他那里我获得了一直以来想要从事的工作的一种新的有效技术。”

皮亚诺强烈地影响了罗素和怀特海合著的《数学原理》一书，对早起的计算理论起了重要的作用。皮亚诺最初用法语发表研究著作，但他对自然语言固有的歧义感到不满。为了解决这个问题，他于1900年左右发明了一套没有歧义的同于语言，称为“无屈折拉丁语”。这门语言后来被称为“国际语”<sup>3</sup>。皮亚诺努力推广他的新语言，但是事实并不如他所愿，几乎没有人愿意读他用国际语重写的《数学公式汇编》。相反倒是他的早期的法语著作使数学家的观点发生了深刻的变化，尤其对法国的布尔巴基学派的纲领，产生了很大影响。后者包含了一大批20世纪世界顶级的数学家，如安德烈·韦伊，亨利·嘉当，舒瓦兹，塞尔，格罗滕迪克，迪厄多内等。

1932年4月20日，皮亚诺因心脏病逝世于都灵。



朱塞佩·皮亚诺 (Giuseppe Peano)  
1858 - 1932。

## 1.3 自然数和计算机程序

现代的计算机系统和在其上构建的程序已经非常的复杂和宏伟了。人们并非是先建立了计算机程序的公理系统，然后逐渐演绎出这些成果的。而是先取得了应用的巨大成功，然后才逐渐将计算

<sup>2</sup>也有人从1开始，而不是0开始计算自然数。在皮亚诺当年的著作中，五条公理的顺序与此不同，其中第五条归纳公理被写在第三的位置上。

<sup>3</sup>国际语的英文为Interlingua，它和世界语（Esperanto）是两种不同的人造语言

机科学的基石数学化、形式化、严谨化的。这种有趣的现象在人类历史上已经不是第一次了。牛顿和莱布尼茨在17世纪发展了微积分，然后在几代数学家的手里应用到了各种领域，包括流体力学，天文学等等。但是直到19世纪才由波尔查诺、阿贝尔、柯西、魏尔斯特拉斯等人将微积分的理论严格化[63]。

我们也模仿一下这样的过程，看看如何根据皮亚诺公理，用计算机程序定义自然数。在一个没有0、1、2、……这些我们熟悉的数字的程序系统中，我们可以这样定义自然数<sup>4</sup>：

```
data Nat = zero | succ Nat
```

这一定义说：一个自然数或者为零，或者是另一自然数的后继。这里符号“|”表示互斥的关系。它自然蕴含了零不是任何自然数后继这一公理。在这一定义下，我们可以进一步定义出自然数的加法。

```
a + zero = a
a + (succ b) = succ (a + b)
```

加法定义包含两部分。首先任何自然数和零相加等于它本身；并且某个自然数和另一个数的后继相加，等于这两个数相加的后继。写成数学符号为：

$$\begin{aligned} a + 0 &= a \\ a + b' &= (a + b)' \end{aligned} \tag{1.1}$$

我们来验证一下 $2+3$ 。自然数2为 $\text{succ}(\text{succ zero})$ ，而3为 $\text{succ}(\text{succ}(\text{succ zero}))$ 。根据加法的定义 $2+3$ 为：

```
succ(succ zero) + succ(succ(succ zero))
= succ(succ(succ zero) + succ(succ zero))
= succ(succ(succ(succ zero) + succ zero))
= succ(succ(succ(succ(succ zero) + zero)))
= succ(succ(succ(succ(succ zero))))
```

最终结果的确是零的五重后继，也就是5。从零开始一次一次地重复使用后继函数的记法很麻烦。如果要表示100，这种记法要写很多行并且容易出错。为此，我们用下面的简单记法表示自然数 $n$ ：

$$n = \text{foldn}(\text{zero}, \text{succ}, n) \tag{1.2}$$

它表示从零开始，不断叠加使用 $\text{succ}$ 函数 $n$ 次。 $\text{foldn}$ 函数可以具体实现如下：

$$\begin{aligned} \text{foldn}(z, f, 0) &= z \\ \text{foldn}(z, f, n') &= f(\text{foldn}(z, f, n)) \end{aligned} \tag{1.3}$$

$\text{foldn}$ 定义了在自然数上的一种操作，只要令 $z$ 为 $\text{zero}$ ，令 $f$ 为 $\text{succ}$ 就可以实现叠加后继若干次，从而获得某个特定的自然数。我们可以用前几个自然数验证一下：

```
foldn(zero, succ, 0) = zero
foldn(zero, succ, 1) = succ(foldn(zero, succ, 0)) = succ zero
foldn(zero, succ, 2) = succ(foldn(zero, succ, 1)) = succ(succ zero)
...
```

定义好加法之后，我们再来定义自然数的乘法：

```
a . zero = zero
a . (succ b) = a . b + a
```

---

<sup>4</sup>本书使用一种理想的计算机语言，并在每一章节的最后给出真实计算机语言的参考代码。

这里我们使用了刚才定义好的加法。这一定义写成数学符号为：

$$\begin{aligned} a \cdot 0 &= 0 \\ a \cdot b' &= a \cdot b + a \end{aligned} \tag{1.4}$$

与通常的观念不同，加法和乘法的交换律、结合律既不是公理，也不是公设。它们都是可以用皮亚诺公理和加法的定义严格证明的定理。我们来看看证明加法结合律的例子。加法结合律是说 $(a + b) + c = a + (b + c)$ 。我们先证明当 $c = 0$ 时它是对的。根据加法定义的第一条规则：

$$\begin{aligned} (a + b) + 0 &= a + b \\ &= a + (b + 0) \end{aligned}$$

然后是递推步骤，假设 $(a + b) + c = a + (b + c)$ 成立，我们要推出 $(a + b) + c' = a + (b + c')$ 。

$$\begin{aligned} (a + b) + c' &= (a + b + c)' && \text{加法定义的规则二, 反向} \\ &= (a + (b + c))' && \text{递推假设} \\ &= a + (b + c)' && \text{加法定义的规则二} \\ &= a + (b + c') && \text{加法定义的规则二, 反向} \end{aligned}$$

这样我们就证明了加法的结合律。但是加法交换律的证明却并不简单，附录一给出了完整的证明。

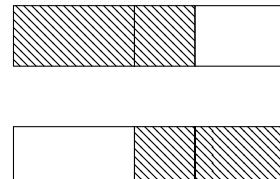


图 1.4: 加法结合律的几何证明。上下面积相等

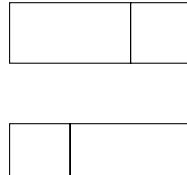


图 1.5: 加法交换律的几何证明。将上方的图形倒过来看，或者在镜中看。

### 练习 1.1

1. 定义0的后继为1，证明对于任何自然数都有 $a \cdot 1 = a$
2. 证明乘法分配律
3. 证明乘法结合律和交换律

## 1.4 自然数的结构

有了加法和乘法，我们就可以定义更复杂的计算。第一个例子是从零开始的累加： $0 + 1 + 2 + \dots$

$$\begin{aligned} \text{sum}(0) &= 0 \\ \text{sum}(n + 1) &= (n + 1) + \text{sum}(n) \end{aligned} \tag{1.5}$$

第二个例子是阶乘 $n!$ 。

$$\begin{aligned} fact(0) &= 1 \\ fact(n+1) &= (n+1) \cdot fact(n) \end{aligned} \tag{1.6}$$

比较这两个例子，我们发现它们非常相似。尽管人工智能日新月异地发展，智慧生命和智能机器的一大区别就是能否“跳出系统”，到更高的层次进行抽象。这是我们人类心智中最强大、神秘、复杂、难以捉摸的一部分[5]。

我们发现累加和阶乘都有一个针对自然数零起始值，累加始自零，阶乘始自一。针对递归情况，它们都将某一操作应用到一个自然数和它的后继上。累加是 $n' + sum(n)$ ，阶乘是 $n' \cdot fact(n)$ 。如果我们把针对零的起始值抽象为 $c$ ，把递归中的操作抽象为 $h$ ，就可以用一个统一的形式概括累加和阶乘。

$$\begin{aligned} f(0) &= c \\ f(n+1) &= h(f(n)) \end{aligned} \tag{1.7}$$

这是一个在自然数上的递归结构。我们观察一下它在前几个自然数上的表现。

$n$	$f(n)$
0	$c$
1	$f(1) = h(f(0)) = h(c)$
2	$f(2) = h(f(1)) = h(h(c))$
3	$f(3) = h(f(2)) = h(h(h(c)))$
...	...
$n$	$f(n) = h^n(c)$

其中， $h^n(c)$ 表示我们叠加在 $c$ 上重复进行 $n$ 次 $h$ 操作。它是原始递归形式的一种 ([6]，第5页)。更进一步，如果观察此前我们在(1.3)定义的函数 $foldn$ ，就会发现它们之间的关系：

$$f = foldn(c, h) \tag{1.8}$$

细心的读者会观察到，我们最初定义的 $foldn$ 带有三个参数，为什么这里只有两个了呢？实际上我们可以写成 $f(n) = foldn(c, h, n)$ 。当我们仅传递给三元函数 $foldn$ 前两个参数，它实际上就成为了接受一个自然数为参数的一元函数了。我们可以这样看待它： $foldn(c, h)(n)$ 。

我们称 $foldn$ 为自然数上的叠加( $fold$ )操作。令 $c$ 为 $zero$ ， $h$ 为 $succ$ ，我们就得到了自然数。如同上面的表格，我们可以得到一个序列：

$$zero, succ(zero), succ(succ(zero)), \dots succ^n(zero), \dots$$

如果 $c$ 不是 $zero$ ， $h$ 不是 $succ$ ，则 $foldn(c, h)$ 就描述了和自然数同构<sup>5</sup>的某种事物。我们来看几个例子：

$$(+m) = foldn(m, succ)$$

这个例子描述了将自然数 $n$ 增加 $m$ 的操作，将它依次作用到自然数上可以产生和自然数同构的序列 $m, m+1, m+2, \dots, n+m, \dots$

$$(\cdot m) = foldn(0, (+m))$$

---

<sup>5</sup>同构(isomorphism)的严格定义将在后继章节给出。本章提到的同构和数学上的意义并不相同。这里特指结构上的一种对应。

这个例子描述了将自然数 $n$ 乘以 $m$ 的操作，将它依次作用到自然数上可以产生和自然数同构的序列 $0, m, 2m, 3m, \dots, nm, \dots$

$$m^0 = foldn(1, (\cdot m))$$

这个例子描述了对自然数 $m$ 取 $n$ 次幂的操作，将它依次作用到自然数上可以产生和自然数同构的序列 $1, m, m^2, m^3, \dots, m^n, \dots$

那么，我们思考出的这个抽象工具 $foldn$ 能否描述累加和阶乘呢？我们观察下面的这个表格：

$n$	0	1	2	3	...	$n'$
$sum(n)$	0	$1 + 0 = 1$	$2 + 1 = 3$	$3 + 3 = 6$	...	$n' + sum(n)$
$n!$	1	$1 \times 1 = 1$	$2 \times 1 = 2$	$3 \times 2 = 6$	...	$n' \cdot (n!)$

这里的关键问题是 $h$ 必须是个二元操作，它能够对 $n'$ 和 $f(n)$ 进行运算。为此，我们将 $c$ 也定义为一个二元数 $(a, b)$ <sup>6</sup>。然后针对二元数 $(a, b)$ 定义某种类似“succ”的操作。最终为了获取结果，我们还需要定义从二元数中抽取 $a$ 和 $b$ 的函数：

$$\begin{aligned} 1st(a, b) &= a \\ 2nd(a, b) &= b \end{aligned} \tag{1.9}$$

这样我们就可以定义累加和阶乘了。首先是累加的定义：

$$\begin{aligned} c &= (0, 0) && \text{二元数的起始值} \\ h(m, n) &= (m', m' + n) && \text{第一个数取后继，第二个数加第一个数的后继} \\ sum &= 2nd \cdot foldn(c, h) \end{aligned}$$

我们看看，从起始值 $(0, 0)$ 开始，会怎样一步一步递推出累加的结果。

$(a, b)$	$(a', b') = h(a, b)$	$b'$
$(0, 0)$	$(0 + 1 = 1, 1 + 0 = 1) = (1, 1)$	1
$(1, 1)$	$(1 + 1 = 2, 2 + 1 = 3) = (2, 3)$	3
$(2, 3)$	$(2 + 1 = 3, 3 + 3 = 6) = (3, 6)$	6
...	...	...
$(m, sum(m))$	$(m + 1, m + 1 + sum(m))$	$sum(m + 1)$

类似地，我们用 $foldn$ 定义出阶乘。

$$\begin{aligned} c &= (0, 1) && \text{阶乘的起始值} \\ h(m, n) &= (m', m'n) && \text{阶乘的递推} \\ fact &= 2nd \cdot foldn(c, h) \end{aligned}$$

在累加和阶乘的定义中，我们使用了符号“ $\cdot$ ”点来连接两个函数 $2nd$ 和 $foldn(c, h)$ 。我们称之为函数组合， $f \cdot g$ 表示先将函数 $g$ 应用到变量上，然后再将函数 $f$ 应用到结果上。即 $(f \cdot g)(x) = f(g(x))$ 。

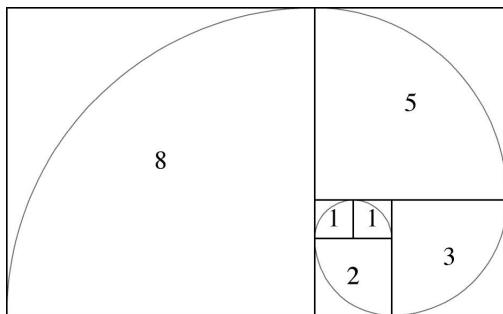
为了展示这一抽象工具的强大，我们再来看一个例子：斐波那契数列。这一数列是用中世纪数学家比萨的列奥纳多命名的。斐波那契来自拉丁文 filius Bonacci 意思是波那契之子。斐波那契的父亲当时是商人，在北非以及地中海一带经商，斐波那契逐渐向当地阿拉伯人学习了印度——阿拉伯的数字系统并通过他的著作《算盘书》（Liber Abaci）将其介绍到欧洲。中世纪的欧洲一直使用罗马数字系统，我们今天在一些钟表盘上仍然可以看到罗马数字。例如2018年的罗马数字表示

<sup>6</sup>在计算机程序中，也称为二元组（tuple），或者对（pair）。

为MMXVIII，其中一个M代表1000，两个M代表2000，X代表10，V表示5，三个I表示3。把这些加起来得到2018。斐波那契引入欧洲的是我们今天仍然使用的位值制十进制系统。它使用了印度数学发明的零，不同的数字在不同的位置上含义不同。这一先进的数字系统极大地方便了计算，广泛应用于记账、利息、汇率等方面。《算盘书》更是对欧洲的数学复兴产生了深远的影响。

斐波那契数列来自《算盘书》中的一个问题：兔子在出生两个月后，就有繁殖能力，一对兔子每个月能生出一对小兔子来。如果所有兔都不死，那么一年以后可以繁殖多少对兔子？开始时有一对兔子。第一个月小兔尚未具备繁殖能力，所以仍然只有一对兔子。第二个月它们生下一对小兔，共有两对。第三个月大兔子又生下一对小兔，而上月生的小兔还在成长，总共有 $2+1=3$ 对。第四个月有两对大兔子产下两对小兔，加上原有的三对兔子，总共有 $3+2=5$ 对。按照这样，我们可以得到一个序列

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \dots$$



(a) 这些正方形的边长组成了斐波那契序列。



(b) 比萨的列奥纳多（斐波那契）1175-1250

这个数列很有规律，从第三项后，任何一项都等于前两项的和。我们可以这样思考它的原理，如果前一个月有 $m$ 对兔子，这个月有 $n$ 对兔子，那么增加的一定都是新产下的小兔，共有 $n - m$ 对，而剩下的都是成年兔子，共 $m$ 对。到下个月时，这 $n - m$ 对小兔刚刚成熟，而 $m$ 对成兔又产下了 $m$ 对小兔。所以下个月的兔子总数等于小兔加上成兔为： $(n - m) + m + m = n + m$ 对。根据这一推理，我们可以给出斐波那契数列的递归定义：

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+2} &= F_n + F_{n+1} \end{aligned} \tag{1.10}$$

习惯上通常将斐波那契数列的起始值定义为0和1<sup>7</sup>。注意到斐波那契数列的起始值是一对自然数，并且递推关系也是一对值。我们可以利用抽象工具`foldn`给出下面的定义<sup>8</sup>：

$$\begin{aligned} F &= 1st \cdot foldn((0, 1), h) \\ h(m, n) &= (n, m + n) \end{aligned} \tag{1.11}$$

也许读者会好奇，真实的计算机程序能实现这样的定义么？这是不是太理想化了？下面方框是一段的Haskell语言的程序代码<sup>9</sup>，执行`fib 10`会输出斐波那契数55<sup>10</sup>。

<sup>7</sup>如果起始值是1和3，我们就得到了卢卡斯数列1, 3, 4, 7, 11, 18, ...

<sup>8</sup>在介绍康托尔的无穷概念时，我们会给出另一个斐波那契数列的定义。

<sup>9</sup>2010年后，Haskell中不再允许使用 $n+k$ 形式的模式匹配，可以修改为：

`foldn z f n = f (foldn z f (n - 1))`

<sup>10</sup>一行代码输出前100个斐波那契数的例子：

`take 100 $ map fst $ iterate (λ(m, n)→(n, m + n)) (0, 1)`

```

foldl z _ 0 = z
foldl z f (n + 1) = f (foldl z f n)

fib = fst . foldl (0, 1) h where
  h (m, n) = (n, m + n)

```

### 练习 1.2

1. 使用 *foldl* 定义  $(\lambda m. m^m)$ ，计算给定自然数的  $m$  次幂。
2. 使用 *foldl* 定义平方。
3. 使用 *foldl* 定义奇数的平方和。它会产生怎样的序列？
4. 地面上有一排洞，一只狐狸藏在某个洞中。每天狐狸会移动到相邻的洞里。如果每天只能检查一个洞，请给出一个捉到狐狸的策略，并证明这个策略有效<sup>1</sup>。

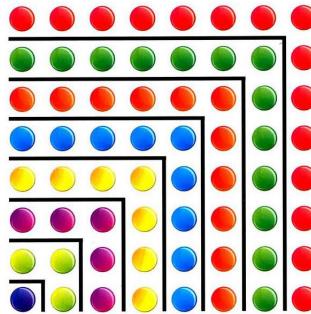


图 1.7: 《无需语言的证明》封面局部

## 1.5 自然数的同构

自然数不仅可以和自己的子集同构，例如奇偶数、平方数、斐波那契数，还可以和其他事物同构。其中一个例子是计算机程序中的数据结构。下面是列表的定义：

```
data List A = nil | cons(A, List A)
```

用数据结构的观点来解释，一个类型为A的列表或者为空，记为nil；或者包含两部分：一个含有类型A数据的节点，和一个包含剩余部分的子列表。函数cons把一个类型为A的元素和另一个类型为A的列表“链接”起来<sup>11</sup>。图1.8描述了一个含有6个节点的列表。

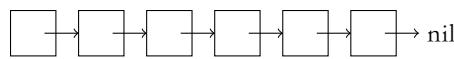


图 1.8: Linked-list

由于这种特点，列表也被称为“链表”。传统的计算机程序中，链表通常定义为一个结构<sup>12</sup>，例如：

<sup>11</sup>名称cons来自Lisp的命名传统。

<sup>12</sup>很多情况下，列表中保存的数据类型是相同的。但也有异构类型的列表，例如Lisp中的列表。

```
Node of A:
key: A
next: Node of A
```

我们也可以用自然数的同构来解释列表。根据皮亚诺公理一，`nil`相当于零；根据皮亚诺公理二，对于任何列表，我们都可以用`cons`，在其左侧链接一个类型为A的新元素。因此`cons`相当于自然数中的`succ`。这里的变化有两点。其一是列表携带类型A的元素，因而`cons(1, cons(2, cons(3, nil)))`和`cons(2, cons(1, cons(3, nil)))`以及`cons(1, cons(4, cons(9, nil)))`以及`cons('a', cons('b', cons('c', nil)))`都是不同的列表。其二是与直觉不同，新元素不是加入到列表的右侧末尾，而是加入到左侧的头部。增长的方向是向左，而非向右。

用嵌套的`cons`表示较长的列表很不方便，我们将`cons(1, cons(2, cons(3, nil)))`简记为`[1, 2, 3]`，用符号“`:`”表示`cons`。因此这一列表也可以写为`1:[2, 3]`或者`1:(2:(3:nil))`。针对A为字母的特殊情况，我们用带双引号的字符串来表示，例如用“hello”来简记表示`['h', 'e', 'l', 'l', 'o']`。

同构于自然数的加法，定义列表的连接运算如下：

$$\begin{aligned} \text{nil} ++ y &= y \\ \text{cons}(a, x) ++ y &= \text{cons}(a, x ++ y) \end{aligned} \tag{1.12}$$

列表的连接运算包含两条规则。首先空列表和任何列表连接的结果仍然等于该列表本身；并且某个列表的“后继”和另一个列表相连接，等于这两个列表连接结果的后继。和自然数的加法对比，它们呈现出有趣的镜像对称形式。

$$\begin{array}{c} \text{nil} ++ y = y \quad | \quad a + 0 = a \\ \text{cons}(a, x) ++ y = \text{cons}(a, x ++ y) \quad | \quad a + \text{succ}(b) = \text{succ}(a + b) \end{array}$$

这种同构提示我们，可以利用递推公理证明列表连接的结合律。为了证明 $(x ++ y) ++ z = x ++ (y ++ z)$ ，我们首先证明 $x = \text{nil}$ 时的起始情况

$$\begin{aligned} (\text{nil} ++ y) ++ z &= y ++ z && \text{列表连接定义的规则一} \\ &= \text{nil} ++ (y ++ z) && \text{列表连接定义的规则一，反向} \end{aligned}$$

然后再证明递推情况。假设 $(x ++ y) ++ z = x ++ (y ++ z)$ ，我们要证明 $((a : x) ++ y) ++ z = (a : x) ++ (y ++ z)$ 。

$$\begin{aligned} ((a : x) ++ y) ++ z &= (a : (x ++ y)) ++ z && \text{列表连接定义的规则二} \\ &= a : ((x ++ y) ++ z) && \text{列表连接定义的规则二} \\ &= a : (x ++ (y ++ z)) && \text{递推假设} \\ &= (a : x) ++ (y ++ z) && \text{列表连接定义的规则二，反向} \end{aligned}$$

这样我们就证明了列表连接操作的结合律。但是和自然数不同，列表不满足交换律<sup>13</sup>。例如 $[2, 3, 5] ++ [7, 11] = [2, 3, 5, 7, 11]$ ，但交换后的结果却是 $[7, 11] ++ [2, 3, 5] = [7, 11, 2, 3, 5]$ 。

考虑和自然数同构，我们也可以定义列表的抽象叠加操作。为此，我们仿照自然数定义一个抽象的起始值 $c$ ，和一个抽象的二元运算 $h$ 。这样就可以定义列表的递归形式：

$$\begin{aligned} f(\text{nil}) &= c \\ f(\text{cons}(a, x)) &= h(a, f(x)) \end{aligned} \tag{1.13}$$

进一步，令 $f = \text{foldr}(c, h)$ ，就可以抽象出列表的叠加操作。我们将其命名为`foldr`以表明这种叠加操作是从右侧开始，自右向左进行的。

$$\begin{aligned} \text{foldr}(c, h, \text{nil}) &= c \\ \text{foldr}(c, h, \text{cons}(a, x)) &= h(a, \text{foldr}(c, h, x)) \end{aligned} \tag{1.14}$$

<sup>13</sup>这也是我们避免使用加号“`+`”来表示列表连接的原因。但是很多编程语言使用了加号，这造成了一些潜在的问题。

使用 $foldr$ , 我们可以定义各种列表上的操作。例如我们可以把一个列表中的各个元素累加或累乘起来:

$$\begin{aligned} sum &= foldr(0, +) \\ product &= foldr(1, \times) \end{aligned} \quad (1.15)$$

我们可以通过一些例子了解 $sum$ 的行为。首先是空列表:  $sum([]) = foldr(0, +, nil) = 0$ ; 然后是若干个元素的列表:

$$\begin{aligned} sum([1, 3, 5, 7]) &= foldr(0, +, 1 : [3, 5, 7]) \\ &= 1 + foldr(0, +, 3 : [5, 7]) \\ &= 1 + (3 + foldr(0, +, 5 : [7])) \\ &= 1 + (3 + (5 + foldr(0, +, cons(7, nil)))) \\ &= 1 + (3 + (5 + (7 + foldr(0, +, nil))))) \\ &= 1 + (3 + (5 + (7 + 0))) \\ &= 16 \end{aligned}$$

我们还可以计算列表的长度。这本质上是把一个列表映射成自然数。

$$\begin{aligned} h(a, n) &= n + 1 \\ length &= foldr(0, h) \end{aligned} \quad (1.16)$$

这样我们就可以用 $|x| = length(x)$ 来计算列表的长度。我们还可以用 $foldr$ 定义连接操作 $\text{++}$ :

$$(\text{++ } y) = foldr(y, cons) \quad (1.17)$$

这相当于自然数的 $(+m)$ 运算, 类似自然数的乘法运算, 我们可以定义列表的“乘法”, 将一个列表的列表全部连接起来。

$$concat = foldr(nil, \text{++ }) \quad (1.18)$$

例如:  $concat([[1, 1], [2, 3, 5], [8]])$  的结果是  $[1, 1, 2, 3, 5, 8]$ 。我们接下来再用 $foldr$ 定义两个列表的重要操作: 选择和逐一映射<sup>14</sup>。选择也称为过滤, 是根据某个条件选择列表中的元素组成一个新的列表。为此, 我们需要引入条件表达式的概念<sup>15</sup>。它通常写为 $(p \mapsto f, g)$ , 也就是给定变量 $x$ , 若条件 $p(x)$ 成立, 则结果为 $f(x)$ , 否则为 $g(x)$ , 我们也会用 $\text{if } p(x) \text{ then } f(x) \text{ else } g(x)$ 来描述条件表达式。

$$filter(p) = foldr(nil, (p \cdot 1st \mapsto cons, 2nd)) \quad (1.19)$$

为了理解这一定义, 我们来看一个例子: 从一组自然数中, 选择出偶数,  $filter(\text{even}, [1, 4, 9, 16, 25])$ 。和 $sum$ 的例子类似, 首先是一系列的展开过程,  $h(1, h(4, h(9, \dots)))$ 直到列表的最右侧 $cons(25, nil)$ 。根据 $foldr$ 的定义, 传入 $nil$ 的结果为 $c$ , 故而接下来, 要计算 $h(25, nil)$ , 而其中 $h$ 为条件表达式。为此我们先将函数 $\text{even} \cdot 1st$ 应用到一对值 $(25, nil)$ 上。 $1st$ 取得 $25$ , 由于它是奇数, 故而 $\text{even}$ 条件不成立。因此接下来对这对值执行 $2nd$ , 得到结果 $nil$ 。此后计算进入上一层 $h(16, nil)$ 。先用 $1st$ 获得偶数 $16$ , 此时 $\text{even}$ 成立。这样就映射到 $cons(16, nil)$ , 结果为 $[16]$ 。然后计算又进入更上一层 $h(9, [16])$ , 通过条件表达式映射到 $2nd$ , 故而结果仍然为 $[16]$ 。这时计算进入 $h(4, [16])$ , 条件表达式映射到 $cons(4, [16])$ , 其结果为 $[4, 16]$ ; 最后计算达到顶层 $h(1, [4, 16])$ , 由条件表达式映射到 $2nd$ 得到最终结果 $[4, 16]$ 。

<sup>14</sup>和一一映射不同, 这里的映射是单向的, 例如从一个词语到它的字符长度的映射, 其逆映射并不存在。

<sup>15</sup>也称麦卡锡条件形式, 是计算机科学家Lisp发明人约翰·麦卡锡于1960年引入的。

逐一映射的概念是将列表中的每个元素通过 $f$ 映射成另一个值，从而组成一个新的列表。即 $\text{map}(f, \{x_1, x_2, \dots, x_n\}) = \{f(x_1), f(x_2), \dots, f(x_n)\}$ 。它可以用 $\text{foldr}$ 定义如下：

$$\begin{aligned} \text{map}(f) &= \text{foldr}(\text{nil}, h) \\ h(x, c) &= \text{cons}(f(x), c) \end{aligned} \quad (1.20)$$

这种把函数映射到一对值中的第一个之上的操作称为 $\text{first}$ ，即 $\text{first}(f, (x, y)) = (f(x), y)$ ，我们在此后讲解范畴的时候还会再仔细讨论它。使用 $\text{first}$ ，逐一映射可以定义为 $\text{map}(f) = \text{foldr}(\text{nil}, \text{cons} \cdot \text{first}(f))$ 。

求列表的长度也可以利用逐一映射来实现。首先将列表的所有元素都映射为1，然后在将这些1加起来就等于列表的长度。

$$\begin{aligned} \text{len} &= \text{sum} \cdot \text{map}(\text{one}) \\ \text{one}(x) &= 1 \end{aligned}$$

### 练习 1.3

1. 表达式 $\text{foldr}(\text{nil}, \text{cons})$ 定义了什么？
2. 读入一串数字（数字字符串），用 $\text{foldr}$ 将其转换成十进制数。如果是16进制怎么处理？如果含有小数点怎么处理？
3. 乔恩·本特利在《编程珠玑》中给出了一个求最大序列和的问题。给定整数序列 $\{x_1, x_2, \dots, x_n\}$ ，求哪段子序列 $i, j$ ，使得和 $x_i + x_{i+1} + \dots + x_j$ 最大。请用 $\text{foldr}$ 解决这道题。
4. 最长无重复字符子串问题。认给一个字符串，求出其中不包含重复字符的最长子串。例如“abcabcbb”的最长无重复字符子串为‘`abc’”。请使用 $\text{foldr}$ 求解。

## 1.6 形式与结构



图 1.9: 拉斐尔《雅典学院》局部

你也许注意到了本章第一节中，每个自然段的开头的第一个汉字连起来是“自然数产生”，我希望用这种形式表达同构的美。亚里士多德说美的主要形式就是秩序、匀称和确定性，这些正是数

学研究的原则。我们用公理系统展示，自然数可以和几何同样建立在五条公理组成的大厦上。我们用自然数和列表的同构同样想表达这种形式上的美。文艺复兴时期的艺术家拉斐尔在创作不朽的作品《雅典学院》时，也采用了同构，画中的历史人物和当时的人物对应。画面中心面向我们走来的是两位伟大学者的柏拉图和亚里士多德，其中柏拉图的原型是文艺复兴时期的艺术家达芬奇，亚里士多德的原型是朱利亚诺·达·桑加洛。画中的柏拉图右手向上指，意思是说人类应该思考永恒。而亚里士多德手向前伸，手掌向下，意思是说人类应该研究世界。这两个对立的手势，表达了他们思想上的分歧。中间台阶下方，倚箱沉思的是古希腊杰出的哲学家赫拉克利特，他是西方最早提出朴素辩证法和唯物论的卓越代表。他的原型是文艺复兴时期的另一位大师米开朗基罗。画面左前方以毕达哥拉斯为中心，他正在专注地书写。毕达哥拉斯右侧有一位身穿白色斗篷的金发青年，被认为是弗朗西斯柯·德拉罗斐尔，他是乌尔宾诺未来的大公。画面右下方中心是手拿圆规的欧几里得（一说为阿基米德），他的周围有手持天球的天文学家托勒密，对面是画家拉斐尔的同乡、建筑家布拉曼特，而最边上那个头戴白帽的人，是画家索多玛，上面露出半个脑袋、头戴深色圆形软帽的青年，就是画家拉斐尔本人。这让人联想起了伟大的音乐家巴赫把自己的名字B-A-C-H通过调式写进了《赋格的艺术》的音乐当中。《雅典学院》通过回忆历史上的黄金时代，表达人类对智慧和真理的追求，同时通过使用文艺复兴时期的人物作为原型，呼应了复兴古希腊艺术和哲学思想的时代主题。这是形式与内容，结构与思想的多重同构。

### 练习 1.4

- 观察斐波那契的叠加定义，它的后继计算 $(m', n') = (n, m + n)$ 相当于一个矩阵乘法：

$$\begin{pmatrix} m' \\ n' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} m \\ n \end{pmatrix}$$

起始值是 $(0, 1)^T$ 。这样斐波那契数列就在矩阵乘方下和自然数同构：

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

设计一个程序，快速计算2阶方阵的幂。求得斐波那契数列的第 $n$ 个元素。



## 第2章 递归

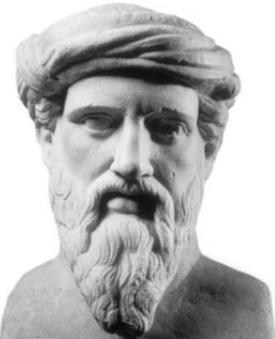
造物神代表造物神-物色的-神怪

——[美]侯世达《哥德尔、埃舍尔、巴赫  
——集异壁之大成》

人们通过深入了解数进而了解自然。在上一章中，我们介绍了自然数的皮亚诺公理。并且展示了一些和自然数有着相同结构的事物，例如“列表”这种基本数据结构。自然数成为了我们进一步前进的基石。但是我们的大厦还不稳固。第一章中，我们不加证明地使用了递归的概念。例如阶乘的定义：

$$\begin{aligned} \text{fact}(0) &= 1 \\ \text{fact}(n+1) &= (n+1)\text{fact}(n) \end{aligned}$$

递归的原理是什么？为什么它是正确的？递归可以在更低的层次被表示么？这些都是我们要在这一章解决的问题。



### 2.1 万物皆数

毕达哥拉斯（约前570——前490）

从数出发研究世间万物的第一人要算是古希腊的数学家和哲学家毕达哥拉斯了。他的名字通过著名的勾股定理（在西方叫毕达哥拉斯定理）而家喻户晓。毕达哥拉斯出生于希腊的萨摩斯（Samos）岛，年轻时他曾去米利都（Miletus）向古希腊哲学的奠基人泰勒斯（Thales）学习。在泰勒斯的建议下，毕达哥拉斯前往东方学习数学。他在埃及学习了13年（一说为22年）。后来波斯帝国征服了埃及，他又随军向东到达了巴比伦，向巴比伦人学习数学和天文知识。或许后来他还到达了更远的印度。不论到了哪里，毕达哥拉斯都不断向有学问的人请教，丰富自己的见解。重要的是，他不仅刻苦学习，而且更善于思考。在经过兼收并蓄、汲取各家之长后，毕达哥拉斯形成并完善了自己的思想<sup>[9]</sup>。

经历了漫长的在外游历后，年近半百的毕达哥拉斯返回了故乡并开始讲学。公元前520年左右，为了摆脱当地的暴政，毕达哥拉斯移居到了意大利南部的克罗顿（Croton）发展。在那里他赢得了人们的信任与景仰并形成了自己的学派。毕达哥拉斯的弟子中还有女性，学派把主要精力都用来研究天文、几何、数论、音乐这四门学科。它们被称为四术（quadrivium），影响了欧洲教育两千多年<sup>[10]</sup>。四术体现了毕达哥拉斯“万物皆数”的哲学思想：星体的运动与几何对应，而几何又以数为基础，数字还可以衍生出音乐。毕达哥拉斯是首个发现纯八度音（octave）在频率上有数学规律的人。他的弟子说他可以“听见天界的乐音”<sup>[1]</sup>。

<sup>1</sup>关于毕达哥拉斯的逝世的说法不一。他领导的学派具有很高的声誉和政治影响，引起了敌对派的忌恨。后来受到民主运动的冲击，学派在克罗顿的活动场所遭到破坏。有人认为毕达哥拉斯被暴徒杀害，也有人说他逃到梅塔蓬图姆（Metapontum）并度过余生。

毕达哥拉斯学派深入研究了数与数、数与自然之间的关系。这开启了数学的重要分支——数论。他们对正整数进行了分类，定义了奇数、偶数、素数、合数等。他们发现某些数的所有真因子<sup>2</sup>之和恰好等于这个数本身，于是将其命名为完全数，并成功地找到两个<sup>3</sup>。最小的完全数是6，因为 $6 = 1 + 2 + 3$ ，下一个是28（等于 $1 + 2 + 4 + 7 + 14$ ）。毕达哥拉斯学派还发现了一大类“形数”（figurate number）<sup>4</sup>。

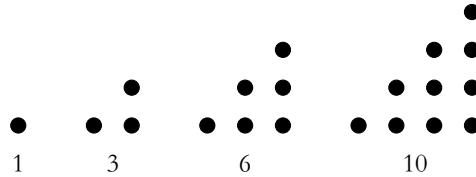


图 2.2: 三角形数 (triangular number)

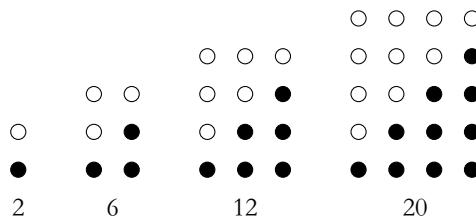


图 2.3: 长方形数 (oblong number)

图2.2和图2.3分别是三角形数和长方形数。很容易看出，每个长方形数都对应三角形数的二倍，而三角形数又是前 $n$ 个正整数之和，这样就得到了正整数累加的求和公式：

$$1 + 2 + 3 + \dots + n = \frac{1}{2}n(n + 1)$$

毕达哥拉斯学派还观察到，所有的奇数可以表示成折尺形（数学上称为“磬折形”），如图2.4，而前 $n$ 个折尺形可以拼成一个正方形，如图2.5。这样他们就发现了前 $n$ 个正奇数的求和公式：

$$1 + 3 + 5 + \dots + (2n - 1) = n^2$$

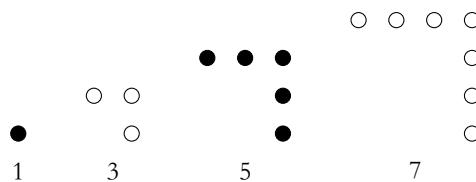


图 2.4: 折尺形数 (gnomon number)

<sup>2</sup>真因子是小于数本身的因子

<sup>3</sup>一说为完美数。经过欧几里得与欧拉的进一步工作，揭示了偶完全数的特征以及完全数和梅森素数的关系。到2018年，人们借助计算机共发现了50个梅森素数和完全数。

<sup>4</sup>毕达哥拉斯学派的门徒通过在地上摆小石子来研究数字，英文的计算calculus一词就是从希腊文“石子”衍生出的[9]。当他们把石子按照某种几何方式排列成图形时，就得到了形数。

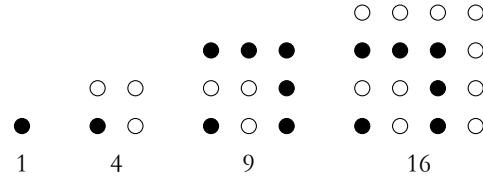


图 2.5: 正方形数 (square number) 与折尺形数的关系

这正是第一章提出的那个习题的答案。就这样，毕达哥拉斯学派发现，很多事物和现象都可以从数的方面进行说明和解释。例如，具有同样张力的两根弦，当它们的长度为简单的整数比时，奏出的乐声就和谐悦耳。由此毕达哥拉斯发展出了最初的音乐理论。音乐与数似乎毫无关系，但它们之间的这种意外联系给毕达哥拉斯很大影响。他从中得到启发并大胆推测：所有的事物都可以用整数或整数的比来解释。毕达哥拉斯学派开始热衷于用数去解释更多的现象，他们相信宇宙的本质就在于“数的和谐”，并且提出“万物皆数”的论断。由此出发，毕达哥拉斯学派试图发展一套以数字为基础的理论，使得几何学可以建立在该理论之上。这种想法实际上就相当于要创建一套基于正整数的统一数学理论。

毕达哥拉斯学派最著名的发现当属勾股定理的证明。至今这一定理在西方仍被称为“毕达哥拉斯定理”。然而，我们将看到勾股定理是一把双刃剑，他的结果最终形成了一个递归的怪圈，使得“万物皆数”的理念出现了漏洞。为此，我们先要引出可公度概念和欧几里得算法。为了将几何纳入“万物皆数”的理论，毕达哥拉斯学派提出了一个概念来定义一条线段可以用另一条线段来度量。这个定义说，如果一条线段A可以通过有限个另一条线段V来表示时，称线段V可用作线段A的量度 (measure)。这本质上是说，我们可以通过整数次拼接产生另一条线段。尽管度量两条不同的线段时，可以使用各自的量度，但是如果想用同一量度测量不同的线段，它必须是二者的公度 (common measure)。即当且仅当线段V可以同时成为线段A和线段B的量度时，它才能成为二者的公度。毕达哥拉斯学派认为，任何情况下都可以找到公度，这样几何就可以建立在整数之上了。

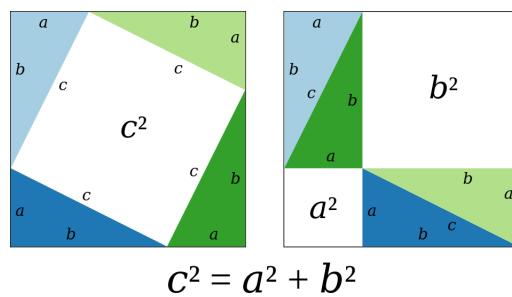


图 2.6: 勾股定理的一种几何证明，两幅图中白色面积相等（来自《周髀算经》，约公元前200年）

## 2.2 欧几里得算法

由于公度可能有多个，为此需要引入最大公度 (greatest common measure) 的概念。如果线段V是线段A和线段B的公度，并且比其它公度都大，则称V是A和B的最大公度。已知两条线段，怎样才能求得最大公度呢？这就引出了历史上著名的递归算法——欧几里得算法（又称辗转相除法）。

它用古希腊伟大的数学家欧几里得的名字命名<sup>5</sup>。在欧几里得的名著《几何原本》第十卷命题三中[11]，详细阐述了这一算法<sup>6</sup>。

### 2.2.1 欧几里得和《几何原本》

欧几里得 (Euclid) 是古希腊数学家，以其所著的《几何原本》闻名于世。对于他的生平，现在知道的很少。柏拉图学派晚期的导师普罗克洛斯 (Proclus) 在《几何学发展概要》中记述了这样的趣事：当时的埃及国王，亚历山大的托勒密一世有一次问欧几里得，学习几何学又没有什么捷径可走。欧几里得回答到：“在几何里，没有专为国王铺设的大道。”(There is no royal road to geometry) 这句话成为千古传颂的学习箴言<sup>7</sup>。斯托比亚斯 (Stobaeus) 记述另一则故事说。一个学生刚开始学习第一个命题，就问欧几里得学习几何后将得到什么。欧几里得说：“给他三个钱币，因为他想在学习中获取实利。”由此可知欧几里得主张学习必须循序渐进、刻苦钻研、不赞成投机取巧的作风，也反对狭隘的实用观点[11]（中文版序）。

欧几里得的《几何原本》是一部划时代的著作。其伟大的历史意义在于它是用公理建立起演绎体系的最早典范。过去所积累下来的数学知识是零碎的、片断的，可以比作木石砖瓦。只有借助于逻辑方法，把这些知识组织起来，加以分类比较，解释彼此间的内在联系，整理在一个严密的系统之中，才能建成巍峨的大厦。《几何原本》完成了这一艰巨的任务，它对整个数学的发展产生了深远的影响。

两千多年来，这部著作在几何教学中一直占据统治地位，在二十世纪初依然用于数学课的基本教材。包括我国在内的许多国家仍将其作为中学的必修科目（现在中学的几何课本是按照法国数学家拉格朗日《几何原本》改写本思路编写的），并作为训练逻辑推理的最有力教育手段[9]。



欧几里得，约公元前300年前后

### 2.2.2 欧几里得算法

**命题 2.2.1** (《几何原本》，卷十，命题三). 已知两个可公度的量，求它们的最大公度量。

欧几里得给出的方法，只需要使用递归和减法。因此本质上可以用尺规作图的方式求出最大公度。这一算法可以形式化如下<sup>8</sup>。

$$gcm(a, b) = \begin{cases} a = b & : a \\ b < a & : gcm(a - b, b) \\ a < b & : gcm(a, b - a) \end{cases} \quad (2.1)$$

设两条线段 $a$ 、 $b$ 可公度，如果它们相等，则最大公度就是其中的任一条线段，此时算法返回 $a$ 作为结果。如果线段 $a$ 比 $b$ 长，就用圆规不断从 $a$ 中截去 $b$ （通过递归），然后求截短的线段 $a'$ 和 $b$ 的最大公度；否则如果线段 $b$ 比 $a$ 长，就反过来不断从 $b$ 中截去 $a$ ，然后求 $a$ 和截短的线段 $b'$ 的最大公度。图2.8描述了这一算法作用于两条线段的计算步骤。我们也可将这一算法应用于整数42和30并与处理线段的过程进行对比，如下表：

<sup>5</sup>印度和中国分别独立发现了欧几里得算法。五世纪末，印度数学家阿耶波多 (Aryabhata) 用这一算法解不定方程（丢番图方程）。在《孙子算经》中，欧几里得算法可以作为中国剩余定理的特例。在1247年，南宋数学家秦九韶在《数书九章》中详细给出了欧几里得算法。

<sup>6</sup>在《几何原本》卷七的命题一中，有针对整数的欧几里得算法。但针对线段的情形实际上覆盖了整数。

<sup>7</sup>也译为“几何无王者之道”

<sup>8</sup>名称gcm是最大公度 (greatest common measure) 的简称。当 $a$ 、 $b$ 是整数时，通常用gcd (greatest common divisor) 作

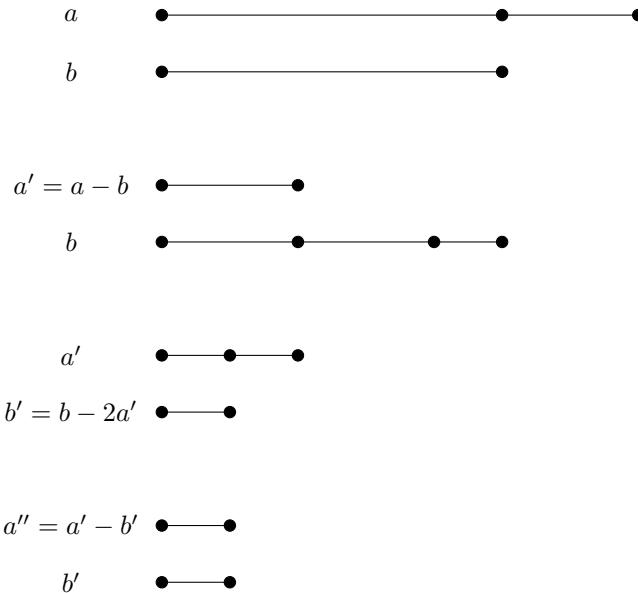


图 2.8: 欧几里得算法的线段示意

$gcm(a, b)$	$a$	$b$
$gcm(42, 30)$	42	30
$gcm(12, 30)$	12	30
$gcm(12, 18)$	12	18
$gcm(12, 6)$	12	6
$gcm(6, 6)$	6	6

将一个量 $b$ 反复从另一个量 $a$ 中减去，最后得到 $a'$ 的过程恰好是带余数除法的定义。即 $a' = a - \lfloor a/b \rfloor$ 或记为 $a' = a \bmod b$ 。因此我们可以用除法和求余运算代替原始欧几里得算法中的反复相减。此外，当一个量是另一个量的整倍数时，例如 $b \leq a$ 且 $b$ 可以整除 $a$ ，我们立即知道最大公度为 $b$ 。此时求余的结果 $a \bmod b = 0$ ，为此我们可以定义 $gcm(0, b) = gcm(b, 0) = b$ 。我们可以先比较 $a$ 和 $b$ 的大小，如果 $a < b$ 就交换两个量。由于我们知道 $a \bmod b$ 一定小于 $b$ ，所以下次递归时可以直接交换为： $gcm(b, a \bmod b)$ 。这样就得到了改进的欧几里得算法。

$$gcm(a, b) = \begin{cases} b = 0 & : a \\ \text{否则} & : gcm(b, a \bmod b) \end{cases} \quad (2.2)$$

为什么这个算法可以求出最大公度呢？我们需要分两步来证明它的正确性。第一步我们要证明这一算法可以求出公度。设 $b \leq a$ ，令整数 $q_0$ 为商， $r_0$ 为余数，即 $a = bq_0 + r_0$ ，如果 $r_0$ 为零，算法就找到了公度，为此我们考虑 $r_0$ 不为零的情况。此时可以进一步列出 $b = r_0 q_1 + r_1$ ，类似地，只要余数不为零，我们可以一直列出这样的式子。

---

为名称。本书按此约定使用这两个名称。

$$\begin{aligned}
 a &= bq_0 + r_0 \\
 b &= r_0q_1 + r_1 \\
 r_0 &= r_1q_2 + r_2 \\
 r_1 &= r_2q_3 + r_3 \\
 &\dots
 \end{aligned}$$

但只要  $a, b$  是可公度的，这些式子不会无限列下去。理由是每次都用圆规截取整数次，即商是整数。同时每次都保证余数小于除数。即  $b > r_0 > r_1 > r_2 > \dots > 0$ ，但是余数不可能小于零。由于起始值是有限的，故最终一定在有限步内得到  $r_{n-2} = r_{n-1}q_n$ 。

接下来我们证明最后一步得到的  $r_{n-1}$  可以同时度量  $a$  和  $b$ 。根据度量的定义，显然  $r_{n-1}$  可以度量  $r_{n-2}$ 。然后考虑倒数第二式  $r_{n-3} = r_{n-2}q_{n-1} + r_{n-1}$ ，由于  $r_{n-1}$  可以度量  $r_{n-2}$ ，所以  $r_{n-1}$  也可以度量  $r_{n-2}q_{n-1}$ ，自然它也可以度量  $r_{n-2}q_{n-1} + r_{n-1}$ ，这个量恰好等于  $r_{n-3}$ 。用同样的方法，我们可以向上逐一证明  $r_{n-1}$  可以度量每个式子左边，一直到  $b, a$ 。这样我们就证明了欧几里得算法得到的答案  $r_{n-1}$  是  $a, b$  的公度。若最大公度为  $g$ ，一定有  $r_{n-1} \leq g$ 。

第二步我们要证明，任何  $a, b$  的公度  $c$ ，一定也可以度量  $r_{n-1}$ 。由于  $c$  是公度，因此  $a$  和  $b$  可以用它来表示，不妨记  $a = mc, b = nc$ ，其中  $m, n$  都是整数。这样第一式  $a = bq_0 + r_0$  就可以写成  $mc = ncq_0 + r_0$ ，我们得知  $r_0 = (m - nq_0)c$ ，因此  $c$  也可以度量  $r_0$ 。类似地，我们可以依次证明  $c$  可以度量  $r_1, r_2, \dots, r_{n-1}$ 。这样我们就证明了任何公度都可以度量  $r_{n-1}$ ，因此最大公度  $g$  也可以度量  $r_{n-1}$ ，即  $g \leq r_{n-1}$ 。

综合第一、二步的结果，即  $r_{n-1} \leq g$  且  $g \leq r_{n-1}$ ，我们得出最大公度  $g = r_{n-1}$ ，也就是说欧几里得算法能够正确地给出最大公度。进一步，我们知道  $g$  是每一对量的最大公度，即：

$$g = \text{gcm}(a, b) = \text{gcm}(b, r_0) = \dots = \text{gcm}(r_{n-2}, r_{n-1}) = r_{n-1} \quad (2.3)$$

### 2.2.3 扩展欧几里得算法

所谓扩展欧几里得算法，就是除了求得两个量  $a, b$  的最大公度  $g$  外，同时找到满足贝祖等式  $ax + by = g$  的两个整数  $x$  和  $y$ 。为什么贝祖等式 (Bézout's identity)<sup>9</sup> 一定成立呢？我们下面给出贝祖等式的一种证明。由  $a, b$  可以构造一个集合，包含它们所有正的线性组合：

$$S = \{ax + by \mid x, y \in \mathbb{Z} \text{ 且 } ax + by > 0\}$$

对于线段来说， $S$  一定不为空，因为它至少包含  $a$  (此时  $x = 1, y = 0$ ) 和  $b$  (此时  $x = 0, y = 1$ )。因为  $S$  的所有元素都为正，所以它一定存在一个最小的元素，我们将其记为  $g = as + bt$ 。我们接下来要证明  $g$  就是  $a$  和  $b$  的最大公度。为此我们将  $a$  表示成  $g$  的商与余数的形式。

$$a = qg + r \quad (2.4)$$

其中余数  $0 \leq r < g$ 。余数要么为 0，要么在集合  $S$  中。这是因为

$$\begin{aligned}
 r &= a - qg && \text{由式(2.4)} \\
 &= a - q(as + bt) && g \text{ 的定义} \\
 &= a(1 - qs) - bqt && \text{合并整理}
 \end{aligned}$$

即  $r$  可以表示为  $a, b$  的线性组合，因此如果它不为 0，则一定在集合  $S$  中。但这是不可能的，因为我们之前假设  $g$  是  $S$  中的最小正元素，而  $r$  却比  $g$  更小，这样就会产生矛盾。因此我们的得知  $r$  一定等于 0。根据式(2.4)， $g$  一定可以度量  $a$ 。用同样的方法，可以证明  $g$  也一定可以度量  $b$ 。因此  $g$  是

<sup>9</sup> 贝祖等式，或称贝祖定理，最早是由法国数学家梅齐里亚克 (Claude Gaspard Bachet de Méziriac, 1581 – 1638) 发现并证明了其对整数成立，法国数学家贝祖证明这一等式对多项式成立。贝祖等式可以推广到任意的整环和主理想环 (PID, Principle Ideal Domain) 上。



(a) 贝祖 (Étienne Bézout, 1730 - 1783)



(b) 梅齐里亚克 (Claude Gaspard Bachet de Méziriac, 1581 – 1638)，最早发现并证明了整数上的贝祖等式。

它们的公度。接下来我们证明 $g$ 是最大公度。令 $c$ 为 $a$ 和 $b$ 的任意公度，根据定义，存在整数 $m$ 、 $n$ 使得 $a = mc$ 、 $b = nc$ 。这样 $g$ 就可以表示为：

$$\begin{aligned} g &= as + bt && \text{由定义} \\ &= mcs + nct && c \text{是 } a, b \text{ 的公度} \\ &= c(ms + nt) && g \text{ 是 } c \text{ 的倍数} \end{aligned}$$

这说明 $c$ 可以度量 $g$ ，也就是说 $c \leq g$ 。这就证明了 $g$ 是最大公度。综上我们就证明了贝祖等式，即存在整数使得 $ax + by = g$ ，并且进一步得知最大公度是所有线性组合的正值中最小的。

使用贝祖等式，我们可以推导出扩展欧几里得算法。

$$\begin{aligned} ax + by &= \text{gcm}(a, b) && \text{贝祖等式} \\ &= \text{gcm}(b, r_0) && \text{欧几里得算法, 式(2.3)} \\ &= bx' + r_0y' && \text{对 } b \text{ 和 } r_0 \text{ 使用贝祖等式} \\ &= bx' + (a - bq_0)y' && \text{利用 } a = bq_0 + r_0 \\ &= ay' + b(x' - y'q_0) && \text{整理为 } a \text{ 和 } b \text{ 的线性组合} \\ &= ay' + b(x' - y'\lfloor a/b \rfloor) && \text{将 } q_0 \text{ 表示为 } a \text{ 与 } b \text{ 的商} \end{aligned}$$

这样就得出了每次递归时的关系：

$$\begin{cases} x = y' \\ y = x' - y'\lfloor a/b \rfloor \end{cases}$$

递归的边界条件出现在 $b = 0$ 的时候，此时 $\text{gcm}(a, 0) = 1a + 0b$ 。把边界条件和递归关系归纳起来就得到了扩展欧几里得算法。

$$\text{gcm}_{ex}(a, b) = \begin{cases} b = 0 & : (a, 1, 0) \\ \text{否则} & : (g, y', x' - y'\lfloor a/b \rfloor) \\ & \quad \text{其中 } (g, x', y') = \text{gcm}_{ex}(b, a \bmod b) \end{cases} \quad (2.5)$$

我们下面给出一个使用扩展欧几里得算法解决的趣题 ([14], 第450页)。有两个水瓶，一个的容量是9升，另一个的容量是4升。问如何才能从河中打出6升水？

这道题目有很多变化形式，瓶子的容积和取水的容量可以是其它数值。有一个故事说解决这道题目的主人公是少年时代的法国数学家泊松（Siméon Denis Poisson）。

使用两个瓶子，共有6种操作。记大瓶子为 $A$ ，容积为 $a$ ；小瓶子为 $B$ ，容积为 $b$ ：

- 将大瓶子 $A$ 装满水；
- 将小瓶子 $B$ 装满水；
- 将大瓶子 $A$ 中的水倒空；
- 将小瓶子 $B$ 中的水倒空；
- 将大瓶子 $A$ 中的水倒入小瓶子 $B$ ；
- 将小瓶子 $B$ 中的水倒入大瓶子 $A$ 。

其中最后两种操作中，任意一个瓶子满或者另一个瓶子空时就停止。表2.1给出了一系列倒水动作的例子，这里假设容积 $b < a < 2b$ 。

$A$	$B$	操作
0	0	开始
0	$b$	倒满 $B$
$b$	0	将 $B$ 倒入 $A$
$b$	$b$	倒满 $B$
$a$	$2b - a$	将 $B$ 倒入 $A$
0	$2b - a$	倒光 $A$
$2b - a$	0	将 $B$ 倒入 $A$
$2b - a$	$b$	倒满 $B$
$a$	$3b - 2a$	将 $B$ 倒入 $A$
...	...	...

表 2.1: 两个瓶子内的水量和倒水操作的对应关系

无论进行何种操作，每个瓶子中的水的容量总可以表示为 $ax + by$ 的形式，其中 $x, y$ 是整数。也就是说，我们能获得的水的体积总是 $a$ 与 $b$ 的线性组合。根据贝祖等式的证明，我们知道线性组合的最小正值恰好是 $a$ 和 $b$ 的最大公度 $g$ 。因此给定两个瓶子的容量，我们立即能够判断是否可以得到体积为 $c$ 的水——只要 $c$ 能够被 $g$ 度量<sup>10</sup>。当然还要求 $c$ 不能超过两个瓶子中较大瓶子的容积。

例如，使用容量为4升和6升的瓶子，我们永远无法得到5升水。这是因为4与6的最大公约数是2，但5不能被2整除。（换个思路想这个问题：用两个容积为偶数升的瓶子，永远无法从河里打到奇数升的水。）如果 $a$ 和 $b$ 是互素的整数，即 $\gcd(a, b) = 1$ ，则可以得到任意自然数 $c$ 升的水。

虽然通过检查 $g$ 是否能度量 $c$ 可以判断是否有解，但是我们并不知道具体的倒水步骤。如果我们可以找到整数 $x$ 和 $y$ ，使得 $ax + by = c$ 。就可以得到一组操作来解决此题。具体思路是这样的：若 $x > 0, y < 0$ ，我们需要倒满瓶子 $A$ 共 $x$ 次，倒空瓶子 $B$ 共 $y$ 次；反之若 $x < 0, y > 0$ ，则需要倒空瓶子 $A$ 共 $x$ 次，倒空瓶子 $B$ 共 $y$ 次。

例如，若大瓶容积 $a = 5$ 、小瓶容积 $b = 3$ ，要取得 $c = 4$ 升水，因为 $4 = 3 \times 3 - 5$ ，即 $x = -1, y = 3$ 我们可以设计下面的一系列操作：

在这一系列操作中，倒满 $B$ 共3次，倒空 $A$ 共1次。因此剩下的问题是如何寻找整数 $x$ 和 $y$ 。使得 $ax + by = c$ ，根据扩展欧几里得算法，我们可以找到满足贝祖等式的一组解 $ax_0 + by_0 = g$ 。因为 $c$ 是最大公度 $g$ 的 $m$ 倍，我们只要把 $x_0$ 和 $y_0$ 相应加大 $m$ 倍即可得到一组特解：

<sup>10</sup>如果容积为整数，当且仅当 $c$ 能够被最大公约数 $g$ 整除。

A	B	操作
0	0	开始
0	3	倒满B
3	0	将B倒入A
3	3	倒满B
5	1	将B倒入A
0	1	将A倒空
1	0	将B倒入A
1	3	倒满B
4	0	将B倒入A

表 2.2: 取得4升水需要进行的操作

$$\begin{cases} x_1 = x_0 \frac{c}{g} \\ y_1 = y_0 \frac{d}{g} \end{cases}$$

根据这组特解，我们可以找到满足不定方程<sup>11</sup>的所有整数解：

$$\begin{cases} x = x_1 - k \frac{b}{g} \\ y = y_1 + k \frac{d}{g} \end{cases} \quad (2.6)$$

这里 $k$ 为整数。这样我们就找到了倒水问题的所有解。进一步，我们可以找到一个特定的 $k$ ，使得 $|x| + |y|$ 的值最小，从而得到最快的倒水步骤<sup>12</sup>。下面是解决这一趣题的Haskell例子程序。

```
import Data.List
import Data.Function (on)

— Extended Euclidean Algorithm
gcmex a 0 = (a, 1, 0)
gcmex a b = (g, y', x' - y' * (a `div` b)) where
  (g, x', y') = gcmex b (a `mod` b)

— Solve the linear Diophantine equation ax + by = c
solve a b c | c `mod` g /= 0 = (0, 0, 0, 0) — no solution
             | otherwise = (x1, u, y1, v)
  where
    (g, x0, y0) = gcmex a b
    (x1, y1) = (x0 * c `div` g, y0 * c `div` g)
    (u, v) = (b `div` g, a `div` g)

— Optimal by minimize |x| + |y|
jars a b c = (x, y) where
  (x1, u, y1, v) = solve a b c
  x = x1 - k * u
  y = y1 + k * v
```

<sup>11</sup>又叫做丢番图方程，是用古希腊亚历山大的数学家丢番图（Diophantus，约200-284）的名字命名的。丢番图在他的著作《算术》中，独创地引入了代数符号系统。被一些数学史家誉为“代数学之父”[12]。

<sup>12</sup>例如，可以将表示解的两条直线画出。取绝对值后，将横轴下方的部分对称翻转。进而找到使得 $|x| + |y|$ 最小的 $k$ 。

```

k = minimumBy (compare `on` (\i -> abs (x1 - i * u) +
                           abs (y1 + i * v))) [-m..m]
m = max (abs x1 `div` u) (abs y1 `div` v)

```

求得两个瓶子倒空和倒满的次数 $x$ 、 $y$ 后，就可以生成一系列倒水的步骤，附录二提供了完整的例子程序。

### 2.2.4 欧几里得算法的意义

在“万物皆数”的哲学思想下，虽然欧几里得算法最初是为了寻找两个整数的最大公约数而产生的。但是经过欧几里得之手，算法却应用到了抽象的几何量上。从整数的最大公约数，到可公度量的最大公度，我们看到了几何量和数的分离<sup>13</sup>。几何不仅没有建立在整数之上，反而独立发展，解决了整数之外的问题。以至于后来古希腊数学形成了这样的传统，任何关于数的结论，都需要给出几何的证明。这一传统直到十六世纪仍然影响着人们。意大利数学家卡尔丹（Gerolamo Cardano）<sup>14</sup>在关于解三次、四次方程的著作《大术》（Ars Magna，1545年出版）中，仍然使用类似立方体填补法的几何论证[12]。

欧几里得算法是最著名的一个递归算法。德国数学家，解析数论创始人狄利克雷（Dirichlet）在他的著作《数论讲义》中评价到：“整个数论的结构都建立在同一个基础之上，这个基础就是最大公约数算法。”现代密码学的RSA加密算法<sup>15</sup>直接使用了扩展欧几里得算法。我们在上一节通过一道趣题展示了如何使用扩展欧几里得算法给出二元线性不定方程 $ax + by = c$ 的整数解。具体来说，就是先求出最大公度 $g$ ，并判断 $g$ 是否整除 $c$ ，若不能，则无整数解。否则，将满足贝祖等式的 $x_0, y_0$ ，扩大 $c/g$ 倍得到一组特解 $x_1, y_1$ 。然后得到一般二元线性不定的通解 $x = x_1 - kb/g$ 和 $y = y_1 + ka/g$ 。

欧几里得算法是一把锋利的宝剑，但是它强大的递归原理被反过来指向了“万物皆数”的基石——万物可公度。一切事物和现象都可以归结为整数与整数的比。从而引发了毕达哥拉斯学派哲学思想的危机。约公元前470年左右，毕达哥拉斯学派的学生希帕索思试图寻找正方形的对角线和边的公度。经过仔细思考他发现不管度量单位取得多么小，这两条线段都无法公度。还有的说法是希帕索思从毕达哥拉斯学派的神秘五角星标志上得到了启发。毕达哥拉斯学派成员用五角星作为学派的徽章和联络标志。有一则故事说，学派的一个成员流落异乡，贫病交迫，无力酬谢房主的款待，临终前要房主在门上画一个五角星。若干年后，有同派的人看到这个标志，询问事情的经过，厚报房主而去[9]。美国迪士尼在1959年的动画片《唐老鸭漫游数学奇境》中，描绘了唐老鸭遇到了毕达哥拉斯和他的朋友们，在了解音乐、艺术与数的关系后，唐老鸭的手掌上也画上了神秘的五角星。如图2.11所示，传说希帕索思发现线段AC和AG也是无法公度的。

十九世纪的苏格兰数学家乔治·克里斯托重建了希帕索思的证明。使用反证法，假设存在一条单位线段 $c$ 能够公度正方形的边和对角线。根据度量的定义，可以令边长为 $mc$ 、对角线长为 $nc$ ，其中 $m, n$ 都是正整数。如图2.11所示，我们以点A为圆心，以边长为半径做圆弧交对角线AC于点E。然后从E出发作垂直于对角线的直线，并交边BC于点F。



希帕索思 (Hippasus of Metapontum)  
约公元前五世纪。

<sup>13</sup>这也是我们使用gcm，而没有使用更常见的gcd作为算法简写的原因。

<sup>14</sup>也译作卡尔达诺

<sup>15</sup>RSA算法是最早的一种公开密钥加密的非对称加密算法，RSA是1977年由罗纳德·李维斯特（Ron Rivest）、阿迪·萨莫尔（Adi Shamir）和伦纳德·阿德曼（Leonard Adleman）一起提出的。当时他们三人都在麻省理工学院工作。RSA就是他们三人姓氏开头字母拼在一起组成的。

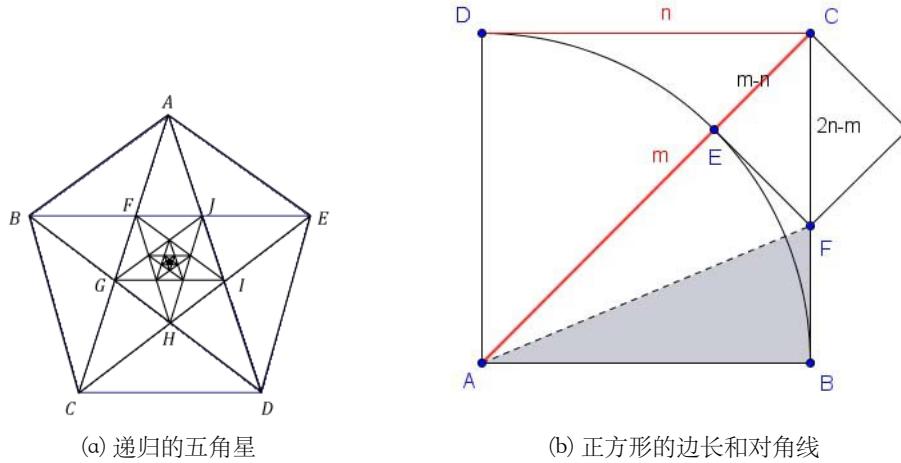


图 2.11

根据圆的定义，线段AE的长度等于正方形的边长，所以线段EC的长度等于 $(m - n)c$ 。因为EF垂直于AC，而角 $\angle ECF = 45^\circ$ ，故三角形ECF是等腰直角三角形。由于等腰三角形两腰相等，故而有 $|EC| = |EF|$ 。接下来我们注意两个直角三角形 $\triangle AEF$ 和 $\triangle ABF$ ，由于边AE等于AB，同时AF是公共边，因此两个直角三角形全等。这样就得到边 $|EF| = |FB|$ 。综合下来，我们有 $|EC| = |EF| = |FB|$ 。这样线段FB的长度也等于 $(m - n)c$ ，所以线段CF的长度等于CB的长度减去FB的长度，等于 $nc - (m - n)c = (2n - m)c$ 。我们把得到的结论列在下面。

$$\left| \begin{array}{l} |AC| = mc \\ |AB| = nc \end{array} \right. \quad \left| \begin{array}{l} |CF| = (2n - m)c \\ |CE| = (m - n)c \end{array} \right.$$

大正方形                           小正方形

由于 $m$ 、 $n$ 都是正整数，显然 $c$ 也可以公度小正方形的对角线 $CF$ 和边 $CE$ 。仿照上面的方法，我们可以继续作出更小的正方形，并且重复作出无穷无尽的更小的正方形。而 $c$ 总可以公度每一个小正方形的斜边和对角线。由于 $m$ 、 $n$ 是有限的正整数，这一过程不可能无限做下去，这样就产生了矛盾。于是我们一开始的假设不成立，即正方形的边和对角线不可公度。

这样毕达哥拉斯万物皆数的理论就出现了一个漏洞：存在线段的长度无法用整数比进行度量。据说希帕索思因为这个发现，而遭到谋杀，毕达哥拉斯学派担心这个秘密被泄露出去，而把希帕索思沉入大海。然而历史的车轮不会倒退，古希腊的哲学家和数学家们正视了这个问题，经过欧多克索斯、亚里士多德和欧几里得等人的工作，终于严格定义了不可公度和无理量，并通过几何将它们纳入了古希腊的数学体系。

**命题 2.2.2** (《几何原本》，卷十，命题二). 如果从两个不等量的大量中连续减去小量，直到余量小于小量，再从小量中连续减去余量直到小于余量，这样一直作下去，当所余的量总不能量尽它前面的量时，则称两个量不可公度。

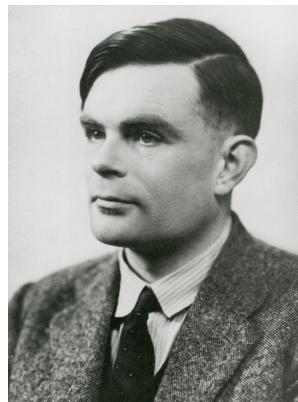
这里出现了一个有趣的现象，不可公度是用欧几里得算法能否停止来定义的。由于欧几里得算法是递归的，也就是说递归能否中止成了判断条件。这再次将我们的注意力引入到递归的本质上。递归究竟是什么？它怎样用形式化的方法表示？

### 练习 2.1

1. 我们给出的欧几里得算法是递归的，请消除递归，只使用循环实现欧几里得算法和扩展欧几里得算法。
2. 大多数编程环境中的取模运算，要求除数、被除数都是整数。但是线段的长度不一定是整数，请实现一个针对线段的取模运算。它的效率如何？
3. 我们在证明欧几里得算法正确性的过程中说：“每次都保证余数小于除数。即 $b > r_0 > r_1 > r_2 > \dots > 0$ ，但是余数不可能小于零。由于起始值是有限的，故最终算法一定中止。”为什么不会出现， $r_n$ 无限接近于零但不等于零的情况？算法一定会中止么？ $a$ 和 $b$ 是可公度的这一前提保证了什么？
4. 对于二元线性不定方程 $ax + by = c$ ，若 $x_1, y_1$ 和 $x_2, y_2$ 为两对整数解。试证明 $|x_1 - x_2|$ 的最小值为 $b/gcm(a, b)$ ，且 $|y_1 - y_2|$ 的最小值为 $a/gcm(a, b)$ 。
5. 试证明图2.11的五角星中的线段AC和AG是不可公度的。使用实数表示，它们比值是什么？

## 2.3 $\lambda$ 演算

如果进行计算的是如我们人类这样的智慧生命，也许不用深究递归的原理。我们只需要进行计算，发现递归时就在自己的思维中螺旋进入下一个层次。当递归终止时，就退回上一层。当人们思考如何用机器帮助我们进行计算时，这一问题才变得重要起来。二十世纪三十年代，当人们在研究可计算问题时，分别独立提出了一些计算模型。最著名的包括图灵提出的图灵机模型（1935年），丘奇（1932到1941年）和克莱尼（Stephen Kleene, 1935年）提出的 $\lambda$ 演算，埃尔布朗（Jacques Herbrand）和哥德尔（Kurt Gödel）提出的递归函数（1934年）等。



艾伦·图灵 (Alan Mathison Turing, 1912 - 1954)

图灵是英国数学家、逻辑学家，深刻地影响了计算机科学的理论发展，他提出使用图灵机来形式化算法和计算的概念，是现代通用计算机的模型。图灵因此被称为计算机科学之父和人工智能之父[15]。第二次世界大战期间，图灵参加了盟军位于布莱切利公园的密码破译中心。他设计了很多技术用以快速破解纳粹德国的密码。图灵研制了一台绰号为“炸弹”（Bombe）的电子机器，使用战前波兰发现的方法，可以在一小时内找到德军恩尼格玛（Enigma）密码机的密钥。密码的成功破译是盟国获胜的一个关键因素，许多历史学家认为缩短了战争达2年之久，并且挽救了成千上万的生命。战争结束后，图灵开始从事“自动计算机”（ACE）的逻辑设计和具体研制工作。在图灵的设计思想指导下，1950年制出了ACE样机，1958年制成大型ACE机。人们认为，通用计算机的概念就是图灵提出来的。1950年，图灵开始考虑机器思维的问题并在论文《计算机与智能》中提出了著名的“图灵测试”。这一划时代的作品，使图灵赢得了“人工智能之父”的桂冠。1951年，由于在可计算数学方面所取得的成就，图灵成为英国皇家学会会员，时年39岁。为了纪念他对计算机科

学的巨大贡献，由美国计算机协会（ACM）于1966年设立一年一度的图灵奖，以表彰在计算机科学中做出突出贡献的人，图灵奖被喻为“计算机界的诺贝尔奖”。

对计算本身进行形式化，被称为“元数学”。这一试图重新为计算树立数学地位的尝试产生了一个杰作—— $\lambda$ 演算。 $\lambda$ 演算名字的由来还有一则有趣的故事。在研究计算本身时，人们意识到应该区分函数和函数的值，例如我们说“如果 $x$ 是奇数那么 $x \times x$ 也是奇数”，这时我们指的是函数的值，而如果我们说“ $x \times x$ 是递增的”，那说的就是这个函数本身。为了区别这两个概念，我们会把函数写成 $x \mapsto x \times x$ 而不单是 $x \times x$ 。

“ $\rightarrow$ ”符号是在1930年前后，由尼古拉·布尔巴基（Nicolas Bourbaki）<sup>16</sup>引入的。二十世纪初，罗素和怀特海在《数学原理》中使用了 $\hat{x}(x \times x)$ 的表示法，1930年代时，丘奇想使用类似的表示法，但是他的出版商不知道如何在 $x$ 上面印出这个“帽子”符号，于是就改成在 $x$ 的前面加上一个与之相似的大写希腊字母 $\Lambda$ ，它后来又变成了小写字母 $\lambda$ 。于是最终表达式就成了今天我们看到的 $\lambda x. x \times x$ <sup>[16]</sup>（页码）。虽然 $x \mapsto x \times x$ 的表示法已经广为接受，人们还是会在逻辑学和计算机科学中使用丘奇的表示法，而这种语言的名字“ $\lambda$ 演算”也正源自于此。



阿隆佐·丘奇 (Alonzo Church, 1903 - 1995)

### 2.3.1 表达式化简

我们先从一些简单的例子开始了解如何用 $\lambda$ 演算形式化算法与计算过程。首先是加减乘除四则运算，我们把它们也看成某种函数。比如加法 $1+2$ ，可以看作用一个名为“ $+$ ”的函数，作用到1和2两个变量上。按照把函数名写在前面的习惯，这一表达式可以写成 $(+ 1 2)$ 。针对表达式的求值，就可以看作是一系列的化简过程，例如：

$$\begin{aligned} & (+ (\times 2 3) (\times 4 5)) \\ \rightarrow & (+ 6 (\times 4 5)) \\ \rightarrow & (+ 6 20) \\ \rightarrow & 26 \end{aligned}$$

这里箭头符号 $\rightarrow$ 读作“化简为”。注意到函数 $f$ 应用到变量 $x$ 上，并没有写成 $f(x)$ 而是写成了 $f x$ 的形式。对于多元函数，如 $f(x, y)$ ，我们不把它写成 $(f(x, y))$ 而是用更加简单一致的方法写成 $((f x) y)$ 。这样为了表达“三加四”这样的加法，需要写成 $((+ 3) 4)$ 。表达式 $(+ 3)$ 实际上表示了一个函数，它把任何传入的变量都加3。这样在整体上这个表达式的含义就是：把加法“ $+$ ”函数先应用到变量3上，这样的结果是一个函数，然后再把这个函数应用到变量4上。这样本质上，我们认为所有的函数都只接受一个参数。这一方法最初是由肖芬格尔（Schönfinkel, 1889 - 1942）在1924年提出，后来经哈斯克尔·克里在1958年后被广泛使用的。因此它被称为函数的“克里化”（Currying）<sup>[17]</sup>（页码）。

严格按照克里化的方式写出的表达式含有很多括弧，为了简化描述，我们在不引起歧义的情况下会省略一些括弧，例如将 $((f ((+ 3) 4)) (g x))$ 简写为 $(f (+ 3 4)) (g x)$ 。

在进行表达式化简时，需要能够理解一些基本含义并做出计算。对于四则运算，我们已经在第一章中介绍的皮亚诺算术的基础上定义了加法和乘法。我们也可以用类似的方式定义其逆运算减法和除法。对于参与运算以及表示结果的常数，我们也有基于零和后继的定义。在这些理论基础上，实现时通常将基本运算和数字内置实现（built-in）以提高性能。通常加以内置实现的还有与或非等逻辑运算、布尔常量真(true)和假(false)。条件表达式可以按照第一章描述的麦卡锡形式 $(p \mapsto f, g)$ 实现，也可以定义为下面的if形式。

<sup>16</sup>“布尔巴基”是一群法国数学家共同使用的笔名。布尔巴基的目的是在集合论的基础上，用最具严格性，最一般的方式来重写整个现代高等数学。布尔巴基学派产生了包括安德烈·韦伊，亨利·嘉当，舒瓦兹，塞尔，格罗滕迪克，迪厄多内等一大批著名数学家。

$$\begin{array}{lll} \text{if } true & \text{then } e_t \text{ else } e_f & \mapsto e_t \\ \text{if } false & \text{then } e_t \text{ else } e_f & \mapsto e_f \end{array}$$

其中 $e_t$ 和 $e_f$ 都是表达式。第一章中通过 $cons$ 定义的复合数据结构，也可以通过函数来抽取其中的各个部分：

$$\begin{array}{l} head (cons a b) \mapsto a \\ tail (cons a b) \mapsto b \end{array}$$

### 2.3.2 $\lambda$ 抽象

我们前面简单介绍了 $\lambda$ 符号的由来，所谓 $\lambda$ 抽象，实际上是一种构建函数的方法。我们通过一个例子来了解 $\lambda$ 抽象的各个组成部分。

$$(\lambda x. + x 1)$$

一个 $\lambda$ 抽象包含四个组成部分，首先是 $\lambda$ 符号，表示“接下来要定义一个函数”。紧随其后的是变量，在本例中就是 $x$ ，被称为形参（formal parameter）。形参之后是一个点，剩余部分是函数体，它向右延伸到最长，在本例中是 $+ x 1$ 。有时为了避免对函数体的右边界产生歧义，可以增加括号，对于本例，可以写成： $(+ x 1)$ 。为了记忆方便，我们可以将 $\lambda$ 抽象的四个部分按照如下方法对应到自然语言上。

$$\begin{array}{cccc} (\lambda & x & . & + x 1) \\ \uparrow & \uparrow & \uparrow & \uparrow \\ \text{函数的} & \text{自变量是}x & \text{它} & \text{将}x\text{和}1\text{相加} \end{array}$$

为了方便，在后续的推导中我们也会等价地使用 $x \mapsto x + 1$ 形式的记法。这里有一点需要澄清， $\lambda$ 抽象并不等同于 $\lambda$ 表达式。 $\lambda$ 抽象只是 $\lambda$ 表达式的一种情况， $\lambda$ 表达式还包括其它三种情况：

$$\begin{array}{lll} <\text{表达式}> = & <\text{常量}> & \text{内置的常量，数字、布尔值等} \\ & | & \text{变量名} \\ & | & <\text{表达式}> <\text{表达式}> & \text{应用} \\ & | & \lambda <\text{变量}> . <\text{表达式}> & \lambda\text{抽象} \end{array}$$

### 2.3.3 $\lambda$ 变换规则

考虑下面的 $\lambda$ 表达式，如果对它化简，我们需要知道“全局”变量 $y$ 的值。与之相对，我们不需要事先知道变量 $x$ 的值，因为它以形参出现在函数中。

$$(\lambda x. + x y) 2$$

比较 $x$ 和 $y$ 的不同之处，我们称 $x$ 是被 $\lambda$ “绑定”的。当将这一 $\lambda$ 抽象应用到参数2时，我们会用2替换掉所有的 $x$ 。相反， $y$ 没有被 $\lambda$ 绑定，我们称变量 $y$ 是自由的。总之，表达式的值是由未被绑定的自由变量的值决定的。一个变量要么是被绑定的，要么是自由的。下面是一个稍复杂点的例子：

$$\lambda x. + ((\lambda y. + y z) 3) x$$

我们可以把它写成箭头记法，这样可以看得更清楚：

$$x \mapsto ((y \mapsto y + z) 3) + x$$

这样就可以看出， $x$ 与 $y$ 是被绑定的，而 $z$ 是自由变量。在更加复杂的表达式中，同一变量的名字，有时是被绑定的，有时又以自由变量的形式出现，例如：

$$+ x ((\lambda x. + x 1) 2)$$

写成箭头形式为：

$$x + ((x \mapsto x + 1) 2)$$

我们看到，第一次出现的  $x$  是个自由变量，而后面出现的  $x$  是被绑定的。在复杂的表达式中，这种同名代表不同的变量的情况会带来麻烦。为了解决名称冲突的问题，我们引入第一条  $\lambda$  变换规则—— $\alpha$ -变换。其中  $\alpha$  是希腊字母阿尔法。这一规则说，我们可以将  $\lambda$  表达式中的一个变量，重新命名为另一个变量。例如：

$$\lambda x. + x 1 \quad \xleftarrow{\alpha} \quad \lambda y. + y 1$$

写成箭头形式为：

$$x \mapsto x + 1 \quad \xleftarrow{\alpha} \quad y \mapsto y + 1$$

我们说  $\lambda$  抽象是一种构建函数的方法，如何将构建好的函数应用到参数值上呢？这就需要引入第二条  $\lambda$  变换规则—— $\beta$ -变换。正向使用这条规则时，把  $\lambda$  抽象函数体中的所有形参的自由出现替换成形参的值。例如：

$$(x \mapsto x + 1) 2$$

根据变换规则，把  $\lambda$  抽象  $x \mapsto x + 1$  应用到自变量 2 上得  $2 + 1$ 。即  $2 + 1$  是将函数体  $x + 1$  中出现的形参  $x$  替换为 2 的结果。用箭头将这一变换表示为：

$$(x \mapsto x + 1) 2 \quad \xrightarrow{\beta} \quad 2 + 1$$

我们称这一特定箭头方向的变换为  $\beta$ -规约 ( $\beta$ -reduction)<sup>17</sup>。而反向使用这条规则称为  $\beta$ -抽象。下面再通过更多的例子了解一下  $\beta$ -规约。首先是形参多次出现的例子：

$$(x \mapsto x \times x) 2 \quad \xrightarrow{\beta} \quad 2 \times 2 \\ \xrightarrow{\beta} \quad 4$$

然后是形参出现零次的情况：

$$(x \mapsto 1) 2 \quad \xrightarrow{\beta} \quad 1$$

这是一个典型的“常量映射”的例子。接下来是一个多重规约的例子：

$$(x \mapsto (y \mapsto y - x)) 2 4 \quad \begin{array}{l} \xrightarrow{\beta} (y \mapsto y - 2) 4 \quad \text{克里化} \\ \xrightarrow{\beta} 4 - 2 \quad \text{内层规约} \\ \xrightarrow{\beta} 1 \quad \text{内置四则运算} \end{array}$$

可以看到，从外向内逐层规约是一个不断克里化的过程。有时我们把多重规约简写如下：

$$(\lambda x. (\lambda y. E)) \Rightarrow (\lambda x. \lambda y. E)$$

其中  $E$  表示函数体。写成箭头形式为：

$$(x \mapsto (y \mapsto E)) \Rightarrow (x \mapsto y \mapsto E)$$

---

<sup>17</sup>也称为  $\beta$ -消解或  $\beta$ -化简

使用 $\beta$ -规约进行函数应用时，参数也可以是另一个函数，例如：

$$\begin{array}{c} (f \mapsto f\ 5) \ (x \mapsto x + 1) \\ \xrightarrow{\beta} \quad (x \mapsto x + 1)\ 5 \\ \xrightarrow{\beta} \quad 5 + 1 \\ \longrightarrow \quad 6 \end{array}$$

最后一个我们要介绍的变换是 $\eta$ -变换。它的定义如下：

$$(\lambda x.F\ x) \quad \xleftrightarrow{\eta} \quad F$$

写成箭头形式为：

$$x \mapsto F\ x \quad \xleftrightarrow{\eta} \quad F$$

其中 $F$ 是函数，且 $x$ 不是 $F$ 中的自由变量。我们来看一个例子：

$$(\lambda x. +\ 1\ x) \quad \xleftrightarrow{\eta} \quad (+\ 1)$$

在这个例子中， $\eta$ -变换两边的行为表现得完全一样。如果应用到一个参数上，效果都是把这个参数增加1。 $\eta$ -变换之所以要求 $x$ 不能是 $F$ 的自由变量是为了避免错误地将 $(\lambda x. +\ x\ x)$ 变换为 $(+\ x)$ 。我们可以看到， $x$ 是 $(+\ x)$ 中的自由变量。同样，限定 $F$ 必须为函数是为了避免错误地将1变换为 $(\lambda x. 1\ x)$ 。在上面的定义中，称从左向右的变换为 $\eta$ -规约。

至此，我们介绍了 $\lambda$ 表达式变换的三大规则，我们小结一下。

1.  $\alpha$ -变换用于改变形参的名字；
2.  $\beta$ -规约用于实现函数应用；
3.  $\eta$ -规约用于去除多余的 $\lambda$ 抽象。

此外，我们称内置函数的化简，如加减乘除四则运算、逻辑上的与或非等为 $\delta$ -变换。在某些文献上，还会看到另外一种 $\lambda$ 表达式变换的简记形式，我们这里简单介绍一下。对表达式 $(\lambda x.E)\ M$ ，进行 $\beta$ -规约时，我们用 $M$ 替换 $E$ 中的 $x$ ，将此结果记为 $E[M/x]$ 。这样三大变换就可以简写成下面的形式：

变换	$\lambda$ 形式	箭头形式
$\alpha$	$(\lambda x.E) \xleftrightarrow{\alpha} \lambda y.E[y/x]$	$x \mapsto E \xleftrightarrow{\alpha} y \mapsto E[y/x]$
$\beta$	$(\lambda x.E)\ M \xleftrightarrow{\beta} E[M/x]$	$(x \mapsto E)\ M \xleftrightarrow{\beta} E[M/x]$
$\eta$	$(\lambda x.E)\ x \xleftrightarrow{\eta} E$	$x \mapsto E\ x \xleftrightarrow{\eta} E$

由于这些转换规则都是双向的，在对一个 $\lambda$ 表达式进行变换时，既可以从左向右进行规约，也可以反过来从右向左进行抽象。这样自然会产生两个问题。第一个问题是，化简过程最终会停止么？第二个问题是，不同的化简方式得到的结果是一致的么？对于第一个问题，答案是不确定的，化简过程不能保证一定会中止<sup>18</sup>。下面就是一个“死循环”的例子： $(D\ D)$ ，其中 $D$ 定义为 $\lambda x.x\ x$ ，写成箭头形式为 $x \mapsto x\ x$ 。如果我们试图化简，就会得到这样的结果：

$$\begin{array}{ll} (D\ D) & \rightarrow (x \mapsto x\ x)\ (x \mapsto x\ x) \quad \text{代入 } D \text{ 的定义} \\ & \xrightarrow{\alpha} (x \mapsto x\ x)\ (y \mapsto y\ y) \quad \text{对第二个 } \lambda \text{ 抽象用 } \alpha \text{-变换} \\ & \xrightarrow{\beta} (y \mapsto y\ y)\ (y \mapsto y\ y) \quad \text{用第二个表达式替换 } x \\ & \xrightarrow{\alpha} (x \mapsto x\ x)\ (x \mapsto x\ x) \quad \text{再用 } x \text{ 替换回 } y \\ & \rightarrow (x \mapsto x\ x)\ (x \mapsto x\ x) \quad \text{重复使用上述步骤} \\ & \dots \end{array}$$

<sup>18</sup>注意答案并不是“否定”，而是不确定。这本质上和图灵停机问题是一致的。即不存在一个可判定过程，确定任意化简是否中止。我们在最后一章会详细讨论这个问题。

更耐人寻味的是这个例子:  $(\lambda x.1)(D D)$ , 如果先化简 $(\lambda x.1)$ , 那么化简过程会中止, 答案为1。反之如果先化简 $(D D)$ , 则根据上面的结论, 这一过程就会陷入“死循环”而无法中止。丘奇和他的学生罗瑟<sup>19</sup>在1936年证明了一对定理, 完整地回答了第二个问题。

**定理 2.3.1 (丘奇-罗瑟定理一).** 若  $E_1 \leftrightarrow E_2$ , 则存在  $E$ , 使得  $E_1 \rightarrow E$  且  $E_2 \rightarrow E$ 。

也就是说, 如果化简过程是可终止的, 则化简结果是一致的。不同的化简方法会化简到同一结果, 如图2.14所示。在此基础上, 丘奇和罗瑟又证明了第二定理。为此我们先要给出“范式”(Normal form)的概念, 所谓范式, 又称 $\beta$ 范式, 是指不能再进行 $\beta$ -规约的形式。简单来讲, 就是表达式中能够应用的函数都已经应用了。更严格的范式是 $\beta - \eta$ 范式, 在这样的范式中, 既不能进行 $\beta$ -规约, 也不能进行 $\eta$ -规约。例如 $(x \mapsto x + 1)y$ 不是范式, 因为它可以进行 $\beta$ -规约变成 $y + 1$ 。下面是范式的递归定义。

$normal((\lambda x.y) z)$	=	false	可进一步 $\beta$ -规约
$normal(\lambda x.(f x))$	=	false	可进一步 $\eta$ -规约
$normal(x y)$	=	$normal(x) \wedge normal(y)$	应用: 函数和参数都是范式
$normal(x)$	=	true	其它情况

**定理 2.3.2 (丘奇-罗瑟定理二).** 若  $E_1 \rightarrow E_2$ , 且  $E_2$  为范式, 则存在从  $E_1$  化简到  $E_2$  的正规顺序。

注意, 这一定理要求化简过程也必须是可终止的。所谓正规顺序 (normal order) 就是从左向右, 从外向内的化简顺序。

## 2.4 递归的定义

利用上一节中介绍的 $\lambda$ 抽象, 我们已经可以定义一些简单函数, 但是如何定义递归函数呢? 比如阶乘, 它可以递归地定义为:

$$fact = n \mapsto \text{if } n = 0 \text{ then } 1 \text{ else } n \times fact(n - 1)$$

但这并不是一个合法的 $\lambda$ 表达式, 问题是由于 $\lambda$ 抽象只定义了匿名函数, 而我们并未定义如何给函数命名。再次观察递归的阶乘定义, 它形如:

$$fact = n \mapsto (\dots fact \dots)$$

我们反向使用 $\beta$ -规约 (即 $\beta$ -抽象), 将其变换为:

$$fact = (f \mapsto (n \mapsto (\dots f \dots))) fact$$

进一步可以抽象为

$$fact = H fact \tag{2.7}$$

其中

<sup>19</sup>罗瑟 (John Barkley Rosser Sr. 1907 - 1989) 是美国数学家、逻辑学家。除了丘奇-罗瑟定理外, 罗瑟还和克莱尼一起还提出了克莱尼-罗瑟悖论, 在数论领域, 他提出了罗瑟筛法, 并证明了罗瑟定理, 该定理指出, 第  $n$  个素数  $p_n > n \ln n$ 。罗瑟在1936年还给出了一种更强的哥德尔第一不完全定理的形式, 他把不可判定命题改进为: “对本命题的任何证明, 都存在一个对否命题的更短证明。”

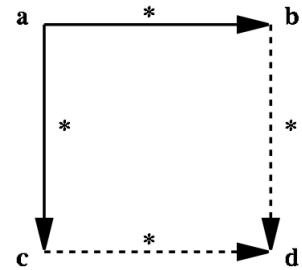


图 2.14: 丘奇-罗瑟定理的示意

$$H = f \mapsto (n \mapsto (\dots f \dots))$$

注意到，经过这样的变换，我们得到的 $H$ 不再是递归的，它是一个普通的 $\lambda$ 表达式。观察式(2.7)，它表示了递归。在形式上它是一个数学方程，让我们联想起“微分方程”的概念。例如解微分方程 $y' = \sin(x)$ 得到 $y = a - \cos(x)$ 。如果我们能将方程 $F = H F$ 解出，就能完整地定义阶乘了。进一步观察，我们发现这个方程的含义是，将 $H$ 应用到 $F$ 上，得到的结果仍然是 $F$ 。这一概念在数学上叫做“不动点”。我们称 $F$ 是 $H$ 的不动点。再举一个不动点的例子： $\lambda$ 表达式 $x \mapsto x \times x$ 的不动点是0和1，这是因为 $(x \mapsto x \times x) 0 = 0$ 且 $(x \mapsto x \times x) 1 = 1$ 。

### 2.4.1 Y组合子

我们希望找到 $H$ 的不动点，显然 $H$ 的不动点只依赖于 $H$ 本身。为此，我们引入一个函数 $Y$ ，它接受一个函数，然后返回这个函数的不动点。 $Y$ 的行为表现如下：

$$Y H = H (Y H) \quad (2.8)$$

为此 $Y$ 被称为“不动点组合子”(fixpoint combinator)。使用 $Y$ ，我们就得到了方程(2.7)的解：

$$fact = Y H \quad (2.9)$$

这样得到的 $fact$ 是一个无递归的定义。我们可以这样验证这个解：

$$\begin{aligned} fact &= Y H && \text{式(2.9)} \\ &= H (Y H) && \text{式(2.8)} \\ &= H fact && \text{式(2.9)反向} \end{aligned}$$

$Y$ 的强大之处在于它可以表达所有的递归函数。但是现在的 $Y$ 仍然是个黑盒子，我们需要用 $\lambda$ 抽象来真正实现它。下面就是 $Y$ 的实现：

$$Y = \lambda h.(\lambda x.h (x x)) (\lambda x.h (x x)) \quad (2.10)$$

写成箭头形式为：

$$Y = h \mapsto (x \mapsto h (x x))(x \mapsto h (x x))$$

我们打开了潘多拉的魔盒，现在来验证一下这个 $Y$ 的 $\lambda$ 抽象是否表现得符合我们的预期： $Y H = H (Y H)$ 。

证明.

$$\begin{aligned} Y H &= (h \mapsto (x \mapsto h (x x))(x \mapsto h (x x))) H && Y \text{的定义} \\ &\xrightarrow{\beta} (x \mapsto H (x x)) (x \mapsto H (x x)) && \beta\text{-规约, 用 } H \text{代换 } h \\ &\xrightarrow{\alpha} (y \mapsto H (y y)) (x \mapsto H (x x)) && \text{对前半部分用 } \alpha\text{-变换} \\ &\xrightarrow{\beta} H ((x \mapsto H (x x)) (x \mapsto H (x x))) && \beta\text{-规约, } y \text{被后半部代换} \\ &\xrightarrow{\beta} H (h \mapsto (x \mapsto h (x x))(x \mapsto h (x x))) H && \beta\text{-抽象, } H \text{抽出为参数} \\ &= H (Y H) && \text{代入 } Y \text{的定义} \end{aligned}$$

□

最终，使用 $Y$ ，我们可以如下实现阶乘的定义：

$$Y (f \mapsto (n \mapsto \text{if } n = 0 \text{ then } 1 \text{ else } n \times f (n - 1)))$$

用 $\lambda$ -抽象来定义 $Y$ 的数学意义远大于其实现的意义。在真实的环境中，通常 $Y$ 被内置实现为函数，可以直接将 $Y H$ 转化为 $H (Y H)$ 。

## 2.5 $\lambda$ 演算的意义

$\lambda$ 演算的意义在于，它用一组简单的规则对复杂的计算过程进行了定义。将欧几里得算法用 $\lambda$ 表示出来，再利用 $\beta$ 规约进行计算。这样的实现可能比较复杂，但确实可行。它不仅可以表达这一函数，而是对所有的可计算函数都使用。图灵后来证明了 $\lambda$ 演算与图灵机的等价性。 $\lambda$ 演算的一大优势在于它仅仅使用了传统的数学概念——函数。因而在十九世纪30年代， $\lambda$ 演算被用来对数学进行形式化。然而在1935年，克莱尼-罗瑟悖论证明了最初的 $\lambda$ 形式系统在逻辑上是不一致的。1936年，丘奇将 $\lambda$ 演算模型中和纯计算有关的部分分离出来，称为的无类型 $\lambda$ 演算。1940年时，丘奇又提出了一个弱化计算，但是逻辑自治的形式系统，被称之为简单类型 $\lambda$ 演算。

我们展示了如何用 $\lambda$ 演算定义基本的四则运算、逻辑运算、定义和应用普通函数以及递归函数。最后，我们给出如何用 $\lambda$ 演算定义复合数据结构。我们用此前定义的 $cons$ 、 $head$ 、 $tail$ 作为例子。以下是它们的 $\lambda$ 抽象定义：

$$\begin{aligned} cons &= (\lambda a. \lambda b. \lambda f. f a b) \\ head &= (\lambda c. c (\lambda a. \lambda b. a)) \\ tail &= (\lambda c. c (\lambda a. \lambda b. b)) \end{aligned}$$

写成箭头形式为：

$$\begin{aligned} cons &= a \mapsto b \mapsto f \mapsto f a b \\ head &= c \mapsto c (a \mapsto b \mapsto a) \\ tail &= c \mapsto c (a \mapsto b \mapsto b) \end{aligned}$$

我们来验证一下 $head (cons p q) = p$ 这一关系。

$$\begin{aligned} head (cons p q) &= (c \mapsto c (a \mapsto b \mapsto a)) (cons p q) \\ &\xrightarrow{\beta} (cons p q) (a \mapsto b \mapsto a) \\ &= ((a \mapsto b \mapsto f \mapsto f a b) p q) (a \mapsto b \mapsto a) \\ &\xrightarrow{\beta} ((b \mapsto f \mapsto f p b) q) (a \mapsto b \mapsto a) \\ &\xrightarrow{\beta} (f \mapsto f \mapsto f p q) (a \mapsto b \mapsto a) \\ &\xrightarrow{\beta} (a \mapsto b \mapsto a) p q \\ &\xrightarrow{\beta} (b \mapsto p) q \\ &\xrightarrow{\beta} p \end{aligned}$$

这说明，理论上复合数据结构不必一定要内置实现，而可以用 $\lambda$ 定义。本章练习中还展示了如何用 $\lambda$ 演算基于皮亚诺公理系统定义自然数、布尔值及逻辑运算。

### 练习 2.2

1. 使用 $\lambda$ 变换规则验证 $tail (cons p q) = q$ 。
2. 可以仅仅使用 $\lambda$ 演算来定义自然数。下面是丘奇数的定义：

$$\begin{aligned} 0 &: \lambda f. \lambda x. x \\ 1 &: \lambda f. \lambda x. f x \\ 2 &: \lambda f. \lambda x. f (f x) \\ 3 &: \lambda f. \lambda x. f (f (f x)) \\ &\vdots \dots \end{aligned}$$

请利用第一章介绍的内容，定义丘奇数的加法和乘法。

3. 以下是丘奇布尔值的定义，以及逻辑运算的一种实现：

$$\begin{aligned}\text{true} &: \lambda x. \lambda y. x \\ \text{false} &: \lambda x. \lambda y. y \\ \text{and} &: \lambda p. \lambda q. p \ q \ p \\ \text{or} &: \lambda p. \lambda q. p \ q \\ \text{not} &: \lambda p. p \ \text{false} \ \text{true}\end{aligned}$$

其中`false`的定义和丘奇数0的定义本质上是相同的。试用 $\lambda$ 变换证明：`and true false = false`；你能给出`if...then...else...`语句的 $\lambda$ 定义么？

## 2.6 更多的递归结构

至此，我们已经将递归函数和递归数据结构建立在完整的数学基础上。利用这些工具，我们可以继续定义一些较复杂的数据结构。例如下面的二叉树。

```
data Tree A = nil | node (Tree A, A, Tree A)
```

这个递归定义说，一棵元素类型为A的二叉树或者为空，或者是一个分支节点。节点包含三部分：两棵元素类型为A的子树和一个类型为A的元素。习惯上我们称这两棵子树为左子树和右子树。A是类型参数，例如自然数。`node(nil, 0, node(nil, 1, nil))`表示了一棵元素为自然数的二叉树。下面我们为二叉树定义抽象的叠加操作`foldt`。

$$\begin{aligned}\text{foldt}(f, g, c, \text{nil}) &= c \\ \text{foldt}(f, g, c, \text{node}(l, x, r)) &= g(foldt(f, g, c, l), f(x), foldt(f, g, c, r))\end{aligned}\tag{2.11}$$

如果函数`f`将类型为A的自变量映射为类型为B的值，我们将其类型记为 $f : A \rightarrow B$ 。克里化的函数`foldt(f, g, c)`的类型为 $foldt(f, g, c) : \text{Tree } A \rightarrow B$ ，其中`c`的类型为`B`，而函数`g`的类型为 $g : (B \times B \times B) \rightarrow B$ ，写成克里化的形式为 $g : B \rightarrow B \rightarrow B \rightarrow B$ 。使用抽象的叠加函数`foldt`我们可以定义针对二叉树的逐一映射`mapt`：

$$\text{mapt}(f) = \text{foldt}(f, \text{node}, \text{nil})\tag{2.12}$$

通过叠加操作，还可以定义函数来统计一棵二叉树中的元素个数：

$$\text{size} = \text{foldt}(\text{one}, \text{sum}, 0)\tag{2.13}$$

其中`one(x) = 1`，是一个常数函数，它对任何变量都返回1。`sum`是一个三元的累加函数： $\text{sum}(a, b, c) = a + b + c$ 。

在此基础上，复合使用第一章定义的列表，我们还可以从二叉树扩展到多叉树，下面是一个多叉树的定义：

```
data MTree A = nil | node (A, List (MTree A))
```

一棵元素类型为A的多叉树或者为空，或者为一个复合节点。复合节点包括一个类型为A的元素，和若干子树。所有子树用一个多叉树的列表来表示。定义多叉树的抽象叠加操作需要相互递归调用列表的叠加操作。

$$\begin{aligned}\text{foldm}(f, g, c, \text{nil}) &= c \\ \text{foldm}(f, g, c, \text{node}(x, ts)) &= \text{foldr}(g(f(x), c), h, ts) \\ h(t, z) &= \text{foldm}(f, g, z, t)\end{aligned}\tag{2.14}$$

### 练习 2.3

1. 不用抽象的叠加操作 $foldt$ , 通过递归定义二叉树的逐一映射 $mapt$ ;
2. 定义一个函数 $depth$ , 计算一棵二叉树的最大深度;
3. 有人认为, 二叉树的抽象叠加操作 $foldt$ 应该这样定义:

$$\begin{aligned} foldt(f, g, c, nil) &= c \\ foldt(f, g, c, node(l, x, r)) &= foldt(f, g, g(foldt(f, g, c, l), f(x)), r) \end{aligned}$$

也就是说,  $g : (B \times B) \rightarrow B$  是一个类似于加法这样的二元函数。能否利用这个 $foldt$ 定义逐一映射 $mapt$ ?

4. 排序二叉树 (又称二叉搜索树) 是一种特殊的二叉树, 如果二叉树的元素类型 $A$ 是可比较的, 并且对任何非空节点 $node(l, k, r)$ 都满足: 左子树 $l$ 中的任何元素都小于 $k$ , 右子树 $r$ 中的任何元素都大于 $k$ 。定义二叉树的插入函数 $insert(x, t) : (A \times TreeA) \rightarrow TreeA$
5. 为多叉树定义逐一映射。能否利用多叉树的叠加操作来定义? 如果不能, 应当怎样修改叠加操作?

## 2.7 递归的形式与结构

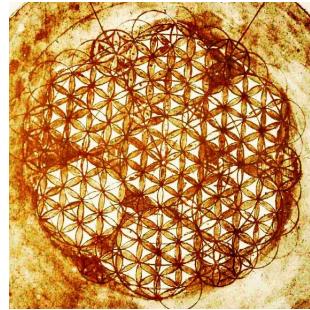
递归不仅存在于古老的欧几里得算法和现代计算机系统中, 它迷人的形式和结构还出现在各种人类文明艺术中。图2.15的平面镶嵌中, 可以看到多重不同的递归形式。



图 2.15: 马拉喀什 (摩洛哥城市) 的瓷砖图案

我们可以看到瓷砖中的多边形图案中递归地含有更小的多边形嵌套。通过不同颜色的瓷砖, 这种递归展现出几何形式的美。这些图案还在更大的范围组成条纹形式, 而各个条纹内部都是不同的递归图形。图2.16左边展示的是文艺复兴时期艺术大师达·芬奇手稿中的递归图样。在圆周上使用相同的半径, 依次可作出六个相互交织的圆, 从而构成一个六瓣的花样图案。而在更大的范围内, 又递归地构成了同样形式的图案。左侧是中国民间手工编织的蝈蝈笼子, 他们展示了类似的递归图样。笼子的网眼是六边形的, 而笼子的整体形状从轴向看去也是六边形的。

递归不仅出现在美术作品中, 还出现在更加抽象的艺术形式中, 例如音乐。复调音乐中的卡农 (Canon) 和赋格 (Fugue) 都是这方面的代表。卡农的所有声部虽然都模仿一个声部, 但不同高度的声部依一定间隔进入, 造成一种此起彼伏, 连绵不断的效果。每一个声部都地归地展示主题但却含有各种变化, 比如音调的升高或降低、逆行重叠、速度变快 (减值) 或变慢 (增值) 、旋律倒影等等。



(a) 达芬奇绘制的递归图案手稿



(b) 手工编织的蝈蝈笼子

图 2.16: 艺术和生活物品中的递归形式

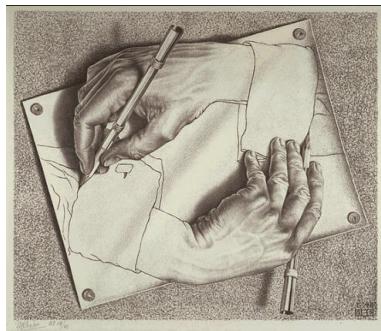


图 2.17: 艾舍尔的作品《画手》，1948年

出例子，它们绝不可能“照公式创造出来”。这两首曲子都具有远比赋格的性质更为深刻的东西 [5]。

这些都是有限递归的例子。图2.17是荷兰艺术大师艾舍尔在1948年创作的《画手》，两只手递归地在描绘着对方。《画手》是无穷递归的典型例子。画中上方的手正在用一支笔描绘着下方的手，而下方的手也在地归地描绘着上方的手。这组相互递归的嵌套一层一层，无穷无尽。

无穷递归的一个数学与艺术完美结合的例子是“分形”(fractal)。例如著名的分形图案“科克雪花”(Koch snowflake)曲线可以通过这样的无穷递归规则来产生：对图形中的每个线段，均分为三段，以第二段为底向上方作出一个等腰三角形，然后再把这个底擦掉。图2.18展示了三重递归后从一个正三角形得到的科克雪花。另一个著名的递归分形图案是谢尔平斯基三角形(Sierpinski)，它的生成规则是对图形中的每个三角形，分别取三边的中点，连成一个内部的小三角形，然后“挖掉这个内部的小三角形”。下图展示了递归四次后的谢尔平斯基三角形。

最后让我们以两张分形图案来结束本章，其中一张是人类思维产生的分形——茱莉亚集(Julia)；另一张是自然界产生的分形。

赋格通常建立在一个主题上，以不同的声部、不同的调子、偶尔也用不同的速度或上下颠倒或从后往前地进行演奏。然而，赋格的概念远不如卡农那么严格，因而允许有更多的感情或艺术的表现。赋格的识别标志的是它的开始方式：单独的一个声部唱出它的主题，唱完后，第二个声部或移高五度或降低四度进入。与此同时，第一个声部继续唱“对应主题”，也叫第二主题，用来在节奏、和声、及旋律方面与主题形成对比。每个声部依次唱出主题，常常是另一个声部伴唱对应主题，其它的声部所起的作用随作曲家的想象而定。当所有的声部都“到齐”了，就不再有什么规则了。当然，还是有一些标准的手法，但它没有严格到只能按照某个公式去创作赋格。《音乐的奉献》中的两首赋格曲就是杰

## 2.8 扩展阅读

韩雪涛的《数学悖论与三次数学危机》[9]对古希腊数学的成就有生动的介绍。欧几里得的《几何原本》中译本[11]是永远的经典。斯捷潘诺夫和罗斯的《数学与泛型编程》[10]中有欧几里得算法算法的多种实现。随着各大主流编程环境引入lambda演算，这方面的介绍材料很多。佩顿琼斯的《函数式编程语言的实现》[17]讲解全面并且深入浅出。侯世达先生的《集异璧》[5]中有大量关于

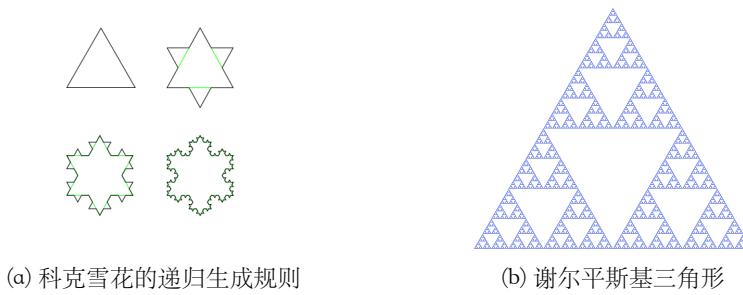


图 2.18: 递归产生的分形图案

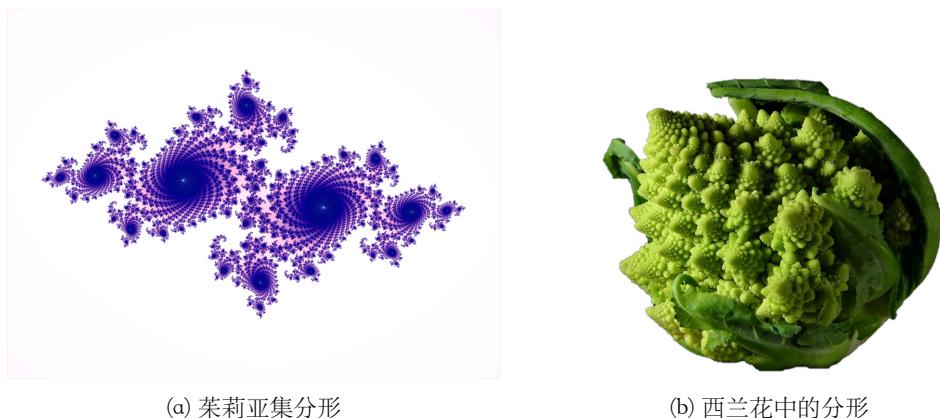


图 2.19: 人类思维和自然界产生的分形

递归的介绍和奇思妙想，曾获得普利策大奖。中译本秉着文化同构的精神进行了非凡的再创作。

## 第3章 群、环、域

你只要能把自己提出的那些“点、线、面”都说的跟“桌子、椅子、啤酒杯子”一样自然连贯就行。

——大卫·希尔伯特

我们人类在长期的生产生活中逐渐养成了对事物分类整理的习惯。不同但相近的东西被归为一类。对整个类适用的性质和方法，对类中的不同事物都有效。这样我们就从解决具体的单一的问题，提高到一下子解决整类的抽象问题，极大地丰富了我们认识掌握世界的能力。

在第一章中，我们曾经从自然数的累加和阶乘中归纳抽象出“叠加”操作。我们观察它们相似的结构，将累加中的0和阶乘中的1抽象成单位元，将累加中的加法和阶乘中的乘法抽象成某种二元运算，这样就在更高的层次抽象出了自然数的叠加。进而可以用这一抽象的叠加解决诸如斐波那契数列这样的一大类和自然数同构的问题。

再举一个例子。我们在第一章中，还定义了列表上的抽象叠加操作 $foldr$ 。可以利用这个抽象工具把一列数累加起来： $sum = foldr(0, +)$ ，也可以把一列数乘到一起： $product = foldr(1, \times)$ 。在计算机程序中，人们可以定义一种叫做“二叉搜索树”的数据结构。第二章中，我们曾经介绍过二叉树的定义。二叉搜索树是一种特殊的二叉树，它要求树中元素的类型是可以比较的<sup>1</sup>，并且对于任意分支节点，节点左侧子树中的所有元素都在此节点的元素之前，而右子树中的所有元素都在此元素之后。由于这种有序性，人们也把它叫作“排序二叉树”。对排序二叉树，我们可以定义一个插入操作：



艾舍尔《解放》

$$\begin{aligned} insert(nil, x) &= node(nil, x, nil) \\ insert(node(l, y, r), x) &= \begin{cases} x < y : node(insert(l, x), y, r) \\ x > y : node(l, y, insert(r, x)) \end{cases} \end{aligned}$$

根据这一定义，将一个元素 $x$ 插入到二叉搜索树时，如果树为空，则结果为一个 $node(nil, x, nil)$ ；否则，需要进一步比较 $x$ 和分支节点中的元素 $y$ ，如果 $x$ 在前（用小于号代表），则递归地插入到左子树，否则递归地插入到右子树。联想累加和阶乘，插入操作和它们有什么相似之处呢？插入也是一个二元操作，而 $nil$ 相当于单位元。这样我们就可以用抽象的叠加操作，将一列数字变成一棵搜索树：

<sup>1</sup>这里比较的含义是抽象的，例如可以比大小，也可以是比较单词在字典中的先后顺序

$$build = foldr(nil, insert)$$

图3.2展示了计算 $build [4, 3, 1, 8, 2, 16, 10, 7, 14, 9]$ 产生的一棵二叉搜索树。

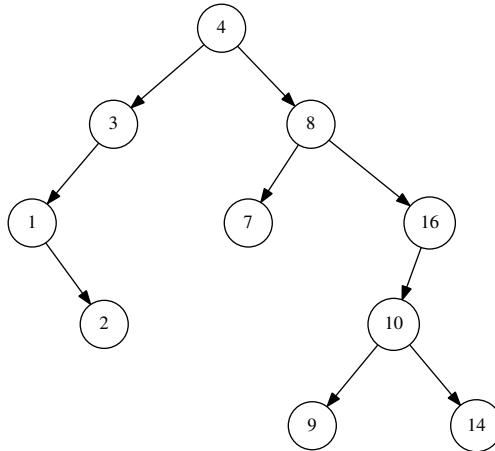


图 3.2: 通过叠加产生的二叉搜索树

我们人类对加法的认识也经历了类似的过程，最初加法是针对具体事物的，例如采集的果实或者猎物。然后逐渐将具体事物的意义去除变成了针对抽象整数的加法。后来对数的认识扩充到了分数，虽然分数的具体相加过程和整数大相径庭，需要先对分母进行通分，然后将分子按照整数的法则相加，最后再进行约分。但是我们进一步将这两种不同的操作抽象成了有理数上的加法，人们深入思考它们的本质和性质，在每一次扩充数的认识时，都重新定义更加抽象的加法。从而有了实数上，复数上的加法。人们发现，对去除了具体意义的抽象事物成立的方法具备更大的应用范围。抽象方法能解决一类问题，而不是单一问题。计算机编程中逐渐发展出的面向对象方法，带泛型的类型系统，和动态类型系统都是某种意义上对抽象的一种支持机制。

在发展抽象的方法和工具时，必须时刻关注一个问题：“某一种抽象的适用范围有多大？什么情况下这一抽象会失效？”如果忽略了这一点，就会产生令人啼笑皆非的结果。比如人们总结抽象出了无穷几何级数的累加公式： $1 + x + x^2 + x^3 + \dots = 1/(1-x)$ 。并用它成功解决了芝诺悖论<sup>2</sup>。但是17世纪的数学家们把-1代入这个公式得到了 $1 - 1 + 1 - 1 + \dots = 1/2$ 这样的结果。这时有人提出了不同的意见： $S = (1 - 1) + (1 - 1) + \dots = 0$ ；还有人认为： $S = 1 + (-1 + 1) + (-1 + 1) + \dots = 1$ 。有人赞同 $1/2$ 的结果，因为 $S = 1 - (1 - 1 + 1 - 1 + \dots) = 1 - S$ ，解方程得 $S = 1/2$ 。意大利数学教授格兰迪（1671 - 1742）还发现了更令人吃惊的结论。通过无穷级数：

$$\frac{1}{1+x+x^2} = 1 - x + x^3 - x^4 + x^6 - x^7 + \dots$$

$$\frac{1}{1+x+x^2+x^3} = 1 - x + x^4 - x^5 + x^8 - x^9 + \dots$$

$$\dots$$

<sup>2</sup>古希腊哲学家芝诺提出了四个著名的悖论，这里是指阿基里斯与乌龟悖论。阿基里斯是古希腊著名的英雄。芝诺假设阿基里斯追赶上前面的乌龟，当他跑到乌龟出发的位置时，乌龟已经向前移动了一小段距离。阿基里斯必须继续跑到乌龟的新位置，但是此时乌龟又向前移动了。重复这一过程，芝诺认为阿基里斯永远赶不上乌龟。本书在第五章详细解释芝诺悖论。

令 $x = 1$ , 可以得到无穷级数 $1 - 1 + 1 - 1 + 1 - 1 + \dots$ 的和可以等于 $1/3, 1/4, \dots$ 就连大数学家莱布尼茨也主张, 它的和可能是0或者1, 而且概率相等, 所以其“真”值应该是它们的平均值 $1/2$ 。格兰迪还做出了另一个“奇妙”的解释: 父亲留给两个儿子一块宝石, 由兄弟二人轮流保存, 每人一年。于是, 交给对方保存的时候, 可以说所有权是0, 自己保存的时候, 可以说所有权是1, 而平均而言, 每人的所有权为 $1/2$ <sup>[9]</sup>。这些奇怪的问题, 要等到法国数学家柯西引入无穷级数收敛性后才得以解决。

编程中也有类似的问题。有一次在编写“自然归并排序”的算法时<sup>3</sup>, 我需要将一列元素分成从大到小的若干组, 例如把序列 $[15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2]$ 分组成 $[[15, 9, 0], [12, 11, 7], [10, 5], [6], [13, 1], [4], [8, 3], [14, 2]]$ 。我知道标准库中提供了一个抽象工具`groupBy`, 它可以根据一个定义好的条件完成分组, 例如: `groupBy (==) "Mississippi"`会得到结果`["M", "i", "ss", "i", "ss", "i", "pp", "i"]`。这样连续相等的元素就被分在一组。然而当我把大于等于作为分组条件, 却得到了一个不正确的结果。`groupBy (>=) [15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2]`产生的结果是`[[15, 9, 0, 12, 11, 7, 10, 5, 6, 13, 1, 4, 8, 3, 14, 2]]`。经过分析, 原来`groupBy`是通过`span`实现的:

```
groupBy _ [] = []
groupBy eq (x:xs) = (x:ys) : groupBy eq zs
  where (ys, zs) = span (eq x) xs

span _ [] = ([] , [])
span p xs@(x:xs')
  | p x      = let (ys, zs) = span p xs' in (x:ys, zs)
  | otherwise = ([] , xs)
```

这样`span`就会连续判断15是否大于等于下一个元素, 由于15恰好是整个列表中最大的, 所以全部被归为了一组。实际上`groupBy`接受的分组条件是一个“抽象的相等”, 它必须满足三个性质: 自反性、对称性、传递性:

- 自反性。 $x = x$ , 即任何元素和它自己相等;
- 对称性。若 $x = y$ , 则 $y = x$ , 即比较的顺序不影响结果。
- 传递性。若 $x = y$ 并且 $y = z$ , 则 $x = z$ , 如果两个元素相等, 并且其中的一个和第三个元素相等, 则这三个元素相等;

当对列表“Mississippi”使用等号作为条件分组时, 三个条件都被满足, 产生了正确的结果。但当将大于等于号( $\geq$ )作为“相等”条件传入时, 就违反了自反性和对称性。这就是得到错误结果的原因。

本章我们介绍一些抽象的代数结构, 它们不仅仅是对数的抽象, 而且是对事物的概念、性质和关系的抽象。是很多伟大的思想和心智的结晶。有些部分难度较大, 挑战我们抽象思维的极限。因此我在其中穿插介绍了前辈数学家们是如何披荆斩棘、取得突破的故事。即使读完这一章没有能领会抽象的知识, 我也希望这些令人感慨的故事能不断激励我们前行。

## 3.1 群

提到群论(Group theory), 就不得不提人类解方程的历史。方程是我们祖先发展出的一个强大工具。人们从古埃及的纸草书和古巴比伦的泥板书中发现, 这些古代文明已经掌握了一元一次和二次方程的解法。但是一般三次方程的解法却一直没有进展。到了16世纪, 经过意大利数学家们的努力, 终于得到了一般三次方程和四次方程的根式解。最终卡尔丹在他的著作《大术》中总结

<sup>3</sup>自然归并排序是一种把一组元素按照大小顺序排列好的方法。它首先将元素分成若干组, 每组中的元素顺序已经排好了。然后不断将这些组合并直到最后全部排好序。详细可参考[14], 第368页。

人类历史上这千余年的结果。这些进展不仅仅是未知数次数的提升，我们对数还有了全新的认识。早期二次方程的负根都被舍弃了，人们认为负数没有意义。不仅如此，人们认为方程的系数也必须是正数，在我们今天看来普通的方程 $x^2 - 7x + 8 = 0$ ，必须用 $x^2 + 8 = 7x$ 的形式来处理。在《大术》中，卡尔丹列出了20种不同类型的四次方程，他说还有67种其它类型的四次方程没有给出，原因仅仅是因为四次方程各项的系数可以为负数和零[18]。直到法国数学家韦达才将方程的形式统一。在解方程时，负数的根式一直被认为是无意义的。但是卡尔丹在解三次方程，例如 $x^3 = 15x + 4$ 时，他的公式给出了 $\sqrt[3]{2 + \sqrt{-121}} + \sqrt[3]{2 - \sqrt{-121}}$ 这样的中间结果。而我们知道这一方程有三个实根 $4, -2 \pm \sqrt{3}$ 。这些问题拓展了我们对虚数的认识，并最终在高斯的手中，建立了代数基本定理<sup>4</sup>。

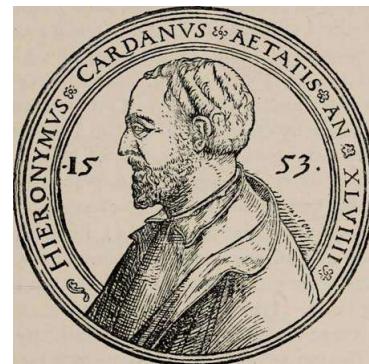


十马克上的高斯像。高斯（1777-1855）

然而接下来在寻找五次及以上方程的根式解时，人们遇到了前所未有的困难。经历了200多年的苦苦寻找，终于在19世纪迎来了出乎意料的结果。法国的天才少年伽罗瓦用他独创的想法，彻底解决了这个问题<sup>5</sup>。

伽罗瓦短短的20年人生上演了一幕悲剧，然而他身后却开辟了抽象代数这一数学分支。1811年10月25日，伽罗瓦生于巴黎。他的母亲精通拉丁文和古典文学。在12岁前他一直由母亲进行启蒙教育。1823年，伽罗瓦进入巴黎的一所久负盛名的中学读书。到了14岁，他开始如饥似渴地学习起数学来。他找来一本勒让德改编的《几何原本》，像读小说一样迅速地掌握了。15岁时，他已经四处寻找拉格朗日的原著来读了。他的水平迅速超出了老师的能力范围。16岁时，他开始尝试解五次方程，经历了失败后他开始反思五次方程是否有根式解。但是这位天才少年除了数学对其他学科都不感兴趣。这让他的老师头疼不已。

1828年6月，伽罗瓦参加了法国最富盛名的巴黎综合工科学校的入学考试。在数学口试中，他认为某些结论很显然，于是不进行解释，结果“在某个领域的知识太多，而在其他领域的知识太少”的伽罗瓦落榜了。他只好回到学校继续读书。



卡尔丹（Gerolamo Cardano, 1501-1576）

<sup>4</sup>高斯一生多次证明了代数基本定理。1799年，22岁的高斯在其博士论文中证明了实系数一元 $n$ 次方程至少有一个复数根。由此推出 $n$ 次方程有且仅有 $n$ 个复根（重根按重数计算）。高斯又在1815和1816年给出了另外两个证明。1849年，在庆祝取得博士学位50周年的纪念会上，高斯又发表了第四个证明，并把系数也推广到了复数。

<sup>5</sup>1770年，法国数学家拉格朗日在研究前人解方程的方法时，引入了根的置换的新思想。遗憾的是，拉格朗日没有考虑这些置换间的组合。1799年，意大利数学家鲁菲尼使用了置换群的方法，而不是单一置换来试图解决一般五次方程的问题。但他的方法存在一处漏洞。1824年，挪威的青年数学家阿贝尔终于独立给出了完整的证明。这一结论现在称为“阿贝尔-鲁菲尼定理”。然而，这一理论只是说一般的五次方程没有根式解，我们知道 $x^5 - 1 = 0$ 是有根式解的。什么情况下一个多项式方程有根式解的完整答案是伽罗瓦完美地解决的[19]。



伽罗瓦 (1811 - 1832)

1829年4月，伽罗瓦发表了第一篇关于连分数的论文。在这一时期，伽罗瓦在多项式方程上取得了重大的发现。他向法国科学院提交了两篇论文。但由于种种原因，论文没有通过<sup>6</sup>。打击接踵而至，1829年7月28日，伽罗瓦的镇长父亲因屈服于政敌的恶意攻击自杀身亡<sup>7</sup>。8月3日，伽罗瓦在服丧期间第二次也是最后一次参加他渴望的巴黎综合工科学校的入学考试。父亲的去世多少对他有影响。在口试阶段，主考官问他一个对数级数的结果是怎样得来的。伽罗瓦跳过了中间步骤说结果很显然，这令考官很恼火，结果他又没有通过。据说伽罗瓦愤怒地把黑板擦投到了主考官的脸上。最后他不得不参加高等师范学校的考试。当时的数学主考官写道：“该生在表达自己的想法时有时很晦涩，但他很聪明，表现出非凡的研究精神。”

1830年，在柯西的建议下，伽罗瓦把论文提交给了科学院秘书傅里叶以角逐法兰西科学院的数学大奖。傅里叶把论文带回家中，但未及审阅就去世了。结果伽罗瓦的论文丢失，科学院于1830年6月把大奖授予了阿贝尔<sup>8</sup>和雅可比<sup>9</sup>[12]。

伽罗瓦生活在政治动荡时期。1830年法国爆发了七月革命<sup>9</sup>。因为激进的革命立场，伽罗瓦被学校开除了。他参加了国民自卫军，一边革命一边进行数学研究。1830年12月，国民自卫军被解散，伽罗瓦曾经一度被捕，但于1831年6月15日被判无罪释放。1831年7月14日法国国庆，伽罗瓦穿上了被解散的国民自卫军军装，拿着上膛的长短枪支上街游行。他因此再次被判入狱6个月。在此之前1831年1月17日，在数学家泊松的建议下，伽罗瓦又一次将他的方程理论提交给法国科学院。结果泊松也没有能理解伽罗瓦的全新思想。7月4日，负责论文审阅的泊松声称：“论证既不够清晰，又不够详尽，使我们无法判断其严格性<sup>10</sup>。”由于伽罗瓦在7月14日被捕，直到10月份他在狱中才收到论文拒绝发表的消息。心高气傲的伽罗瓦反应强烈，他放弃了正规学术途径，转而求助于他的朋友奥古斯特·谢瓦利耶 (Auguste Chevalier) 私人发表论文。伽罗瓦注意到了泊松的建议，他在狱中开始整理数学手稿，并不断完善他的思想，直到1832年4月29日出狱。

出狱后不久，伽罗瓦因为爱情卷入了一场无谓的决斗。5月29日决斗前夕，深信自己将在决斗中死去的伽罗瓦写了三封信。最长的第三封信是写给朋友谢瓦利耶的，短短7页中包含了一份数学思想纲要和三份手稿。这份纲要十分简洁，但内容却极为丰富。只有在后来人把它展开的时候，其重要性才渐渐为人所理解。而且预见了很久以后的发现，证明了伽罗瓦深刻的洞察力。数学家赫尔曼·外尔认为：“从创新程度与思想深度来看，这封信有可能是有史以来份量最重的一篇文字。”信中还有一些后人读来无比惋惜的文字：“这些题目并非我已研究的全部……我没有时间，在我有兴趣的领域里，我已宣布但尚未证明，从而使人怀疑的定理确实是太多了。”其中最令人难忘的，也是最悲伤的话语是：“我没有时间了。”在信的末尾，他要求他的朋友“请求雅可比或者高斯就这些定理的重要性（而不是正确与否）公开发表他们的看法。我希望将来有人能意识到它们是有益的。”

1832年5月30日早晨，伽罗瓦在决斗受了重伤，一个路过的农民发现了他。9点半，他被送到医院。他拒绝了牧师的祈祷，对赶来的弟弟说：“阿尔弗莱德，不要哭。在20岁就死去，需要我全部的勇气。”次日上午10时，伽罗瓦停止了呼吸。关于决斗的原因，有各种不同的说法。人们不知道

<sup>6</sup>有说法认为由于审稿人柯西的疏忽这些论文丢失了。柯西实际上推荐了伽罗瓦的论文，但认为不够清晰，不能发表。人们普遍认为柯西认识到了伽罗瓦工作的重要性，他只是建议将这两篇论文合并成一个，以便能够角逐科学院的数学大奖。柯西是当时著名的数学家，虽然与伽罗瓦的政治观点不同，但仍认为伽罗瓦能够赢得大奖。[20]

<sup>7</sup>他与村里的牧师发生了激烈的政治争执后自杀[20]。

<sup>8</sup>年轻的阿贝尔已于1829年4月6日在贫病交加中去世了，年仅27岁。他是椭圆函数领域的开拓者，阿贝尔函数的发现者。第一个证明了五次方程无根式解。他还研究了更广的一类代数方程，后人发现这是具有交换的伽罗瓦群的方程。为了纪念他，后人称交换群为阿贝尔群。

<sup>9</sup>拿破仑在滑铁卢惨败后，波旁王朝复辟。查理十世清洗了军队中曾为拿破仑效力的军人，引起了人民的不满。并在政治经济、文化宗教和外交中不断失政。查理十世在7月25日颁布圣卢克法令，宣布限制出版自由，解散议会，修改选举法。直接引发了人民的武装起义，通过革命推翻波旁王朝，拥戴路易·菲利普登上王位，建立七月王朝。

<sup>10</sup>但是，泊松在论文审阅的报告结尾处鼓励道：“我们建议作者应公布他的全部工作，以便形成一个明确的意见。”

这是一个悲惨的爱情事件还是政治谋杀。无论是哪一种，一位世界上最杰出的数学家在他20岁时被杀死了，他研究数学只有5年，而其全部数学成果仅有67页。

谢瓦利耶和伽罗瓦的弟弟按照他的遗愿将其工作发表在《百科评论》上，但没有对当时的发展产生影响。也许是因为它太晦涩简略了<sup>11</sup>。直到14年后的1846年，法国数学家刘维尔领悟到伽罗瓦思想的价值，将这些论文编辑发表在极有影响的《纯粹与应用数学》杂志上。在对论文的介绍中，刘维尔对伽罗瓦的悲剧进行了反思：“过分地追求简洁是导致这一缺憾的原因。人们处理像纯粹代数这样抽象和神秘的事物时，应该首先尽力避免这样做。事实上，当你试图引导读者远离习以为常的思路进入较为困惑的领域时，清晰性是绝对必须的……但是现在一切都改变了，伽罗瓦再也回不来了！我们不要再过分地做无用地批评，让我们把缺憾抛开，找一找有价值的东西……我的热心得到了好报，在填补了一些细小的缺陷后，我看出了伽罗瓦用来证明这个定理的方法是美妙和完全正确的，在那个瞬间，我体验到了一种强烈的愉悦。”1870年法国数学家约当根据伽罗瓦的思想，撰写了《置换与代数方程》一书<sup>12</sup>，伽罗瓦的最主要成就是提出了“群”的概念，并用群论彻底解决了根式求解代数方程的问题，而由此发展出了一整套关于群和域的理论。为了纪念他，人们称之为伽罗瓦理论。正是这套理论创立了抽象代数学，标志着数学发展现代阶段的开始。有人评论道：“伽罗瓦一心想要参加的是政治革命，然而实际上所引发的却是数学革命。<sup>[10]</sup>”

### 3.1.1 群的定义

下面我们来了解一下群的定义。

**定义 3.1.1.** 群是一个集合 $G$ 与其上定义的某种二元运算“·”。它遵循四条公理：

1. 封闭性公理：对任何 $a, b \in G$ ，运算结果 $a \cdot b \in G$ ；
2. 结合性公理：对任何 $a, b, c$ ，有 $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ ；
3. 单位元公理： $G$ 中存在一个元素 $e$ ，使得对任何 $a \cdot e = e \cdot a = a$ ；
4. 消去公理：对任何 $a \in G$ ，都存在一个逆元 $a^{-1}$ 使得 $a \cdot a^{-1} = a^{-1} \cdot a = e$ 。

方便起见，我们有时称二元运算为“乘法”，并且省略掉点，将 $a \cdot b$ 写成 $ab$ 。我们称 $e$ 为单位元。一个群的元素个数可以有限也可以无限，分别称为有限群和无限群。元素的个数叫做这个群的阶。

群的“乘法”运算并不一定像数的乘法运算那样满足交换律。例如所有元素为实数的可逆矩阵与矩阵乘法组成一个群。但矩阵乘法是不可逆的。如果群中的二元运算满足交换律，则称为交换群。为了纪念挪威数学家阿贝尔，人们称交换群为阿贝尔群。

为了帮助理解这一抽象定义，我们来看一些具体的群的例子。

1. 整数加法群：群元素是全体整数，二元运算是加法。简称为整数加群；
2. 所有整数除以5的余数构成的集合，也就是 $\{0, 1, 2, 3, 4\}$ 。二元运算是相加后再除以5取余数。例如 $3 + 4 = 7 \bmod 5 = 2$ 。这样构成的群称为整数模5加法群，记为 $Z_5$ 。通过取余数对整数进行分类叫做模 $n$ 剩余类；
3. 转动魔方所形成的群：群元素是各种魔方转动的方式<sup>13</sup>，二元运算是先进行某种转动后再进行另一种转动。这种运算常被称作转动的合成；

<sup>11</sup>同样的教训也发生在阿贝尔身上，1824年他自费印刷发表关于一般五次方程没有根式解的论文。为了省钱，他拼命把论文压缩到6页纸内。结果由于太过简略晦涩，在阿贝尔生前，几乎没有人注意到这一成果。

<sup>12</sup>伽罗瓦理论无疑是相当艰深，并且超越于他的时代的。刘维尔在1846年发表整理伽罗瓦的论文时，在注释中完全忽略了群的核心思想。塞雷参加了刘维尔的讲座后，在他的书中开始介绍伽罗瓦理论。约当是塞雷的学生。伽罗瓦理论在非法语国家被接受得更晚。英国要到一个世纪后，而德国直到哥廷根时代，才由戴德金在课上讲授。

<sup>13</sup>魔方的旋转方式共有18种。可以沿着正面、反面、顶面、底面、左面、右面旋转，分别用字母 $F, B, T, D, L, R$ 代表，每面可以旋转90度、180度、-90度。例如左面旋转90度、180度、-90度分别可以记为： $L, L^2, L'$ <sup>[21]</sup>。再加上恒等变换，一共19个元素。

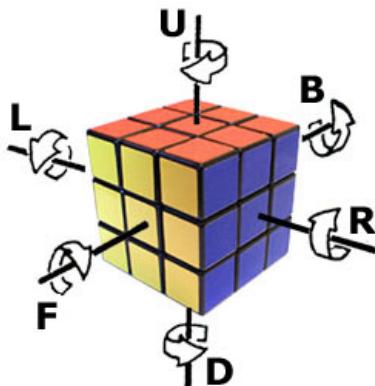


图 3.6: 魔方六个面共有18种转动方式，再加上恒等变换。这些转动方式在合成下构成一个群。

- 平面绕某点的所有旋转构成一个群。群元素是所有的旋转角度，二元运算是先旋转某一角度，然后再旋转另一角度。

而下面这些不是群：

- 所有不等于零的整数和乘法不构成群，我们不能规定单位元为1，因为3没有逆元（ $1/3$ 不是整数）；
- 同理，模5的余数集合 $\{0, 1, 2, 3, 4\}$ 和模5的乘法不构成一个群，但如果把5的整倍数去掉，则集合 $\{1, 2, 3, 4\}$ 和模5的乘法构成一个群。可以从下面的“乘法表”看出来：

	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

因此，单位元是1，它的逆元是其本身；2和3互为逆元，4的逆元还是4；

- 虽然模5的非零余数在模乘法下构成一个群，但是模4的非零余数在模乘下却不构成群。观察模4的乘法表：

	1	2	3
1	1	2	3
2	2	0	2
3	3	2	1

注意到 $(2 \times 2) \bmod 4 = 0$ ，不在集合 $\{1, 2, 3\}$ 中。这个反例告诉我们，只有和 $n$ 互素的余数集合，在模 $n$ 乘法下才能构成群，这种群称为整数模 $n$ 乘法群。自然，如果 $p$ 是素数，则 $\{1, 2, \dots, p-1\}$ 构成模 $p$ 乘法群。

- 全体有理数在乘法下不构成一个群。尽管任何形如 $p/q$ 的有理数都有逆元 $q/p$ ，但是0没有逆元。所有非零有理数在乘法下才构成一个群。

### 练习 3.1

- 全体偶数在加法下是否构成一个群？

2. 能否找到一个整数的子集，使得它在整数乘法下构成一个群？
3. 所有正实数在乘法下是否构成一个群？
4. 整数在减法下是否构成一个群？
5. 举一个只有两个元素的群的例子。
6. 魔方群的单位元是什么？ $F$ 的逆元是什么？

### 3.1.2 么半群与半群

群的限制条件比较严格，从上一节的反例中，我们看到有一些常用的代数结构无法满足群的全部条件。有时我们并不需要一定能够取逆元，如果放宽条件，就可以得到么半群（monoid）这种结构。

**定义 3.1.2.** 么半群是一个集合 $S$ 和其上定义的二元运算 $\cdot$ ，它们遵循两条公理：

1. 结合性公理： $S$ 中和任何三个元素满足 $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ ；
2. 单位元公理： $S$ 中存在一个元素 $e$ ，使得对任何 $a \cdot e = e \cdot a = e$ 。

可以看到么半群与群的定义类似，只是去掉了取逆运算的消去公理。可以立即看到，上一节群的反例中有不少是么半群，例如整数乘法构成么半群，单位元是1。么半群在编程中很常见。我们在下一章介绍范畴论时还会再仔细介绍么半群这一特别重要的结构。下面是一些么半群的例子：

1. 给定字符集后，长度有限的字符串在拼接操作下构成么半群。么半群的元素是字符串，二元运算是拼接操作。单位元是空串。
2. 由字符串推广到列表（List A），在连接操作下构成么半群。么半群的元素是列表，二元运算是连接操作（ $+$ ）。单位元是空列表`nil`。

为此，我们可以写出以下的程序，用么半群来定义字符串和列表的代数结构。

```
instance Monoid (List A) where
  e = nil
  (*) = (++)
```

这样针对字符串和列表的“叠加”操作，完全可以将抽象程度提升到么半群上来<sup>14</sup>。例如下面的“合并”操作定义对任何么半群都有效：

$$\text{concat} = \text{foldr } e (*)$$

这样就可以用`concat`合并列表，如

`concat [[1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3], [1, 3, 2]]`的结果是`[1, 2, 3, 1, 2, 1, 3, 2, 3, 1, 2, 3, 1, 3, 2]`。

3. 堆（heap）是编程中一种常见的数据结构，堆顶元素总是最小的堆称为最小堆（或小顶堆、小根堆）。有一种“斜堆”（skew heap）可以用上一章定义的二叉树来实现([14]第7.3节)：

```
data SHeap A = nil | node (SHeap A, A, SHeap A)
```

<sup>14</sup> 我们将在下一章讲解范畴论时介绍如何实现这样的抽象叠加操作

可以看出，除了名称，斜堆的定义和二叉树完全一样。非空堆的最小元素在树的根节点中。定义斜堆的“归并”（merge）运算如下：

$$\begin{aligned} \text{merge}(\text{nil}, h) &= h \\ \text{merge}(h, \text{nil}) &= h \\ \text{merge}(h_1, h_2) &= \begin{cases} k_1 < k_2 : \text{node}(\text{merge}(r_1, h_2), k_1, l_1) \\ \text{否则} : \text{node}(\text{merge}(h_1, r_2), k_2, l_2) \end{cases} \end{aligned}$$

如果其中任一堆为空，则归并结果为另一个堆。非空情况下， $h_1$ 和 $h_2$ 分别可以表示为 $\text{node}(l_1, k_1, r_1)$ 和 $\text{node}(l_2, k_2, r_2)$ 。我们比较根节点，选择较小的作为新的根。然后把含有较大元素的树合并到某一子树上。最后再把左右子树交换。若 $k_1 < k_2$ ，选择 $k_1$ 作为新的根。我们既可以将 $h_2$ 和 $l_1$ 合并，也可以将 $h_2$ 和 $r_1$ 合并。不失一般性，我们合并到 $r_1$ 上。然后交换左右子树，最后的结果为 $(k_1, \text{merge}(r_1, h_2), l_1)$ 。注意这个归并的二元运算是递归的。这样所有斜堆组成的集合，在这一归并运算下构成一个幺半群。单位元是空堆nil。

4. 堆也可以用上一章介绍的多叉树实现。有一种堆叫做配对堆（pairing heap）它的定义如下（[14]第9.4节）：

```
data PHeap A = nil | node (A, List (PHeap A))
```

除了名称外，这个定义和上一章的多叉树的结构完全相同。这是一个递归定义，一个配对堆要么为空，要么是一棵多叉树，包含一个根节点和一组子树。非空堆的最小元素在多叉树的根节点中。我们可以定义配对堆的归并操作：

$$\begin{aligned} \text{merge}(\text{nil}, h) &= h \\ \text{merge}(h, \text{nil}) &= h \\ \text{merge}(h_1, h_2) &= \begin{cases} k_1 < k_2 : \text{node}(k_1, h_2 : ts_1) \\ \text{否则} : \text{node}(k_2, h_1 : ts_2) \end{cases} \end{aligned}$$

如果任一堆为空，则归并结果为另一个堆。非空情况下，两个堆 $h_1$ 和 $h_2$ 分别可以表示为 $\text{node}(k_1, ts_1)$ 和 $\text{node}(k_2, ts_2)$ 。我们比较两个堆的根节点元素，令根节点较大的一个作为另一个的新子树。这样所有的配对堆组成的集合，在归并操作下构成一个幺半群，单位元是空堆nil。

如果再进一步把单位元的限制也去掉，我们就得到了半群（semigroup）这种代数结构。

**定义 3.1.3.** 半群是一个集合和定义在其上的可结合的二元运算。

半群的二元运算是可结合的，所以满足结合律：半群中的任何三个元素 $a$ 、 $b$ 、 $c$ 满足 $(ab)c = a(bc)$ 。半群的条件更为宽松，下面是一些半群的例子：

1. 全体正整数构成的加法半群、乘法半群；
2. 全体偶数构成的加法半群、乘法半群。

如前所述，人们常将定义在群、幺半群、半群上的二元运算叫做“乘法”，因此我们可以用“乘方”来表示连续的二元运算。如 $x \cdot x \cdot x = x^3$ 。一般地，群和幺半群的“幂”可以递归地定义如下：

$$x^n = \begin{cases} n = 0 : e \\ \text{其它} : x \cdot x^{n-1} \end{cases}$$

但是半群没有单位元，所以 $n$ 必须是非零正整数：

$$x^n = \begin{cases} n = 1 : & x \\ \text{其它} : & x \cdot x^{n-1} \end{cases}$$

### 练习 3.2

1. 布尔值构成的集合{True, False}，在“逻辑或”运算 $\vee$ 下构成一个幺半群。称为任意（Any）逻辑幺半群。它的单位元是什么？
2. 布尔值构成的集合{True, False}，在“逻辑与”运算 $\wedge$ 下构成一个幺半群。称为全部（All）逻辑幺半群。它的单位元是什么？
3. 对可比类型的元素进行比较时，会有三种结果，我们把它们抽象为{<, =, >}<sup>15</sup>，针对这个集合，定义一个二元运算使得它们构成一个幺半群。这个幺半群的单位元是什么？
4. 证明群、幺半群、半群的幂满足交换律： $x^m x^n = x^n x^m$

#### 3.1.3 群的性质

抽象代数最强大的思想在于我们可以不关心抽象概念所代表的实际物体的含义，而研究抽象结构间内在的规律。这些规律对所有被抽象的物体都适用。如果我们了解了一般群的内在规律，而群元素代表几何的点、线、面，则我们就获得了几何的规律；如果群元素是魔方的旋转，我们就获得了魔方变换的规律；如果群元素是编程中的某种数据结构，我们就得到了这种数据结构上的算法。我们这一节介绍一些群上的重要性质。

**定理 3.1.1.** 群的单位元是唯一的。

证明. 假设存在另一单位元 $e'$ ，使得对任意元素 $a$ 满足 $e'a = ae' = a$ 。我们有 $e = ee' = e'$ 。这样就证明了单位元的唯一性。□

不仅群的单位元是唯一存在的，而且每个元素的逆也是唯一存在的。我们有下面的定理。

**定理 3.1.2.** 逆元唯一存在。对群中的任意元素 $a$ ，都存在且仅存在一个 $a^{-1}$ 使得 $aa^{-1} = a^{-1}a = e$ 。我们称 $a^{-1}$ 为 $a$ 的逆元。

证明. 根据群的消去公理，我们知道逆元存在。所以我们只需要证明逆元唯一。假设存在另一元素 $b$ ，也满足 $ab = ba = e$ ，我们在等式右侧“乘以” $a^{-1}$ 得：

$$\begin{aligned} aba^{-1} &= baa^{-1} = ea^{-1} \\ &\Rightarrow be = a^{-1} && \text{对第二项用结合律} \\ &\Rightarrow b = a^{-1} && \text{逆元唯一} \end{aligned}$$

□

我们在前面定义了群的阶，群的元素也可以定义阶。对群中的一个元 $a$ ，能够满足 $a^m = e$ 的最小正整数 $m$ 叫做 $a$ 的阶。如果这样的 $m$ 不存在，我们说 $a$ 的阶是无限的。例如前面例子中魔方旋转群中， $F$ 旋转4次就回到了原位，所以 $F$ 的阶是4，而 $F'$ 旋转两次回到原位，所以它的阶是2。再比如整数模5的乘法群，除了1以外，其他元素的4次幂都模5余1，所以他们的阶都是4。我们有如下有趣的定理。

**定理 3.1.3.** 有限群的每个元素都有有限的阶。

<sup>15</sup>一些编程语言，如C、C++、Java用负数、零、正数表示这三种关系。Haskell中用GT, EQ, LE表示。



图 3.7: 魔方的F变换, 连续转动4次回到原位。

证明. 如果有限群 $G$ 的阶是 $n$ , 对任意元素 $a$ , 构造集合 $\{a, a^2, \dots, a^{n+1}\}$ 。这个集合有 $n+1$ 个元, 但是群的阶是 $n$ , 所以根据鸽笼原理, 一定有两个元素是相等的, 不妨记这两个元素为 $a^i$ 和 $a^j$ , 其中 $0 < i < j \leq n+1$ 。我们有:

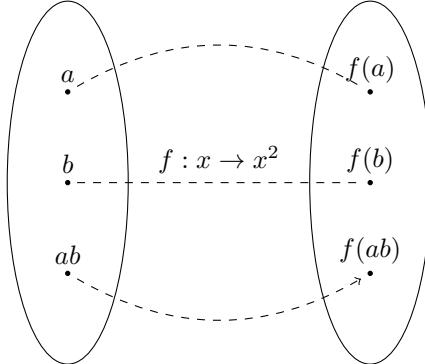
$$\begin{aligned} a^j a^{-i} &= a^{-i} a^i \quad \text{由 } a^i = a^j \\ a^j a^{-i} &= e \quad a^i \text{ 和 } a^{-i} \text{ 互为逆元} \\ a^{j-i} &= e \quad a \text{ 的阶为 } j-i \end{aligned}$$

所以 $a$ 的阶为 $j-i$ 是有限的。  $\square$

在第一章, 我们比较随意地使用了“同构”一词来形容具有相同内在结构的事物。现在我们可以给出同态和同构的概念了。假设存在从某个集合 $A$ 到另一个集合 $B$ 的映射 $f$ 。 $a$ 和 $b$ 是 $A$ 中的两个元,  $f(a)$ 和 $f(b)$ 是它们在 $B$ 中的像。我们考虑 $A$ 上的二元封闭运算产生的元 $a \cdot b$ , 在映射下在 $B$ 中的像 $f(a \cdot b)$ 。如果对 $B$ 上的二元封闭运算总有

$$f(a) \cdot f(b) = f(a \cdot b)$$

$A$ : 自然数乘法半群       $B$ : 平方数乘法半群



$$\begin{aligned} a &= 2 & f(a) &= 4 \\ b &= 3 & f(b) &= 9 \\ ab &= 6 & f(ab) &= f(a)f(b) = 36 \end{aligned}$$

图 3.8: 同构

我们就说 $f$ 是从 $A$ 到 $B$ 的同态映射 (homomorphism)。如果 $f$ 是满射 (surjection, 即 $B$ 中所有元素都在 $A$ 中有原像)，则称为同态满射。举一个例子，考虑一个奇偶判定函数 $odd : \mathbb{Z} \rightarrow \text{Bool}$ ，它接受一个整数，如果是奇数就返回真值True，否则返回假False。整数在加法下构成一个群。而布尔集合 $\{\text{True}, \text{False}\}$ 在逻辑异或运算下也构成一个群。我们可以很容易验证：

1.  $a$ 和 $b$ 都是奇数， $odd(a)$ 和 $odd(b)$ 都为真。它们的和是偶数， $odd(a + b)$ 为假。满足 $odd(a) \oplus odd(b) = odd(a + b)$ ；
2.  $a$ 和 $b$ 都是偶数， $odd(a)$ 和 $odd(b)$ 都为假。它们的和也是偶数， $odd(a + b)$ 也为假。满足 $odd(a) \oplus odd(b) = odd(a + b)$ ；
3.  $a$ 和 $b$ 一奇一偶， $odd(a)$ 和 $odd(b)$ 一真一假。它们的和为奇数， $odd(a + b)$ 为真。满足 $odd(a) \oplus odd(b) = odd(a + b)$ ；

如果 $f$ 不仅是满射，还是单射 (injection)。那么它就是一一映射。这种情况下，我们称 $f$ 是从 $A$ 到 $B$ 的同构映射，简称同构 (isomorphism)。记为 $A \cong B$ 。同构是一种非常强大的关系，不仅两个群可以同构，半群、幺半群以及其他代数结构也可以同构，如图3.8所示。如果 $A$ 和 $B$ 同构，那么抽象地来看，它们没有什么区别，只有命名上的不同。如果 $A$ 上有一个代数性质，那么在 $B$ 上也有一个完全类似的性质[22] (页码)。另外， $A$ 与 $A$ 之间的同构映射称为 $A$ 的自同构 (automorphism)。例如整数加群在取相反数下成为自同构。

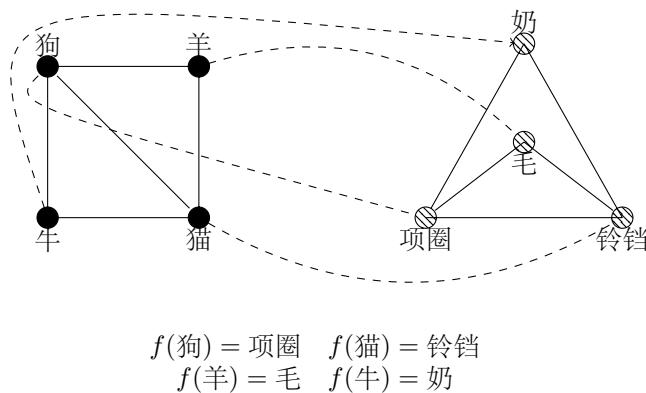


图 3.9: 图的同构 (graph isomorphism) 具有不同的定义。它反映了看似不同的两个图，具有相同的结构。

面对群这种抽象的代数结构时，具体的例子可以帮助我们理解。我们会不自觉地选出一个自己熟悉的代表，例如整数加群，然后看看各种概念，性质在其上是怎样的。但有时这容易形成一种错觉。感觉群的元素大多是一些实体，例如各种数；并且二元运算大多像普通加法、乘法那样可以交换。我们接下来介绍的“变换群”是一个例外，一方面，它不是阿贝尔群，运算不可交换；另一方面群元素不是数，而是变换。

所谓变换，就是一个集合 $A$ 到 $A$ 自身的映射。记为 $\tau : A \rightarrow A$ 。它把集合 $A$ 的元素 $a$ 映射为 $\tau(a)$ ，即 $a \rightarrow \tau(a)$ 。一个集合可以有多个不同的变换，例如下面是布尔集合的全部变换，我们记真为 $T$ 、假为 $F$ 。

$$\begin{aligned}\tau_1 &: T \rightarrow T, \quad F \rightarrow T \\ \tau_2 &: T \rightarrow F, \quad F \rightarrow F \\ \tau_3 &: T \rightarrow T, \quad F \rightarrow F \\ \tau_4 &: T \rightarrow F, \quad F \rightarrow T\end{aligned}$$

其中变换 $\tau_3$ 和 $\tau_4$ 是一一变换。针对一个集合 $A$ ，我们将它的全体变换放到一起构成一个新的集合：

$$S = \{\tau, \lambda, \mu, \dots\}$$

现在我们要规定一个 $S$ 的二元代数运算，把它叫做乘法。为了方便起见，我们将 $\tau(a)$ 用另一个符号表达：

$$\tau : a \rightarrow a^\tau = \tau(a)$$

这里 $a^\tau$ 不是 $a$ 的 $\tau$ 次方的意思，它只是一个符号记法，表示变换的意思。我们观察 $S$ 的两个元 $\tau$ 和 $\lambda$ ，

$$\tau : a \rightarrow a^\tau, \lambda : a \rightarrow a^\lambda$$

那么 $a \rightarrow (a^\tau)^\lambda = \lambda(\tau(a))$ 显然也是 $A$ 的一个变换，现在我们规定把这个变换叫做 $\tau$ 和 $\lambda$ 的乘积。

$$\tau\lambda : a \rightarrow (a^\tau)^\lambda = a^{\tau\lambda}$$

读着不妨从前面布尔变换的集合里取几个变换来计算它们的乘积，不难验证这一乘法适合结合律，因为：

$$\begin{aligned} \tau(\lambda\mu) : a &\rightarrow (a^\tau)^{\lambda\mu} = ((a^\tau)^\lambda)^\mu \\ (\tau\lambda)\mu : a &\rightarrow (a^{\tau\lambda})^\mu = ((a^\tau)^\lambda)^\mu \end{aligned}$$

现在不难看出当初我们选择乘方符号来表达变换的好处了。选择一套强大的符号系统是多年来数学发展的传统和法宝。欧拉、莱布尼茨都是选择和使用符号的大师。对这个乘法来说， $S$ 的单位元就是 $A$ 的恒等变换 $\epsilon : a \rightarrow a$ 。不难验证：

$$\begin{aligned} \epsilon\tau : a &\rightarrow (a^\epsilon)^\tau = a^\tau \\ \tau\epsilon : a &\rightarrow (a^\tau)^\epsilon = a^\tau \end{aligned}$$

所以 $\epsilon\tau = \tau\epsilon = \tau$ 。这样 $S$ 对这个乘法来说差不多已经构成一个群了。可惜，虽然说是差不多，到底还是差一点。因为任意变换 $\tau$ 不一定有逆元。例如布尔变换集合中的 $\tau_1$ ，它把任何布尔值都变换为真，不管用这4个变换中的哪一个，都无法把 $\tau_1$ 变换回去。所以 $\tau_1$ 没有逆元。

虽然 $S$ 无法构成群，但是峰回路转，是它的一个子集 $G$ 却有可能构成群。事实上，如果 $G$ 只包含 $A$ 的一一变换，则它在这个乘法下构成群。

我们称，一个集合 $A$ 的若干个一一变换对于上述规定的乘法所作成的群叫做 $A$ 的一个变换群（transform group）。并且我们有以下重要的定理：

**定理 3.1.4.** 一个集合 $A$ 的所有一一变换构成一个变换群 $G$ 。

变换群一般不是交换群（阿贝尔群），我们可以很容易地找到反例。考虑 $\tau_1$ 是平面上的平移，它把原点 $(0, 0)$ 移动到 $(1, 0)$ ，变换 $\tau_2$ 是绕原点旋转 $\pi/2$ ，但是：

$$\begin{aligned} \tau_1\tau_2 : (0, 0) &\rightarrow (0, 1) \\ \tau_2\tau_1 : (0, 0) &\rightarrow (1, 0) \end{aligned}$$

因此这一变换群不是阿贝尔群。变换群应用极广，十分重要，我们有一个很强的结论：

**定理 3.1.5.** 任何一个群都同一个变换群同构。

我们这里略去了证明过程。这个定理告诉我们，任意一个抽象的群都能在变换群里找到一个具体的实例。换一句话说，我们不必害怕，将来会找到一个抽象群，这个群完全是我们脑子里造出来的空中楼阁[22](页码)。

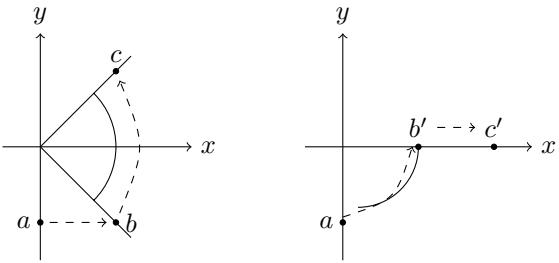


图 3.10: 变换顺序不同, 结果也不同的例子

### 练习 3.3

1. 奇偶判断函数在整数加群( $Z, +$ )和布尔逻辑与群( $Bool, \wedge$ )下是否构成同态? 去除0元素的整数乘法群呢?
2. 假定两个群 $G$ 和 $G'$ 在映射下同态, 群 $G$ 中的元 $a \rightarrow a'$ , 那么 $a$ 和 $a'$ 的阶是否相同?
3. 证明一个变换群的单位元一定是恒等变换。

#### 3.1.4 置换群

我们现在介绍伽罗瓦用来判定方程是不是有根式解的群——置换群。这种群是变换群的一种特例。我们首先定义置换的概念。一个有限集合的一一变换叫做一个**置换**。一个有限集合的若干个置換作成的一个群叫做一个**置換群** (permutation group, 顾名思义, 它相当于对集合的元素进行重新排列)。进一步, 一个包含 $n$ 个元素的集合的全体置換作成的群叫做 $n$ 次**对称群** (symmetric group)。这个群用 $S_n$ 来表示。

由高中的排列组合我们知道,  $n$ 元的置換一共有 $n!$ 个。所以 $n$ 次对称群 $S_n$ 的阶是 $n!$ 。一个置換把集合的元 $a_i$ 映射为 $a_{k_i}$ , 其中 $i = 1, 2, \dots, n$ 。所以这一置換完全可以由 $(1, k_1), (2, k_2), \dots, (n, k_n)$ 这 $n$ 对数来确定。我们可以把这一置換写成:

$$\begin{pmatrix} 1 & 2 & \dots & n \\ k_1 & k_2 & \dots & k_n \end{pmatrix}$$

例如

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 4 & 3 & 1 \end{pmatrix}$$

就表示了一个置換。明显这样第一行总是 $1, 2, \dots, n$ 的形式, 所以可以进一步将置換的记法简化为 $(2, 5, 4, 3, 1)$ , 表示置換后, 原来第2个元素变为新的第1个元素, 原来第5个元素变为现在的第2个元素等等。我们可以按照这种方法列出3个元素集合的全部置換, 也就是 $S_3$ 群的元素:

$$(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)$$

计算乘法, 也就是两个置換的复合时, 我们可以这样确定每个位置上元素。针对第 $i$ 个位置, 我们先看在第一个置換上它被映射到几, 例如 $j$ , 然后再到第二个置換中看第 $j$ 个位置被映射到哪里, 例如 $k$ , 这样乘法的结果中, 第 $i$ 个位置上的元素就是 $k$ 。我们可以进一步取两个元素相乘, 举例看一下 $S_3$ 是否是阿贝尔群(交换群):

$$\begin{aligned} (1, 3, 2)(2, 1, 3) &= (2, 3, 1) \\ (2, 1, 3)(1, 3, 2) &= (3, 1, 2) \end{aligned}$$

所以 $S_3$ 不是阿贝尔群。它是一个最小的有限非阿贝尔群。一个有限非阿贝尔群至少要有6个元素。我们观察5个元素的一个置换 $(2, 3, 1, 4, 5)$ , 发现只有前三个元素变化了, 而后两个元素保持不变。而前三个元素的变化很有特点:  $2 \rightarrow 1$ 、 $3 \rightarrow 2$ 、 $1 \rightarrow 3$ , 恰好是循环地画了个圈。

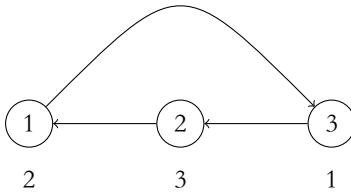


图 3.11: 变换 $(2\ 3\ 1)$ 是一个循环置换

为此我们可以把置换 $(2, 3, 1, 4, 5)$ 进一步简写为 $(2\ 3\ 1)$ 。注意这种写法元素间没有逗号, 并且认为 $(1\ 2\ 3)$ 、 $(2\ 3\ 1)$ 、 $(3\ 1\ 2)$ 都表示同一个3循环置换<sup>16</sup>。一般地, 我们定义 $k$ -循环置换 $(i_{j_1} i_{j_2} \dots i_{j_k})$ , 它将后一个元素映射到前一个的位置, 第一个元素映射到最后。组成循环:

$$i_{j_2} \rightarrow i_{j_1}, i_{j_3} \rightarrow i_{j_2}, \dots, i_{j_1} \rightarrow i_{j_k}$$

注意,  $k$ -循环的元素并不一定相邻, 例如 $(3\ 9\ 4)$ , 也不一定有固定的顺序, 例如 $(2\ 4\ 1\ 3)$ 。如果循环只有两个元素 $(i\ j)$ 称其为对调 (transposition)。有个特殊情况, 就是恒等置换 $(1, 2, 3, 4, 5)$ , 我们把它记为 $\epsilon = (1)$ 。并且有:

$$\epsilon = (1) = (2) = \dots = (n)$$

现在我们观察置换 $(2, 1, 4, 5, 3)$ , 它有两个循环, 一个是2循环置换 $(1\ 2)$ , 另一个是3循环置换 $(3\ 4\ 5)$ 。为此我们可以将它表示为两个置换的乘法:

$$(2, 1, 4, 5, 3) = (1\ 2)(3\ 4\ 5)$$

事实上, 每一个 $n$ 元的置换 $\pi$ 都可以写成若干个互相没有共同数字的 (不相连的) 循环置换的乘积。采用这种方法的好处是, 虽然一般的置换是不可交换的, 但是由于这样的 $k$ -循环置换彼此没有共同数字, 所以它们是可交换的。例如 $(1\ 2)(3\ 4\ 5) = (3\ 4\ 5)(1\ 2)$ 。

下面是 $S_3$ 的全体循环置换用这种记法的列表:

$$(1), (1\ 2), (1\ 3), (2\ 3), (1\ 2\ 3), (1\ 3\ 2)$$

任意给一个置换 $(k_1, k_2, \dots, k_n)$ , 如何将其转化为若干个不相连的循环置换的乘积呢? 我们可以按照这样的规则来操作。首先从左到右依次比较置换的每个元素, 如果 $k_i$ 和 $i$ 相等, 说明这个元素无需置换; 否则先写下左括号, 然后将第 $k_i$ 个元素的值 $k_j$ 写到括弧中, 接着顺着 $k_j$ 继续寻找, 将其和 $j$ 比较, 如果不等, 就写入括弧, 再检查第 $k_j$ 个元素。重复这个步骤直到发现某个元素形成了循环。这时可以写下右括号, 完成一个循环。此后再继续从左向右比较置换中的元素, 直到处理完所有的[23] (页码)。如果所有的 $k_i$ 都和 $i$ 相等, 我们可以写下 $(1)$ 表示恒等置换。这一过程特别适合用编程的方法实现, 下面就是相应的算法, 和一种对应的源代码。

```
function k-cycles( $\pi$ )
     $r \leftarrow []$ 
    for  $i \leftarrow 1$  to  $|\pi|$  do
         $p \leftarrow []$ 
        while  $i \neq \pi[i]$  do
```

<sup>16</sup>也有用不带空格的 $(231)$ 来表示的, 但是如果元素个数超过10个, 不带空格的表示会产生歧义。可以被理解为 $(23\ 1)$

```

 $p \leftarrow \pi[i] : p$ 
Exchange  $\pi[i] \leftrightarrow \pi[\pi[i]]$ 
if  $p \neq []$  then
     $r \leftarrow r : (\pi[i] : reverse(p))$ 
if  $r \neq []$  then
    return  $r$ 
else
    return [[1]]                                ▷ 返回恒等变换

```

将任意置换转换为不相连的 $k$ -循环的乘积：

```

def kcycles(a):
    r = []
    n = len(a)
    for i in range(n):
        p = []
        while i + 1 != a[i]:
            p.append(a[i])
            j = a[i] - 1
            a[i], a[j] = a[j], a[i]
        if p != []:
            r.append([a[i]] + p)
    return r if r != [] else [[1]]

```

由上一小节的定理，我们有

**定理 3.1.6.** 每一个有限群都与一个置换群同构。

这样对于任何有限群，例如方程的根，我们都可以用置换群加以研究。而置换群又是一种比较容易计算的群，这正是伽罗瓦解决方程根式解问题的方法。

### 练习 3.4

1. 列出 $S_4$ 的全体元素。
2. 将 $S_3$ 的所有元写成不相连的循环置换的乘积。
3. 编程将 $k$ -循环的乘积转换回置换。

#### 3.1.5 循环群

在各种群中，循环群（cyclic group）是最简单的。它是人们已经完全解决了的一类群。一个群 $G$ 中的所有元素会不会都是某一个固定元素的乘方？这是有可能的。举一个例子，我们看去掉0元素的模5乘法群。它包含4个元素 $\{1, 2, 3, 4\}$ ，其中2的各个乘方模5的结果如下：

$$\begin{aligned} 2^1 \bmod 5 &= 2 \\ 2^2 \bmod 5 &= 4 \\ 2^3 \bmod 5 &= 3 \\ 2^4 \bmod 5 &= 1 \end{aligned}$$

所以2的乘方生成了这个群中的所有元素。为此我们定义：

**定义 3.1.4.** 若一个群的每个元素都是某个固定元素 $a$ 的乘方，称 $G$ 为**循环群**。我们也说， $G$ 是由元 $a$ 所生成的，并且用符号

$$G = (a)$$

来表示， $a$ 叫做 $G$ 的一个生成元。

我们举两个循环群的重要例子：

**例 3.1.1.** 整数加群中的全体整数都是1的“乘方”，这里的二元运算是加法，所以乘方就是不断进行相加。我们看任意正整数 $m$ ：

$$\begin{aligned} 1^m &= 1 \cdot 1 \cdot 1 \cdots 1 \quad m\text{次} \\ &= 1 + 1 + \dots + 1 \quad \text{整数加群中乘法为+} \\ &= m \end{aligned}$$

在整数加群中，1的逆元是-1，这是因为 $1 + (-1) = 0$ ，而0是单位元。所以对任意负整数 $-m$ 有：

$$\begin{aligned} 1^{-m} &= (-1)^m \quad \text{逆元} \\ (-1)^m &= (-1) \cdot (-1) \cdot (-1) \cdots (-1) \quad m\text{次} \\ &= -1 + (-1) + \dots + (-1) \quad \text{整数加群中乘法为+} \\ &= -m \end{aligned}$$

最后0是单位元，按照定义有 $0 = 1^0$ 。综合上述三种情况，我们有 $Z = (1)$ 。

这是一个无限循环群的例子，我们再举一个有限循环群的例子。

**例 3.1.2.** 我们看整数模 $n$ 剩余类。我们用符号 $[a]$ 表示整数 $a$ 所在的剩余类。定义二元运算为模 $n$ 的加法。

$$[a] + [b] = [a + b]$$

例如 $[3] + [4]$ 在模5时，结果为 $[2]$ 。我们不难验证它符合结合律，单位元是 $[0]$ ，并且每个元素都存在逆。我们把这个群叫做整数模 $n$ 的剩余类加群。包含元素 $[0], [1], \dots, [n-1]$ 。这个群是一个循环群，因为每个元 $[i]$ 都可以写成 $i$ 个 $[1]$ 的乘方。

$$[i] = [1] + [1] + \dots + [1] \quad i\text{个}$$

这两个例子不是随便选的。实际上，由这两个例子，我们已经认识了所有的循环群！因为有如下定理：

**定理 3.1.7.** 若 $G$ 是由元素 $a$ 所生成的循环群，那么 $G$ 的构造完全可以由 $a$ 的阶来决定：

- 如果 $a$ 的阶无限，那么 $G$ 与整数加群同构；
- 如果 $a$ 的阶是整数 $n$ ，那么 $G$ 与整数模 $n$ 剩余类加群同构。

证明. 如果 $a$ 的阶无限，我们有 $a^h = a^k$ 当且仅当 $h = k$ 。否则，如果 $h \neq k$ ，不妨令 $h > k$ ，我们有 $a^{h-k} = e$ ，这与 $a$ 的阶无限矛盾。所以我们可以构造一一映射：

$$f : a^k \rightarrow k$$

从而建立循环群 $G = (a)$ 到整数加群 $Z$ 之间的同构。即： $a^h a^k \rightarrow h + k$ 。

如果 $a$ 的阶是整数 $n$ ，即 $a^n = e$ 。这时 $a^h = a^k$ 当且仅当 $h \equiv k \pmod{n}$ 的时候。我们可以构造一一映射：

$$f : a^k \rightarrow [k]$$

从而建立循环群 $G = \langle a \rangle$ 到整数模 $n$ 剩余类加群间的同构。即：

$$a^h a^k = a^{h+k} \rightarrow [h+k] = [h] + [k]$$

□

所以，抽象地来看，生成元的阶是无限大的循环群只有一个，生成元的阶是某个正整数的群也只有一个。至于这些循环群的构造，我们也知道得很清楚：

- $a$ 的阶无限大
  - 群元素是： $\dots, a^{-2}, a^{-1}, a^0, a^1, a^2, \dots$
  - 群乘法是： $a^h a^k = a^{h+k}$
- $a$ 的阶是 $n$ 
  - 群元素是： $a^0, a^1, a^2, \dots, a^{n-1}$
  - 群乘法是： $a^h a^k = a^{(h+k) \bmod n}$

### 练习 3.5

1. 证明循环群一定是阿贝尔群。

#### 3.1.6 子群

我们接下来介绍子群的概念。它可以帮助我们通过一个群的子集来推测整个群的性质。

**定义 3.1.5.** 一个群 $G$ 的一个子集 $H$ 叫做 $G$ 的一个**子群**，如果 $H$ 对于 $G$ 的乘法也构成一个群。

对于任何群 $G$ ，至少有两个子群，一个是 $G$ 本身，另外是只包含一个单位元的集合 $\{e\}$ 。这两个子群称为平凡子群。对于整数加群来说，偶数加群构成一个子群，而奇数和加法不构成子群（为什么？）。我们再举一个例子，考虑上一节介绍的置换群 $S_3$ 的子集 $H = \{(1), (1 2)\}$ ， $H$ 在置换复合乘法下构成一个子群。我们下面来验证一下：

1. 乘法的封闭性。 $(1)(1) = (1)$ ,  $(1)(1 2) = (1 2)$ ,  $(1 2)(1) = (1 2)$ ,  $(1 2)(1 2) = (1)$ ；其中最后一个乘法表示头两个元素对调后再对调会相当于恒等置换。
2. 结合律对 $S_3$ 的所有元素都成立，故对 $H$ 的元素也成立；
3. 单位元 $(1) \in H$ ；
4. 每个元素的逆元都存在： $(1)(1) = (1)$ 、 $(1 2)(1 2) = (1)$ 。

这样依次验证群的所有性质比较麻烦，我们有一个方便的工具：

**定理 3.1.8.** 一个群 $G$ 的非空子集 $H$ 构成子群的充分必要条件是：

1. 若任意 $a, b \in H$ ，则 $ab \in H$ ；
2. 任意元素 $a \in H$ ，有 $a^{-1} \in H$ ；

我们把这一定理的证明留作练习。这个定理可以直接得到一个推论，如果 $H$ 是 $G$ 的子群，那么 $H$ 的单位元就是 $G$ 的单位元， $H$ 中任意元素 $a$ 的逆元就是它在 $G$ 中的逆元。上述定理中的两个条件也可以用一个合并的条件来代替：

**定理 3.1.9.** 一个群 $G$ 的非空子集 $H$ 构成子群的充分必要条件是。若任意 $a, b \in H$ , 则 $ab^{-1} \in H$ 。

证明. 首先证明充分性, 设 $a, b \in H$ , 由上一定理的条件2, 有 $b^{-1} \in H$ 。再由条件1, 有 $ab^{-1} \in H$ ;

现在反过来证明必要性。设 $a \in H$ , 所以 $aa^{-1} = e \in H$ , 于是 $ea^{-1} = a^{-1} \in H$ ; 这就是前面定理中的条件1; 接下来若 $a, b \in H$ , 由刚刚证明的结果,  $b^{-1} \in H$ 。所以 $a(b^{-1})^{-1} = ab \in H$ 。这就证明了前面定理中的条件2。□

假如子集 $H$ 是有限集, 那么 $H$ 构成子群的条件要更简单:

**定理 3.1.10.** 一个群 $G$ 的非空有限子集 $H$ 构成子群的充分必要条件是。若任意 $a, b \in H$ , 则 $ab \in H$ 。

我们曾经利用一个整数 $n$ 把全体整数分成模 $n$ 的剩余类。我们可以把类似的思路扩展抽象到群上。我们把整数加群叫做 $Z$ , 把所有包含 $n$ 的倍数的集合叫做 $H$ , 即 $H = kn$ , 其中 $k = 0, \pm 1, \pm 2, \dots$ 。那么对于 $H$ 的任意两个元 $hn$ 和 $kn$ 来说,  $hn + (-k)n = (h - k)n \in H$ 。而 $-kn$ 恰好是 $kn$ 在 $Z$ 中的逆元, 并且 $+$ 是整数加群上的二元运算。所以由上述定理3.1.9,  $H$ 是 $Z$ 的子群。

当我们把整数划分成模 $n$ 剩余类时, 所利用的等价关系是这样规定的:

$$a \equiv b \pmod{n}, \text{ 当且仅当 } n|(a - b)$$

利用子群 $H$ , 这一等价关系也可以这样定义:

$$a \equiv b \pmod{n}, \text{ 当且仅当 } (a - b) \in H$$

这样, 我们就利用子群 $H$ 实现了 $Z$ 的剩余类划分。我们现在将这一情形推广到利用子群 $H$ 对一个群 $G$ 进行分类。为此, 我们需要先用子群定义元素的一种等价关系~:

$$a \sim b, \text{ 当且仅当 } ab^{-1} \in H$$

给了 $a$ 和 $b$ , 我们可以唯一确定 $ab^{-1}$ 是不是属于 $H$ 。为什么说~是一种等价关系呢? 因为它满足等价关系的全部三条性质:

1. 因为 $aa^{-1} = e \in H$ , 所以 $a \sim a$ 。也就是自反性成立;
2. 如果 $ab^{-1} \in H$ , 则它的逆 $(ab^{-1})^{-1} = ba^{-1} \in H$ , 所以 $a \sim b \Rightarrow b \sim a$ 。也就是对称性成立;
3. 若 $ab^{-1} \in H, bc^{-1} \in H$ , 我们有 $(ab^{-1})(bc^{-1}) = ac^{-1} \in H$ , 所以 $a \sim b, b \sim c \Rightarrow a \sim c$ 。也就是传递性成立。

**定义 3.1.6.** 由这一等价关系~所决定的类叫做子群 $H$ 的**右陪集** (right coset), 包含元素 $a$ 的右陪集用符号 $Ha$ 来表示。

具体来说, 用 $a$ 从右边去乘 $H$ 的每个元素, 就得到了包含 $a$ 的类。所以 $Ha$ 正好包含所有可以写成 $ha$ , 其中 $h \in H$ 形式的 $G$ 中的元素, 即:

$$Ha = \{ha | h \in H\}$$

如图3.12所示, 令 $Z$ 为整数加群,  $H$ 为所有3的倍数 $0, \pm 3, \pm 6, \dots$ , 它在加法下构成一个子群。我们用 $Z$ 中的元素0从右侧加<sup>17</sup> $H$ 中的每个元素得到 $H0$ , 显然这一结果还等于 $H$ , 是所有模3余0的整数, 记为[0]; 用 $Z$ 中的元素1从右侧加 $H$ 中的每个元素得到 $H1$ , 它包含所有模3余1的整数, 记为[1]; 用 $Z$ 中的元素2从右侧加 $H$ 中的每个元素得到 $H2$ , 它包含所有模3余2的整数, 记为[2]。如果用

<sup>17</sup>由于整数加群是阿贝尔群, 加法满足交换律, 所以左右陪集相同。

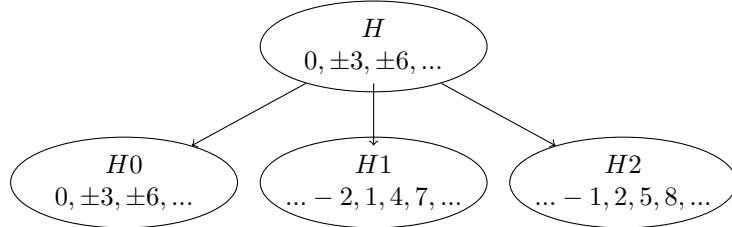


图 3.12: 右陪集。整数加群的子群 \$H\$ 包含所有3的倍数，用0、1、2分别加所有3的倍数可以得到三个互不相交子集。它们恰好是整数的一个分类。

3加所有 \$H\$ 的元素，结果和 \$H\_0\$ 一样。事实上，如果我们用 \$H\_0\$ 中的任何元素 \$a\$ 获得的右陪集 \$Ha\$ 都等于 \$H\_0\$；用 \$H\_1\$ 中的任何元素 \$b\$ 获得的右陪集 \$Hb\$ 都等于 \$H\_1\$，用 \$H\_2\$ 中的任何元素 \$c\$ 获得的右陪集 \$Hc\$ 都等于 \$H\_2\$。\$H\_0, H\_1, H\_2\$ 三个右陪集放在一起恰好是全体整数 \$Z\$。它们的确是 \$Z\$ 的一个分类：[0], [1], [2]。

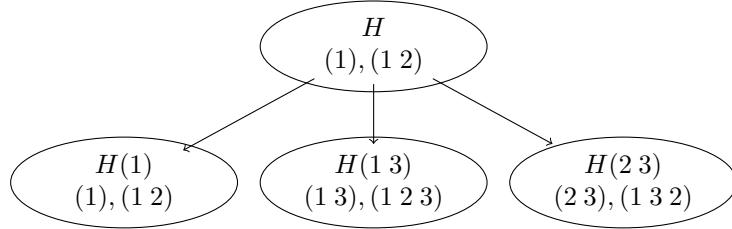


图 3.13: 有限非阿贝尔群的右陪集。

我们再看一个有限非阿贝尔群的例子。考虑3阶置换群

$$G = S_3 = \{(1), (1 2), (1 3), (2 3), (1 2 3), (1 3 2)\},$$

及其子群 \$H = \{(1), (1 2)\}\$。我们用单位元恒等置换 \$(1)\$，和另外两个置换 \$(1 3), (2 3)\$ 右乘 \$H\$ 得到3个右陪集：

$$\begin{aligned} H(1) &= \{(1), (1 2)\} \\ H(1 3) &= \{(1 3), (1 2 3)\} \\ H(2 3) &= \{(2 3), (1 3 2)\} \end{aligned}$$

我们也可以用另外三个元来作右陪集：

$$H(1 2), H(1 2 3), H(1 3 2)$$

同样由于 \$(1 2) \in H(1), (1 2 3) \in H(1 3), (1 3 2) \in H(2 3)\$，所以一定有

$$\begin{aligned} H(1) &= H(1 2) \\ H(1 3) &= H(1 2 3) \\ H(2 3) &= H(1 3 2) \end{aligned}$$

这样，子群 \$H\$ 把整个群 \$G = S\_3\$ 分成 \$H(1), H(1 3), H(2 3)\$ 三个不同的右陪集，这三个右陪集放在一起正是 \$G\$，它们是 \$G\$ 的一个分类。

与右陪集对称，我们也可以定义左陪集。规定对称的等价关系 \$\sim'\$：

$$a \sim' b \text{ 当且仅当 } b^{-1}a \in H$$

这样，由等价关系 \$\sim'\$ 所决定的类叫做子群 \$H\$ 的左陪集。包含元素 \$a\$ 的左陪集用符号 \$aH\$ 来表示。它包含所有形如：\$ah, h \in H\$ 形式的 \$G\$ 的元。因为一个群的乘法不一定满足交换律，所以一般来

说 $\sim$ 和 $\sim'$ 两个等价关系并不相同， $H$ 的左右陪集也不相同。但是一个子群的左右陪集之间有一个共同点：

**定理 3.1.11.** 子群 $H$ 的左右陪集个数相等，他们或者都是无限大，或者都有限并相等。

要想证明这一点，我们可以构造一个从 $H$ 的右陪集到左陪集间的映射： $f : Ha \rightarrow a^{-1}H$ 。容易验证，这是一个一一映射。任意 $Ha = Hb$ ，都有 $ab^{-1} \in H$ ，所以 $(ab^{-1})^{-1} = ba^{-1} \in H$ 。因此 $a^{-1}H = b^{-1}H$ 。既然一一映射存在，自然左右陪集的个数是相等的。

为此，我们可以将子群 $H$ 的陪集个数（左或者右）定义为 $H$ 在 $G$ 中的指數。

一般情况下，子群的右陪集 $Ha$ 未必等于左陪集 $aH$ ，如果相等，这个子群叫做正规子群。正是伽罗瓦第一个引入了正规子群来分析方程的可解性。

**定义 3.1.7.** 一个群 $G$ 的子群 $N$ 叫做一个**正规子群**（或者不变子群），加入对 $G$ 的每个元 $a$ 都有：

$$Na = aN$$

一个正规子群的左（或右）陪集叫做 $N$ 的**陪集**。

由于其对称性，正规子集也叫做群的中心。判断一个群是否是正规子群，我们有以下两个定理：

**定理 3.1.12.** 一个群 $G$ 的子群 $N$ 是一个不变子群的充分必要条件是对 $G$ 中的任何一个元 $a$ 都有：

$$aN a^{-1} = N$$

**定理 3.1.13.** 一个群 $G$ 的子群 $N$ 是一个不变子群的充分必要条件是对 $G$ 中的任何一个元 $a$ ， $N$ 中的任何一个元 $n$ 都有：

$$ana^{-1} \in N$$

我们可以把正规子群 $N$ 的所有陪集做成一个集合 $\{aN, bN, cN, \dots\}$ ，并且定义这个集合上的乘法为：

$$(xN)(yN) = (xy)N$$

可以验证，正规子群的陪集在这个乘法下构成一个群，叫做**商群**，用 $G/N$ 表示。在正规子群、商群和同态映射之间存在着重要的关系。首先一个群 $G$ 同它的每个商群 $G/N$ 同态。要证明这一点，我们可以构造一个映射 $a \rightarrow aN$ ,  $a \in G$ 。这显然是一个从 $G$ 到商群的满射，对于 $G$ 中任何两个元素 $a, b$ ，都有 $ab \rightarrow abN = (aN)(bN)$ 。所以它是一个同态满射。这样我们既可以通过正规子群，也可以通过商群推测群 $G$ 的性质。为了做到这一点，我们定义一下“核”的概念。

**定义 3.1.8.** 若 $f$ 是群 $G$ 到另一个群 $G'$ 的同态满射， $G'$ 的单位元 $e'$ 在映射 $f$ 之下的逆像集合是 $G$ 中的一个子集。这个子集叫做同态满射 $f$ 的**核**。

我们有如下定理，如果 $G$ 和 $G'$ 同态，那么这个同态满射的核 $N$ 是 $G$ 的正规子群。并且其商群 $G/N \cong G'$ 。这样一个群不仅和它的每一个商群同态，并且抽象地来看， $G$ 只能和它的商群同态。有时候，我们发现群 $G$ 和 $G'$ 同态，但是 $G'$ 的性质我们并不清楚，这时候，我们一定能找到 $G$ 的一个正规子群 $N$ ，使得 $G'$ 的性质和商群 $G/N$ 完全一样。从这里我们能看出不变子群和商群的意义。伽罗瓦正是通过这一点，想出并定义了方程的群的可解性。

### 练习 3.6

1. 证明子群的判定定理3.1.8。
2. 列出图3.13中 $H$ 的左陪集。

### 3.1.7 拉格朗日定理

拉格朗日定理是特别体现抽象代数特点的一个定理。在完全无须了解群中元素和运算的具体意义的情况下，我们仍可以揭示抽象结构的内在规律。

拉格朗日全名为约瑟夫·路易斯·拉格朗日，是法国著名数学家、物理学家。1736年1月25日生于意大利都灵。拉格朗日的祖先是法国军官，到意大利后与罗马家族的人结婚定居。他的父亲曾任都灵的公共事务和防务局会计，但后来由于经商破产，家道中落。据拉格朗日本人回忆，如果幼年时家境富裕，他也就不会作数学研究了，因为父亲一心想把他培养成为一名律师。拉格朗日在都灵大学读书时，并没有对数学发生兴趣，他觉得欧几里得几何很枯燥。

17岁时，拉格朗日碰巧看到了英国天文学家哈雷的一篇介绍牛顿微积分的文章。他一下子被吸引住了。开始全身心投入学习数学。1755年拉格朗日19岁时，在探讨数学难题“等周问题”的过程中，他以欧拉的思路和结果为依据，用纯分析的方法求变分极值。第一篇论文“极大和极小的方法研究”，发展了欧拉所开创的变分法，为变分法奠定了理论基础。变分法的创立，使拉格朗日在都灵声名大振，并使他在19岁时就当上了都灵皇家炮兵学校的教授，成为当时欧洲公认的第一流数学家。1756年，受欧拉的举荐，拉格朗日被任命为普鲁士科学院通讯院士。

1766年德国的腓特烈大帝向拉格朗日发出邀请时说，在“欧洲最大的王”的宫廷中应有“欧洲最大的数学家”。于是他应邀前往柏林，任普鲁士科学院数学部主任，居住达20年之久，开始了他一生科学的研究的鼎盛时期。在此期间，他完成了《分析力学》一书，这是牛顿之后的一部重要的经典力学著作。书中运用变分原理和分析的方法，建立起完整和谐的力学体系，使力学分析化了。

腓特烈大帝非常欣赏拉格朗日，常常和他谈论规律生活的重要性。在皇帝的影响下，拉格朗日每天晚上都想好明天要完成的具体任务。每当工作取得进展，他都写一段简短的分析看看哪里还能进一步改进。拉格朗日在撰写论文前总是深思熟虑，他通常能一气呵成而不用修改。但拉格朗日不太习惯柏林的气候，他的身体一直不太好。1783年，他的妻子维多利亚因病去世。

1786年一直支持他的腓特烈大帝也去世了。拉格朗日接受了路易十六的邀请来到巴黎。他得到了特别的接待，国王甚至在卢浮宫里为他安排了一处住所。1792年，拉格朗日的朋友，天文学家勒莫尼埃（LeMonnier）的女儿，24岁蕾妮（Renée-Francoise- Adelaide）坚持要嫁给孤单一人的拉格朗日。他们结婚后虽未生儿女，但家庭幸福。

这时法国大革命爆发了。1793年9月革命政府决定逮捕所有在敌国出生的人，著名化学家拉瓦锡在身处危险的情况下竭力奔走相助，终于才把拉格朗日作为例外。1794年5月4日法国雅各宾派开庭审判波旁王朝税务人员，把包括拉瓦锡在内的28名成员全部处以死刑。拉格朗日等人尽力地挽救，请求赦免，但是遭到了革命法庭副长官考费那尔（J.B.Coffinhal）的拒绝，全部予以驳回，并宣称，“共和国不需要学者，而只需要为国家而采取的正义行动！”5月8日早晨，拉格朗日痛心地说：“他们可以一眨眼就把拉瓦锡的头砍下来，但他那样的头脑一百年也再长不出一个来了。”<sup>[24]</sup>

1799年雾月政变后，拿破仑热情支持法国的科学的研究。他提名拉格朗日等著名科学家成为上议院议员及新设的勋级会荣誉军团成员，封为伯爵；还在1813年4月3日授予他帝国大十字勋章。此时拉格朗日已重病在身，终于在4月11日晨逝世。在葬礼上，由议长拉普拉斯代表上议院，院长拉赛佩德（Lacépède）代表法兰西研究院致悼词。意大利各大学都举行了纪念活动。

拉格朗日是18世纪的伟大科学家，在数学、力学和天文学三个学科中都有历史性的重大贡献。但他主要是数学家，拿破仑曾称赞他是“一座高耸在数学界的金字塔”，他最突出的贡献是在把数学分析的基础脱离几何与力学方面起了决定性的作用。使数学的独立性更为清楚，而不仅是其他



拉格朗日纪念邮票

学科的工具。同时在使天文学力学化、力学分析化上也起了历史性作用，促使力学和天文学（天体力学）更深入发展。

拉格朗日在柏林的前十年，把大量时间花在代数方程和超越方程的解法上。他在代数方程解法中有历史性贡献。他把前人解三、四次代数方程的各种解法，总结为一套标准方法，而且还分析出一般三、四次方程能用代数方法解出的原因。三次方程有一个二次辅助方程，其解为三次方程根的函数，在根的置换下只有两个值；四次方程的辅助方程的解则在根的置换下只有三个不同值，因而辅助方程为三次方程。拉格朗日称辅助方程的解为原方程根的预解函数（是有理函数）。他继续寻找5次方程的预解函数，希望这个函数是低于5次的方程的解，但没有成功。尽管如此，拉格朗日的想法已蕴含着置换群概念，而且使预解（有理）函数值不变的置换构成子群，子群的阶是原置换群阶的因子。这正是群论中著名的拉格朗日定理。拉格朗日是群论的先驱。他的思想为后来的阿贝尔和伽罗瓦采用并发展，终于解决了五次以上的一般方程为何不能用代数方法求解的问题。

在介绍拉格朗日定理前，我们首先看一个引理：

**引理 3.1.14.** 一个子群  $H$  与其每一个右陪集  $Ha$  之间都存在一一映射。

由于陪集的左右对称性，这一结论对于左陪集也成立。要想证明它，我们可以构造映射  $f : h \rightarrow ha$ ，它就是一个从子群到右陪集的一一映射，因为：

1.  $H$  中的每个元素  $h$  都有一个唯一的像  $ha$ ；
2.  $Ha$  中的每个元素  $ha$  都是子群  $H$  中元素  $h$  的像；
3. 任意  $h_1a = h_2a$ ，都有  $h_1 = h_2$ 。

由于一一映射的存在，我们知道有限群  $G$  中，陪集中元素的个数一定与子群  $H$  的阶相等。而且由于陪集的分类特性，我们知道群中的每个元素都可以在某一个陪集中找到。这样我们就可以利用陪集发现子群  $H$  与有限群  $G$  之间的一个关系：

**定理 3.1.15. 拉格朗日定理：**有限群  $G$  的阶，能够被其子群  $H$  的阶整除。

证明. 首先，我们知道  $G$  能够为  $H$  的陪集全部覆盖；并且由于陪集之上定义的等价关系，我们知道陪集之间是没有交集的。如果存在元素  $c$  既属于  $Ha$ ，也属于  $Hb$ ，那么  $c \sim a, c \sim b$ ，所以  $a \sim b$ ，所以  $Ha = Hb$ 。再加上子群  $H$  与陪集存在一一映射的引理，我们知道每个陪集的大小都等于  $H$  的阶  $|H| = n$ ，如果陪集的个数（也就是  $H$  的指数）等于  $m$ ，一定有

$$|G| = mn$$

□

注意，拉格朗日定理的逆定理不一定成立。我们无法将群的阶做任意因子分解，然后对每个因子都找到子群和陪集。从拉格朗日定理，我们可以得到很多有用的推论。

**推论 3.1.16.** 一个有限群  $G$  中的任意元素  $a$  的阶都能整除  $G$  的阶。

这是因为  $a$  生成一个阶为  $n$  的子群，所以  $n$  能整除  $|G|$ 。

**推论 3.1.17.** 如果  $G$  的阶是素数，那么  $G$  是循环群。

这是因为任何不等于单位元的元素  $a$ ，它所生成的子群的阶一定等于  $G$  的阶。所以  $a$  是  $G$  的生成元，即  $G = (a)$ 。

**推论 3.1.18.** 有限群  $G$  中的任何元素  $a$ ，都有  $a^{|G|} = e$ 。

这是因为 $a$ 的阶 $n$ 能够整除 $|G|$ , 不妨令 $|G| = nk$ 。所以

$$a^{|G|} = a^{nk} = (a^n)^k = e^k = e$$

群论中的拉格朗日定理可以用来证明数论中著名的费马小定理和欧拉定理。费马小定理是法国数学家费马于1636年发现的。他在1640年10月18日写给友人<sup>18</sup>的信中首次提出了这个定理。1736年, 欧拉给出了费马小定理的一个证明。但从莱布尼茨未发表的手稿中发现他在1683年以前已经得到几乎相同的证明。

**定理 3.1.19. 费马小定理:** 若 $p$ 是素数, 对任何满足 $0 < a < p$ 的整数 $a$ 都有,  $a^{p-1} - 1$ 能被 $p$ 整除。

证明. 考虑整数模 $p$ 乘法群。这个群的元素由所有模 $p$ 的非零余数构成, 因为 $p$ 是素数, 所以群元素为 $1, 2, \dots, p-1$ , 单位元 $e = 1$ , 群的阶为 $p-1$ 。根据拉格朗日定理的推论3.1.18, 有:

$$a^{p-1} = e$$

因为单位元为 $1$ , 所以上式可写为:

$$a^{p-1} \equiv 1 \pmod{p}$$

故 $p$ 整除 $a^{p-1} - 1$ 。 □

和这一证明对比, 初等数论方法的证明要复杂得多 ([10]第5.2到5.4)。我们再介绍另一种有趣的“项链”证法[25], 供读者参考。

证明. 考虑有 $a$ 颗不同颜色的珍珠, 我们要串一条长为 $p$ 的珍珠串。其中 $p$ 为素数。显然一共有 $a^p$ 种不同的串, 这是因为串中每个位置都可以在 $a$ 颗珍珠中选择, 共选 $p$ 次。

例如有两种颜色 $A, B$ 的珍珠, 要串长度为5的串。即 $a = 2, p = 5$ , 一共有 $2^5 = 32$ 种不同的串:

AAAAA, AAAAB, AAABA, ..., BBBBA, BBBBB.

我们要证明, 在这 $a^p$ 串珍珠中, 如果去掉 $a$ 个颜色完全相同的串 (在上述例子中是串AAAAA和BBBBB), 剩下的 $a^p - a$ 串珍珠可以分成若干组, 每个组恰好有 $p$ 串珍珠。也就是说 $p$ 整除 $a^p - a$ 。

如果把每串珍珠首尾连接起来做成项链。原来有些不同的串会变成相同的项链。如果一个串可以通过旋转变换成另一个, 这两个串必然做成相同的项链。例如下面的5个串珍珠可以做成相同的项链:

AAAAB, AAABA, AABAA, ABAAA, BAAAA.

用这种方法, 上面例子中的32串珍珠可以分成5种不同颜色的项链和2种同色的项链:

[AAABB, AABBA, ABBAA, BBAAA, BAAAB];  
 [AABAB, ABABA, BABAA, ABAAB, BAABA];  
 [AABBB, ABBBA, BBBAA, BBAAB, BAABB];  
 [ABABB, BABBA, ABBAB, BBABA, BABAB];  
 [ABBBB, BBBBA, BBBAB, BBABB, BABBB];  
 [AAAAA];  
 [BBBBB].

一串项链可以代表多少种不同的串呢? 如果一个串 $S$ 由若干相同的子串 $T$ 复制连接而成, 而 $T$ 无法再继续分拆成相同的子串, 则由 $S$ 构成的项链可代表 $|T|$ 种不同的串, 其中 $|T|$ 表示串 $T$ 的长度。例如串 $S = ABBABBABB$ , 由相同的子串 $T = ABB$ 复制多次而成, 如果我们一次旋转一颗珍珠, 一共可以得到3串不同的结果:

<sup>18</sup>费马的朋友法国数学家贝西 (Bernard Frénicle de Bessy)



(a) 一串项链代表7种不同的串: ABCBAAC, BCBAACA, CBAACAB, BAACABC, AACACB, ACABCBA, CABCBAA  
(b) 相同珍珠串成的项链只代表一种串: AAAAAAA

图 3.15: 通过项链对珍珠串进行分类

ABBABBABBABB,  
BBABBABBABBA,  
BABBBABBBABAB.

除去这3种之外，不可能再有其它不同种类的串了。**ABB**的长度为3，再次旋转必然会循环得到相同的结果。这样，所有的 $a^p$ 串珍珠可分成两类。一类是 $a$ 串颜色相同的串；另一类颜色不同，但是由于这些串的长度 $p$ 是素数，它绝不可能由若干子串复制连接出来。所以任何一个这样的串连成的项链，总共代表 $p$ 种不同的串。而这样的串总共有 $a^p - a$ 种，它们可以通过连成项链后分组，每组恰好 $p$ 个串，代表一种不同的项链。因此 $a^p - a = a^{p-1}$ 一定可以被 $p$ 整除。□

项链证明法可能是费马小定理已知证法中最容易的，它不需要太多的数学知识。核心思想是利用两种不同的方式数数的结果必然是一样的。

费马小定理从提出到欧拉成功证明经过了整整100年。这是费马的一贯风格，费马大定理（也称费马的最后定理）难住了无数天才数学家，历经358年才由英国数学家安德鲁·怀尔斯在1995年成功攻克。而怀尔斯的主要武器包括椭圆曲线、模形式和伽罗瓦表示[12]。可以说费马留下的这些猜想是一笔丰富的数学遗产。



皮埃尔·德·费马 1601-1665

费马是法国著名数学家，1601年8月生于法国图卢兹一个富有的皮革商人家中。他是一位全职律师，一直在司法部门工作，后来还当过图卢兹最高法院的大法官。费马在成年后才开始利用业余时间研究数学。但费马取得的数学成就，堪称那个时代的高峰。1630年，费马独立得到了解析几何的基本原理<sup>19</sup>，因而与笛卡尔分享创立解析几何的荣誉；他与帕斯卡在一段漫长而有趣的通信中一起奠定了古典概率论的基础，提出了数学期望的概念，因而与帕斯卡被公认为是概率论的创始人；他提出光学中的“费马原理”<sup>20</sup>，给后来变分法的研究以极大的启示；他是创建微积分学的杰出先驱者。而费马本人最感兴趣的领域是数论。

费马对数论的兴趣是由古希腊数学家丢番图的名著《算术》启发的。1621年，巴谢（Bachet）校订注释的希腊——拉丁文对照本在法国出版。费马在巴黎买到此书，并被其中的数论问题深深吸引。值得庆幸的是，这一版的每一页都留有宽大的空白

<sup>19</sup> 费马的8页论文《平面与立体轨迹引论》在1630年完成，但在1679年费死后14年才出版。

<sup>20</sup> 又称最小作用原理，或最短时间作用原理。

页边，于是这书同时成了费马的笔记本。在研究丢番图的问题和解答时，他会受到启发去思考和解决更多、更深入的问题，并草草地在这些空白书边上写下自己的评注。

费马在其生前，几乎没有公开发表过任何他的数学成果。这在“不发表就发霉”的今天是很难被理解的。除了把自己的研究成果写在书籍的页边，费马也常常按照当时流行的风气，以书信的形式向一些学者朋友报告。究其原因，费马是被“发现新的数学奥秘”这类强烈的念头所驱使的。研究数学对于他是一种消遣，纯粹是由于他对数学的爱好。当创造出他从未触及的结果时，费马会获得真正的愉悦与自我满足。因而公开发表和被人们承认对他而言没有多大意义[12]。有趣的是，这位缄默的天才有时喜好捉弄人，他经常在信中向其他数学家发出挑战，要求他们证明自己发现的某些数学结果。

当费马在1665年1月去世时，他的研究成果散落各处。费马的长子塞缪尔（Samuel）花了5年时间整理信件收集注记，1670年将他父亲的成果出版为一个特殊版本的《算术》。在封面图片上方，写着“附有费马的评注”的小字。这一版本包括费马所做的48个评注。1679年，他又整理了出版了费马的第二卷著作。费马生前的这些研究成果终于得以流传，极大地丰富了十七世纪的数学宝库，推动了后来的数学发展。

在费马之前，数论基本上是一些相关问题的汇集。费马振兴了数论的研究，系统地提出了数量众多的数论定理，并引入了一般化的方法和原理，从而把数论引上了近代发展的轨道。可以说，正是费马的系统化工作，数论才真正开始成为一门数学分支。费马也因此奠定近代数论的基础，而被称为“近代数论之父”。在高斯的《算术研究》出版之前，数论的发展始终是跟费马的推动联系在一起的。

然而，费马生前提出的种种结论常常仅含有证明的一些关键部分，有时甚至根本没有证明。有些内容后来发现是错误的<sup>21</sup>。因此在找到严格的数学证明前，这些结论只能称之为“猜想”。这些猜想有很多是被欧拉攻克的，不仅如此，欧拉还在费马的基础上发展了更多的结果。

我们接下来要介绍的数论中的欧拉定理，就是比费马小定理更加普遍的结果。在成功证明了费马小定理后，欧拉并不满足，仔细研究了合数的情况，得出并证明了下面的定理。

**定理 3.1.20. 欧拉定理：**如果  $0 < a < n$  且  $a$  与  $n$  互素，那么  $a^{\phi(n)} - 1$  能够被  $n$  整除。

其中  $\phi(n)$  是欧拉函数<sup>22</sup>，它的定义是所有小于  $n$  且与  $n$  互素的正整数个数。即：

$$\phi(n) = |\{i | 0 < i < n \text{ 且 } \gcd(i, n) = 1\}|$$

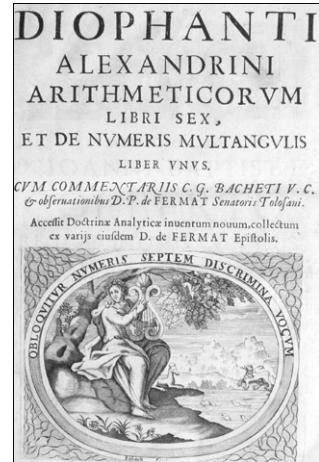
欧拉使用初等数论的方法证明了这一定理。我们展示如何利用群论和拉格朗日定理给出一个简洁的证明。

证明. 考虑整数模  $n$  的非零余数。我们把模  $n$  乘法之下可以互逆的所有余数挑选出来。它们构成一个模  $n$  乘法群。按照欧拉函数的定义， $\phi(n)$  的值，是所有小于  $n$  且与  $n$  互素的正整数的个数。而这些正整数，恰好对应这个乘法群中的元素。所以这个群的阶就是  $\phi(n)$ 。根据拉格朗日定理的推论3.1.18，有：

$$a^{\phi(n)} = e$$

<sup>21</sup>例如费马数。费马在1640年宣称  $2^{2^n} + 1$  形式的数都是素数，并且当  $n$  为  $0, 1, 2, 3, 4$  时都是对的。它们分别是  $3, 5, 17, 257, 65537$ 。但欧拉在1732年算出  $2^{2^5} + 1 = 641 \times 6700417$ 。目前（2017年）人们已经找到243个反例，却没有再发现第6个费马素数。是否还有其它费马素数至今没有人能够解决。

<sup>22</sup>也称为欧拉总计函数（Euler totient function）或者欧拉  $\phi$  函数



丢番图《算术》1670年版，带有费马评注

如果  $p$  不是素数会怎样呢？欧拉仔

故而  $a^{\phi(n)} \equiv 1 \pmod{n}$ , 所以  $n$  一定整除  $a^{\phi(n)} - 1$ 。  $\square$

我们举个例子, 下表是模10的非零余数的乘法表。我们可以从表中找到单位元1, 用下划线标出。然后从其所对应的行列找到模乘的两个数, 加粗标出。这些数都和10互素。反之, 那些不和10互素的余数所在的行列中都有0元素, 而0不是群中的元。可以看到和10互素的余数恰好是群中的元素1, 3, 7, 9。

	1	2	3	4	5	6	7	8	9
1	<u>1</u>	2	3	4	5	6	7	8	9
2	2	4	6	8	0	2	4	6	8
3	3	6	9	2	5	8	<u>1</u>	4	7
4	4	8	2	6	0	4	8	2	6
5	5	0	5	0	5	0	5	0	5
6	6	2	8	4	0	6	2	8	4
7	7	4	<u>1</u>	8	5	2	9	6	3
8	8	6	4	2	0	8	6	4	2
9	9	8	7	6	5	4	3	2	<u>1</u>

任给一个整数  $n$ , 如何求出其欧拉函数的值呢? 因为任何大于1的整数都可以写成素数幂的乘积形式, 我们首先来看, 对于素数  $p$  的  $m$  次幂,  $\phi(p^m)$  如何计算。也就是求出从1到  $p^m - 1$  中, 有多少个数和  $p^m$  互素。显然我们只要把  $p$  的倍数这些数除去即可。这些数分别是:  $p, 2p, 3p, \dots, p^m - p$ , 我们把它们分别除以  $p$ , 就可以得到自然数序列:  $1, 2, 3, \dots, p^{m-1} - 1$ 。显然共有  $p^{m-1} - 1$  个。于是素数的整数次幂的欧拉函数值为:

$$\begin{aligned}\phi(p^m) &= (p^m - 1) - (p^{m-1} - 1) \\ &= p^m - p^{m-1} \\ &= p^m(1 - \frac{1}{p})\end{aligned}$$

接下来我们考虑  $n = p^u q^v$ , 也就是两个不同素数幂的积。我们首先从1到  $n - 1$  中, 减去  $p$  的所有倍数, 然后再减去  $q$  的所有倍数, 但是有些整数既是  $p$  的倍数, 也是  $q$  的倍数, 所以最后要把这些  $pq$  的倍数再加回来 (组合中的容斥原理)。这样有:

$$\begin{aligned}\phi(p^u q^v) &= (n - 1) - (\frac{n}{p} - 1) - (\frac{n}{q} - 1) + (\frac{n}{pq} - 1) \\ &= n(1 - \frac{1}{p})(1 - \frac{1}{q}) \\ &= p^u(1 - \frac{1}{p})q^v(1 - \frac{1}{q}) \\ &= \phi(p^u)\phi(q^v)\end{aligned}$$

特别地, 当指数  $u, v$  都是1的时候, 我们有  $\phi(pq) = \phi(p)\phi(q)$ 。并且我们可以把这个结果推广到多个素数幂的情况, 如果  $n = p_1^{k_1} p_2^{k_2} \dots p_m^{k_m}$ , 则其欧拉函数的值为:

$$\begin{aligned}\phi(n) &= n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \dots (1 - \frac{1}{p_m}) \\ &= \phi(p_1^{k_1})\phi(p_2^{k_2}) \dots \phi(p_m^{k_m})\end{aligned}$$

根据这一结论, 我们可以找到快速求欧拉函数的方法, 读者可以通过本节的习题实现这一算法。

欧拉是伟大的瑞士数学家和自然科学家。人们一般把他与阿基米德、牛顿、高斯并列为数学史上最伟大的四位数学家。1707年4月15日欧拉出生于瑞士的巴塞尔。牧师家庭的父亲曾希望他学习神学。欧拉13岁考入巴塞尔大学神学院，主修哲学和法律。欧拉很快在数学方面表现出兴趣，每周六下午都和当时欧洲最优秀的数学家约翰·伯努利学习数学。16岁时，欧拉取得了他的哲学硕士学位。之后，他遵从了父亲的意愿进入了神学系，并准备成为一名牧师。但最终约翰·伯努利说服欧拉的父亲允许欧拉学习数学，并使他相信欧拉注定能成为一位伟大的数学家。



各国发行的欧拉纪念邮票。欧拉 (Leonhard Euler) 1707 - 1783

1727年，欧拉成为俄罗斯圣彼得堡科学院的成员。他以旺盛的精力投入研究，在俄国的14年中，他在分析学、数论和力学方面作了大量出色的工作。1741年受腓特烈大帝的邀请到柏林科学院工作，达25年之久。在柏林期间他的研究内容更加广泛，涉及行星运动、刚体运动、热力学、弹道学、人口学，这些工作和他的数学研究相互推动。欧拉这个时期在微分方程、曲面微分几何以及其他数学领域的研究都是开创性的。1766年，叶卡捷琳娜二世在位期间，欧拉应邀重新返回圣彼得堡，并在那里直到逝世。

欧拉渊博的知识，无穷无尽的创作精力和空前丰富的著作，都是令人惊叹不已的。他从19岁开始发表论文，直到76岁，半个多世纪写下了浩如烟海的书籍和论文。几乎每一个数学领域都可以看到欧拉的名字。欧拉是科学史上最多产的一位杰出的数学家，一生发表论文共计856篇，专著31部。这还不包括1771年圣彼得堡火灾中失去的一部分。（欧拉的记录直到20世纪才由匈牙利数学家保罗·埃尔德什打破。后者发表的论文1525篇，著作32部。[\[26\]](#)）

欧拉具有超出常人的坚强意志。1735年，他的右眼开始失明。1771年原本正常的左眼也完全失明了。正如失聪没有阻止贝多芬的音乐创作一样，失明也同样没有阻止欧拉的数学探索。[\[12\]](#)。在书记员的帮助下，欧拉在多个领域的研究其实变得更加高产了。在1775年，他平均每周就完成一篇数学论文。欧拉一生中有一半著作都是在双目完全失明后口述完成的。欧拉的多产并不是偶然的，他有着惊人的记忆力。不但能够记住前100个素数，而且还能记住它们的平方、立方甚至更高次方。他还可以进行复杂的心算。法国物理学家阿拉戈 (François Arago) 说“欧拉计算时就像人在呼吸、鹰在翱翔一样轻松”。欧拉可以在任何不良的环境中工作，他常常抱着孩子在膝上完成论文，也不顾孩子在旁边喧哗。

欧拉的著作中既有难度很高的专著，也有专为普通大众所写的读物。他还特意为青少年写过一本书——《给德国公主的信》。并为非数学专业的读者写了一本初等代数教程，这本书至今仍在印刷。欧拉特别重视表达的清晰易懂。我们今天很多熟知的数学符号，都是欧拉精心选用的，例如 $\pi$ （1736年），虚数单位*i*（1777年），*e*（1748年），三角函数 $\sin$ 和 $\cos$ （1748年）， $\tg$ （1753年）， $\Delta x$ （1755年），求和 $\sum$ （1755年），表示函数的 $f(x)$ （1734年）等[12]。

1783年9月18日，欧拉与朋友们吃饭。那天天王星刚发现不久，欧拉列出计算天王星轨道的要领。晚餐后，欧拉一边喝着茶，一边和小孙女玩耍，突然之间，烟斗从他手中掉了下来。他说了一声：“我的烟斗”，并弯腰去捡，结果再也没有站起来，他喃喃地说了一句：“我死了……”就这样“停止了计算和生命”<sup>23</sup>。

今天，费马小定理已经走进人们的日常生活中，不管是网络购物还是电子交易。1976年，美国斯坦福大学的教授马丁·赫尔曼（Martin Hellman）和惠特菲尔德·迪菲（Whitfield Diffie）提出了非对称公钥加密算法的思想。1977年美国麻省理工学院的罗纳德·李维斯特（Ron Rivest）、阿迪·萨莫尔（Adi Shamir）和伦纳德·阿德曼（Len Adleman）提出了构造单向函数的数论方法，从而产生了以这三个人姓氏首字母命名的RSA算法。

RSA算法的核心思想是人们可以容易地将两个大素数乘在一起得到一个合数，然而在不事先知道这两个素数的情况下，对这个合数做因数分解却非常困难。对于一个200位以上的大数做因式分解，即使用强大的超级计算机，所耗费的时间也要超过宇宙的年龄。因此，如果能够迅速地找到大素数，就可以构造难以破解的密钥。但是素数的存在规律是神秘的，人们没有找到素数的“通项公式”。最原始的办法是挑选一个数 $n$ ，然后逐一验证从1到 $\sqrt{n}$ 之间的整数能否整除 $n$ 。但这种方法非常低效，对大数进行素数检测，同样会超过宇宙年龄所需的时间。稍好的方法是埃拉托斯特尼筛法。列出从2开始到 $n$ 之间的整数，然后从2开始，先筛除所有2的倍数，然后再筛除3的倍数，这样每次都从没有被筛除的第一个数开始重复这一步骤，直到把不大于 $\sqrt{n}$ 的所有倍数都筛除为止。这样就可以获得 $n$ 以内的所有素数。但这样方法同样只是对较小的整数 $n$ 有效。无法达到大素数检测的目的。

费马小定理恰好给出了一种大素数检验的办法。对于一个大整数 $n$ ，我们可以随机挑选一个小于 $n$ 的正整数 $a$ ，将其称为“证人”（witness），然后检查 $a^{n-1} \bmod n$ 的余数是否等于1，如果不是1，根据费马小定理， $n$ 一定不是素数。如果等于1，则 $n$ 有可能是素数。

根据这一思想构造的“费马素数检测”算法如下：

```
function primality(n)
    随机选择正整数  $a < n$ 
    if  $a^{n-1} \equiv 1 \pmod n$  then
        return 素数
    else
        return 合数
```

我们并不需要真的计算 $a$ 的 $n - 1$ 次方然后再求除 $n$ 的余数，而是通过模乘运算。并且还可以进一步利用中间的计算结果加速，例如当我们得到 $b = a^2 \bmod n$ 时，可以直接计算 $b^2 \bmod n$ 从而得到 $a^4 \bmod n$ 。假设要计算 $a^{11} \bmod n$ ，因为：

$$a^{11} = a^{8+2+1} = ((a^2)^2)^2 a^2 a \bmod n$$

所以我们真正需要计算的只有 $a^2 \bmod n$ ,  $(a^2)^2 \bmod n$ ,  $((a^2)^2)^2 \bmod n$ 。为此，我们可以把 $n$ 表示为2进制，然后仅仅迭代计算数字为1所在位上的模乘结果，这是一个复杂度为 $O(\lg n)$ 的算法。因此费马素数检测的速度很快。

但某个数即使通过了费马素数检测，仍然不一定是素数，例如 $341 = 11 \times 31$ ，但是 $2^{340} \equiv 1 \pmod{341}$ 。为了减少费马检验的“假阳性”，人们进行了一系列改进。首先是适当增加证人的数量。人们发现，如果一个数无法通过费马检验，那么至少存在一半小于 $n$ 的数都无法通过费马检验[28]（页码）。

---

<sup>23</sup>法国数学家孔多塞语

**定理 3.1.21.** 正整数  $a$  小于  $n$ , 且和  $n$  互素, 如果  $a^{n-1} \not\equiv 1 \pmod{n}$ , 则所有  $a < n$  的选择中, 至少有一半也是这样。

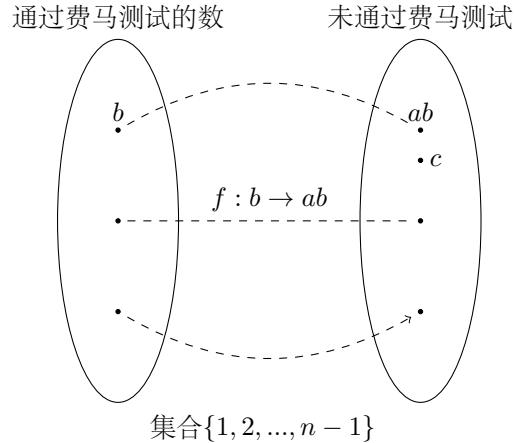


图 3.19: 从通过费马测试的集合向未通过的集合做映射

证明. 若某个  $a$  使得  $a^{n-1} \not\equiv 1 \pmod{n}$ , 对于任何可以同行过费马检测的证人  $b$  (即  $b^{n-1} \equiv 1 \pmod{n}$ ), 都可以构造一个费马检测的反例  $ab$

$$(ab)^{n-1} \equiv a^{n-1}b^{n-1} \equiv a^{n-1}1 \not\equiv 1 \pmod{n}$$

并且, 由于  $i \neq j$ , 有  $a \cdot i \not\equiv a \cdot j$ , 所以这些反例都是彼此不同的。

如图 3.19 所示, 如果存在不通过费马测试的整数, 则这样的数至少和通过的一样多。□

为此, 我们可以多次选取不同的证人  $k$  次执行费马检验, 这样就可以把  $n$  不是素数的概率降低到  $\frac{1}{2^k}$ 。但实际上存在这样的合数  $n$ , 使得任何小于  $n$  且和  $n$  互素的数  $a$  都有  $a^{n-1} \pmod{n}$ 。也就是说无论选什么样的  $a$ , 这样的合数都能通过费马检测。卡迈克尔 (Carmichael) 在 1910 年发现了第一个这样的数  $561 = 3 \times 11 \times 17$ , 这样的数现在被称为卡迈克尔数, 或者费马伪素数<sup>24</sup>。埃尔德什曾经猜测有无穷多个卡迈克尔数, 1994 年人们证明了对足够大的  $n$ , 在 1 到  $n$  之间至少存在  $n^{2/7}$  个卡迈克尔数。从而说明存在无穷多的卡迈克尔数[27]。

实际的 RSA 算法采用“米勒——拉宾”进行素数测试。它也是一种概率算法<sup>25</sup>。根据上述定理, 如果选择超过 100 个证人, 错误率会低于  $\frac{1}{2^{100}}$ 。高德纳说: “该测试的错误率要比计算机因为宇宙辐射而丢失某个二进制位的概率还要低。”

到目前为止, 我们介绍过的群、半群、么半群的关系总结如下。

### 练习 3.7

1. 今天是星期日,  $2^{100}$  天以后是星期几?
2. 任给两个串 (字符串或者列表), 如何通过编程判断它们可以连成相同的项链?
3. 编程实现埃拉托斯特尼筛法。

<sup>24</sup>捷克数学家西摩尔卡 (Václav Šimerka) 在 1885 年发现了前 7 个费马伪素数:  $561 = 3 \times 7 \times 11$ ,  $1105 = 5 \times 13 \times 7$ ,  $1729 = 7 \times 13 \times 19$ ,  $2465 = 5 \times 17 \times 29$ ,  $2821 = 7 \times 13 \times 31$ ,  $6601 = 7 \times 23 \times 41$ ,  $8911 = 7 \times 19 \times 67$ 。但是他的工作不为人知。

<sup>25</sup>米勒——拉宾素数检验存在一个确定性算法的版本, 但是其正确性依赖于黎曼假设 (黎曼猜想) [29]。

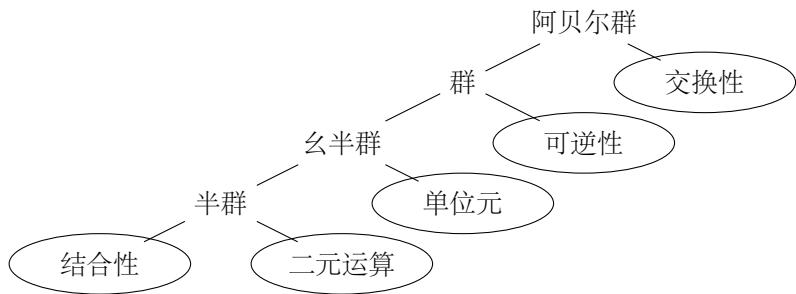


图 3.20: 群、半群、幺半群

4. 利用埃拉托斯特尼筛法的思想，编程产生2到100内正整数的欧拉 $\phi$ 函数表。
5. 编程实现模乘的幂运算，并实现费马素数检测。

## 3.2 环与域

现代抽象数学中环和域的概念是由德国女数学家埃米·诺特所发展的。1882年3月23日，诺特生于德国大学城埃尔朗根的一个犹太人家庭。她的父亲是埃尔朗根大学的数学家。诺特本来打算将来教授法文和英文，但在父亲的影响下，她逐渐对数学产生了兴趣。1900年冬天，18岁的诺特考进了爱尔朗根大学。当时，大学里不允许女生注册，女生顶多只有自费旁听的资格。大学的几百名学生中只有两名女生，诺特大大方方地坐在教室前排，认真听课，刻苦学习，后来，她勤奋好学的精神感动了主讲教授，破例允许她与男生一样参加考试。1903年7月，诺特顺利通过了毕业考试，男们都取得了文凭，而她却成了没有文凭的大学毕业生。诺特生活在公开歧视妇女发挥数学才能的制度下。毕业以后她在埃尔朗根数学研究院工作，长达7年却没有任何工资。1907年12月，她以优异的成绩通过了博士考试，成为第一位女数学博士。此后，她在著名的数学家高丹、费叶尔的指引下，在数学的不变式领域进行了深入的研究。1915年，应著名数学家希尔伯特和克莱因的邀请，诺特来到数学圣地哥廷根大学。不久，她就以希尔伯特教授的名义，在哥廷根大学讲授数学课程。希尔伯特十分欣赏诺特的才能，想帮她在哥廷根大学找一份正式的工作。当时的哥廷根大学没有专门的数学系，数学、语言学、历史学都划在哲学系里，聘请必须经过哲学教授会议批准。希尔伯特的努力遭到教授会议中语言学家和历史学家的极力反对，他们出于对妇女的传统偏见，连聘为“私人讲师”这样的请求也断然拒绝。希尔伯特屡次据理力争都没有结果，他在一次教授会上愤愤地说：“我简直无法想象候选人的性别竟成了反对她升任讲师的理由。先生们，别忘了这里是大学而不是洗澡堂！”

在希尔伯特等人的影响下，不到两年时间，诺特就发表了两篇重要论文。诺特为爱因斯坦的广义相对论给出了一种纯数学的严格方法。并提出了现代物理学中称为“诺特定理”的观点。就这样，诺特以她出色的科学成就，迫使那些歧视妇女的人也不得不于1919年准许她升任讲师。此后，诺特走上了完全独立的数学道路。1921年，她从不同领域的相似现象出发，把不同的对象加以抽象化、公理化，然后用统一的方法加以处理，完成了《环中的理想论》这篇重要论文。这是一项非常了不起的数学创造，它标志着抽象代数学真正成为一门数学分支，或者说标志着这门数学分支现代化的开端。诺特也因此获得了极大的声誉，被誉为是“现代数学代数化的伟大先行



埃米·诺特 1882-1935

者”，“抽象代数之母”。1931年，她的学生荷兰数学家的范德瓦尔登系统总结了整个诺特学派的成就，出版了《近世代数》一书，影响了许多当时的青年数学家。1932年，诺特的科学声誉达到了顶点。在这一年于苏黎世举行的第9届国际数学家大会上，诺特作了长达1小时的大会发言，受到广泛的赞扬。

然而，巨大的声誉并未改善诺特的艰难处境。在不合理的制度下，灾难和歧视像影子一样缠住了她。1922年，由于大数学家希尔伯特等人的推荐，诺特终于在清一色的男子世界——哥廷根大学取得教授称号。不过，那只是一种编外教授，没有正式工资，于是，这位历史上最伟大的女数学家，只能从学生的学费中支取一点点薪金，来维持极其简朴的生活。希特勒上台后，德国法西斯对犹太人的迫害愈演愈烈。1929年，诺特被撵出居住的公寓。1933年4月，法西斯当局剥夺了诺特教书的权利，将一批犹太人教授逐出校园，诺特只好辗转逃往美国。尽管她是世界知名的数学家，但由于是女性，诺特没有获得大型研究院校的聘约。最后，她在一家名为布林莫尔（Bryn Mawr）的女校以访问学者的身份任教。1935年，诺特做了卵巢囊肿摘除手术，4月14日不幸去世，终年53岁。

诺特善于通过透彻的洞察建立优雅的抽象概念，再将之漂亮地形式化。被帕维尔·亚历山德罗夫、爱因斯坦、迪厄多内、外尔和维纳形容为数学史上最重要的女性。她彻底改变了环、域和代数的理论。在物理学方面，诺特定理解释了对称性和守恒定律之间的根本联系，她还被称为“现代数学之母”，她允许学者们无条件地使用她的工作成果，也因此被人们尊称为“当代数学文章的合著者”<sup>[30]</sup>。

### 3.2.1 环的定义

我们接下来看一看环（ring）的定义。

**定义 3.2.1.** 一个集合  $R$  叫做一个环，如果满足以下条件：

1.  $R$  是一个加群，也就是说  $R$  对于其上定义的某种加法构成一个阿贝尔群；
2.  $R$  上还定义了乘法，并且对于这个乘法是封闭的；
3. 乘法满足结合律，任意  $a, b, c$ ，有： $(ab)c = a(bc)$ ；
4. 满足分配律，任意  $a, b, c$  都有：

$$\begin{aligned} a(b+c) &= ab+ac \\ (b+c)a &= ba+ca \end{aligned}$$

显然，全体整数对于普通加法和乘法构成一个环。另外多项式和矩阵对于加法和乘法也构成一个环。在环的定义中，我们只要求加法满足交换律，构成阿贝尔群。而对乘法没有要求。如果乘法也适合交换律，我们称这样的环为交换环。在一个交换环里，对于正整数  $n$  和任何两个元都有：

$$a^n b^n = (ab)^n$$

在环的定义中，我们也没有要求环中的乘法一定要有单位元。如果  $R$  中存在一个元素  $e$  的，使得对任何元素都有：

$$ea = ae = a$$

则称  $e$  为环的单位元。一般来说，一个环未必有单位元。习惯上我们常用  $1$  来代表单位元，当然这只是一个符号，它并不是数字  $1$ 。有了单位元，自然也可以规定逆元。如果  $ab = ba = 1$ ，我们称  $b$  是  $a$  的逆元。从群的性质我们可以推知，如果一个环有单位元，这个单位元必然是唯一的。如果一个元素有逆元，这个逆元也必然是唯一的。但是并非任何元素都逆元，例如整数环有单位元，但除了  $\pm 1$  外，其它整数都没有逆元。

由于环上有结合律，所以我们有：

$$(a - a)a = aa - aa = 0$$

也就是说，如果  $a$  或  $b$  中有一个是 0，则  $ab = 0$ 。但是这一结论的逆命题却不成立。也就是说，由  $ab = 0$ ，并不能推知  $a$  或  $b$  中存在一个 0。我们来看一个反例。考虑模  $n$  的剩余类，其中加法是模  $n$  的加法： $[a] + [b] = [a + b]$ ，即相加后对  $n$  取模，其构成一个交换加群。模  $n$  的乘法定义为： $[a][b] = [ab]$ ，即相乘后对  $n$  取模。容易验证这构成一个环，称为模  $n$  的剩余类环。

如果  $n$  不是素数，例如 10，我们发现两个非 0 元素  $[5][2] = [5 \times 2] = [0]$ 。事实上，把  $n$  分解成两个因子，其模乘的结果必然是零。

在一个环里，如果  $a \neq 0, b \neq 0$ ，但却有  $ab = 0$ ，我们说  $a$  是环的一个左零因子， $b$  是环的一个右零因子。如果环是交换环，那么一个左零因子同时也是右零因子。当然，一个环可能没有零因子，例如整数环。只有在不存在零因子的环里，我们才能从  $ab = 0$  推知  $a$  或  $b$  等于 0。并且我们有如下定理：

**定理 3.2.1.** 一个没有零因子的环里，以下两个消去律都成立：

- 若  $a \neq 0, ab = ac$ , 则  $b = c$ ;
- 若  $a \neq 0, ba = ca$ , 则  $b = c$ ;

反之，如果一个环里任一个消去律成立，则另一个也必然成立，并且环中没有零因子。

这样，我们认识了一个环的三种附加条件：一是乘法满足交换律；二是单位元存在；三是不存在零因子。同时满足这三个附加条件的环有一个特殊名称，叫做整环（integral domain）。整数环显然是一个整环。

有些情况下，环的条件太强了。我们并不需要加法一定存在逆元（即加法取反，additive inverse）。如果放宽条件，我们就得到半环这种代数结构。

**定义 3.2.2.** 一个集合  $R$ ，以及其上定义的某种加法和乘法称作半环，如果它满足以下条件。

1.  $R$  上的加法构成一个交换幺半群，并且存在一个称为 0 的单位元；
2.  $R$  上的乘法构成了一个幺半群，并且存在一个称为 1 的单位元；
3. 满足分配律，任意  $a, b, c$  都有：

$$\begin{aligned} a(b+c) &= ab+ac \\ (b+c)a &= ba+ca \end{aligned}$$

4. 0 元乘以任何元都得 0。即  $a0 = 0a = 0$

自然数  $N$  就是半环的典型例子。布尔运算是一个只有两个元素的半环。

### 练习 3.8

1. 证明本节的定理，一个没有零因子的环里，两个消去律成立。
2. 证明所有形如  $a + b\sqrt{2}$ ，其中  $a, b$  是整数的实数对于普通加法和乘法构成一个整环。

## 3.2.2 除环和域

我们知道一个环里不一定所有的元素都有逆元。如果某个环中所有的元素都有逆元，就构成一种特殊的环。例如全体有理数对于普通加法和乘法来说显然是一个环，这个环中任意不等于零的元  $a$  都有逆元  $\frac{1}{a}$ 。

**定义 3.2.3.** 一个环  $R$  如果满足以下条件，则叫做除环：

1.  $R$ 至少包含一个不等于零的元；
2.  $R$ 有一个单位元；
3.  $R$ 的每个不等于零的元都有一个逆元。

**定义 3.2.4.** 一个交换除环叫做一个域。

按照这一定义，全体有理数构成一个域。同样，全体实数或者全体复数的集合对于普通加法和乘法也各自构成一个域<sup>26</sup>。除环和域有一些有趣的性质。一个除环没有零因子。这是因为，如果  $a \neq 0$ ，并且  $ab = 0$ 。我们在两边左乘  $a$  的逆元：

$$a^{-1}ab = b = 0$$

这样必然有  $b$  等于零，所以除环不含有零因子。第二个性质是，一个除环中的所有非零元，对于乘法构成一个群  $R^*$ ，我们称  $R^*$  叫做除环  $R$  的乘群。这样一个除环是由两个群：加群与乘群组合而成的；分配率好像是一座桥梁，使得两个群发生联系。到目前为止，我们介绍过的环、半环、整环、除环和域的关系总结如图 3.22。

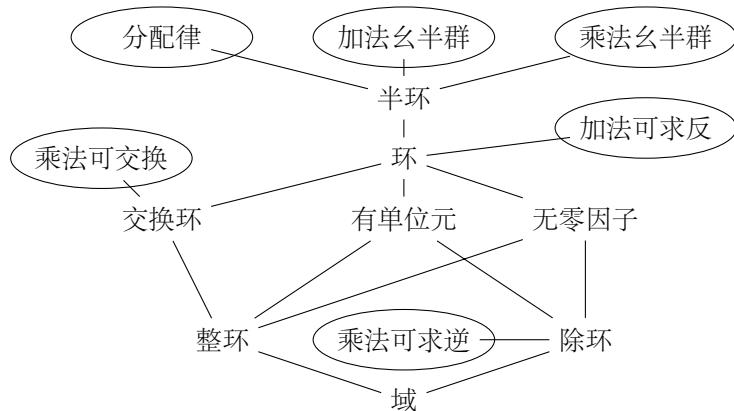


图 3.22: 半环、环、整环和域

还有一些重要的概念，如子环、理想、主理想环与欧几里得环限于篇幅我们不在这里介绍了。

### 3.3 伽罗瓦理论

我们分别介绍了域和群，而伽罗瓦理论像是一座桥梁把它们连接到了一起。域里面有加减乘除四种运算和无穷多个元素，是很复杂的对象。而伽罗瓦理论可以把域中的一些问题化简到只有一种运算的有限群里去。整个伽罗瓦理论的核心思想就是这一点。

伽罗瓦理论无疑是相当艰深难懂的。今天我们读到的伽罗瓦理论不再模糊费解，而是经过不下二十位大师的发展和处理的。其中最主要的贡献当属若当、戴德金和阿廷。若当和戴德金分别在法国和德国最早系统地整理伽罗瓦理论。今天使用的伽罗瓦群的定义是由戴德金给出的。阿廷使得伽罗瓦理论取得了现代的形式<sup>[31]</sup>。为了在这么短的一个章节介绍伽罗瓦理论，我采用了对初学者比较容易理解的一种解释<sup>[32]</sup>。对于一般的读者，可能仍然无法一遍就弄懂它。开卷有益，生活不是线性发展的。我建议大家不断回过头去重温此前介绍的一些概念，并阅读那些名著。每次也许有不同的体会，让我们一起享受那种一览众山小的感觉。

<sup>26</sup>我们在后面章节会看到，有理数域是可数的，而实数、复数域是不可数的

### 3.3.1 扩域

根据域的定义，我们知道全体有理数构成一个域，记作 $Q$ 。我们现在考虑所有形如 $a + b\sqrt{2}$ 的数组成的集合，其中 $a, b$ 都是有理数<sup>[33]</sup>。显然有理数是这一集合的子集（只要让 $b = 0$ 即可）。容易验证，任何两个这样的数做加、减、乘法都仍然可以写成 $a + b\sqrt{2}$ 的形式。除法稍微难些。观察 $x = \frac{1}{a + b\sqrt{2}}$ ，我们可以将分子分母同时乘以 $a - b\sqrt{2}$ ，这样就有：

$$\begin{aligned} x &= \frac{a - b\sqrt{2}}{(a + b\sqrt{2})(a - b\sqrt{2})} \\ &= \frac{a - b\sqrt{2}}{a^2 - 2b^2} \end{aligned}$$

令 $p = a^2 - 2b^2$ ，这样 $x$ 就可以表示成 $(a/p) - (b/p)\sqrt{2}$ 。这样除法也就验证了。的的确确这一集合构成一个域。我们将其记为 $Q[\sqrt{2}]$ 。同理 $Q[\sqrt{3}]$ 也是一个域。这就引出了扩域的概念。

**定义 3.3.1.** 如果域 $E$ 包含域 $F$ ，我们称 $E$ 是 $F$ 的扩域，记作 $F \subseteq E$ 或者 $E/F$ 。

例如实数域就是有理数域的扩域， $Q[\sqrt{2}]$ 是有理数域 $Q$ 的扩域。一般来说，如果 $F$ 是一个域， $\alpha \in F$ ，但是 $\sqrt{\alpha} \notin F$ ，则 $F_1 = F[\sqrt{\alpha}]$ 仍是一个域。这样，我们就可以重复使用这个方法不断扩张。若 $\beta \in F_1$ ，但 $\sqrt{\beta} \notin F_1$ ，则：

$$\begin{aligned} F_2 &= F_1[\sqrt{\beta}] \\ &= F[\sqrt{\alpha}][\sqrt{\beta}] \\ &= F[\sqrt{\alpha}, \sqrt{\beta}] \end{aligned}$$

即所有形如 $a + b\sqrt{\beta}$ 的数，其中 $a, b \in F_1$ 的数构成了更高层次的扩域。这样我们就可以从有理数域开始，得到一个扩域系列： $Q \subset F_1 \subset F_2 \dots \subset F_n$ 。

扩域有何意义呢？举个例子，方程 $x^2 - 2 = 0$ 在有理数域上无解，但是它在有理数域的扩域 $Q[\sqrt{2}]$ 上有一对解 $(x = \pm\sqrt{2})$ 。我们进一步再看一个例子，方程 $x^4 - 5x^2 + 6 = 0$ 在有理数域上无解，在扩域 $Q[\sqrt{2}]$ 上有两个解 $\pm\sqrt{2}$ ，在扩域 $Q[\sqrt{2}, \sqrt{3}]$ 上有全部四个解 $\pm\sqrt{2}, \pm\sqrt{3}$ 。于是就自然引出了根域的重要概念：

**定义 3.3.2.** 包含方程 $p(x) = 0$ 全部根的最小扩域叫做 $p(x)$ 的分裂域。也称为根域。

这样方程 $x^2 - 2 = 0$ 的分裂域就是 $Q[\sqrt{2}]$ 。为什么叫做“分裂”呢？多项式 $x^2 - 2$ 在有理数域 $Q$ 上是不可约的，但是在扩域 $Q[\sqrt{2}]$ 上可以“分裂”成：

$$(x + \sqrt{2})(x - \sqrt{2})$$

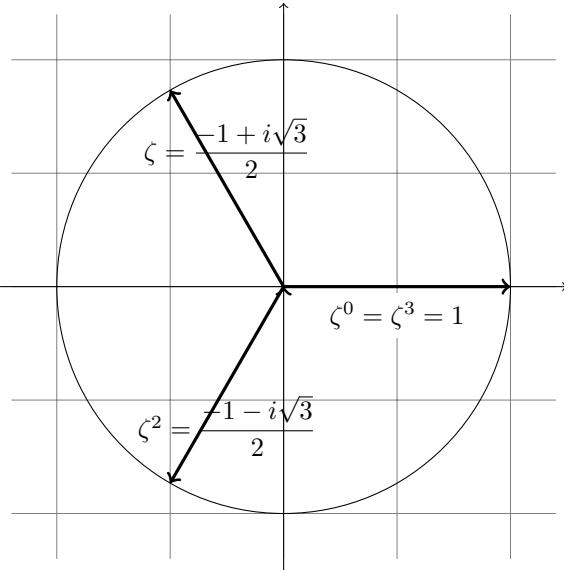
对于某一多项式方程，如果我们能够从基本域有理数域开始，通过一系列上述的扩域，最终到达分裂域，那么自然这个方程是可以用根式解的。

我们给出了平方根的例子，显然还有更加复杂的情况。3次方程可能要开3次方根，有些方程甚至还需要虚数单位 $i$ 。从高中数学我们知道，最简单的方程 $x^p - 1 = 0$ 含有 $p$ 个根 $1, \zeta, \zeta^2, \dots, \zeta^{p-1}$ ，其中 $\zeta \neq 1$ 是单位圆上的一个复根。为了涵盖这些情况，我们需要更加严格地描述扩域。

如果 $\alpha$ 的整数次方等于域 $F$ 中的某个元素 $b$ ，也就是说 $\alpha^m = b \in F$ ，这样 $\alpha$ 就可以写成 $b$ 的根式形式 $\alpha = \sqrt[m]{b}$ 。我们可以逐一用这样的一组根式进行扩域，从而得到 $F[\alpha_1][\alpha_2] \dots [\alpha_k]$ 。如果每次扩张所用的 $\alpha_i$ 都是根式，我们称扩域 $F[\alpha_1, \alpha_2, \dots, \alpha_k]$ 为 $F$ 的根式扩域（radical extension）。设方程的根是 $x_1, x_2, \dots, x_n$ （注意，这里我们没有说 $x_i$ 一定是根式），如果其分裂域 $E = Q[x_1, x_2, \dots, x_n]$ 是根式扩域，我们说方程是根式可解的。

### 练习 3.9

- 证明 $Q[a, b] = Q[a][b]$ ，其中 $Q[a, b]$ 是所有由 $a, b$ 组成的表达式如 $2ab, a + a^2b$ 等。

图 3.23: 单位根,  $x^3 - 1 = 0$ 

### 3.3.2 自同构和伽罗瓦群

到目前为止, 我们的思路是从方程系数所在的有理数域开始, 通过逐步根式扩域, 向较高层次的分裂域前进。如何进一步化简难题, 找到答案呢? 伽罗瓦在此进行逆向思维。他转而去考虑方程根之间的对称性。例如方程 $x^2 - 2 = 0$ 有一对根 $\pm\sqrt{2}$ 。显然用 $-\sqrt{2}$ 去代替 $\sqrt{2}$ , 方程仍然成立。除此之外, 将等式 $\sqrt{2}^2 + \sqrt{2} + 1 = 3 + \sqrt{2}$ 中的 $\sqrt{2}$ 替换成 $-\sqrt{2}$ 也是成立的。也就是说, 这对根 $\pm\sqrt{2}$ , 对于关系 $\alpha^2 + \alpha + 1 = 3\alpha$ 是对称的。进一步说, 对于任何仅仅对 $\sqrt{2}$ 进行乘法和加法的表达式, 这对根都可以进行互换。今天我们知道, 描述对称的最有力武器是群。具体说是置换群 (如果记不清了, 请回过头去再看一下3.1.4)。于是伽罗瓦引入了域上的自同构概念, 将扩域和群联系了起来。

我们还是用 $Q[\sqrt{2}]$ 举例子。如果我们定义一个从 $Q[\sqrt{2}]$ 到 $Q[\sqrt{2}]$ 的函数 $f : Q[\sqrt{2}] \rightarrow Q[\sqrt{2}]$ 。它的具体定义为:

$$f(a + b\sqrt{2}) = a - b\sqrt{2}$$

则 $f$ 就是一个域上的自同构。

**定义 3.3.3. 域的自同构** (Field Automorphism) 是一个可逆函数 $f$ , 它满足 $f(x + y) = f(x) + f(y)$ ,  $f(ax) = f(a)f(x)$ ,  $f(1/x) = 1/f(x)$ 。

我们可以验证, 前面那个例子 $f(a + b\sqrt{2}) = a - b\sqrt{2}$ 满足这三个条件。域的自同构背后的思想是, 我们可以重新调换域中的元素, 而完全不影响域的结构。

**定义 3.3.4.  $F$ -自同构:** 进一步, 如果 $E$ 是 $F$ 的扩域, 并且在域 $E$ 的自同构 $f$ 的基础上还满足一条额外的性质: 对 $F$ 中的任何元素 $x$ 都有 $f(x) = x$ , 则称为 $E$ 上的 $F$ -自同构。

这样就非常精确地定义了根的对称性。 $F$ -自同构对 $F$ 中的所有元素都原样保持不动, 而仅仅调换扩域 $E$ 中新元素。对域中的元素进行调换恰好是置换 (permutation) 的思想, 置换后保持不变恰好是对称的概念。对于 $Q[\sqrt{2}]$ 这个例子, 只有两个 $Q$ -自同构: 一个是恒等变换 $g(x) = x$ , 另外一个就是我们前面定义的 $f$ 。

现在我们把目前得到的结果用例子 $p(x) = x^2 - 2$ 总结一下:

1.  $p(x)$  的分裂域是  $Q[\sqrt{2}]$ ;
2.  $p(x)$  的  $Q$ -自同构代表了根的对称性, 它包含两个变换:  $f(a + b\sqrt{2}) = a - b\sqrt{2}$  和  $g(x) = x$ 。

但是仅仅扩展到分裂域并不能保证对所有的根完全对称。例如  $x^4 - 5x + 6 = 0$  的分裂域是  $Q[\sqrt{2}, \sqrt{3}]$ , 虽然我们有能力调换  $\sqrt{2}$  的自同构  $f$ , 但是不存在能够调换  $\sqrt{2}$  和  $\sqrt{3}$  的自同构。否则根据自同构的定义  $f(x) = x$ , 如果  $f(\sqrt{2}) = \sqrt{3}$ , 则  $f(\sqrt{2})^2 = \sqrt{3}^2$ , 即  $\sqrt{2} = \sqrt{3}$ 。这显然是错误的。另外, 考虑根式扩域  $Q[x_1, \dots, x_n, \sqrt{x_1}]$ , 它包含  $x_1$  的平方根, 但是不含  $x_2$  的平方根。因此不存在能够调换  $x_1$  和  $x_2$  的自同构。为了恢复对称性, 我们可以继续把  $\sqrt{x_2}, \dots, \sqrt{x_n}$  扩展到域里。

**定理 3.3.1.** 对于每个  $Q[x_1, \dots, x_n]$  的根式扩域  $E$ , 都存在一个更大的根式扩域  $E \subseteq \bar{E}$ , 使得所有  $x_1, \dots, x_n$  的置换都有自同构  $\sigma$ 。

至此, 我们已经把扩域和自同构联系了起来。如果把所有的自同构做成一个集合, 这些自同构间的二元运算是复合 (composite) 变换, 恒等变换是单位元, 我们就得到了一个群。这个群被称作伽罗瓦群。

**定义 3.3.5. 伽罗瓦群:** 对于  $F$  的扩域  $E$ , 我们有一组  $E$  的  $F$ -自同构的集合  $G$ 。对任意  $G$  中的两个  $F$ -自同构  $f, g$ , 定义二元运算  $(f \cdot g)(x) = f(g(x))$ 。我们称  $G$  为扩域  $E/F$  的伽罗瓦群。记为  $Gal(E/F)$ 。

我们还是用  $p(x) = x^2 - 2$  举例。伽罗瓦群  $G = Gal(p) = \{f, g\}$ , 包含两个元素, 一个是  $f(a + b\sqrt{2}) = a - b\sqrt{2}$ , 另一个是  $g(x) = x$ 。其中单位元是  $g$ 。我们来验证是否有  $f \cdot f = g$ :

$$\begin{aligned} (f \cdot f)(a + b\sqrt{2}) &= f(f(a + b\sqrt{2})) \quad \text{群运算的定义} \\ &= f(a - b\sqrt{2}) \quad \text{对内层括号用 } f \text{ 的定义} \\ &= a + b\sqrt{2} \quad \text{再次用 } f \text{ 的定义} \\ &= g(a + b\sqrt{2}) \quad \text{恒等变换 } g \text{ 的定义} \end{aligned}$$

这说明  $G$  的确是一个群, 并且在同构的意义下等同于二阶循环群, 它也同构于二阶对称群  $S_2$ 。这个群可以写成  $\{f, f^2\}$ , 这是因为  $f^2 = f \cdot f = 1$ , 是群的单位元。也可写成置换群的形式  $\{(1), (1 2)\}$ , 其中 (1) 是保持两个根不变, (1 2) 是对调两个根。

### 练习 3.10

1. 试证明: 对于有理数系数的任何多项式  $p(x)$ , 若  $E/Q$  是扩域,  $f$  是  $E$  上的  $Q$ -自同构, 则有  $f(p(x)) = p(f(x))$ 。
2. 考虑复数, 多项式  $p(x) = x^4 - 1$  的分裂域是什么? 它的  $Q$ -自同构中有哪些变换?
3. 尝试写出二次方程  $x^2 - bx + c = 0$  的伽罗瓦群。
4. 证明, 如果  $p$  是素数, 则方程  $x^p - 1$  的伽罗瓦群是  $q - 1$  阶的循环群  $C_{q-1}$ 。

### 3.3.3 伽罗瓦基本定理

现在我们进入了伽罗瓦理论的核心部分了。从方程系数所在的域  $F$  开始, 我们可以进行一系列扩域一直到达分裂域  $F \subset F_1 \subset F_2 \dots \subset E$ 。对应的伽罗瓦群为  $Gal(E/F)$ 。伽罗瓦发现, 这个群的所有子群和这些中间域  $F_1, F_2, \dots$  之间存在着反序的一一对应。

**定理 3.3.2. 伽罗瓦基本定理:** 令  $E/F$  为扩域<sup>27</sup>,  $G$  是对应的伽罗瓦群。这个群的子群和中间域之间存在一一对应。若  $F \subset L \subset E$ , 有  $Gal(E/L) = H$ , 则  $H$  是  $Gal(E/F)$  的子群。

<sup>27</sup>严格说, 要求是伽罗瓦扩张, 我们这里略去了伽罗瓦扩张的定义。

之所以说是反序，是因为，随着域的扩张，对应的群是减小的。扩张的起点是域 $F$ ，此时对应的群是完整的伽罗瓦群 $G = Gal(E/F)$ ；扩张的终点是分裂域 $E$ ，此时的群只有一个元素，就是恒等变换 $\{1\}$ 。扩张的中间域 $L$ ，对应的群是 $H = Gal(E/L)$ ，它是子群，有 $Gal(E/L) \subset Gal(E/F)$ 。并且如果 $H$ 是正规子群（如果记不清了，可回去看一下3.1.7），则它的商群 $G/H = Gal(L/F)$ 。

我们举一个具体的例子[34]（页码）。考虑方程 $x^4 - 8x^2 + 15 = 0$ ，它也可以表示成 $(x^2 - 3)(x^2 - 5) = 0$ 。它的系数域是有理数域 $Q$ ，方程的分裂域是 $E = Q[\sqrt{3}, \sqrt{5}]$ 。它的伽罗瓦群 $Gal(E/Q)$ 的阶是4，同构于一个4阶循环群。可以找到3个中间扩域分别是 $Q[\sqrt{3}]$ ,  $Q[\sqrt{5}]$ 和 $Q[\sqrt{15}]$ ，这3个中间扩域对应的子群的阶都是2。而方程的伽罗瓦群，也就是4阶循环群只有一个2阶子群。除此之外，它不再有其它非平凡子群了。根据伽罗瓦基本定理，我们知道，除了那3个中间扩域之外，再也没有其它中间扩域了。换个角度看，分裂域上的任何元素都可以表示为 $\alpha = a + b\sqrt{3} + c\sqrt{5} + d\sqrt{15}$ ，其中 $a, b, c, d$ 是有理数；对于3个中间扩域上的任何元素， $b, c, d$ 中至少有两个为0。

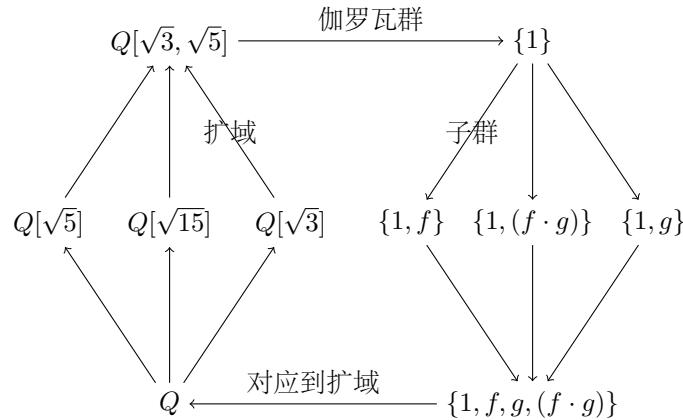


图 3.24: 伽罗瓦对应

如图3.24所示，分裂域上的任意元可以写成如下形式：

$$\begin{aligned}\alpha &= (a + b\sqrt{3}) + (c + d\sqrt{3})\sqrt{5} \\ &= a + b\sqrt{3} + c\sqrt{5} + d\sqrt{15}\end{aligned}$$

其中 $a, b, c, d$ 都是有理数。我们定义如下自同构：

变换 $f$ 将 $\sqrt{3}$ 反号：

$$\begin{aligned}f((a + b\sqrt{3}) + (c + d\sqrt{3})\sqrt{5}) &= (a - b\sqrt{3}) + (c - d\sqrt{3})\sqrt{5} \\ &= a - b\sqrt{3} + c\sqrt{5} - d\sqrt{15}\end{aligned}$$

变换 $g$ 将 $\sqrt{5}$ 反号

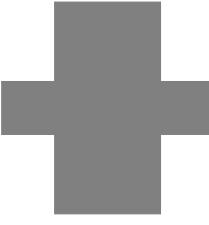
$$\begin{aligned}g((a + b\sqrt{3}) + (c + d\sqrt{3})\sqrt{5}) &= (a + b\sqrt{3}) - (c + d\sqrt{3})\sqrt{5} \\ &= a + b\sqrt{3} - c\sqrt{5} - d\sqrt{15}\end{aligned}$$

复合变换 $f \cdot g$ 同时将 $\sqrt{3}$ 和 $\sqrt{5}$ 反号

$$\begin{aligned}(f \cdot g)((a + b\sqrt{3}) + (c + d\sqrt{3})\sqrt{5}) &= (a - b\sqrt{3}) - (c - d\sqrt{3})\sqrt{5} \\ &= a - b\sqrt{3} - c\sqrt{5} + d\sqrt{15}\end{aligned}$$

再加上恒等变换1，基本域 $Q$ 上的伽罗瓦群一共有4个元素： $G = \{1, f, g, (f \cdot g)\}$ 。这个群同构于克莱因群，而克莱因群相当于是两个2阶循环群的积。它共有5个子群，每个都对应着一个扩域：

- 只含有单位元{1}的子群，对应于分裂域 $Q[\sqrt{3}, \sqrt{5}]$ ；
- $G$ 自身，对应于有理数域 $Q$ ；
- 二阶子群 $\{1, f\}$ ，对应于扩域 $Q[\sqrt{5}]$ ， $f$ 只反转 $\sqrt{3}$ 而固定 $\sqrt{5}$ 不变；
- 二阶子群 $\{1, g\}$ ，对应于扩域 $Q[\sqrt{3}]$ ， $f$ 只反转 $\sqrt{5}$ 而固定 $\sqrt{3}$ 不变；
- 二阶子群 $\{1, (f \cdot g)\}$ ，对应于扩域 $Q[\sqrt{15}]$ ， $(f \cdot g)$ 同时反转 $\sqrt{5}$ 和 $\sqrt{3}$ ，而固定 $\sqrt{15}$ 不变。



克莱因群的对称性。左右翻转，上下翻转，或者同时上下左右翻转都是对称的。

我们也可以把这个方程的伽罗瓦群写成置换群的形式： $\{(1), (12), (34), (12)(34)\}$ 。其中置换(1)表示保持4个根都不变；(1 2)表示交换 $\pm\sqrt{3}$ 这两个根；(3 4)表示交换 $\pm\sqrt{5}$ 这两个根；(1 2)(3 4)表示同时交换这两对根。

通过伽罗瓦基本定理，我们已经成功地将方程在域上的问题，转换为对称群的问题了。接下来要做的就是最后一击，通过群揭示可解性的本质。

### 3.3.4 可解性

通过伽罗瓦对应，我们将根式扩域的结构和群的结构联系起来了。为了简化问题，我们增加一些额外的限制。首先，我们假设每次根式扩域 $F[\alpha_i]$ 中加入的 $\alpha_i$ 都是素数次方根。例如对 $\sqrt[p]{\alpha}$ ，我们把它拆成 $\sqrt[p]{\alpha} = \beta$ 和 $\sqrt[p]{\beta}$ ，然后分两次进行根式扩域。其次，如果 $\alpha_i$ 是 $p$ 次方根，若扩域 $F[\alpha_1, \dots, \alpha_{i-1}]$ 中不含有 $p$ 次单位根，我们可以认为 $F[\alpha_1, \dots, \alpha_i]$ 中也不含有 $p$ 次单位根。除非 $\alpha_i$ 本身就是 $p$ 次单位根。举个例子，假设要加入 $\alpha = \sqrt[3]{2}$ 进行根式扩域，如果上一个域 $F$ 中不含有如图3.23中的 $\zeta = \frac{-1 + i\sqrt{3}}{2}$ ，我们可以认为扩域 $F[\sqrt[3]{2}]$ 中也不含有 $\zeta$ 。除非 $\alpha^3 = 1$ 即 $\alpha$ 本身就等于 $\zeta$ 。如果这一点不成立，我们可以在加入 $\alpha_i$ 之前先用 $\zeta$ 进行扩域，这样域 $F[\alpha_1, \dots, \alpha_{i-1}, \zeta]$ 中就包含了所有 $p$ 次单位根： $1, \zeta, \zeta^2, \dots, \zeta^{p-1}$ 。以上这两个对根式扩域的修改并不会改变最终扩域的结果 $F[\alpha_1, \dots, \alpha_k]$ 。

这样，任何根式扩域 $F[\alpha_1, \dots, \alpha_k]$ 都组成一系列递增的链条：

$$F = F_0 \subseteq F_1 \subseteq F_2 \subseteq \dots \subseteq F_k = F[\alpha_1, \dots, \alpha_k]$$

其中每个 $F_i = F_{i-1}[\alpha_i]$ ，而 $\alpha_i$ 是 $F_{i-1}$ 中某个元素的素数次方根。并且满足上面关于单位根的条件。和这一系列扩域链对应的，是一系列递减的子群链：

$$\text{Gal}(F_k/F_0) = G_0 \supseteq G_1 \supseteq \dots \supseteq G_k = \text{Gal}(F_k/F_k) = \{1\}$$

其中 $G_i = \text{Gal}(F_k/F_i) = \text{Gal}(F_k/F_{i-1}[\alpha_i])$ 。每一步从 $G_{i-1}$ 前进到它的子群 $G_i$ ，都反应了向域 $F$ 中扩张素数次方根 $\alpha_i$ 。用群论的术语说， $G_i$ 是前一个群 $G_{i-1}$ 的正规子群，而商群 $G_{i-1}/G_i$ 是阿贝尔群（可交换的）。我们下面来证明这点。

**定理 3.3.3.** 如果扩域 $B \subseteq E[\alpha] \subseteq E$ 中， $\alpha^p \in B$ ， $p$ 是素数。并且满足上面的单位根的条件，则 $\text{Gal}(E/B[\alpha])$ 是群 $\text{Gal}(E/B)$ 的正规子群，并且商群

$$\text{Gal}(E/B)/\text{Gal}(E/B[\alpha])$$

是阿贝尔群。

这一段的证明较难，一般的读者可以跳过框中的内容

证明。我们要利用在3.1.7中介绍的群同态定理来完成这一证明。我们要找到一个 $Gal(E/B)$ 上的同态映射，它的核是 $Gal(E/B[\alpha])$ ，它映射到一个阿贝尔群上。以 $Gal(E/B[\alpha])$ 为核的映射显然限制在域 $B[\alpha]$ 上，我们将这一映射记为 $|_{B[\alpha]}$ ，根据伽罗瓦群的元素是自同构的定义：

$$\sigma \in Gal(E/B[\alpha]) \iff \sigma|_{B[\alpha]} \text{是恒等变换}$$

伽罗瓦群 $Gal(E/B)$ 中所有这样的自同构 $\sigma$ 满足同态映射的性质：

$$\sigma' \sigma|_{B[\alpha]} = \sigma'|_{B[\alpha]} \sigma|_{B[\alpha]}$$

由于 $\sigma$ 固定所有 $B$ 中的元素不变，所以 $\sigma|_{B[\alpha]}$ 完全由 $\sigma(\alpha)$ 决定。这里有两种情况。第一种情况是 $\alpha$ 恰好等于 $p$ 次单位根 $\zeta$ 。此时：

$$(\sigma(\alpha))^p = \sigma(\alpha^p) = \sigma(\zeta^p) = \sigma(1) = 1$$

所以 $\sigma(\alpha) = \zeta^i \in B[\alpha]$ ，这是因为每个 $p$ 次单位根都是 $\zeta^i$ 的形式。第二种情况下， $\alpha$ 不是 $p$ 次单位根，此时：

$$(\sigma(\alpha))^p = \sigma(\alpha^p) = \alpha^p$$

根据根式扩域的规则， $\alpha^p$ 是 $B$ 中的元素，所以 $\sigma(\alpha) = \zeta^j \alpha$ 。其中 $\zeta$ 是 $p$ 次单位根，并且 $\zeta \in B$ 。所以也有 $\sigma(\alpha) \in B[\alpha]$ 。两种情况合并到一起，说明映射在每个 $\sigma$ 下都是封闭的。

这就说明了 $|_{B[\alpha]}$ 将伽罗瓦群 $Gal(E/B)$ 映射到 $Gal(E/B[\alpha])$ 。接下来我们要证明群 $Gal(B[\alpha]/B)$ 是可交换的。我们仍然分两种情况来看：第一种情况 $\alpha$ 是单位根，我们看到每个群中的元素，也就是自同构 $\sigma_i = \sigma|_{B[\alpha]} \in Gal(B[\alpha]/B)$ 都可以写成 $\sigma_i(\alpha) = \alpha^i$ 的形式，所以：

$$\sigma_i \sigma_j(\alpha) = \sigma_i(\alpha^j) = \alpha^{ij} = \sigma_j \sigma_i(\alpha)$$

第二种情况 $\alpha$ 不是单位根，这样每个群中的元素，也就是自同构 $\sigma_i$ 都可以写成 $\sigma_i(\alpha) = \zeta^i \alpha$ 的形式，所以：

$$\sigma_i \sigma_j(\alpha) = \sigma_i(\zeta^j \alpha) = \zeta^{i+j} \alpha = \sigma_j \sigma_i(\alpha)$$

由于 $\zeta \in B$ ，所以 $\zeta$ 在自同构下固定不变。这样两种情况下，我们都证明了交换性。  $\square$

这样回过头来看伽罗瓦群的子群链：

$$Gal(F[\alpha_1, \dots, \alpha_k]/F) = G_0 \supseteq G_1 \supseteq \dots \supseteq G_k = \{1\}$$

如果这链条中，每个群 $G_i$ 都是上一个群 $G_{i-1}$ 的正规子群，并且每个商群 $G_{i-1}/G_i$ 都是可交换的，我们称伽罗瓦群 $Gal(F[\alpha_1, \dots, \alpha_k]/F)$ 是可解的。

现在我们来看一般的 $n$ 次方程。

$$x^n - a_1 x^{n-1} + a_2 x^{n-2} \dots \pm a_n = 0$$

包含系数的基本域为 $Q[a_1, \dots, a_n]$ 。如果它的 $n$ 个根（不一定为根式）是 $x_1, \dots, x_n$ ，即

$$(x - x_1) \dots (x - x_n) = x^n - a_1 x^{n-1} \dots \pm a_n$$

方程的根域为 $Q[x_1, \dots, x_n]$ 。如果无论怎样对基本域做根式扩域，都无法包含方程的根域，我们就说方程是无法用根式求解的。

**定理 3.3.4.** 当 $n \geq 5$ 时， $Q[a_1, \dots, a_n]$ 的根式扩域不包含根域 $Q[x_1, \dots, x_n]$ 。

证明. 我们用反证法, 假设某个根式扩域 $E$ 包含根域 $Q[x_1, \dots, x_n]$ , 根据此前的定理, 还存在更大个根式扩域 $\bar{E} \supseteq E$ , 使得对应的伽罗瓦群 $G_0 = Gal(\bar{E}/Q[a_1, \dots, a_n])$ 包括所有根的置换的自同构。这样我们就可以构造一个子群的链条:

$$G_0 \supseteq G_1 \supseteq \dots \supseteq G_k = \{1\}$$

每个 $G_i$ 都是上一个群 $G_{i-1}$ 的正规子群, 并且商群 $G_{i-1}/G_i$ 是可交换的。我们要证明这是不可能的。正规子群 $G_i$ 是同态映射下 $G_{i-1}$ 中的核, 所以任意 $G_{i-1}$ 中的两个自同构 $\sigma, \tau$ 都有:

$$\sigma^{-1}\tau^{-1}\sigma\tau \in G_i$$

当 $n$ 大于等于5时,  $G_0$ 包含所有根的置换, 同构于对称群 $S_n$ , 它必然包含3循环的置换 $(a\ b\ c)$ 。假设 $G_{i-1}$ 中也包含3循环置换, 我们有:

$$(a\ b\ c) = (d\ a\ c)^{-1}(c\ e\ b)^{-1}(d\ a\ c)(c\ e\ b) \in G_i$$

这说明 $G_i$ 也包含3循环置换。根据数学归纳法, 子群链条中所有的群都包含3循环置换。但这是不可能的, 因为最后一个群是 $\{1\}$ , 它不含3循环置换。□



对称群 $S_5$ 是交错群 $A_5$ 与二阶循环群的积。可以描述一个足球的对称

这样我们就证明了一般5次方程是不能用根式解的。我们也可以从另一个角度做出解释。对称群 $S_5$ 有120个元素, 它的唯一正规子群是 $A_5$ , 其中 $A_5$ 叫做交错群, 有60个元素。 $A_5$ 的唯一正规子群是 $\{1\}$ 。这一子群链是 $S_5 \supset A_5 \supset \{1\}$ 。但是商群 $A_5/\{1\}$ 不是阿贝尔群(不可交换)。因此它不是可解群。这样它对应的一般5次方程不是根式可解的。

一般4次方程, 对称群 $S_4$ 的正规子群是 $A_4$ , 交错群 $A_4$ 有一个正规子群

$$\{(1), (12)(34), (13)(24), (14)(23)\}$$

而这个群和克莱因群同构, 所以是可解的。

对于三次方程, 对称群 $S_3$ 的正规子群 $A_3$ , 交错群 $A_3$ 同构于3阶循环群。所以也是可解的。

至此我们简单介绍了伽罗瓦理论。伽罗瓦理论是抽象代数中的强大工具, 它可以解决很多问题, 如证明代数基本定理、证明当年高斯用尺规作图画出的正17边形是可行的。证明无法用尺规作图完成古希腊三大作图问题——圆化方、三分角、倍立方等等。我们希望读者能通过这一章, 领略到抽象思维的深邃和优美。

### 练习 3.11

- 考虑5次方程 $x^5 - 1 = 0$ , 它是根式可解的。它的伽罗瓦群和对应的子群链是什么?

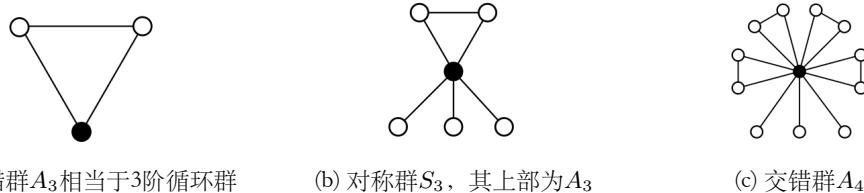


图 3.27: 对称群和交错群, 黑点是单位元, 循环用闭环表示

### 3.4 扩展阅读

对于抽象代数的基本概念, 张禾瑞先生的《近世代数基础》[22]是本入门的教材。这本书不到200页, 提纲挈领地减少了群、环、域的基本概念。阿姆斯特朗的《群与对称》[23]同样也是小薄本, 对于对称性的讲解是其特色。迈克尔·阿廷的《代数》是一本广泛使用的教材, 最后一章讲解了伽罗瓦理论。爱德华兹的《伽罗瓦理论》, 以历史的发展为主要脉络, 用古典的方法来讲述。并且还把伽罗瓦手稿的英文译文置于附录, 使读者能了解当初是怎样一步一步解决问题的。章璞先生的《伽罗瓦理论: 天才的激情》[31]则用现代的方法讲解伽罗瓦理论。埃米尔·阿廷的《伽罗瓦理论》是这个领域中的经典, 有李同孚先生翻译的中译本。韩雪涛的《好的数学》系列中有很多数学家的传记故事, 生动有趣。

## 第4章 范畴论

数学是赋予不同事物相同名字的艺术。

——昂利·庞加莱

欢迎来到范畴世界！我建议你小小的奖励自己一下。你已经迈过了第一道门槛，正在通往神奇的抽象王国之路上。这条路是世界上许多最聪明的心智披荆斩棘开辟出来的。如果说，人们将具体的事物，抽象成不带具体意义的数与形是原始阶段；将数、形与计算的意义去除，抽象成代数结构（例如群）和代数关系（例如同构）是第一阶段；范畴可以算是抽象的第二阶段。

你也许会问，什么是范畴？为什么要了解范畴？这和编程有什么关系？范畴是数学家在1940年代研究同调代数时的“副产品”——顺手发明的理论。最近十几年，范畴论由于其强大的抽象，几乎普适于任何问题，正向各种各样的领域渗透。不要说各大编程语言纷纷引入lambda演算和闭包等结构，有超过20种语言已经实现了单子（Monad）<sup>[36]</sup>——用范畴的语言说，叫做“函子范畴上的幺半群”。

更重要的是：我们需要抽象。赫尔曼·外尔说现代数学在过去几十年不断沉湎在抽象和形式化上。编程领域何尝不是如此呢？现代计算机科学解决的问题空前复杂，大数据量、分布式、高并发、还要保证数据和计算的安全。仅仅靠着前几十年的传统方法——暴力求解、务实的工程实践再加上一点聪明的头脑已经不够了。这逼迫着我们去吸取其它科学和数学中的新方法和新工具。

正如迪厄多内所说：“这种抽象绝不是来自数学家的反常意愿，似乎他们想通过使用深奥莫测的语言来把自己与其他人隔开。数学家是被经典对象和关系的本质特性逼着去锻造新的抽象工具，来解决过去看来是不可攻克的问题。”<sup>[35]</sup>（中文版第2页，英文版第9页）

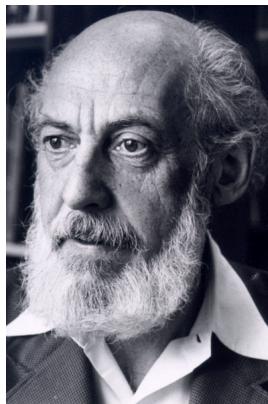
范畴论是数学家艾伦伯格和麦克兰恩在1940年代创立的。

塞缪尔·艾伦伯格于1913年生于波兰华沙的一个犹太人家庭。他于1936年在华沙大学获得博士学位。艾伦伯格主要研究代数拓扑。他与诺曼·斯廷罗德（Norman Steenrod）一起对同调理论进行了公理化。艾伦伯格在与麦克兰恩合作研究同调代数的过程中一起创立了范畴论。艾伦伯格也是著名的布尔巴基小组成员。他与昂利·嘉当合作，在1956年完成了经典著作《同调代数》。艾伦伯格后来移居美国，在纽约哥伦比亚大学做教授，他的主要工作是发展纯粹的范畴论。他是该领域的奠基者之一。1986年他获得沃尔夫奖。1998年艾伦伯格逝世于美国纽约市。

艾伦伯格还是著名的亚洲艺术品收藏家。他收集了来自印度、印度尼西亚、尼泊尔、泰国、柬埔寨、斯里兰卡和中亚的雕塑和艺术品。1992年，他把自己收藏的400多件艺术品捐赠给了纽约大都会博物馆<sup>[37]</sup>。



艾舍尔《露珠》1948



艾伦伯格 (Samuel Eilenberg,  
1913 - 1998)



麦克兰恩 (Saunders Mac  
Lane, 1909 - 2005)

桑德斯·麦克兰恩1909年生于美国康涅狄格州的诺维奇市。麦克兰恩受洗时的名字是“雷斯利·桑德斯·麦克兰恩”。但是他的父母不喜欢这个名字，所以来后就把雷斯利去掉了。麦克兰恩名字的英文本来是MacLane，他的妻子在打字时总是习惯加上一个空格变成Mac Lane，索性后来麦克兰恩就将错就错了。

麦克兰恩在高中时最喜欢化学。他的父亲在这时去世了，只好由祖父来照顾他。1926年，麦克兰恩的一个远房叔叔资助他到耶鲁学习。他的数学老师希尔带领他参加数学竞赛，并且一路过关斩将获得优胜。从此麦克兰恩下定了从事数学的决心。1930年，麦克兰恩从耶鲁毕业，获得了数学和物理学的双学位。毕业前一年，有一次在新泽西召开耶鲁橄榄球队的球迷聚会。麦克兰恩在会上被授予耶鲁优秀毕业成绩奖[38]。恰好芝加哥大学的新任校长哈钦斯也在这次聚会中，他鼓励麦克兰恩到芝加哥大学深造。麦克兰恩于是来到了他后来毕生工作的芝加哥大学<sup>1</sup>。1931年他获得了硕士学位，接着获得了前往世界数学圣地——哥廷根大学进修的机会。

麦克兰恩幸运的成为了最后一批前往哥廷根的美国人，不久纳粹德国就开始禁止美国人前来学习。

在哥廷根，麦克兰恩师从大数学家保罗·伯奈斯、埃米·诺特、赫尔曼·外尔。在他即将获得博士学位的前夕，导师伯奈斯因为是犹太人，被纳粹政府赶出了校园，于是只好由外尔接替伯奈斯。1934年麦克兰恩获得了哥廷根数学研究所的博士学位，并返回美国<sup>2</sup>。

在1944年与1945年，麦克兰恩领导了在第二次世界大战期间有卓越贡献的哥伦比亚大学应用数学小组。麦克兰恩曾任美国科学院与美国哲学会的副主席，美国数学会的主席。在领导美国数学会的期间，他提倡对现代数学教学改进的研习活动。而在1974年至1980年间，他担任了美国政府的科学顾问。1976年，以他为首的美国数学家访问团访问了中国，考察了当时中国数学学术发展。麦克兰恩于1949年获选为美国科学院院士，并在1989年获得美国国家科学奖章。

麦克兰恩的早期研究方向为域论与赋值论。1941年，麦克兰恩在访问密歇根大学时遇到了艾伦伯格。两人开始在代数和拓扑学上进行合作，并结出了累累硕果。1943年，他与艾伦伯格在研究同调代数时一起创立了范畴论。

2005年，麦克兰恩逝世于美国的旧金山。

<sup>1</sup>这里有一段小插曲，聚会后不久，哈钦斯就答应给麦克兰恩一笔奖学金。但是麦克兰恩竟然忘记申请研究生课程就直接去了芝加哥大学。当然最后他被获准入学。

<sup>2</sup>取得学位后，麦克兰恩与来自芝加哥的多罗茜·琼斯结婚，不久后，他们夫妻一起返回了芝加哥。

## 4.1 范畴

让我们用非洲大草原来了解范畴。在大草原这个范畴中，有很多种动植物。狮子、鬣狗、猎豹、羚羊、水牛、斑马、秃鹫、蜥蜴、眼镜蛇、蚂蚁……我们管这些动物叫做对象<sup>3</sup>，每种动物和植物都是一个对象。这些动植物之间会产生联系，例如狮子会吃羚羊，羚羊吃掉草。我们很容易构造一个食物链。可以从羚羊画一个箭头指向狮子，从草画一个箭头指向羚羊来表示这样的关系。这样对象加上箭头就构成了一个有结构的，彼此联系在一起的系统。

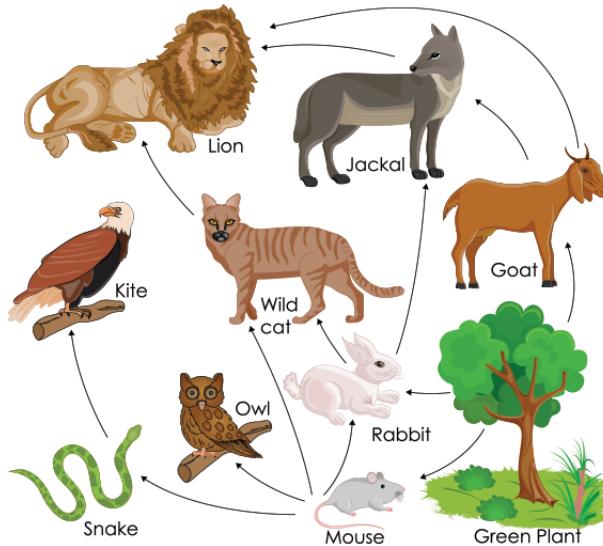
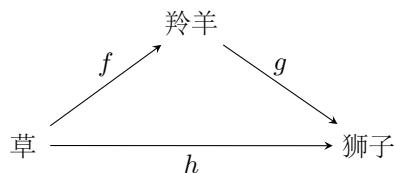


图 4.4: 动植物对象与食物关系箭头组成了一个有结构的系统

这样的系统要想成为范畴，还必须满足两点。第一点是每个对象都有指向自己的箭头。这种特殊的箭头称为恒等箭头。对于草原上的动植物，从羚羊到狮子箭头说明羚羊在狮子食物链的下游。我们可以认为每种动植物都处于自己物种食物链的下游（别误会，并不是指吃同类，而是指不吃同类）。这样每个物种就都有自指的箭头了。第二点是箭头之间是可组合的。什么叫做组合呢？草有一个指向羚羊的箭头  $f$ ，羚羊有一个指向狮子的箭头  $g$ 。我们即可以把这两个箭头组合起来成为  $g \circ f$ ，表示草处于狮子食物链的下游。进一步，把两个箭头组合起来后，还可以和第三个箭头再次组合。例如狮子死后会被秃鹫吃掉，这样我们可以从狮子画一个指向秃鹫的箭头  $h$ 。组合在一起就是  $h \circ (g \circ f)$ 。这个箭头的含义表示草处于秃鹫食物链的下游，它等价于  $(h \circ g) \circ f$ 。也就是说，食物链的上下游关系是满足结合性的。

在组合的基础上，还有一个叫作“可交换”的概念。例如下面图中，



<sup>3</sup>和编程中的“面向对象”无关。这里指抽象事物。

可以沿着两条通路从草到达狮子，一条是箭头组合 $g \circ f$ ，表示草在羚羊的下游，羚羊又在狮子的下游；另一条是箭头 $h$ ，表示草在狮子的下游。这样我们就相当于得到了一对平行的箭头：

$$\begin{array}{ccc} & g \circ f & \\ \text{草} & \xrightarrow{\hspace{2cm}} & \text{狮子} \\ & h & \end{array}$$

如果这两个平行的箭头是等效的，我们称它们是可交换的。用符号表示就是：

$$h = g \circ f$$

这样，我们说草原上的动植物，在食物链关系这个箭头下，构成了一个范畴。现在我们给出较为正式的范畴的定义。

**定义 4.1.1.** 一个范畴 $\mathbf{C}$ 包括一组对象 (Object)<sup>4</sup>，记为 $A, B, C, \dots$ ，和一组箭头 (Arrow)，记为 $f, g, h, \dots$ 。它们之上定义了以下四种操作：

- 两个全操作<sup>5</sup>，称为源 (source) 和目标 (target)<sup>6</sup>，这两个操作都将对象指定到箭头上，记为 $A \xrightarrow{f} B$ ，表示箭头 $f$ 的源是 $A$ ，目标是 $B$ ；
- 第三个全操作叫恒等箭头<sup>7</sup>，对于任何对象 $A$ ，恒等箭头都指向 $A$ 自己。记为： $A \xrightarrow{id_A} A$ ；
- 第四个操作是一个部分操作，称为组合。它将两个箭头组合起来。如果有箭头 $B \xrightarrow{f} C$ 和 $A \xrightarrow{g} B$ ，则 $f$ 和 $g$ 的组合为 $f \circ g$ ，读作“ $g$ 然后 $f$ ”。它表示 $A \xrightarrow{f \circ g} C$ 。

除此之外，范畴还必须满足以下两条公理：

- **结合性公理**：箭头组合是可结合的。对任何三个可组合的箭头 $f, g, h$ 都有：

$$f \circ (g \circ h) = (f \circ g) \circ h$$

我们可以将其写为 $fg h$ 。

- **单位元公理**：恒等箭头对于组合运算来说相当于单位元。对于任何箭头 $A \xrightarrow{f} B$ 来说，都有：

$$f \circ id_A = f = id_B \circ f$$

和大草原上的食物链比起来，范畴和箭头的定义是很抽象的。我们可以通过一些较为具体的例子，来加深对这个定义的理解。

<sup>4</sup>和编程中的“面向对象”无关。这里指抽象事物。

<sup>5</sup>全操作 (total operation) 是指对于所有的对象，无一例外都存在这一操作。与之相对的是部分操作 (partial operation)，对于某些对象，这一操作没有定义。例如对于全体整数的集合，取相反数 $x \mapsto -x$ 是全操作，而取倒数 $x \mapsto 1/x$ 由于对0没有定义，所以是部分操作。

<sup>6</sup>不应把源和目标理解为名词，而应理解为动词。表示“指定源为……”，“指定目标为……”

<sup>7</sup>同样这里恒等箭头 (identity arrow) 应理解为动词，表示“为……指定恒等箭头”。

### 4.1.1 范畴的例子

在数学上，如果一个集合带有特殊的单位元，元素间定义了可结合的二元运算。我们说这样的集合构成一个幺半群。例如全体整数中，令0是单位元，二元运算是加法，则构成了整数加法幺半群。

幺半群的集合不仅可以包含数，也可以是其它东西。例如英文的词语和句子（在编程中称为字符串）构成一个集合。如果我们把两个英文词句连在一起称为二元运算，例如“red”+“apple”=“red apple”，则这些英文词句构成一个幺半群。其中单位元是空词句“”。不难验证它满足幺半群的条件：

$$\text{red} \text{ ++ } (\text{apple} \text{ ++ } \text{tree}) = (\text{red} \text{ ++ } \text{apple}) \text{ ++ } \text{tree}$$

词句连接满足结合律。

$$\text{""} \text{ ++ } \text{apple} = \text{apple} = \text{apple} \text{ ++ } \text{"”}$$

任何词句连接上空词句（单位元）都仍然等于它本身。

把词句幺半群放在一边，再考虑另一个幺半群。群元素是字母组成的集合（在编程中称为字符集），二元操作是集合的并，单位元是空集。将两个字母集合并在一起可以获得一个更大的集合，例如：

$$\{a, b, c, 1, 2\} \cup \{X, Y, Z, 0, 1\} = \{a, b, c, X, Y, Z, 0, 1, 2\}$$

再加上最初的例子，整数加法幺半群，我们就有三个幺半群了。接下来我们建立这三个幺半群之间的转换关系<sup>8</sup>，首先是从词句幺半群到字母集合幺半群之间的转换关系。任给一个英文词句，我们可以把这句英文中用到的所有不同字母组成一个集合。可以把这个操作叫做“取字母”。可以验证，取字母满足以下条件：

$$\text{取字母}(\text{red} \text{ ++ } \text{apple}) = \text{取字母}(\text{red}) \cup \text{取字母}(\text{apple}) \quad \text{和} \quad \text{取字母}(\text{""}) = \emptyset$$

也就是说，两句英文字词连在一起所涵盖的字母，等于各个字词中字母的并；空串不含有任何字母（空集）。

接下来再定义从字母集合到整数加法幺半群之间的转换关系。任给一个字母集合，我们可以数出它含有多少字母，空集含有0个字母。我们把这个操作叫做“数个数”。

这样就有了两个变换：“取字母”和“数个数”，我们检查一下它们的组合：

$$\text{词句} \xrightarrow{\text{取字母}} \text{字母集合} \xrightarrow{\text{数个数}} \text{整数}$$

和

$$\text{词句} \xrightarrow{\text{数个数} \circ \text{取字母}} \text{整数}$$

显然组合后“数个数◦取字母”也是一个转换关系。它表示先从英文词句中抽取出不同的字母，然后再统计不同字母的个数。这样，我们就得到了**幺半群范畴Mon**。其中对象是各种各样的幺半群，箭头是彼此间的转换关系，数学上叫做态射。恒等箭头从一个幺半群指向它自己。

这是一个非常巨大的范畴，它包含宇宙中**所有的**幺半群<sup>9</sup>。另一方面，范畴也可以很小，我们接下来看的例子只含有一个幺半群。考虑英文词句这个幺半群，它只有一个对象，就是全体英文的集合（请原谅，这看起来也不怎么小）。对于任何一个英文词句，例如hello，我们都可以定义一个前缀操作，把hello添加到任何其它英文词句前面。我们管这个操作叫“加前缀hello”。例如：“加前缀hello”作用到单词Alice上就变成了helloAlice；类似地还可以定义“加前缀hi”，把它作用到Alice上就得到了hiAlice。如果认为这是两个不同的箭头，则它们的组合就是：

<sup>8</sup>数学上叫做态射（morphism）

<sup>9</sup>你可能想到了罗素悖论。确切地说，Mon包含了宇宙中所有“小”的幺半群

$$\text{加前缀hello} \circ \text{加前缀hi} = \text{加前缀hellohi}$$

表示先增加前缀hi，然后再增加前缀hello。不难验证，任意三个这样的箭头满足结合律。接下来我们还需要检查一下单位元，也就是空词句“”。以空词句作为前缀等于没有变化，也就是恒等变换。这样我们就得到了只有一个对象的么半群范畴，如图4.5所示。

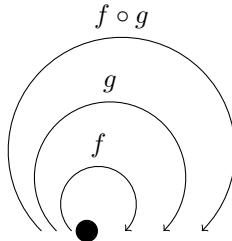


图 4.5: 只有一个对象的么半群范畴

同样是么半群，我们既看到了包含宇宙中全部么半群的范畴**Mon**，也看到了只包含一个么半群的范畴。可谓一沙一世界，一花一叶一如来。更有意思的是，对于任何范畴**C**，和其中任何的对象 $A$ ，定义集合 $\text{hom}(A, A)$ 为所有从 $A$ 指向 $A$ 的箭头。则这一箭头的集合在组合运算下又构成一个么半群，其中单位元是恒等箭头。这种对偶值得我们仔细体会。

我们举的第二个例子是集合。我们让每个集合都成为一个对象<sup>10</sup>，而箭头是从一个集合 $A$ 到另一个集合 $B$ 的函数（或映射）。我们称 $A$ 为函数的定义域， $B$ 为值域<sup>11</sup>。组合运算就是函数的组合。即 $y = f(x)$ 与 $z = g(y)$ 的组合为 $z = (g \circ f)(x) = g(f(x))$ 。不难验证，函数的组合满足结合律，单位元是恒等函数 $\text{id}(x) = x$ 。这样我们就获得了全体集合和函数组成的范畴**Set**。

我们举的第三个例子包括一对概念。分别叫做**偏序集**和**预序集**。给定一个集合，所谓预序（pre-order），是说集合中的两个元素之间可以进行比较。我们用二元关系符号 $\leq$ 来表示，这个符号不一定是大于小于关系，它可能表示一个集合是另一个集合的子集，一个词句是另一个词句的后缀，一个人是另一个人的后代等等。如果关系 $\leq$ 满足以下两条性质，我们说这是一个预序关系：

- **自反性**: 任何集合中的元素 $a$ 都有 $a \leq a$ ;
- **传递性**: 若 $a \leq b$ 且 $b \leq c$ ，则 $a \leq c$ ;

如果在此基础上还满足反对称性，则这一关系叫做偏序（partial order）：

- **反对称性**: 若 $a \leq b$ 且 $b \leq a$ ，则 $a = b$ ;

我们称满足预序关系的集合叫做预序集，满足偏序关系的集合叫做偏序集，分别记作：

*preset*      *poset*

在偏序集中，并非任何两个元素都能够进行比较。例如，《红楼梦》中贾家按照祖先关系构成一个偏序集。如果4.6所示。可以看到巧姐 $\leq$ 贾琏，但是贾宝玉和贾探春之间，贾迎春和贾惜春之间无法用 $\leq$ 进行比较。在这棵家族树中，尽管任何人都有祖先（位于树根的人根据自反性，可以认为自己是自己的祖先），但是平辈的人之间无法进行比较。不同枝干上的人之间也无法进行比较。

如图4.7所示，一个集合 $\{x, y, z\}$ 的所有子集在包含关系下构成一个偏序集。尽管图中任意元素，都可以找到其子集，但是图中同一层级上的元素间无法比较，另外 $\{x\}$ 和 $\{y, z\}$ 间也无法比较。

<sup>10</sup>一旦开始考虑全部集合的集合，就会对导致“全部不包括自身的集合”这样的矛盾，称之为罗素悖论。我们将在第7章详细讲述罗素悖论。

<sup>11</sup>确切地说是全函数，即定义域中的每个元素都可以应用的函数

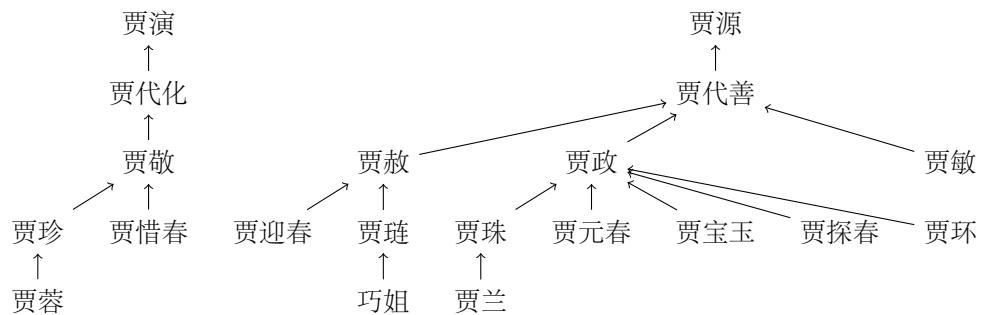


图 4.6: 《红楼梦》贾家的家族树

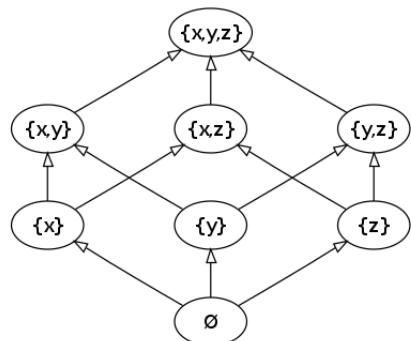


图 4.7: 一个集合的子集在包含关系下构成一个偏序集

一般来说，任何一个偏序集都是一个预序集，但是反过来却不一定成立。一个预序集不一定是一个偏序集。高中时我们都学过单调函数，就是那种只增不减的函数，如果 $x \leq y$ ，则 $f(x) \leq f(y)$ 。把这种单调函数组合起来用到偏序集和预序集上，我们就获得了一对范畴

***Pre***      ***Pos***

它们的对象分别是所有的*preset*和*poset*，两个范畴中的箭头都是单调映射。由于恒等映射也是单调映射，所以范畴中的恒等箭头为恒等映射。

幺半群范畴和预序集范畴这两个例子不是随便挑选的，我们有着特殊的用意。这两个范畴可以说是最简单的例子。通过幺半群范畴，我们可以学习组合；通过预序集范畴，我们可以学习比较。对数学结构的组合与比较是整个范畴论的核心。在这一意义上，任何范畴都是幺半群与预序集的某种混合形式（[39]第13页）。

***Pre***是包含所有预序集的巨大范畴。另一方面预序集范畴也可以很小。小到只含有一个预序集。考虑一个集合，对象是集合中的元素 $i, j, k, \dots$ 。如果 $i, j$ ，可比较并且 $i \leq j$ ，我们就定义箭头

$$i \longrightarrow j$$

这样任何对象间或者没有箭头，表示它们不可比较；或者存在一个箭头，表示它们有 $\leq$ 关系。总之，任何两个对象间最多只有一个箭头。可以验证这样定义的箭头可以组合，并且每个元素都有 $i \leq i$ ，因此有指向自己的箭头。这样一个预序集本身就是一个范畴。我们再次看到了预序集范畴和幺半群范畴的对偶。幺半群作为范畴只含有一个对象，却含有丰富的箭头；预序集作为范畴对象很多，但是对象间的箭头却最多只有一个。

### 练习 4.1

- 验证幺半群 $(S, \cup, \emptyset)$ （群元素是集合，二元运算是集合的并，单位元是空集）和 $(N, +, 0)$ （群元素是自然数，二元运算是加法，单位元是零）都是只含有一个对象的范畴。
- 第一章中我们介绍了自然数的皮亚诺公理，并且介绍了和皮亚诺算术同构的其它结构，例如链表等。这些完全可以用范畴来解释。这一结论是德国数学家戴德金发现的，尽管当时还没有范畴论。我们今天将这一范畴命名为皮亚诺范畴**Pno**。范畴中的对象为 $(A, f, z)$ ，其中 $A$ 为元素的集合，对于自然数来说这个集合是全体自然数 $N$ ； $f : A \rightarrow A$ 是后继函数，对于自然数来说，就是 $succ$ ； $z \in A$ 是起始元素，对于自然数来说是0。任给两个皮亚诺对象 $(A, f, z)$ 和 $(B, g, c)$ ，现在定义从 $A$ 到 $B$ 的态射

$$A \xrightarrow{\phi} B$$

它满足

$$\phi \circ f = g \circ \phi \quad \text{且} \quad \phi(z) = c$$

试验证**Pno**的确是一个范畴。

#### 4.1.2 箭头≠函数

在此前的例子中，箭头要么是一般意义上的函数，要么是类似函数意义的映射和态射。这容易造成一种错觉，认为箭头等同于函数。我们接下来看的例子有助于打破这种错觉。有一种名叫关系的范畴。范畴中的对象是集合。从集合 $A$ 到 $B$ 的箭头 $A \xrightarrow{R} B$ 的定义为：

$$R \subseteq B \times A$$

我们来看看这个箭头的含义是什么。集合 $B \times A$ 代表了所有 $B$ 和 $A$ 中元素对的组合。也叫做 $B$ 和 $A$ 的积：

$$B \times A = \{(b, a) | b \in B, a \in A\}$$

我们以红楼梦人物为例，集合 $A = \{\text{贾蓉, 贾惜春, 贾琏}\}$ ，集合 $B = \{\text{秦可卿, 王熙凤}\}$ ，则 $B \times A$ 的积为 $\{(秦可卿, 贾蓉), (秦可卿, 贾惜春), (秦可卿, 贾琏), (王熙凤, 贾蓉), (王熙凤, 贾惜春), (王熙凤, 贾琏)\}$ 。集合 $R$ 是 $B \times A$ 的子集，它可以看作 $A$ 与 $B$ 的某种关系。如果 $A$ 中的元素 $a$ 和 $B$ 中的元素 $b$ 满足关系 $R$ ，则有 $(b, a) \in R$ ，我们将其记为 $bRa$ 。具体到这个例子，我们令 $R = \{(秦可卿, 贾蓉), (王熙凤, 贾琏)\}$ ，则 $R$ 代表婚姻关系。

这样从对象 $A$ 到 $B$ 的所有箭头构成的集合就代表了从 $A$ 到 $B$ 的各种可能的关系。现在我们考虑箭头的组合

$$A \rightarrow B \rightarrow C$$

如果存在某个中间集合中的元素 $b$ ，同时使得两个关系 $bRa, cSb$ 都成立，我们就说存在箭头间的组合。具体到红楼梦的例子，令集合 $C = \{\text{王夫人, 薛姨妈, 邢夫人}\}$ ，关系 $S = \{(王夫人, 王熙凤), (薛姨妈, 王熙凤)\}$ 表示姑妈关系。这样组合箭头 $S \circ R$ 的结果是： $\{(王夫人, 贾琏), (薛姨妈, 贾琏)\}$ ，表示 $c$ 的侄女嫁给了 $b$ 的关系。也就是说王夫人、薛姨妈和贾琏都通过王熙凤使得两个关系同时成立。恒等箭头的定义很简单，所有元素都和自己产生恒等关系。

我们还可以从任何已知的范畴产生新的范畴，例如把一个范畴 $\mathbf{C}$ 中的所有箭头反向就得到了一个对偶的范畴 $\mathbf{C}^{op}$ 。这样我们了解了一个范畴，就同时了解了它的对偶范畴。

## 4.2 函子

我们说范畴论是对抽象代数结构的“二次抽象”。上一节我们看到如何把所有的集合和映射、群和态射、偏序集和单调函数等抽象成范畴。接下来的问题是，如何在这些范畴之间架设桥梁、进行比较？函子<sup>12</sup>（functor）就是用来对范畴及其内在的关系（对象和箭头）进行比较的。

### 4.2.1 函子的定义

在某种意义上说，函子好比是范畴之间的变换关系（态射）。但是它不仅把一个范畴中的对象映射为另一范畴中的对象，它还将一个范畴中的箭头映射到另一个范畴中。这一点是和普通的态射（例如群之间的态射）不同的。

函子通常用符号 $\mathsf{F}$ 表示。既然说函子好比是范畴间的态射，那么函子必须能够忠实保持范畴间的结构和关系，这是如何做到的呢？为此函子必须满足两条性质。

- 第一条性质，是函子必须保持恒等箭头仍然变换为恒等箭头。用图来表示就是：

$$A \xrightarrow{id} A \quad \mapsto \quad \mathsf{F}A \xrightarrow{id} \mathsf{F}A$$

- 第二条性质，函子必须能够保持箭头的组合仍然变换为箭头的组合<sup>13</sup>。



<sup>12</sup>C++语言的一些资料中，用英文functor来命名函数对象，即function object。这与范畴论中的函子完全无关。

<sup>13</sup>实际存在两种变换方式，一种叫做协变，一种叫做反变。这里仅考虑了协变。

$$\mathsf{F}(g \circ f) = \mathsf{F}(g) \circ \mathsf{F}(f)$$

函子之间也可以进行组合，例如函子 $(\mathsf{FG})(f)$ 表示先用函子 $\mathsf{G}$ 对箭头 $f$ 进行变换，然后再用 $\mathsf{F}$ 对这一新范畴中的箭头进行变换。

### 4.2.2 函子的例子

让我们用具体的例子来更好地理解函子的概念。如果一个函子从一个范畴映射到这个范畴本身上，这样的函子被称为**自函子**（endofunctor）<sup>14</sup>。最简单的函子称为“恒等函子”，它是一个自函子，记为 $id : \mathbf{C} \rightarrow \mathbf{C}$ 。它可以作用到任何范畴上，将对象 $A$ 映射为对象 $A$ ，将箭头 $f$ 映射为箭头 $f$ 。

其次简单的函子叫做“常函子”，它的行为像一个黑洞，我们可以把它记作 $K_B : \mathbf{C} \rightarrow \mathbf{B}$ 。它可以作用到任何范畴上，把所有的对象都映射为黑洞中的对象 $B$ ，把所有的箭头都映射为黑洞中的恒等箭头 $id_B$ 。黑洞范畴中只有一个恒等箭头，它显然也满足箭头组合的性质： $id_B \circ id_B = id_B$ 。

**例 4.2.1.** 我们接下来举的例子叫做“可能函子”（maybe functor）。

计算机科学家，快速排序算法的提出者，1980年图灵奖得主霍尔<sup>15</sup>曾说过一段有趣的话<sup>16</sup>。



计算机科学家霍尔

“我在1965年发明了空引用（null reference）。这是一个导致数十亿美元损失的发明。当时，我在为一种面向对象语言（ALGOL W）设计最早的类型系统。我希望能够保证所有的引用，经过编译器的自动检查，都是绝对安全的。但是我无法抗拒空引用的想法，它太容易实现了。此后，空引用导致了无数的错误、漏洞和系统崩溃。在过去的40年，由此导致的痛苦损失估计高达数十亿美元。”[40]

2015年后主流的编程环境都纷纷将Maybe概念引入以取代null来获取更安全的方法<sup>17</sup>。

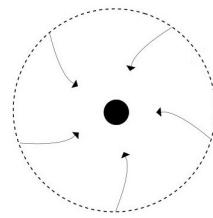
下图描述了Maybe函子的行为。

<sup>14</sup>这和代数中的自同构概念很类似。如果记不清了，请回顾一下上一章。

<sup>15</sup>英文是Hoare，不是美国物理学家霍尔（Hall）。

<sup>16</sup>2009年在伦敦举行的QCon会议上

<sup>17</sup>例如Java和C++中的Optional<T>。



常数函子的行为像一个黑洞

$$\begin{array}{ccc} A & \xrightarrow{\quad} & \text{Maybe } A \\ f \downarrow & & \downarrow \text{Maybe}(f) \\ B & \xrightarrow{\quad} & \text{Maybe } B \end{array}$$

图中左侧的对象 $A, B$ 是一些数据类型，比如整形，布尔型。而右侧的对象，是通过函子`Maybe`映射成的类型。如果 $A$ 代表`Int`，则右侧对应是`Maybe Int`， $B$ 如果代表`Bool`，则右侧对应的是`Maybe Bool`。可能函子是怎样完成对象的映射呢？它的对象映射部分是这样定义的：

```
data Maybe A = Nothing | Just A
```

也就是说，若对象是类型 $A$ ，则映射到的对象是类型`Maybe A`。注意，这里的对象是类型，而不是值。而类型`Maybe A`的值要么是一个空值`Nothing`，要么是一个用`Just`构造的值。

例如，对象是类型`Int`，可能函子映射后对象是类型`Maybe Int`，它的值可能是`Nothing`或者`Just 5`。

例如有一个元素类型为 $A$ 的二叉搜索树，在其中搜索一个值时，可能会搜不到。所以搜索的结果类型是`Maybe A`<sup>18</sup>。

$$\begin{aligned} \text{lookup } \text{Nil } &= \text{Nothing} \\ \text{lookup } (\text{Br } l \ k \ r) \ x &= \begin{cases} x < k : \text{lookup } l \ x \\ x > k : \text{lookup } r \ x \\ x = k : \text{Just } k \end{cases} \end{aligned}$$

在处理`Maybe`数据类型时，必须处理可能的两种值，例如：

$$\begin{aligned} \text{elem } \text{Nothing} &= \text{False} \\ \text{elem } (\text{Just } x) &= \text{True} \end{aligned}$$

函子既映射对象，也映射箭头。以上我们看到了`Maybe`如何映射对象，那么它是如何映射箭头的呢？上面图中的左侧从上到下有一个箭头 $A \xrightarrow{f} B$ ，而右侧也有一个箭头 $\text{Maybe } A \xrightarrow{\text{Maybe}(f)} \text{Maybe } B$ 。我们不妨把右侧的箭头叫做 $f'$ 。假设我们知道左侧箭头 $f$ 的行为，那么右侧箭头的行为是什么呢？我们说过处理`Maybe`类型的数据，必须处理两种可能的值，所以 $f'$ 的行为应该是这样的：

$$\begin{aligned} f' \text{ Nothing} &= \text{Nothing} \\ f' (\text{Just } x) &= \text{Just } (f x) \end{aligned}$$

这种给定 $f$ ，映射成 $f'$ 的行为恰好就是`Maybe`函子对箭头进行的映射。实际的编程环境中，通常使用`fmap`来定义函子对箭头的映射。我们可以定义所有函子 $F$ 满足：

$$\text{fmap} : (A \rightarrow B) \rightarrow (F A \rightarrow F B)$$

也就是说，如果 $F$ 是一个函子，它把从 $A$ 到 $B$ 的箭头映射成从 $F A$ 到 $F B$ 的箭头。因此对于`Maybe`函子，相应的`fmap`定义如下：

$$\begin{aligned} \text{fmap} &: (A \rightarrow B) \rightarrow (\text{Maybe } A \rightarrow \text{Maybe } B) \\ \text{fmap } f \text{ Nothing} &= \text{Nothing} \\ \text{fmap } f (\text{Just } x) &= \text{Just } (f x) \end{aligned}$$

---

<sup>18</sup>完整的代码见本章附录

回到前面二叉搜索树的例子，假如树中元素的类型是自然数，我们在树中搜索一个值，如果搜到，就将这个值转换成二进制，否则返回*Nothing*。如果我们以前已经写过一个将十进制自然数转换成二进制的函数：

$$\text{binary}(n) = \begin{cases} n < 2 : [n] \\ \text{否则: } \text{binary}(\lfloor \frac{n}{2} \rfloor) ++ [n \bmod 2] \end{cases}$$

下面是一个相应的Haskell代码实现。为了提高性能，这个实现使用了尾递归。

```
binary = bin □ where
  bin xs 0 = 0 : xs
  bin xs 1 = 1 : xs
  bin xs n = bin ((n `mod` 2) : xs) (n `div` 2)
```

有了函子，就可以直接将这个函数箭头“举”到上面去，如下图所示：

$$\begin{array}{ccc} \text{Maybe Int} & \xrightarrow{\text{fmap binary}} & \text{Maybe [Int]} \\ \uparrow & & \uparrow \\ \text{Int} & \xrightarrow{\text{binary}} & [\text{Int}] \end{array}$$

这样，我们就可以直接利用Maybe函子和*binary*箭头操作二叉树搜索的结果：

$$\text{fmap binary (lookup t x)}$$

**证明.** 一般的读者可以跳过这个框中的内容。为了验证Maybe的确是一个函子，我们还需要验证箭头映射的两条性质：

$$\begin{aligned} \text{fmap id} &= \text{id} \\ \text{fmap } (f \circ g) &= \text{fmap } f \circ \text{fmap } g \end{aligned}$$

首先验证第一条性质。*id*的定义为：

$$\text{id } x = x$$

因此：

$$\begin{aligned} \text{fmap id Nothing} &= \text{Nothing} && \text{fmap的定义} \\ &= \text{id Nothing} && \text{反向用id的定义} \end{aligned}$$

并且

$$\begin{aligned} \text{fmap id (Just } x) &= \text{Just } (\text{id } x) && \text{fmap的定义} \\ &= \text{Just } x && \text{id的定义} \\ &= \text{id } (\text{Just } x) && \text{反向用id的定义} \end{aligned}$$

接下来验证第二条性质：

$$\begin{aligned}
 fmap(f \circ g) Nothing &= Nothing && fmap\text{的定义} \\
 &= fmap f Nothing && \text{反向使用 } fmap\text{的定义} \\
 &= fmap f (fmap g Nothing) && \text{反向使用 } fmap\text{的定义} \\
 &= (fmap f \circ fmap g) Nothing && \text{反向用函数组合的定义}
 \end{aligned}$$

并且

$$\begin{aligned}
 fmap(f \circ g)(Just x) &= Just((f \circ g)x) && fmap\text{的定义} \\
 &= Just(f(gx)) && \text{函数组合的定义} \\
 &= fmap f(Just(gx)) && \text{反向使用 } fmap\text{的定义} \\
 &= fmap f(fmap g(Just x)) && \text{反向使用 } fmap\text{的定义} \\
 &= (fmap f \circ fmap g)(Just x) && \text{反向用函数组合的定义}
 \end{aligned}$$

因此Maybe的确是一个函数。 □

**例 4.2.2.** 接下来介绍的例子是列表函数。在第一章中我们介绍了列表的定义：

```
data List A = Nil | Cons(A, List A)
```

从编程的观点看，这定义了“单向链表”的数据结构，链表中元素的类型是  $A$ ，这样的链表称为类型为  $A$  的列表。从范畴的观点来看，一个列表函数要分别定义对象和箭头的映射。在这里，对象是类型，箭头是全函数。下图描述了列表函数的行为：

$$\begin{array}{ccc}
 A & \xrightarrow{\quad} & \text{List } A \\
 f \downarrow & & \downarrow \text{List}(f) \\
 B & \xrightarrow{\quad} & \text{List } B
 \end{array}$$

图中左侧的对象  $A, B$  是数据类型，例如整型、布尔型甚至是  $\text{Maybe Char}$  这样的复杂类型。而右侧的对象，是通过列表函数映射成的类型，如果  $A$  为整形  $\text{Int}$ ，则右侧对应  $\text{List Int}$ ，如果  $B$  表示  $\text{Char}$ ，则右侧对应的是  $\text{List Char}$ ，也就是  $\text{String}$ 。

请再次注意这里的对象是类型，而不是值， $A$  可以是  $\text{Int}$ ，但不是具体的值 5。因此  $\text{List } A$  对应整形列表，是一个类型，而不是一个具体的值，如：[1, 1, 2, 3, 5]。那么如何产生具体的列表值呢？那要靠  $\text{Nil}$  和  $\text{Cons}$  两个具体的函数来产生空值或者诸如  $\text{List}(1, \text{List}(1, \text{List}(2, \text{Nil})))$  这样的具体列表。

以上是列表函数在对象映射时的行为，那么它是如何对箭头进行映射的呢？也就是说，给定一个函数  $f : A \rightarrow B$ ，如何通过列表函数得到另一个函数  $g : \text{List } A \rightarrow \text{List } B$  呢？与可能函数  $\text{Maybe}$  类似，我们可以定义一个  $fmap$  实现从箭头  $f$  到箭头  $g$  的映射，对于列表函数它的类型为：

$$fmap : (A \rightarrow B) \rightarrow (\text{List } A \rightarrow \text{List } B)$$

接下来需要仔细分析  $g$  的行为。首先考虑最简单的情况，不管箭头  $f$  的定义如何，如果  $\text{List } A$  是一个空列表  $\text{Nil}$ ，则应用  $g$  后的结果也必然是空列表。所以有：

$$fmap f \text{Nil} = \text{Nil}$$

接下来考虑递归的情况  $\text{Cons}(x, xs)$ ，其中  $x$  是类型为  $A$  的某个值，而  $xs$  是类型为  $\text{List } A$  的子列表。如果  $f(x) = y$ ，将类型  $A$  的值  $x$  映射为类型  $B$  的值  $y$ ，那么我们就将  $f$  应用到  $x$  上，然后在将其

递归地应用到子列表 $xs$ 上从而得到一个元素类型为 $B$ 的子列表 $ys$ , 最后再将 $y$ 和 $ys$ 链接起来得到最终的结果:

$$fmap f \text{Cons}(x, xs) = \text{Cons}(f x, fmap f xs)$$

综上, 我们得到了 $fmap$ 的完整定义:

$$\begin{aligned} fmap &: (A \rightarrow B) \rightarrow (\text{List } A \rightarrow \text{List } B) \\ fmap f \text{Nil} &= \text{Nil} \\ fmap f \text{Cons}(x, xs) &= \text{Cons}(f x, fmap f xs) \end{aligned}$$

如果使用第一章介绍的简记法, 用冒号“: $\cdot$ ”表示 $Cons$ , 用中括号表示列表, 用“[]”表示 $Nil$ , 则列表函子的箭头部分映射可以简写为:

$$\begin{aligned} fmap &: (A \rightarrow B) \rightarrow (\text{List } A \rightarrow \text{List } B) \\ fmap f [] &= [] \\ fmap f (x : xs) &= (f x) : (fmap f xs) \end{aligned}$$

仔细观察这个定义, 再和第一章中列表的“逐一映射”定义对比, 会发现它们除了名字之外完全一样, 因此我们可以复用列表的“逐一映射”来定义列表函子, 例如Haskell中列表函子的定义就是通过复用 $map$ 实现的。

```
instance Functor [] where
  fmap = map
```

作为本例的结尾, 我们来验证一下列表函子的箭头映射部分是否满足组合和恒等的两条性质。一般的读者可以跳过这一证明部分。

$$\begin{aligned} fmap id &= id \\ fmap(f \circ g) &= fmap f \circ fmap g \end{aligned}$$

证明. 我们用数学归纳法来验证恒等性质, 首先针对空列表的情况:

$$\begin{aligned} fmap id \text{Nil} &= \text{Nil} && \text{fmap的定义} \\ &= id \text{Nil} && \text{反向用} id \text{的定义} \end{aligned}$$

针对 $(x : xs)$ 的递归情况, 令递归假设为 $fmap id xs = id xs$ , 我们有:

$$\begin{aligned} fmap id (x : xs) &= (id x) : (fmap id xs) && \text{fmap的定义} \\ &= (id x) : (id xs) && \text{递归假设} \\ &= x : xs && id \text{的定义} \\ &= id (x : xs) && \text{反向用} id \text{的定义} \end{aligned}$$

同样, 我们用数学归纳法来验证组合性质, 对于空列表, 我们有:

$$\begin{aligned} fmap(f \circ g) \text{Nil} &= \text{Nil} && \text{fmap的定义} \\ &= fmap f \text{Nil} && \text{反向使用} fmap \text{的定义} \\ &= fmap f (fmap g \text{Nil}) && \text{反向使用} fmap \text{的定义} \\ &= (fmap f \circ fmap g) \text{Nil} && \text{反向用函数组合的定义} \end{aligned}$$

针对 $(x : xs)$ 的递归情况, 令递归假设为 $fmap(f \circ g) xs = (fmap f \circ fmap g) xs$ , 我们有:

$$\begin{aligned}
 fmap(f \circ g)(x : xs) &= ((f \circ g)x) : (fmap(f \circ g)xs) && fmap\text{的定义} \\
 &= ((f \circ g)x) : ((fmap f \circ fmap g)xs) && \text{递归假设} \\
 &= (f(gx)) : (fmap f (fmap g xs)) && \text{函数组合的定义} \\
 &= fmap f ((g x) : (fmap g xs)) && \text{反向使用 } fmap\text{ 的定义} \\
 &= fmap f (fmap g (x : xs)) && \text{再次反向使用 } fmap\text{ 的定义} \\
 &= (fmap f \circ fmap g)(x : xs) && \text{反向用函数组合的定义}
 \end{aligned}$$

这就验证了List的确是一个函子。 □

### 练习 4.2

1. 请使用叠加操作 $foldr$ 来定义列表函子的箭头映射。
2. 证明可能函子和列表函子的组合 $\text{Maybe} \circ \text{List}$ 与 $\text{List} \circ \text{Maybe}$ 仍然是函子。
3. 证明任意函子的组合 $G \circ F$ 仍然是函子。
4. 思考一个预序集范畴上的函子的例子。
5. 回顾第二章中介绍的二叉树，请定义一个二叉树函子。

## 4.3 积和余积

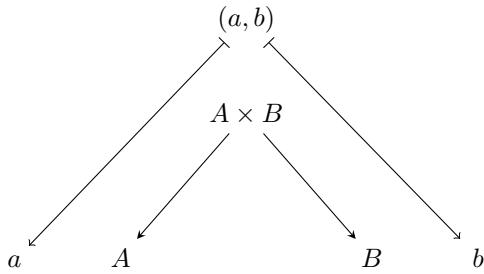
在介绍更复杂的范畴和函子的例子之前，我们先了解一下积和余积的概念。这里的积是指范畴的积。为了容易理解，我们先从集合的积入手。两个集合 $A$ 和 $B$ 的笛卡尔积 $A \times B$ ，是所有有序对 $(a, b)$ 的集合。其中 $a \in A, b \in B$ 。即：

$$\{(a, b) | a \in A, b \in B\}$$

例如有限集合 $\{1, 2, 3\}$ 和 $\{a, b\}$ 的积为

$$\{(1, a), (2, a), (3, a), (1, b), (2, b), (3, b)\}$$

如果集合 $A, B$ 是同类的代数结构，如群、环等，我们可以定义如下图的对象和箭头。



笛卡尔积又称直积，是用法国哲学家、数学家、物理学家笛卡尔的名字命名的。在笛卡尔的时代，拉丁文是学术上广泛使用的语言。笛卡尔也有一个拉丁化的名字——卡提修斯（Cartesius）。正因为如此，笛卡尔积也称卡氏积，笛卡尔坐标系也称卡氏坐标系。

笛卡尔1596年生于法国一个地位较低的贵族家庭。一岁时他的母亲患肺结核去世，而他也受到传染，造成体弱多病。母亲去世后，父亲移居他乡并再婚，笛卡尔由外祖母带大，自此父子很少见面，但是父亲一直提供金钱方面的帮助，使他能够受到良好的教育。



勒内·笛卡尔（1596-1650）。哈尔斯的画布油画，现藏于卢浮宫

笛卡尔后来进入位于拉弗莱什的耶稣会的皇家大亨利学院学习。在那里，他学习到了数学和物理学，包括伽利略的工作。1616年毕业后，他遵从父亲的意愿，进入普瓦捷大学学习法律，并获得业士学位和文凭。毕业后笛卡尔一直对职业选择不定，又决心游历欧洲各地，专心寻求“世界这本大书”中的智慧。1618年，笛卡尔加入荷兰的拿骚的毛里茨的军队。

笛卡尔对结合数学与物理学的兴趣，是在荷兰当兵期间产生的。1618年，他偶然在路旁公告栏上，看到用佛莱芒语提出的数学问题征答。这引起了他的兴趣，并且让身旁的人，将他不懂的佛莱芒语翻译成拉丁语。这位身旁的人就是大他八岁的以撒·贝克曼（Isaac Beeckman）。贝克曼在数学和物理学方面有很高造诣，很快成为了他的导师。4个月后，他写信给贝克曼：“你是将我从冷漠中唤醒的人……”。

1622年，当他26岁时，笛卡尔变卖掉父亲留下的资产，用4年时间游历欧洲。他先在意大利住了2年，随后迁往于巴黎。当时法国教会势力庞大，不能自由讨论宗教问题，因此笛卡尔在1628年移居荷兰，在那里住了20多年。在此期间，笛卡尔致力于哲学研究，他发表了多部重要的文集，包括《方法论》、《形而上学的沉思》和《哲学原理》等，成为欧洲最有影响力的哲学家之一。

1649年笛卡尔受瑞典克里斯蒂娜女王之邀来到斯德哥尔摩担任女王的私人教师，但不幸在这片“熊、冰雪与岩石的土地”上得了肺炎，在1650年2月去世，享年54岁。

笛卡尔留下名言“我思故我在”，提出了“普遍怀疑”的主张。他开拓了欧洲理性主义哲学，是西方现代哲学的奠基人。他的哲学思想深深影响了之后的几代人。笛卡尔对现代数学的发展做出了重要的贡献，他创立的解析几何，成功地将当时完全分开的代数和几何学联系到了一起。

相对于集合的笛卡尔积，还有一种对偶的构造。我们可以从两个集合 $A, B$ 产生一个不相交的并集（disjoint union，也称为和） $A + B$ 。为了分清和中的每个元素究竟是来自 $A$ 还是 $B$ ，可以给元素增加一个标记（tag）：

$$A + B = (A \times \{0\}) \cup (B \times \{1\})$$

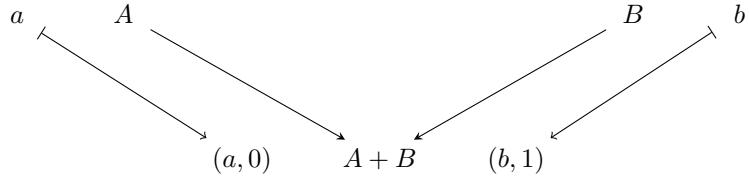
当从 $A + B$ 中取出一个元素 $(x, tag)$ ，如果 $tag$ 为0，我们就知道 $x$ 本来属于 $A$ ，如果 $tag$ 为1，就知道 $x$ 本来属于 $B$ 。这样有限集合 $\{1, 2, 3\}$ 和 $\{a, b\}$ 的和就是

$$\{(1, 0), (2, 0), (3, 0), (a, 1), (b, 1)\}$$

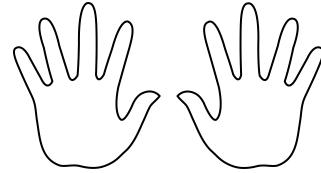
$A + B$ 也可以用程序等价地这样定义：

$$A + B = \text{zip } A \{0, \dots\} ++ \text{zip } B \{1, \dots\}$$

当 $A, B$ 是同类的代数结构时，我们可以定义如下的对象和箭头：



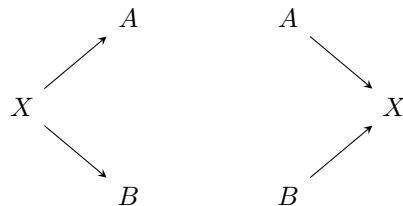
对比这两种构造我们发现它们是对偶的。如果把两张图旋转90度，把 $A$ 画在上面， $B$ 画在下面，这样两个问题一个在左侧，一个在右侧。如同双手一样对称。我们后面会发现，这种现象在范畴的相关概念中很常见。甚至我们的宇宙中也充满了对称。这样当我们了解了一个事物，就对偶地了解了另一个事物。



### 4.3.1 积和余积的定义

**定义 4.3.1.** 对于范畴 $\mathbf{C}$ 中的一对对象 $A$ 和 $B$ ，一个

指向 $A, B$ 的      从 $A, B$ 出发的  
楔形 (wedge) 是范畴 $\mathbf{C}$ 中的一个对象 $X$ 和一对箭头：

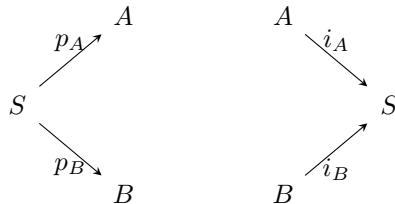


给定范畴中的一对对象 $A$ 和 $B$ ，可能有不止一个向左的或者向右的楔形。我们关心那个“最好”的楔形，它距离两个对象“最近”。本质上，我们在寻找一个泛性的 (universal) 楔形。这就引出了下面的 (一对) 定义：

**定义 4.3.2.** 对于范畴 $\mathbf{C}$ 中的一对对象 $A$ 和 $B$ ，一个

积 (product)      余积 (coproduct)

是一个特殊的楔形：



这个楔形满足如下泛性性质：对于任意楔形，

$$\begin{array}{ccc} & A & \\ f_A \nearrow & & \searrow f_A \\ X & & X \\ \searrow f_B & & \nearrow f_B \\ & B & \end{array}$$

都存在相应唯一的箭头：

$$X \xrightarrow{m} S \quad S \xrightarrow{m} X$$

使得下面图中的箭头可交换。

$$\begin{array}{ccc} & A & \\ f_A \nearrow & & \downarrow p_A \\ X \xrightarrow{m} S & \uparrow p_B & \downarrow i_A \\ \searrow f_B & & \uparrow i_B \\ & B & \end{array} \quad \begin{array}{ccc} & A & \\ \downarrow f_A & & \searrow m \\ S & \xrightarrow{m} & X \\ \uparrow i_B & & \nearrow f_B \\ B & & \end{array}$$

其中箭头  $m$  叫做楔形  $X$  的媒介箭头 (mediating arrow 或者 mediator)

在这一定义中，积或者余积并不仅仅是一个对象  $S$ ，而是一个对象加上一对箭头。其次，对于任意给定的  $X$ ，媒介箭头  $m$  都是唯一的。所谓箭头可交换，我们用左侧的积来举例，是说箭头间的关系满足：

$$\begin{aligned} f_A &= p_A \circ m \\ f_B &= p_B \circ m \end{aligned}$$

这立刻告诉我们，下面的特例一定成立：如果  $X$  等于  $S$ ， $m$  是一个自指箭头 (endo-arrow)

$$S \xrightarrow{m} S$$

则  $m$  一定是恒等箭头。此时的范畴图简化为：

$$\begin{array}{ccc} & A & \\ p_A \nearrow & & \uparrow p_A \\ S \xrightarrow{id_S} S & \downarrow p_B & \downarrow i_A \\ \searrow p_B & & \uparrow i_B \\ & B & \end{array} \quad \begin{array}{ccc} & A & \\ \downarrow i_A & & \searrow id_S \\ S & \xrightarrow{id_S} & S \\ \uparrow i_B & & \nearrow i_B \\ B & & \end{array}$$

在众多的楔形中，积和余积是特殊的——它们是泛性的，是“最接近”或者说“最好”的一个楔形。并且可以证明（见书后附录）积和余积是唯一的。但是积和余积并不总是同时存在，甚至有可能都不存在。现在我们回过头来再看一下集合的情形。

**引理 4.3.1.** 若  $A, B$  是两个集合, 则它们的

笛卡尔积  $A \times B$  不相交并集  $A + B$

在范畴 **Set** 中构成

积 余积

详细的证明过程可以参考本书后面的附录。在一些编程环境, 例如Haskell中, 积的确是通过二元组  $(a, b)$  以及函数  $fst$  和  $snd$  来实现的。但是, 余积却是以单独的数据类型来实现的。

```
data Either a b = Left a | Right b
```

这样的好处是, 我们不需要再通过 0, 1 这样的标记 (tag) 来了解一个 `Either a b` 的元素  $x$  究竟是来自 `a` 还是 `b`。例如下面的 Haskell 例子代码使用模式匹配来决定如何处理余积的元素:

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x)      =  f x
either _ g (Right y)    =  g y
```

举个具体的例子, 假设我们有一个 `Either String Int` 类型的余积, 它或者为一个字符串, 例如 `s = Left "hello"`, 或者是一个整数, 例如 `n = Right 8`。如果是字符串, 我们希望得到它的长度; 如果是整数, 我们希望将它加倍。为此我们可以这样调用 `either` 函数: `either length (*2) x`。

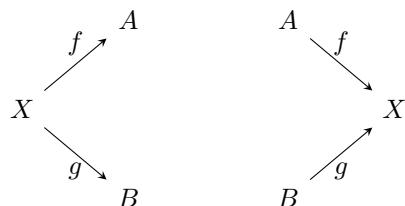
这样 `either length (*2) s` 会计算 “hello” 的长度, 结果为 5; 而 `either length (*2) n` 会将 8 加倍, 结果为 16。有些编程环境中, 使用联合体 (union) 或者枚举 (enum) 来部分实现余积的概念。方便起见, 我们有时把集合余积的这两个箭头称为 *left* 和 *right*。

### 4.3.2 积和余积的性质

在积和余积的定义中, 对于任意的  $X$  和箭头组成的楔形, 媒介箭头  $m$  是唯一确定的。媒介箭头在集合范畴 **Set** 中可以这样定义:

$$\begin{array}{ccc} \text{积} & & \text{余积} \\ m(x) = (a, b) & \quad \begin{cases} m(a, 0) = p(a) \\ m(b, 1) = q(b) \end{cases} & \end{array}$$

在更一般的范畴中更, 媒介箭头如何定义呢? 为此我们引入两个特殊的箭头运算符号。任意给定楔形:



分别定义

$$m = \langle f, g \rangle \quad m = [f, g]$$

使得它们满足：

$$\begin{cases} fst \circ m = f \\ snd \circ m = g \end{cases} \quad \begin{cases} m \circ left = f \\ m \circ right = g \end{cases}$$

这样，下面范畴图中的箭头是可交换的：

通过这对范畴图，我们立即可以得到一些积和余积的性质。首先是消去律：

$$\begin{cases} fst \circ \langle f, g \rangle = f \\ snd \circ \langle f, g \rangle = g \end{cases} \quad \begin{cases} [f, g] \circ left = f \\ [f, g] \circ right = g \end{cases}$$

对于积，如果  $f$  恰好等于  $fst$ ，而  $g$  恰好等于  $snd$ ，我们就得到了上一小节中恒等箭头的特例；同样对于余积，如果  $f$  恰好等于  $left$ ，而  $g$  恰好等于  $right$ ，则媒介箭头也是恒等箭头。我们称这条性质为反射律（reflection law）：

$$id = \langle fst, snd \rangle \quad id = [left, right]$$

如果存在另一个楔形  $Y$  和箭头  $h, k$ ，它们和  $f, g$  间满足关系：

积	余积
$\begin{cases} h \circ \phi = f \\ k \circ \phi = g \end{cases}$	$\begin{cases} \phi \circ h = f \\ \phi \circ k = g \end{cases}$

我们用

$$\langle h, k \rangle \circ \phi \quad \phi \circ [h, k]$$

代入  $m$ ，并使用消去律，这样就得到了融合律（fusion law）：

积	余积
$\begin{cases} h \circ \phi = f \\ k \circ \phi = g \end{cases} \Rightarrow \langle h, k \rangle \circ \phi = \langle f, g \rangle$	$\begin{cases} \phi \circ h = f \\ \phi \circ k = g \end{cases} \Rightarrow \phi \circ [h, k] = [f, g]$

也就是说：

$$\langle h, k \rangle \circ \phi = \langle h \circ \phi, k \circ \phi \rangle \quad \phi \circ [h, k] = [\phi \circ h, \phi \circ k]$$

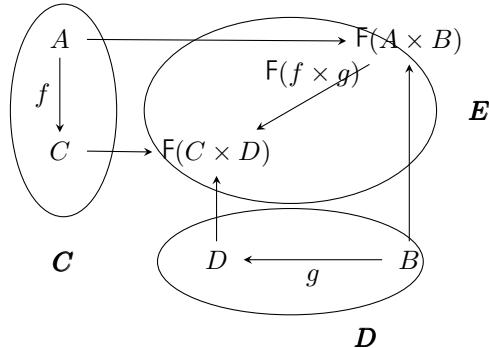
我们在下一章将看到，这些性质和定律对于我们进行算法推导、程序化简、性能优化会起很重要的作用。

### 4.3.3 积和余积作为函子

了解了范畴的积，就自然引出了**二元函子**（bifunctor或binary functor）的概念。我们此前介绍的函子，将一个范畴中的对象转换成另一个范畴中的对象，同时将一个范畴中箭头转成另一个范畴中的箭头。所谓二元函子，它作用于两个范畴 $\mathbf{C}$ 和 $\mathbf{D}$ 的积，也就是说它的源是 $\mathbf{C} \times \mathbf{D}$ 。所以对于对象部分，二元函子的转换关系为：

$$\begin{array}{ccc} \mathbf{C} \times \mathbf{D} & \longrightarrow & \mathbf{E} \\ A \times B & \longmapsto & \mathsf{F}(A \times B) \end{array}$$

但是函子除了作用于对象之外，也作用于箭头，对于范畴 $\mathbf{C}$ 中的箭头 $f$ ，和 $\mathbf{D}$ 中的箭头 $g$ ，二元函子是怎样对应的呢？我们观察下面的范畴图：



这个范畴图告诉我们，箭头 $A \xrightarrow{f} C$ 和 $B \xrightarrow{g} D$ 被函子 $\mathsf{F}$ 对应到范畴 $\mathbf{E}$ 中的新箭头，它的源是对象 $\mathsf{F}(A \times B)$ ，目标是对象 $\mathsf{F}(C \times D)$ 。为此我们可以先定义两个箭头 $f, g$ 的积，它作用于 $A$ 和 $B$ 的积，对于任意 $(a, b) \in A \times B$ ，它的行为如下：

$$(f \times g)(a, b) = (f a, g b)$$

然后二元函子 $F$ 作用于这两个箭头的积 $f \times g$ ，将其对应到新的箭头 $\mathsf{F}(f \times g)$ 上。于是最终的二元函子的定义如下：

$$\begin{array}{ccc} \mathbf{C} \times \mathbf{D} & \longrightarrow & \mathbf{E} \\ A \times B & \longmapsto & \mathsf{F}(A \times B) \\ f \times g & \longmapsto & \mathsf{F}(f \times g) \end{array}$$

既然是函子，我们还必须验证这个定义满足函子的两条性质：恒等性质和组合性质。一般的读者可以略过框中的内容。

$$\begin{aligned} \mathsf{F}(id \times id) &= id \\ \mathsf{F}((f \circ f') \times (g \circ g')) &= \mathsf{F}(f \times g) \circ \mathsf{F}(f' \times g') \end{aligned}$$

证明。我们将积看作一个对象，这样只要等价地证明下面的两条性质，再用一元函子的已知结论就可以证明二元函子的情形了：

$$\begin{aligned} id \times id &= id \\ (f \circ f') \times (g \circ g') &= (f \times g) \circ (f' \times g') \end{aligned}$$

我们先证明恒等性质。对于任意 $(a, b) \in A \times B$ ，有：

$$\begin{aligned}
 (id \times id)(a, b) &= (id(a), id(b)) && \text{箭头积的定义} \\
 &= (a, b) && id \text{的定义} \\
 &= id(a, b) && \text{反向用 } id \text{ 的定义}
 \end{aligned}$$

接着证明组合性质。

$$\begin{aligned}
 ((f \circ f') \times (g \circ g'))(a, b) &= ((f \circ f') a, (g \circ g') b) && \text{箭头积的定义} \\
 &= (f(f'(a)), g(g'(b))) && \text{箭头组合的定义} \\
 &= (f \times g)(f'(a), g'(b)) && \text{反向用箭头积的定义} \\
 &= (f \times g)((f' \times g')(a, b)) && \text{反向用箭头积的定义} \\
 &= ((f \times g) \circ (f' \times g'))(a, b)
 \end{aligned}$$

这样就证明了二元函子满足所有函子的性质。  $\square$

与  $fmap$  类似，在一些编程环境中，无法用同一个符号既做对象的映射，也做箭头的映射。为此可以专门为二元函子定义一个  $bimap$ 。我们要求所有的二元函子  $F$  满足：

$$bimap : (A \rightarrow C) \rightarrow (B \rightarrow D) \rightarrow (F A \times B \rightarrow F C \times D)$$

也就是说，如果  $F$  是一个二元函子，它把两个箭头  $A \xrightarrow{g} C$ 、 $B \xrightarrow{h} D$  映射为从  $F A \times B$  到  $F C \times D$  的箭头。

有了二元函子的概念，就可以定义积函子和余积函子了。我们用中缀的形式，将

积函子记为“ $\times$ ”      余积函子记为“ $+$ ”

对于两个对象，积函子将其映射为对象的积；余积函子将其映射为对象的余积。对于箭头，定义：

$$f \times g = \langle f \circ fst, g \circ snd \rangle \quad f + g = [left \circ f, right \circ g]$$

接下来需要验证函子的两条性质，一般的读者可以略过框中的内容。首先是恒等性质。将  $f, g$  都用  $id$  代入：

$$\begin{aligned}
 &id \times id && id + id \\
 &= \langle id \circ fst, id \circ snd \rangle && \times \text{的定义} && = [left \circ id, right \circ id] && + \text{的定义} \\
 &= \langle fst, snd \rangle && id \text{的性质} && = [left, right] && id \text{的性质} \\
 &= id && \text{积的反射律} && = id && \text{余积的反射律}
 \end{aligned}$$

其次验证组合性质。

$$(f \times g) \circ (f' \times g') = f \circ f' \times g \circ g' \quad (f + g) \circ (f' + g') = f \circ f' + g \circ g'$$

为此，我们先证明积和余积的吸收律 (absorption law)：

$$\begin{array}{ll}
 \text{积的吸收律} & \text{余积的吸收律} \\
 (f \times g) \circ \langle p, q \rangle = \langle f \circ p, g \circ q \rangle & [p, q] \circ (f + g) = [p \circ f, q \circ g]
 \end{array}$$

我们只给出左手边积的情况，余积的情况与此对称，我们将其留作练习。

$$\begin{aligned}
 & (f \times g) \circ \langle p, q \rangle \\
 = & \langle f \circ fst, g \circ snd \rangle \circ \langle p, q \rangle && \times \text{的定义} \\
 = & \langle f \circ fst \circ \langle p, q \rangle, g \circ snd \circ \langle p, q \rangle \rangle && \text{积的融合律} \\
 = & \langle f \circ p, g \circ q \rangle && \text{积的消去律}
 \end{aligned}$$

使用吸收律，令  $p = f' \circ fst, q = g' \circ snd$ ，就可以验证组合性质：

$$\begin{aligned}
 & (f \times g) \circ (f' \times g') \\
 = & (f \times g) \circ \langle f' \circ fst, g' \circ snd \rangle && \text{对第二项用 } \times \text{的定义} \\
 = & (f \times g) \circ \langle p, q \rangle && \text{用 } p, q \text{ 代换} \\
 = & \langle f \circ p, g \circ q \rangle && \text{吸收率} \\
 = & \langle f \circ f' \circ fst, g \circ g' \circ snd \rangle && \text{把 } p, q \text{ 代换回} \\
 = & \langle (f \circ f') \circ fst, (g \circ g') \circ snd \rangle && \text{结合律} \\
 = & (f \circ f') \times (g \circ g') && \text{反向用 } \times \text{的定义}
 \end{aligned}$$

积函子是二元函子的一个特例。我们可以为积函子定义 *bimap* 如下：

$$\begin{aligned}
 bimap : (A \rightarrow C) \rightarrow (B \rightarrow D) \rightarrow (A \times B \rightarrow C \times D) \\
 bimap f g (x, y) = (f x, g y)
 \end{aligned}$$

如果使用 *Either* 类型实现余积函子，则相应的 *bimap* 可以定义如下：

$$\begin{aligned}
 bimap : (A \rightarrow C) \rightarrow (B \rightarrow D) \rightarrow (\text{Either } A B \rightarrow \text{Either } C D) \\
 bimap f g (\text{left } x) = \text{left } (f x) \\
 bimap f g (\text{right } y) = \text{right } (g y)
 \end{aligned}$$

### 练习 4.3

1. 考虑偏序集 (poset) 中的两个对象，它们的积是什么？余积是什么？
2. 证明余积的吸收率，并验证余积函子的组合性质。

## 4.4 自然变换

艾伦伯格和麦克兰恩在1940年代发展范畴论时，他们想解释为何某些构造是“自然”的，而某些不是。现在这一术语被命名为“自然变换”。麦克兰恩曾说：“我并非为了研究函子而发明了范畴，我发明它们是为了研究自然变换。”通过前面介绍的范畴和函子，我们可以说在范畴论中，箭头用于比较对象，而函子用于比较范畴。那么我们用什么比较函子呢？自然变换正是用于比较函子的工具。

考虑下面的两个函子：

$$\begin{array}{ccc}
 & \xrightarrow{F} & \\
 Src & \xrightarrow{\quad\quad\quad} & Trg \\
 & \xrightarrow{G} &
 \end{array}$$

它们联系两个同样的范畴，或者都是协变的，或者都是反变的。我们怎样比较它们呢？由于函子既映射对象，又映射箭头，所以我们也得既比较函子映射后的对象，也比较函子映射后的箭头。考虑 *Src* 中的任意对象  $A$ ，它被两个函子映射成 *Trg* 中的两个对象  $FA$  和  $GA$ 。我们要研究的是从  $FA$  到  $GA$  之间的箭头

$$\mathbf{F}A \xrightarrow{\phi_A} \mathbf{G}A$$

并且除了  $A$  外，我们对  $\mathbf{Src}$  中的所有对象做同样的研究。

**定义 4.4.1.** 任给如上图所示的一对函子  $\mathbf{F}, \mathbf{G}$ ，一个自然变换

$$F \xrightarrow{\phi} G$$

是由  $A$  索引的在  $\mathbf{Trg}$  中的一族箭头

$$\mathbf{F}A \xrightarrow{\phi_A} \mathbf{G}A$$

并且对  $\mathbf{Src}$  中的任意箭头  $A \xrightarrow{f} B$ ，下面的方形都是可交换的。

$$\begin{array}{ccc} \begin{array}{c} \mathbf{F}A \xrightarrow{\phi_A} \mathbf{G}A \\ \downarrow \mathbf{F}(f) \qquad \downarrow \mathbf{G}(f) \\ \mathbf{F}B \xrightarrow{\phi_B} \mathbf{G}B \end{array} & \begin{array}{c} A \\ \downarrow f \\ B \end{array} & \begin{array}{c} \mathbf{F}A \xrightarrow{\phi_A} \mathbf{G}A \\ \uparrow \mathbf{F}(f) \qquad \uparrow \mathbf{G}(f) \\ \mathbf{F}B \xrightarrow{\phi_B} \mathbf{G}B \end{array} \\ \text{协变} & & \text{反变} \end{array}$$

从自然变换的范畴图中，我们看到任何一个箭头  $f$ ，都对应一个可交换的方形。对于协变的情况，可交换意味着对于任何箭头  $f$ ，下面的关系成立：

$$\mathbf{G}(f) \circ \phi_A = \phi_B \circ \mathbf{F}(f)$$

#### 4.4.1 自然变换的例子

自然变换是在范畴、箭头、函子之上的又一层抽象。我们举一些具体的例子来帮助理解这一重要的概念。

**例 4.4.1.** 第一个例子是前缀枚举函数  $inits$ 。任给一个字符串或者列表， $inits$  可以枚举出所有的前缀，例如：

```
inits "Mississippi" = ["", "M", "Mi", "Mis", "Miss", "Missi", "Missis",
                        "Mississ", "Mississi", "Mississip", "Mississipp", "Mississippi"]
```

```
inits [1, 2, 3, 4] = [[], [1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]
```

概括来说， $inits$  的行为如下：

$$inits[a_1, a_2, \dots, a_n] = [[], [a_1], [a_1, a_2], \dots, [a_1, a_2, \dots, a_n]]$$

考虑集合范畴  $\mathbf{Set}$ ，对于任何对象，也就是集合  $A$ （或者说类型  $A$ ），都存在被  $A$  索引的  $inits$  箭头：

$$inits_A : \mathbf{List}A \rightarrow \mathbf{List}(\mathbf{List}A)$$

其中一个函子是列表函子  $\mathbf{List}$ ，另一个函子是嵌套列表函子  $\mathbf{List} \mathbf{List}$ 。  
如果我们用“ $[]$ ”简记法，这个箭头也可以表示为：

$$[A] \xrightarrow{\text{inits}_A} [[A]]$$

我们接下来要验证，对任何函数  $A \xrightarrow{f} B$ ，有：

$$\text{List}(\text{List}(f)) \circ \text{inits}_A = \text{inits}_B \circ \text{List}(f)$$

也就是说，要验证如下的方形范畴图是可交换的。

$$\begin{array}{ccccc} A & & [A] & \xrightarrow{\text{init}_A} & [[A]] \\ f \downarrow & \text{List}(f) \downarrow & \downarrow & & \downarrow \text{List}(\text{List}(f)) \\ B & & [B] & \xrightarrow{\text{init}_B} & [[B]] \end{array}$$

证明。我们利用前面小节中针对列表函子定义的  $fmap$  来证明可交换性。对任意  $A$  中的  $n$  个元素  $a_1, a_2, \dots, a_n$ ，令  $B$  中的  $n$  个元素  $b_1, b_2, \dots, b_n$  满足： $f(a_1) = b_1, f(a_2) = b_2, \dots, f(a_n) = b_n$ 。我们有：

$$\begin{aligned} & \text{List}(\text{List}(f)) \circ \text{init}_A[a_1, \dots, a_n] \\ = & \text{fmap}_{[]} (f) \circ \text{init}_A[a_1, \dots, a_n] && \text{使用 } fmap \\ = & \text{fmap}_{[]} (f)[[], [a_1], \dots, [a_1, a_2, \dots, a_n]] && \text{inits 的定义} \\ = & \text{map}(\text{map } f)[[], [a_1], \dots, [a_1, a_2, \dots, a_n]] && \text{列表函子的 } fmap \text{ 等价于 } map \\ = & [\text{map } f[], \text{map } f[a_1], \dots, \text{map } f[a_1, a_2, \dots, a_n]] && \text{map 的定义} \\ = & [[], [f(a_1)], \dots, [f(a_1), f(a_2), \dots, f(a_n)]] && \text{对每个子列表应用 } map \text{ } f \\ = & [[], [b_1], \dots, [b_1, b_2, \dots, b_n]] && f \text{ 的定义} \\ = & \text{init}_B[b_1, b_2, \dots, b_n] && \text{反向用 } init \text{ 的定义} \\ = & \text{init}_B[f(a_1), f(a_2), \dots, f(a_n)] && \text{反向用 } f \text{ 的定义} \\ = & \text{init}_B \circ \text{map}(f)[a_1, a_2, \dots, a_n] && \text{反向用 } map \text{ } f \\ = & \text{init}_B \circ \text{fmap}_{[]} (f)[a_1, \dots, a_n] && \text{列表函子的 } fmap \text{ 等价于 } map \\ = & \text{init}_B \circ \text{List}(f)[a_1, \dots, a_n] \end{aligned}$$

□

所以  $\text{inits} : \text{List} \rightarrow \text{List} \circ \text{List}$  的确是一个自然变换。

**例 4.4.2.** 我们举的第二个例子叫做  $\text{safeHead}$ ，它的行为是安全地获取列表中的第一个元素。所谓“安全”，就是说它能够处理空列表  $\text{Nil}$  的情况。为此我们可以使用前面介绍过的可能函子  $\text{Maybe}$ 。它的定义如下：

$$\begin{aligned} \text{safeHead} &: [A] \rightarrow \text{Maybe } A \\ \text{safeHead } [] &= \text{Nothing} \\ \text{safeHead } (x : xs) &= \text{Just } x \end{aligned}$$

同样在集合范畴，任何类型  $A$  是一个对象，它索引的箭头  $\text{safeHead}$  为：

$$[A] \xrightarrow{\text{safeHead}_A} \text{Maybe } A$$

这里的两个函子分别是列表函子和可能函子。我们接下来要验证，对于任何箭头（函数） $A \xrightarrow{f} B$ ，下面的方形范畴图是可交换的：

$$\begin{array}{ccc}
 A & & [A] \xrightarrow{\text{safeHead}_A} \text{Maybe } A \\
 f \downarrow & \text{List}(f) \downarrow & \downarrow \text{Maybe}(f) \\
 B & & [B] \xrightarrow{\text{safeHead}_B} \text{Maybe } B
 \end{array}$$

也就是要证明：

$$\text{Maybe}(f) \circ \text{safeHead}_A = \text{safeHead}_B \circ \text{List}(f)$$

证明. 我们分两种情况证明。第一种情况是空列表：

$$\begin{aligned}
 & \text{Maybe}(f) \circ \text{safeHead}_A [] \\
 = & \text{Maybe}(f) \text{ Nothing} & \text{safeHead的定义} \\
 = & \text{Nothing} & \text{fmap } f \text{ Nothing的定义} \\
 = & \text{safeHead}_B [] & \text{反向用safeHead的定义} \\
 = & \text{safeHead}_B \circ \text{List}(f) [] & \text{反向用fmap } f [] \text{ 的定义}
 \end{aligned}$$

第二种情况是非空列表( $x : xs$ )：

$$\begin{aligned}
 & \text{Maybe}(f) \circ \text{safeHead}_A (x : xs) \\
 = & \text{Maybe}(f) (\text{Just } x) & \text{safeHead的定义} \\
 = & \text{Just } f(x) & \text{fmap } f \text{ Just } x \text{ 的定义} \\
 = & \text{safeHead}_B (f(x) : \text{fmap } f xs) & \text{反向用safeHead的定义} \\
 = & \text{safeHead}_B \circ \text{List}(f) (x : xs) & \text{反向用fmap } f (x : xs) \text{ 的定义}
 \end{aligned}$$

综合这两种情况，就证明了 $\text{safeHead} : \text{List} \rightarrow \text{Maybe}$ 的确是自然变换。□

我们总结一下这两个自然变换的例子。从范畴中的任意对象 $A$ 开始（对于集合范畴，相当于一个集合 $A$ ；对于编程，相当于一个类型 $A$ ），通过一个函子 $F$ ，将其映射为对象 $FA$ （对于集合范畴， $FA$ 是另一个集合；对于编程， $FA$ 是另一个类型），而另一函子 $G$ 将以映射为 $GA$ 。自然变换 $\phi$ 经由 $A$ 索引的箭头（在集合范畴中是一个映射；在编程中是一个函数）<sup>19</sup>，形如：

$$\phi_A : FA \rightarrow GA$$

现在，我们说不仅对 $A$ ，而是对所有对象进行抽象，这样就得到了一族箭头（在编程中，就是一个多态函数<sup>20</sup>）：

$$\phi : \forall A . FA \rightarrow GA$$

例如在Haskell中，可以这样写<sup>21</sup>：

```
phi :: forall a . F a -> G a
```

通常我们不需要明确标明`forall a`，这样自然变换就可以写为：

```
phi : F a -> G a
```

<sup>19</sup>也称在 $A$ 上的部分 (The component at  $A$ )

<sup>20</sup>我们可以联想到面向对象编程中的多态函数和泛型编程中的模板函数。

<sup>21</sup>Haskell中有一个`ExplicitForAll`的选项，我们在下一章介绍构造——叠加融合律 (build/fold fusion law) 时会再次遇到它。

回顾此前的两个例子，我们可以知道 $\text{inits}$ 和 $\text{safeHead}$ 的类型分别为：

```
inits :: [a] -> [[a]]
```

```
safeHead :: [a] -> Maybe a
```

它们仅仅是把 $\text{phi}$ 换成了各自的名字，把 $F, G$ 换成了各自的函子。

#### 4.4.2 自然同构

我们说自然变换是用来比较函子的。仿照抽象代数中同构的含义，我们需要定义什么情况下认为两个函子是“等价”的。

**定义 4.4.2.** 我们称两个函子 $F$ 和 $G$ 间的**自然同构**是一个自然变换：

$$F \xrightarrow{\phi} G$$

使得对每个源范畴中的对象 $A$ ，被索引的箭头

$$FA \xrightarrow{\phi_A} GA$$

都是目的范畴中的同构。

有时我们也说自然同构的两个函子是自然等价的。

我们举一个极端的例子 $\text{swap}$ 。对于任意两个对象的积 $A \times B$ ， $\text{swap}$ 将其对换为 $B \times A$ ：

$$\begin{aligned} \text{swap} : A \times B &\rightarrow B \times A \\ \text{swap}(a, b) &= (b, a) \end{aligned}$$

$\text{swap}$ 是一个自然变换，它作用于两个二元函子，恰巧这两个二元函子都是积函子，即： $F = G = \times$

由于是二元函子，所以对任意两个箭头 $A \xrightarrow{f} C$ 和 $B \xrightarrow{g} D$ ，我们需要下面的自然条件使得范畴图可交换：

$$(g \times f) \circ \text{swap}_{A \times B} = \text{swap}_{C \times D} \circ (f \times g)$$

$$B \xrightarrow{g} D$$

$$\begin{array}{ccc} A & & A \times B \xrightarrow{\text{swap}_{A \times B}} B \times A \\ f \downarrow & & f \times g \downarrow \\ C & & C \times D \xrightarrow{\text{swap}_{C \times D}} D \times C \end{array}$$

证明这一点很简单，只要任选两个对象的积 $(a, b)$ 和 $(c, d)$ 分别代入自然条件的左右两侧即可。我们把它留给读者作为练习。

虽然两个函子都是积函子，我们还是来证明一下它们是自然同构的。对于任何两个对象的积 $A \times B$ ，注意到：

$$\text{swap}_{A \times B} \circ \text{swap}_{B \times A} = id$$

也就是说 $swap$ 是一个一一映射，因此它在目标范畴中是同构的。这样就证明了自然同构。

以上的三个例子 $inits$ ,  $safeHead$ ,  $swap$ 都是多态函数，它们也都是自然变换。这不是一个巧合。在函数式编程中，所有的多态函数都是自然映射[41]。

### 练习 4.4

1. 证明 $swap$ 满足自然变换的条件 $(g \times f) \circ swap = swap \circ (f \times g)$
2. 证明多态函数 $length$ 是一个自然变换，其定义如下：

$$\begin{aligned} length : [A] &\rightarrow Int \\ length [] &= 0 \\ length (x : xs) &= 1 + length xs \end{aligned}$$

3. 自然变换也可以进行组合，考虑两个自然变换 $F \xrightarrow{\phi} G$ 和 $G \xrightarrow{\psi} H$ ，对于任意箭头 $A \xrightarrow{f} B$ ，试画出自然变换组合 $\phi \circ \psi$ 的范畴图，并列出可交换性的条件。

## 4.5 数据类型

我们已经简单介绍了范畴论中的最基本概念，包括范畴、函子、自然变换。接下来我们了解一下如何通过这些基本组件实现较复杂的数据类型。

### 4.5.1 起始对象和终止对象

让我们先认识两个最简单的数据类型，起始对象（initial object）和终止对象（terminal object或final object）。这两个对象也像左右手一样是对偶的，它们虽然简单，可是并不容易理解。

**定义 4.5.1.** 在任意范畴 $C$ 中，如果有一个特殊的对象 $S$ ，使得范畴中的每个对象 $A$ 都有一个唯一的箭头

$$S \longrightarrow A \qquad A \longrightarrow S$$

也就是说 $S$ 存在

$$\text{指向所有其它对象} \qquad \text{从所有其它对象指向自己}$$

的唯一箭头，我们称这个对象 $S$ 为

$$\text{起始对象} \qquad \text{终止对象}$$

习惯上，我们用0表示起始对象，1表示终止对象。所以对任何对象 $A$ 有：

$$0 \longrightarrow A \qquad A \longrightarrow 1$$

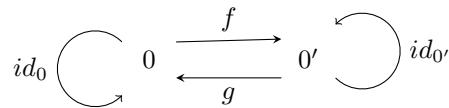
为什么说起始对象和终止对象是对偶的呢？如果 $S$ 是范畴 $C$ 中的起始对象，我们只要把所有的箭头都反向，则 $S$ 就成了范畴 $C^{op}$ 中的终止对象。一个范畴可能没有起始对象或终止对象，或者两个都没有，但如果有的话，起始对象或者终止对象在同构的意义下是唯一的。

我们只证明起始对象的同构唯一性，终止对象可以通过对偶性得到证明。

证明. 假设除0外还存在另一个起始对象 $0'$ 。考虑起始对象0, 根据定义一定存在从0指向 $0'$ 的箭头 $f$ ; 反过来, 当考虑起始对象 $0'$ 的时候, 也一定存在从 $0'$ 指向0的箭头 $g$ 。但是根据范畴公理, 一定也有从0指向自己的箭头 $id_0$ 和从 $0'$ 指向自己的箭头 $id_{0'}$ 。根据起始对象定义中箭头的唯一性, 一定有:

$$id_0 = f \circ g \quad \text{和} \quad id_{0'} = g \circ f$$

这一关系可从下面的范畴图看出。



这就证明了0和 $0'$ 是同构的。换言之, 在同构的意义下, 起始对象是唯一的。  $\square$

特别地, 如果一个对象既是起始对象, 又是终止对象, 我们称之为零对象 (zero object或null object)。一个范畴并不一定有零对象。

接下来我们通过一些例子, 由简到难说明起始对象和终止对象对应着什么样的数据类型。

**例 4.5.1.** 考虑一个偏序集, 箭头为序关系。如果一个偏序集存在最小值, 则这个最小值就是起始对象; 类似地, 如果存在最大值, 则最大值就是终止对象。例如《红楼梦》人物的有限集合{贾宝玉、贾政、贾代善、贾源}中, 序关系为祖先关系。这样贾宝玉就是起始对象, 贾源就是终止对象。而斐波那契数列组成的集合{1, 1, 2, 3, 5, 8, ...}, 序关系为小于等于。1是最小值, 是起始对象; 但是没有终止对象。注意斐波那契数列中有两个1, 但是它们在小于等于关系下是同构的。考虑全体实数 $R$ 构成的偏序集, 序关系为小于等于。它既没有最小值也没有最大值, 所以既没有起始对象也没有终止对象。考虑图4.6中《红楼梦》家族树中所有人物组成的偏序集, 序关系仍然是祖先关系。由于没有公共祖先, 这样就即不存在起始对象也不存在终止对象。

**例 4.5.2.** 考虑所有群构成的范畴**Grp**。所谓平凡群就是只含有单位元的群{e} (见上一章中的定义)。群间的箭头为态射。任何态射都把单位元映射为另一个群中的单位元。所以从{e}出发, 到任何群 $G$ 都有:  $e \mapsto e_G$ , 其中 $e_G$ 是 $G$ 中的单位元。因此从{e}出发到任何群都有唯一的箭头。

$$\{e\} \longrightarrow G$$

另一方面, 从任何一个群 $G$ 出发, 都存在一个唯一的态射, 将群中所有的元素都映射到 $e$ 上, 即:  $\forall x \in G, x \mapsto e$ 。因此从任何群 $G$ 都有到{e}的唯一箭头。

$$G \longrightarrow \{e\}$$

因此{e}即是起始对象, 又是终止对象。换言之, {e}是一个零对象。特别地观察下面的箭头组合:

$$G \longrightarrow \{e\} \longrightarrow G'$$

组合的结果是一个零箭头, 它将三个群 $G, \{e\}, G'$ 这样串起来: 所有 $G$ 中的元素都映射到 $e$ 上, 然后再进一步映射到 $e_{G'}$ 上。如下图所示。这就是零对象这个名称的由来。

平凡群中唯一的元素是什么名字不重要, 它可以是 $e$ , 可以是1 (例如整数乘法群的平凡子群), 可以是0 (例如整数加群的平凡子群), 可以是 $I$  (例如方阵乘法群中的单位矩阵), 可以是(1) (置换群中的恒等置换), 可以是 $id$ ……在同构的意义下它们都是等价的。

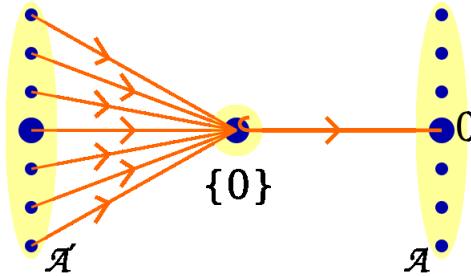


图 4.12: 零对象

**例 4.5.3.** 现在我们把难度增加一点，考虑全体集合的范畴 **Set**，箭头是全函数。终止对象相对容易找到，就是只含有一个元素的集合（singleton） $\{\star\}$ 。仿照群的例子，对于任何集合  $S$ ，我们让集合中所有的元素都映射到这个唯一的元素上去： $\forall x \in S, x \mapsto \star$ 。显然这个箭头是唯一的。

$$S \longrightarrow \{\star\}$$

但是问题来了，空集  $\emptyset$  怎样映射到  $\{\star\}$  上？事实上，空集是集合范畴中的起始对象。对于任何集合  $S$ ，我们都可以定义箭头

$$\emptyset \xrightarrow{f} S$$

这一点比较难理解，请停下来仔细思考一下。可以从空集的  $id$  箭头出发思考这个问题， $\emptyset \xrightarrow{id_{\emptyset}} \emptyset$ 。根据范畴公理， $id$  箭头是任何范畴中任何对象都有的。空集是到任何集合（包括它自己）都存在唯一箭头的对象。只不过这个箭头所表示的全函数没有任何参数。它不能写成  $f(x) : x \mapsto y$  的形式。

回答刚才的问题，由于空集到任何集合都有唯一的箭头，所以空集到  $\{\star\}$  也有唯一的箭头。这样从任何集合（包括空集）都存在到  $\{\star\}$  的唯一箭头，因此  $\{\star\}$  的确是集合范畴中的终止对象。

为什么不能像群一样，让只含有一个元素的集合  $\{\star\}$  成为起始对象呢？原因是，从  $\{\star\}$  到任意集合  $S$  的箭头可能不唯一。考虑有若干元素的集合  $S = \{x, y, z, \dots\}$ ，我们可以定义箭头（全函数） $\{\star\} \xrightarrow{\vec{x}} S$ ，将唯一元素映射到  $x$  上，也可以定义箭头  $\{\star\} \xrightarrow{\vec{y}} S$  和  $\{\star\} \xrightarrow{\vec{z}} S$ ，将其映射到其它元素上。

我们选择符号  $\{\star\}$  的用意是说，元素是什么不重要，只要它是只含有一个元素的集合（singleton set），在同构意义上就是等价的。

从终止对象  $\{\star\}$  到某一集合  $S$  的箭头

$$\{\star\} \longrightarrow S$$

也有着特定的含义。它表明我们可以从集合  $S$  中选出一个元素，例如上述的箭头  $\vec{x}$ ，从集合中选出元素  $x$ 。我们把这样的箭头叫做 **选择箭头**。

**例 4.5.4.** 在最后一个例子中，我们来到编程领域。编程环境的类型系统本质上是集合范畴 **Set**。一个集合相当于是数据类型。例如整形 **Int** 是所有整数的集合；布尔型是两个元素  $\{True, False\}$  组成的集合。既然集合范畴的终止对象是  $\{\star\}$ ，那么什么数据类型是编程中的终止对象呢？由于终止对象在同构的意义下是唯一的，所以任何只含有一个值的数据类型都是编程中的终止对象。

例如在 Haskell 中，特别定义了一个数据类型叫做 “0”，它只含有一个元素，也叫做 “0”。

```
data () = ()
```

在同构的意义下 $\{\star\} = \{()\}$ ，我们稍后会看到用“0”的好处。可以这样定义所有类型（集合）到这个终止对象的箭头（全函数）：

```
unit :: a -> ⊥
unit _ = ⊥
```

我们也可以自己定义终止对象——任何只有一个值的数据类型，在同构的意义上它和0是等价的<sup>22</sup>。

```
data Singleton = S

proj :: a -> Singleton
proj _ = S
```

看起来这像一个常函数，把任何值都映射到一个常数的值，但是并非所有常函数都指向终止对象，例如下面两个常函数：

```
yes :: a -> Bool
yes _ = True

no :: a -> Bool
no _ = False
```

因为数据类型Bool包含两个元素，所以任何其他数据类型都有yes, no两个箭头指向Bool，这样就不满足终止对象箭头的唯一性要求。

在集合范畴中，起始对象是空集 $\emptyset$ ，那么在编程中它对应着什么数据类型呢？既然数据类型本质上是集合，那么空集就相当于没有任何值的数据类型。在编程中，相当于我们声明了一个数据类型，但是却没有对它进行定义。

例如在Haskell中，声明了一个类型Void，但是却没有为它定义任何值：

```
data Void
```

这样Void就表示一个空集。但是起始对象必须有到所有其它对象的唯一箭头。这就要求我们必须定义从Void到所有其他类型的函数。

```
absurd :: Void -> a
absurd _ = undefined
```

函数的具体实现并不重要，因为根本不存在一个实际的参数来调用这个函数。因此在Haskell标准库的某个版本中还有这样的实现：

```
absurd :: Void -> a
absurd a = case a of {}
```

我们也可以自己定义起始对象，只要在同构的意义上是等价的就可以。方法就是只声明类型，不定义任何值。例如：

```
data Empty

f :: Empty -> a
f _ = undefined

iso :: Void -> Empty
```

<sup>22</sup>在其它编程环境，例如C++, Java, Scala中，可以定义“单子”（Single）对象，如果把强制类型转换作为箭头，则这也构成了终止对象。

```
iso_ = undefined
iso' :: Empty -> Void
iso' _ = undefined
```

我们明确定义了 $\text{iso}$ 和 $\text{iso}'$ ，使得 $\text{Empty}$ 和 $\text{Void}$ 是同构的。并且容易验证 $\text{iso} \circ \text{iso}' = \text{id}$ 。

集合范畴的例子中，我们说从终止对象到任何集合的箭头叫做选择箭头，它在编程中意味着什么呢？集合对应着类型，它表示我们可以从任何类型中选出特定的值。例如下面的函数从 $\text{Int}$ 中分别选出0和1：

```
zero :: () -> Int
zero () = 0

one :: () -> Int
one () = 1
```

现在我们看到用“0”的好处了，调用的时候，就像传入0个参数的函数那样： $\text{zero}()$ 返回0， $\text{one}()$ 返回1。

### 练习 4.5

- 在本节的例子中，我们说在一个偏序集中，如果存在最小值（或最大值），则最小值（或最大值）就是起始对象（或终止对象）。考虑全体偏序集构成的范畴 $\mathbf{Poset}$ ，如果存在起始对象，它是什么？如果存在终止对象，它是什么？
- 皮亚诺范畴 $\mathbf{Pno}$ （参见本章第一节的习题2）中，什么样的对象 $(A, f, z)$ 是起始对象？终止对象是什么？

#### 4.5.2 幂

起始对象和终止对象，相当于0和1，积和余积相当于 $\times$ 和 $+$ ，如果再有了指数（幂），我们就可以在范畴这个抽象层面中拥有和基本算术运算，甚至和多项式“同构”的强大工具。

观察一个定义在自然数上二元函数 $f(x, y) = z$ ，例如 $f(x, y) = x^2 + 2y + 1$ 。其类型为 $f : N \times N \rightarrow N$ 。我们会自然联想到，可以把这个类型写成：

$$f : N^2 \rightarrow N$$

对于函数 $f$ ，我们能否把 $N^2$ 看成一个对象，而不是两个参数呢？也就是说，看成 $f(x, y)$ ，而不是 $f(x, y)$ 。如果把参数看成一个“洞”，则前者是 $f \bullet$ ，然后把一个对 $(x, y)$ 填入洞中；而后者是 $f(\bullet, \bullet)$ ，然后分别把 $x, y$ 填入到两个洞中。另一方面，在第二章我们介绍了克里化，可以把 $f$ 看成 $f : N \rightarrow (N \rightarrow N)$ ，也就是传入 $x$ 后， $f x$ 会返回另一个函数，这个新函数把一个自然数映射到另一个自然数。但使用克里化，我们反而得不到幂的概念了。峰回路转，如果我们把克里化后返回的东西，也就是 $N \rightarrow N$ 看成一个事物，比方叫做“函数对象”会怎样呢？ $N$ 是一个无穷集，想起来比较困难。我们后退一步，构造一个简单点的例子——只有两个元素的有限集 $\text{Bool}$ ——来分析一下。

**例 4.5.5.** 存在无穷多个从 $\text{Bool}$ 到 $\text{Int}$ 的函数（箭头）组成的集合 $\{\text{Bool} \xrightarrow{f} \text{Int}\}$ ，我们从这个集合中挑选一个元素，也就是一个例子函数来看看：

```
ord : Bool -> Int
ord False = 0
ord True = 1
```

它的结果是一对整数(0,1)，这是一个代表，我们可以把所有上述箭头集合中的元素写成这样 的形式：

$$\begin{aligned} f : \text{Bool} &\rightarrow \text{Int} \\ f \text{ False} &= \dots \\ f \text{ True} &= \dots \end{aligned}$$

不管 $f$ 怎样千变万化，结果总是一对整数 $(a, b)$ 。我们可以说箭头 $f$ 的集合，同构于一对整数 $(a, b)$ 的集合，即：

$$\begin{aligned} \{\text{Bool} \xrightarrow{f} \text{Int}\} &= \{(a = f \text{ False}, b = f \text{ True})\} \\ &= \text{Int} \times \text{Int} \\ &= \text{Int}^2 = \text{Int}^{\text{Bool}} \end{aligned}$$

这告诉我们，箭头的集合，也就是箭头的类型<sup>23</sup>： $\text{Bool} \rightarrow \text{Int}$ ，相当于一个幂 $\text{Int}^{\text{Bool}}$ 。也就是说 $\text{Bool} \rightarrow \text{Int} = \text{Int}^{\text{Bool}}$ 。为什么在上面的推理中，我们可以用 $\text{Bool}$ 替换 $2$ ，从而把 $\text{Int}^2$ 变成 $\text{Int}^{\text{Bool}}$ 呢？原因是同构，上述的等号的真正含义是同构，所以严格来说，应该用“ $\cong$ ”符号，而不是“ $=$ ”。 $\text{Int}^2$ ，是两个整数积的自然记法，也就是一对 $\text{Int}$ 值的集合。一对 $\text{Int}$ 值的集合可以看成是从一个名叫 $2$ 的，拥有 $2$ 个元素的索引集合 $\{0, 1\}$ 到 $\text{Int}$ 的映射。

$$\{0, 1\} \xrightarrow{f} \text{Int} = \{(f(0), f(1))\} = \text{Int}^2$$

而索引集合 $2 = \{0, 1\}$ 正是 $\text{Bool}$ 的同构（例如在 $\text{ord}$ 函数下）。

**例 4.5.6.** 再举一个例子，考虑所有从字符 $\text{Char}$ 到布尔值 $\text{Bool}$ 的函数（箭头）集合 $\text{Char} \rightarrow \text{Bool}$ 。这个集合里有很多函数，例如 $\text{isDigit}(c)$ 可以判断传入的字符是否是数字。它可以实现为：

```
isDigit : Char → Bool
...
isDigit '0' = True
isDigit '1' = True
...
isDigit '9' = True
isDigit 'a' = False
isDigit 'b' = False
...
```

虽然这个实现比较的笨拙，但是它反映了一个事实：如果 $\text{Char}$ 中有 $256$ 个不同的字符（例如英语ASCII码），则函数 $\text{isDigit}$ 的结果本质上和一个 $256$ 元组 $(\text{False}, \dots, \text{True}, \text{True}, \dots, \text{True}, \text{False}, \dots)$ 同构。这个元组中对应数字字符的位置是 $\text{True}$ ，其余是 $\text{False}$ 。在 $\text{Char} \rightarrow \text{Bool}$ 中的众多函数中， $\text{isUpper}$ ,  $\text{isLower}$ ,  $\text{isWhitespace}$ 等等分别对应一个特定的元组。例如 $\text{isUpper}$ 对应的元组中，大写字符所在的位置上为 $\text{True}$ ，其余为 $\text{False}$ 。这样尽管有无穷多种方式来定义一个从 $\text{Char}$ 到 $\text{Bool}$ 的函数，但从结果上看，本质上只有 $2^{256}$ 个不同的函数。也就是说 $\text{Char} \rightarrow \text{Bool}$ 和 $256$ 元组的布尔值集合 $\text{Bool}^{\text{Char}}$ 同构。

现在我们把例子中的集合推进到无限集。在这一节的开头，我们说克里化后的函数 $f : N \rightarrow (N \rightarrow N)$ ，是从自然数 $N$ 到“函数对象”的函数。我们把函数对象记为 $(\Rightarrow)$ ，则 $f$ 的类型为 $N \xrightarrow{f} (\Rightarrow)$ 。也就是从索引集 $\{0, 1, \dots\}$ 到 $(\Rightarrow)$ 的映射，它是一个无穷长元组的集合，元组的每个值都是一个函数对象。记为 $(\Rightarrow)^N$ 。

<sup>23</sup>严格来说，我们应该去掉上面箭头两侧的大括号，增加大括号是为了容易理解

一般来说，我们把函数的集合 $f : B \rightarrow C$ 记为 $C^B$ 。从这个集合中取出一个函数 $f \in C^B$ ，然后在从集合 $B$ 中取出一个值 $b \in B$ ，我们可以用一个名叫 $apply$ 的函数将 $f$ 应用到 $b$ 上，从而得到一个 $C$ 中的值 $c = f(b)$ 。即<sup>24</sup>：

$$apply(f, b) = f(b)$$

聪明的读者可能会问：符号 $A$ 哪里去了？因为 $A$ 还有别的用处。我们还要把克里化考虑进来。对于二元函数 $g : A \times B \rightarrow C$ ，传入一个 $A$ 中的值 $a$ 后，就得到了一个克里化的函数 $g(a, \bullet) : B \rightarrow C$ 。它是一个函数对象，属于 $C^B$ 。这样，对于任何二元函数 $A \times B \xrightarrow{g} C$ ，都存在唯一的一元函数 $A \xrightarrow{\lambda g} C^B$ ，将 $a \in A$ 送入 $g(a, \bullet) : B \rightarrow C$ 中。我们称 $\lambda g$ 为 $g$ 的幂转换（exponential transpose）<sup>25</sup>。并且，有这样的关系：

$$apply(\lambda g(a), b) = g(a, b)$$

也就是说， $apply$ 把一个类型为 $C^B$ 的函数对象 $\lambda g(a)$ 与类型为 $B$ 的参数 $b$ 组合在一起，最终得到类型为 $C$ 的结果 $c$ 。所以箭头 $apply$ 的类型为：

$$C^B \times B \xrightarrow{apply} C$$

现在，我们终于可以给出完整的幂对象（Exponentials或Exponential object）定义了。

**定义 4.5.2.** 如果范畴 $\mathcal{C}$ 中存在终止对象和积，则一个幂对象是一对对象和箭头

$$(C^B, apply)$$

它们满足对于任何的对象 $A$ 和箭头 $A \times B \xrightarrow{g} C$ ，都存在唯一的转换箭头

$$A \xrightarrow{\lambda g} C^B$$

使得下面的范畴图可交换

$$\begin{array}{ccc} A & & A \times B \\ \downarrow \lambda g & & \downarrow \lambda g \times id_B \\ C^B & & C^B \times B \\ & & \xrightarrow{\quad apply \quad} C \end{array}$$

即：

$$apply \circ (\lambda g \times id_B) = g$$

第二章中我们没有给出克里化的完整定义，现在利用幂对象，我们可以给出 $curry$ 的定义了：

$$\begin{aligned} curry : (A \times B \rightarrow C) &\rightarrow A \rightarrow (C^B) \\ curry g &= \lambda g \end{aligned}$$

因此， $curry g$ 就是 $g$ 的幂转换。把这一关系代入上面的范畴图，我们有：

<sup>24</sup>有些文献，例如[43]第111-112页，[44]使用 $eval$ ，也就是求值（evaluation）作为名称，这里采用和[6]（第72页）一致的命名。这样和Lisp中的命名传统一致。

<sup>25</sup>有的文献中用 $g$ 来表示，根据第二章中对 $\lambda$ 的介绍，以及关于丘奇出版商的小插曲，我认为用 $\lambda g$ 更为传神，它表示 $a \mapsto \lambda \bullet \cdot g(a, \bullet)$ 。

$$\text{apply} \circ (\text{curry } g \times \text{id}) = g$$

换言之，我们就得到了这样的泛性性质：

$$f = \text{curry } g \quad \equiv \quad \text{apply} \circ (f \times \text{id}) = g$$

并且我们还能说明为什么幂转换箭头是唯一的，假设还存在另外的箭头  $A \xrightarrow{h} C^B$ ，使得  $\text{apply} \circ (h \times \text{id}) = g$ ，根据上述泛性性质，我们立刻得到  $h = \text{curry } g$ 。

我们还可以从范畴的角度来理解幂对象，在范畴  $\mathbf{C}$  中，如果固定对象  $B, C$ ，我们可以构造一个名叫  $\mathbf{Exp}$  的范畴，这个范畴中的对象是  $A \times B \rightarrow C$  这样的箭头，范畴中的箭头是这样定义的。

$$\begin{array}{ccc} A & & A \times B \xrightarrow{h} C \\ f \downarrow & & \downarrow j \\ D & & D \times B \xrightarrow{k} C \end{array}$$

如果在  $\mathbf{C}$  中存在  $A \xrightarrow{f} D$  的箭头，则  $\mathbf{Exp}$  中的箭头为  $h \xrightarrow{j} k$ 。当且仅当把上面范畴图中右侧的  $C$  合并时，箭头之间可交换：

$$\begin{array}{ccc} A & & A \times B \\ f \downarrow & & \downarrow f \times \text{id}_B \\ D & & D \times B \xrightarrow{k} C \end{array}$$

即： $k \circ (f \times \text{id}_B) = h$ 。在这个范畴  $\mathbf{Exp}$  中，存在一个终止对象，它恰好是  $C^B \times B \xrightarrow{\text{apply}} C$ 。我们现在来验证一下，根据幂对象的定义，从任何其它对象  $A \times B \xrightarrow{g} C$ ，都有到  $\text{apply}$  的箭头  $\lambda g = \text{curry } g$ 。另外根据终止对象的性质，从终止对象到自己的箭头一定是  $\text{id}$ ，所以我们有反射律：

$$\text{curry apply} = \text{id}$$

### 练习 4.6

1. 验证  $\mathbf{Exp}$  的确是一个范畴，指出  $\text{id}$  箭头和箭头的组合。
2. 反射律  $\text{curry apply} = \text{id}$  中， $\text{id}$  的下标是什么？请用另一种方法证明它。
3. 我们称下面的等式

$$(\text{curry } f) \circ g = \text{curry}(f \circ (g \times \text{id}))$$

为克里化的融合律。请画出它的范畴图并证明它。

### 4.5.3 笛卡尔闭和对象算术

有了起始对象0，终止对象1，余积代表的加法，积代表的乘法，再加上幂对象代表的指数幂，我们终于拥有了在范畴上进行运算甚至构造抽象多项式的能力。但是且慢，让我们停下来看一下这样能跑多远。正如第三章开头我们讲到的，必须时刻关注一个问题：“某一种抽象的适用范围有多大？什么情况下这一抽象会失效？”

并非所有的范畴都有起始或终止对象，也并非所有的范畴都有幂对象。如果一个范畴存在有限积、对于任何对象 $A$ 和 $B$ 都存在幂 $A^B$ ，我们称这样的范畴为**笛卡尔闭的**（Cartesian closed）。一个笛卡尔闭范畴必须包含：

1. 一个终止对象 (1)；
2. 任何一对对象都有积 ( $\times$ )；
3. 任何一对对象都有幂 ( $A^B$ )

我们既可以把终止对象1，想象成一个对象的零次幂： $A^0 = 1$ ，也可以把它想象成零个对象的积。非常幸运的是，编程时我们所在的范畴——集合和全函数组成的范畴是笛卡尔闭的。一个笛卡尔闭范畴可以作为简单类型 $\lambda$ -演算 (simply typed lambda calculus) 的数学模型（见第二章），从而成为所有带有类型的编程语言的基础([42]第148页)。

如果一个笛卡尔闭的范畴还同时支持终止对象的对偶——起始对象，和积的对偶——余积。并且支持积和余积的分配律，我们称其为**双笛卡尔闭范畴** (Bicartesian closed)：

5. 一个起始对象 (0)；
6. 任何一对对象都有余积 (+)；
7. 积可以从左右两侧分配到余积上：

$$\begin{aligned} A \times (B + C) &= A \times B + A \times C \\ (B + C) \times A &= B \times A + C \times A \end{aligned}$$

现在我们可以看看基本算术运算在一个笛卡尔闭的范畴上，特别是编程中都意味着什么了。这一理论称为**对象算术理论** (Equational theory)。

#### 4.5.3.1 0次幂

$$A^0 = 1$$

0代表了起始对象，1代表了终止对象， $A$ 的0次幂可以解读为类型为 $0 \rightarrow A$ 的所有箭头的集合。但既然0是起始对象，所以它到任何对象只有唯一的箭头。因此集合 $\{0 \rightarrow A\}$ 仅仅含有一个元素 (singleton)，而仅有一个元素的集合 $\{\star\}$ 恰巧是（集合范畴的）终止对象1。下面的推导中等号应理解为同构。

$$\begin{aligned} A^0 &= \{0 \rightarrow A\} && \text{幂对象的定义} \\ &= \{\star\} && \text{起始对象到任何对象的箭头唯一} \\ &= 1 && \{\star\} \text{是终止对象} \end{aligned}$$

这样，算术中的任何数的0次幂等于1在范畴中也得到了解读。

## 4.5.3.2 1的幂

$$1^A = 1$$

$1$ 代表了终止对象，所以幂对象 $1^A$ 表示所有从 $A$ 出发，到终止对象的箭头的集合 $\{A \rightarrow 1\}$ 。根据终止对象的定义，任何对象到终止对象只有唯一的箭头，所以这个箭头集合仅仅含有一个元素，而仅有一个元素的集合同构于 $\{\star\}$ 。它恰好又是（集合范畴中的）终止对象。

$$\begin{aligned} 1^A &= \{A \rightarrow 1\} && \text{幂对象的定义} \\ &= \{\star\} && \text{任何对象到终止对象的箭头唯一} \\ &= 1 && \{\star\} \text{是终止对象} \end{aligned}$$

## 4.5.3.3 1次幂

$$A^1 = A$$

这恰好是前面介绍过的“[选择箭头](#)”。 $1$ 是终止对象，所以幂 $A^1$ 表示了从终止对象到 $A$ 的箭头集合 $\{1 \rightarrow A\}$ 。如果 $A$ 是一个集合，我们可以针对集合中的任何元素 $a \in A$ 构造一个从终止对象 $1$ 到 $a$ 的一个函数：

$$f_a : 1 \rightarrow a$$

这种选择函数，能从集合 $A$ 中选出一个元素 $a$ 。所有从 $1$ 到 $A$ 中元素的选择函数的集合为 $\{f_a : 1 \mapsto a | a \in A\}$ ，它恰好就是集合 $1 \rightarrow A$ 。另一方面，集合 $\{f_a\}$ 是和 $A = \{a\}$ 一一对应的，也就是说它们是同构的。

$$\begin{aligned} A^1 &= \{1 \rightarrow A\} && \text{幂对象的定义} \\ &= \{f_a : 1 \mapsto a | a \in A\} && \text{从 } 1 \text{ 到 } A \text{ 中元素的映射的集合} \\ &= \{a | a \in A\} = A && \text{一一映射同构} \end{aligned}$$

## 4.5.3.4 幂的和

$$A^{B+C} = A^B \times A^C$$

幂对象 $A^{B+C}$ 表示从余积 $B + C$ 到 $A$ 的箭头的集合 $\{B + C \rightarrow A\}$ 。我们借助[Either B C](#)来帮助我们思考<sup>26</sup>，当我们实现任何[Either B C → A](#)函数时，都可以写成这样的形式：

$$\begin{aligned} f : \text{Either } B \text{ } C &\rightarrow A \\ f \text{ left } b &= \dots \\ f \text{ right } c &= \dots \end{aligned}$$

也就是说，任何这样的函数，都可以看作一对映射 $(b \mapsto a_1, c \mapsto a_2)$ ，而 $\{(b \mapsto a_1, c \mapsto a_2)\}$ 恰好是 $B \rightarrow A$ 和 $C \rightarrow A$ 的积。因此 $\{B + C \rightarrow A\} = \{B \rightarrow A\} \times \{C \rightarrow A\}$ 。另一方面，根据 $\{B \rightarrow A\}$ 可以表示为幂对象 $A^B$ ，同样 $\{C \rightarrow A\}$ 可以表示为 $A^C$ 。这样就解读了幂的和：

$$\begin{aligned} A^{B+C} &= \{B + C \rightarrow A\} && \text{幂对象的定义} \\ &= \{(b \mapsto a_1, c \mapsto a_2) | a_1, a_2 \in A, b \in B, c \in C\} && B \text{ 和 } C \text{ 到 } A \text{ 的箭头对} \\ &= \{B \rightarrow A\} \times \{C \rightarrow A\} && \text{笛卡尔积} \\ &= A^B \times A^C && \text{幂对象} \end{aligned}$$

<sup>26</sup>也可以用标记(tag)来推理：

$$\begin{aligned} f : B + C &\rightarrow A \\ f(b, 0) &= \dots \\ f(c, 1) &= \dots \end{aligned}$$

### 4.5.3.5 幂的幂

$$(A^B)^C = A^{B \times C}$$

先看右侧，幂对象  $A^{B \times C}$  实际就是二元函数  $B \times C \xrightarrow{g} A$  的集合。交换积的次序  $C \times B \xrightarrow{g \circ swap} A$ ，显然是一个自然同构（参见自然同构一节的  $swap$  自然变换）。如果克里化，就是  $\{curry(g \circ swap)\} = \{C \rightarrow A^B\}$ ，再一次用幂对象表示就是  $(A^B)^C$ 。

$$\begin{aligned} A^{B \times C} &= \{B \times C \xrightarrow{g} A\} && \text{幂对象的定义} \\ &= \{C \times B \xrightarrow{g \circ swap} A\} && \text{自然同构} \\ &= \{C \xrightarrow{curry(g \circ swap)} A^B\} && \text{克里化后仍然同构} \\ &= (A^B)^C && \text{幂对象} \end{aligned}$$

### 4.5.3.6 积的幂

$$(A \times B)^C = A^C \times B^C$$

幂对象  $(A \times B)^C$  是箭头  $C \rightarrow A \times B$  的集合，它可以认为是返回一对值的函数集合  $\{c \mapsto (a, b)\}$ ，其中  $c \in C, a \in A, b \in B$ 。它显然同构于  $\{(c \mapsto a, c \mapsto b)\}$ ，而这恰恰是箭头  $C \rightarrow A$  和  $C \rightarrow B$  的积。最后再分别将  $C \rightarrow A$  和  $C \rightarrow B$  表示为幂对象就得到了幂的积。

$$\begin{aligned} (A \times B)^C &= \{C \rightarrow A \times B\} && \text{幂对象的定义} \\ &= \{c \mapsto (a, b) | a \in A, b \in B, c \in C\} && \text{箭头的集合} \\ &= \{(c \mapsto a, c \mapsto b)\} && \text{箭头对} \\ &= \{C \rightarrow A\} \times \{C \rightarrow B\} && \text{笛卡尔积} \\ &= A^C \times B^C && \text{幂对象} \end{aligned}$$

### 4.5.4 多项式函子

以上介绍的在笛卡尔闭范畴上的算术主要是针对对象的。如果我们把函子考虑进来，会怎样呢？由于函子既作用于对象，也作用于箭头，这样就产生了多项式函子的概念。多项式函子是使用常函子（参见[函子的例子](#)中的“黑洞”函子）、积函子和余积函子递归构造的。

- 恒等函子  $id$  和常函子  $K_A$  是多项式函子；
- 若函子  $F$  和  $G$  是多项式函子，则它们的组合  $FG$ ，和  $F + G$  与积  $F \times G$  也是多项式函子。其中和与积定义如下：

$$\begin{aligned} (F + G)h &= Fh + Gh \\ (F \times G)h &= Fh \times Gh \end{aligned}$$

举一个例子，若函子  $F$  对于对象和箭头的行为是：

$$\begin{cases} \text{对象: } & FX = A + X \times A \\ \text{箭头: } & Fh = id_A + h \times id_A \end{cases}$$

其中  $A$  是某个固定的对象。则函子  $F$  是一个多项式函子。这是因为它可以表达为多项式：

$$F = K_A + (id \times K_A)$$

### 4.5.5 F-代数

为了构造复杂的带有递归结构的代数数据类型，我们还需要最后一块砖石。这就是F-代数（F-algebras）。观察抽象代数中的概念，如幺半群、群、环、域，它们不仅仅是抽象的对象，而且带有结构。正是这些结构之间的关系，使得它们区别于过去那些具体的对象，例如数、点、线、面。如果关系和结构研究清楚了，不仅仅是数、点、线、面，所有具有同样关系和结构的对象（用希尔伯特的比喻，连同桌子、椅子、啤酒杯）就都尽在掌握了。

**例 4.5.7.** 我们从比较简单的幺半群作为开始的例子，一步一步导出F-代数的概念。一个幺半群是一个集合 $M$ ，并且在集合上定义了可结合的二元运算和单位元。

如果用 $\oplus$ 表示二元运算， $1_M$ 表示单位元，则幺半群的两条公理可以描述如下：

$$\begin{cases} \text{结合性公理: } & (x \oplus y) \oplus z = x \oplus (y \oplus z), \forall x, y, z \in M \\ \text{单位元公理: } & x \oplus 1_M = 1_M \oplus x = x, \forall x \in M \end{cases}$$

第一步，我们将二元运算 $\oplus$ 表示为函数，将 $1_M$ 表示为选择函数，定义：

$$\begin{cases} m(x, y) = x \oplus y \\ e() = 1_M \end{cases}$$

这两种箭头的类型分别为：

$$\begin{cases} \text{二元运算: } & \text{类型 } M \times M \rightarrow M & \text{幂对象 } M^{M \times M} \\ \text{选择运算: } & 1 \rightarrow M & M^1 \end{cases}$$

第一种箭头是说，两个幺半群的元素进行二元运算，其结果仍是这个幺半群中的元素（二元运算是封闭的）；第二种箭头是“选择函数”， $1$ 是终止对象<sup>27</sup>。

现在我们就可以用函数 $m$ 和 $e$ 来替换幺半群公理了：

$$\begin{cases} \text{结合性公理: } & m(m(x, y), z) = m(x, m(y, z)), \forall x, y, z \in M \\ \text{单位元公理: } & m(x, e()) = m(e(), x) = x, \forall x \in M \end{cases}$$

第二步，把所有的 $x, y, z$ 等具体的元素和对象 $M$ 都去掉。这样就得到了纯用函数（箭头）表示的幺半群公理：

$$\begin{cases} \text{结合性公理: } & m \circ (m, id) = m \circ (id, m) \\ \text{单位元公理: } & m \circ (id, e) = m \circ (e, id) = id \end{cases}$$

这两条公理意味着下面的两个范畴图中的箭头可交换：

现在可以说，任何幺半群都可以用一个三元组 $(M, m, e)$ 来表示了。其中 $M$ 是集合， $m$ 是二元运算函数， $e$ 是单位元选择函数。

第三步， $(M, m, e)$ 不仅规定了幺半群的集合，还规定了在此集合上的二元运算和单位元，因此整个幺半群的代数结构就都确定了。组成幺半群的所有箭头，也就是代表所有可能的 $m$ 和 $e$ 的箭头集合，是两种幂对象的积

$$M^{M \times M} \times M^1 = M^{M \times M + 1}$$

我们再把右侧从幂对象写回箭头的形式，这样集合 $M$ 上定义的幺半群代数运算一定是下面的形式：

---

<sup>27</sup>我们借鉴Haskell中的符号，特意用“0”表示终止对象，在同构的意义下 $1 = \{\star\} = \{()\}$ 。这样的好处是， $e()$ 看起来像是个无参数的函数调用，实际接受了终止对象中的唯一元素 $0$ 作为参数。

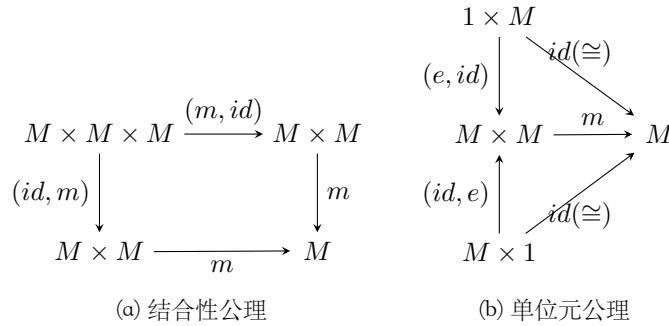


图 4.13: 幺半群公理对应的范畴图

$$\begin{array}{ccc} \alpha : 1 + M \times M & \longrightarrow & M \\ 1 & \longmapsto & 1 \\ (x, y) & \longmapsto & x \oplus y \end{array} \quad \begin{array}{l} \text{和 (余积)} \\ \text{单位元} \\ \text{二元运算} \end{array}$$

通过余积用和来表示这种关系就是 $\alpha = e + m$ , 用多项式函子来表示就是 $F M = 1 + M \times M$ 。总结一下, 对于幺半群这个例子, 幺半群上的代数结构由三部分组成:

1. 对象 $M$ , 就是用于携带幺半群上代数结构的集合, 叫做**携带对象** (carrier object) ;
2. 函子 $F$ , 定义了幺半群上的代数运算。是一个多项式函子 $F M = 1 + M \times M$ ;
3. 箭头 $F M \xrightarrow{\alpha} M$ , 它是单位元箭头 $e$ 和二元运算箭头 $m$ 的和 (余积),  $\alpha = e + m$ 。

我们称这样定义了一个幺半群的 $F$ -代数( $M, \alpha$ )。

例如在Haskell中, 可以这样定义一个 $F$ -代数箭头的类型:

```
type Algebra f a = f a -> a
```

这本质上是给箭头 $F A \rightarrow A$ 起了一个别名 (alias), 名叫 $\text{Algebra } F, A$ 。对于幺半群, 还需要定义一个函子<sup>28</sup>:

```
data MonoidF a = MEmptyF | MAppendF a a
```

当 $a$ 是字符串时, 我们可以这样定义一个 $F$ -代数( $\text{String}$ ,  $\text{evals}$ )的箭头实现:

```
evals :: Algebra MonoidF String
evals MEmpty = e []
evals (MAppendF s1 s2) = m s1 s2

e :: [] -> String
e [] = ""

m :: String -> String -> String
m = (++)
```

当然, 也可将 $e$ 和 $m$ 内嵌到 $\text{evals}$ 中简化实现为:

<sup>28</sup> Haskell中有一个对幺半群的标准定义 (见本章附录)。但这是一个幺半群代数结构的定义, 而不是幺半群函子。

```
evals :: Algebra MonoidF String
evals MEmpty = ""
evals (MAppendF s1 s2) = s1 ++ s2
```

`evals`是箭头 $\alpha : \text{FA} \rightarrow A$ 的一种实现，其中 $\text{F}$ 为`MonoidF`， $A$ 为`String`。当然还可以有别的实现，只要它能够满足幺半群的单位元公理和结合性公理。

**例 4.5.8.** 在幺半群之上，增加一条可逆性公理，就得到了群。我们仍然分步骤引出群的F-代数。首先列出群元素集合 $G$ 上的三条公理，我们用 $1_G$ 表示单位元，点号表示二元运算， $(\cdot)^{-1}$ 表示求逆元：

$$\begin{cases} \text{结合性公理: } & (x \cdot y) \cdot z = x \cdot (y \cdot z), \forall x, y, z \in G \\ \text{单位元公理: } & x \cdot 1_G = 1_G \cdot x = x, \forall x \in G \\ \text{可逆性公理: } & x \cdot x^{-1} = x^{-1} \cdot x = 1_G, \forall x \in G \end{cases}$$

第一步，将二元运算、求单位元、求逆元都表示为函数。定义：

$$\begin{cases} m(x, y) = x \cdot y \\ e() = 1_G \\ i(x) = x^{-1} \end{cases}$$

这三种函数的类型和幂对象分别为：

$$\begin{cases} \text{二元运算: } & G \times G \rightarrow G & G^{G \times G} \\ \text{选择运算: } & 1 \rightarrow G & G^1 \\ \text{求逆运算: } & G \times G & G^G \end{cases}$$

前两个箭头和幺半群一样，第三个箭头是说群元素的逆元仍是群中的元素。现在就可以用 $e, m, i$ 替换群公理中的 $1_G$ ，点号和逆元符号了。

$$\begin{cases} \text{结合性公理: } & m(m(x, y), z) = m(x, m(y, z)), \forall x, y, z \in G \\ \text{单位元公理: } & m(x, e()) = m(e(), x) = x, \forall x \in G \\ \text{可逆性公理: } & m(x, i(x)) = m(i(x), x) = e(), \forall x \in G \end{cases}$$

第二步，把所有的具体元素 $x, y, z$ 和群元素集合 $G$ 去掉。得到纯用箭头表示的群公理：

$$\begin{cases} \text{结合性公理: } & m \circ (m, id) = m \circ (id, m) \\ \text{单位元公理: } & m \circ (id, e) = m \circ (e, id) = id \\ \text{可逆性公理: } & m \circ (id, i) = m \circ (i, id) = e \end{cases}$$

这样，任何群都可以用一个四元组 $(G, m, e, i)$ 表示。

第三步，利用余积求 $m, e, i$ 的和 $\alpha = e + m + i$ ，通过多项式函子 $\text{FA} = 1 + A + A \times A$ ，并令 $A = G$ ，描述群 $G$ 上的代数运算。

$$\begin{array}{rcl} \alpha : 1 + A + A \times A & \longrightarrow & A \quad \text{和 (余积)} \\ 1 & \longmapsto & 1 \quad \text{单位元} \\ x & \longmapsto & x^{-1} \quad \text{逆元} \\ (x, y) & \longmapsto & x \cdot y \quad \text{二元运算} \end{array}$$

所以，群上的代数结构由三部分组成：

1. 携带对象 $G$ ，用于携带群上代数结构的集合；

2. 多项式函子  $\mathsf{F}A = 1 + A + A \times A$ , 定义了群上的代数运算;
3. 箭头  $\mathsf{F}A \xrightarrow{\alpha = e + m + i} A$ , 它是单位元箭头  $e$ 、二元运算箭头  $m$ 、逆元箭头  $i$  的和。

我们称这样定义了一个群上的F-代数  $(G, \alpha)$ 。

现在我们可以给出F-代数的定义了。在范畴论中,许多概念是对偶的,成对出现的。这就像是买一送一,我们定义了F-代数,就同时得到了F-余代数(F-coalgebra)<sup>29</sup>。关于F-余代数,我们暂时只给出定义。稍后将在第6章介绍无穷的概念时,再给出相应的例子。

**定义 4.5.3.** 如果  $\mathbf{C}$  是一个范畴,  $\mathbf{C} \xrightarrow{\mathsf{F}} \mathbf{C}$  是范畴  $\mathbf{C}$  上的一个自函子。对于范畴中的对象  $A$  和态射  $\alpha$ :

$$\mathsf{F}A \xrightarrow{\alpha} A \quad A \xrightarrow{\alpha} \mathsf{F}A$$

构成一对元组  $(A, \alpha)$  叫做

$$\begin{array}{c} \text{F-代数} \\ \text{F-余代数} \end{array}$$

其中  $A$ ,叫做携带对象。

我们可以把

$$\begin{array}{cc} \text{F-代数} & \text{F-余代数} \\ (A, \alpha) & (A, \alpha) \end{array}$$

本身看成对象。在上下文清楚的情况下,我们通常用二元组  $(A, \alpha)$  表示对象。两个对象间的箭头定义如下:

**定义 4.5.4.** F-态射是F-代数或F-余代数对象间的箭头:

$$(A, \alpha) \longrightarrow (B, \beta)$$

如果携带对象间的箭头  $A \xrightarrow{f} B$ ,使得下面的范畴图可交换:

$$\begin{array}{ccc} \mathsf{F}A & \xrightarrow{\alpha} & A \\ \mathsf{F}(f) \downarrow & & \downarrow f \\ \mathsf{F}B & \xrightarrow{\beta} & B \end{array} \quad \begin{array}{ccc} A & \xrightarrow{\alpha} & \mathsf{F}A \\ f \downarrow & & \downarrow \mathsf{F}(f) \\ B & \xrightarrow{\beta} & \mathsf{F}B \end{array}$$

即:

$$f \circ \alpha = \beta \circ \mathsf{F}(f) \quad \beta \circ f = \mathsf{F}(f) \circ \alpha$$

F-代数和F-态射, F-余代数和F-态射分别构成了

$$\begin{array}{cc} \text{F-代数范畴} \mathbf{Alg}(\mathsf{F}) & \text{F-余代数范畴} \mathbf{CoAlg}(\mathsf{F}) \end{array}$$

### 练习 4.7

1. 画出群的可逆性公理的范畴图。
2.  $p$ 是一个素数, 使用群的F-代数, 为整数模  $p$  乘法群(可以参考上一章) 定义一个  $\alpha$  箭头。
3. 参考上一章环的定义, 定义环的F-代数。
4. F-代数范畴上的  $id$  箭头是什么? 箭头组合是什么?

<sup>29</sup>也译作上代数和共代数。

### 4.5.5.1 递归和不动点

在第一章中，我们介绍了自然数和皮亚诺公理。并且介绍了和自然数同构的事物。实际上，我们可以把所有类似自然数的东西都用F-代数描述。

假设有一个集合 $A$ ，具有自然数的代数结构。例如第一章中介绍过的斐波那契数列，其中 $A$ 是数对( $\text{Int}, \text{Int}$ )。起始对象是 $(1, 1)$ ，后继函数是 $h(m, n) = (n, m + n)$ 。

为了用F-代数描述这一类代数结构，我们需要三样东西：函子，携带对象和 $\alpha$ 箭头。根据皮亚诺公理，自然数的函子应当这样定义：

```
data NatF A = ZeroF | SuccF A
```

这个函子是一个多项式函子 $\text{NatF}A = 1 + A$ 。现在我们提出一个问题。如果令 $A' = \text{NatF}A$ ，将其代入 $\text{NatF}$ 函子，那么 $\text{NatF}A'$ 是什么？应该是两重函子 $\text{NatF}(\text{NatF}A)$ 。我们还可以重复这一过程得到三重的 $\text{NatF}(\text{NatF}(\text{NatF}A))$ 。我们可以重复无穷多次，现在我们给重复无穷多次后产生类型 $\text{NatF}(\text{NatF}(\dots))$ 起名为 $\text{Nat}$ ，即：

data Nat =	$\text{NatF}(\text{NatF}(\dots))$	无穷重
=	$\text{ZeroF}   \text{SuccF}(\text{SuccF}(\dots))$	无穷重 $\text{SuccF}$
=	$\text{ZeroF}   \text{SuccF Nat}$	无穷重 $\text{SuccF}$ 的类型是 $\text{Nat}$
=	$\text{Zero}   \text{Succ Nat}$	重命名

这恰恰是我们在第一章定义的自然数类型。我们把 $\text{Nat}$ 和 $\text{Nat A}$ 并列写在一起：

```
data Nat = Zero | Succ Nat
```

```
data NatF A = ZeroF | SuccF A
```

这是函子层面的递归，在第二章中，我们曾经遇到过类似的概念。将自函子应用到自身无穷次产生了不动点：

$$\text{Fix } F = F(\text{Fix } F)$$

自函子 $F$ 的不动点是 $\text{Fix } F$ ，将 $F$ 应用到不动点上，得到的仍然是不动点。因此， $\text{Nat}$ 是函子 $\text{NatF}$ 的不动点。也就是说 $\text{Nat} = \text{Fix } \text{NatF}$ 。

### 练习 4.8

1. 可否把类自然数函子写成如下递归的形式？谈谈你的看法。

```
data NatF A = ZeroF | SuccF (NatF A)
```

2. 我们可以为 $\text{NatF} \text{Int} \rightarrow \text{Int}$ 定义一个 $\alpha$ 箭头，名叫 $\text{eval}$ ：

```
eval : NatF Int → Int
eval ZeroF = 0
eval (SuccF n) = n + 1
```

如果迭代地将 $A' = \text{NatF}A$ 代入 $\text{NatF}$ 函子 $n$ 次。我们把这样得到的函子记为 $\text{NatF}^n A$ 。试思考能否定义下面的 $\alpha$ 箭头：

```
eval : NatF^n Int → Int
```

#### 4.5.5.2 初始代数和向下态射

在F-代数组成范畴 $\mathbf{Alg}(F)$ 中，如果存在初始对象，它会有什么特殊的性质呢？初始对象到其它对象的唯一箭头表示什么样的关系呢？F-代数范畴中的所有对象可表示为二元组 $(A, \alpha)$ ，我们用类自然数的F-代数来举例。在 $\mathbf{Alg}(\mathbf{NatF})$ 中，所有的对象都表示为 $(A, \alpha)$ 。其中 $\mathbf{NatF}$ 的定义如同上小节；携带对象是 $A$ ，箭头是 $\mathbf{NatF}A \xrightarrow{\alpha} A$ 。

例如 $(\text{Int} \times \text{Int}, \text{fib})$ 是一个F-代数对象。其中携带对象是整数的积，箭头 $\mathbf{NatF}(\text{Int} \times \text{Int}) \xrightarrow{\text{fib}} (\text{Int} \times \text{Int})$ 的定义为：

$$\begin{aligned}\text{fib} &: \mathbf{NatF}(\text{Int} \times \text{Int}) \rightarrow (\text{Int} \times \text{Int}) \\ \text{fib ZeroF} &= (1, 1) \\ \text{fib } (\text{SuccF } (m, n)) &= (n, m + n)\end{aligned}$$

这是一种紧凑的定义。我们可以把它也写成和（余积）的形式 $\text{fib} = [\text{start}, \text{next}]$ ，其中 $\text{start}$ 总返回 $(1, 1)$ ，而 $\text{next}(m, n) = (n, m + n)$ 。

如果范畴 $\mathbf{NatF}$ 有起始对象 $0$ ，我们把它表示为二元组 $(I, i)$ 。起始对象到任何对象 $(A, \alpha)$ 都有唯一的箭头。即存在箭头 $I \xrightarrow{f} A$ ，使得下面的范畴图可交换<sup>30</sup>：

$$\begin{array}{ccc}\mathbf{NatF}A & \xrightarrow{\alpha} & A \\ \mathbf{NatF}(f) \uparrow & & \uparrow f \\ \mathbf{NatFI} & \xrightarrow{i} & I\end{array}$$

因为起始对象到任何对象都有唯一的箭头，所以它到自己的递归——由 $\mathbf{NatF}(\mathbf{NatFI})$ ——构成的对象也一定有唯一的箭头。具体来说这个递归的F-代数具有的三要素是：

1. 共同的函子 $\mathbf{NatF}$ ；
2. 携带对象是 $\mathbf{NatFI}$ ；
3. 箭头 $\mathbf{NatF}(\mathbf{NatFI}) \rightarrow \mathbf{NatFI}$ 。因为函子既作用于对象，也作用于箭头。所以我们可以通过函子 $\mathbf{NatF}$ ，将初始对象中的箭头*i*“举”上去。因此，这个递归的F-代数的箭头就是 $\mathbf{NatF}(i)$ 。

这样，这个递归的F-代数可以记为： $(\mathbf{NatFI}, \mathbf{NatF}(i))$ 。因为从初始对象到它也有唯一的箭头，所以存在 $I \xrightarrow{j} \mathbf{NatFI}$ ，使得下面的范畴图可交换：

$$\begin{array}{ccc}\mathbf{NatF}(\mathbf{NatFI}) & \xrightarrow{\mathbf{NatF}(i)} & \mathbf{NatFI} \\ \mathbf{NatF}(j) \uparrow & & \uparrow j \\ \mathbf{NatFI} & \xrightarrow{i} & I\end{array}$$

现在我们考虑沿着 $\mathbf{NatF}$ 递归路径上的两个箭头：

$$\mathbf{NatF}(\mathbf{NatFI}) \xrightarrow{\mathbf{NatF}(i)} \mathbf{NatFI} \xrightarrow{i} I$$

为了明显，我们把它画成弯的：

<sup>30</sup>起始的英文是initial，所以选择符号 $(I, i)$ 代表起始对象中的集合 $I$ 和箭头*i*

$$\begin{array}{ccc} \text{NatF}(\text{NatFI}) & \xrightarrow{\text{NatF}(i)} & \text{NatFI} \\ & & \downarrow i \\ & & I \end{array}$$

和上面的范畴图对比，我们发现*i*和*j*的方向相反。如果把它们两个组合起来，就得到了一个从*I*出发指回自己的箭头。并且由于起始对象到自己的唯一箭头就是*id*，所以：

$$i \circ j = id$$

同样的分析对NatFI也成立，因为起始对象到递归对象的箭头也是唯一的。所以从NatFI出发到*I*的箭头*i*，然后再从箭头*j*返回NatFI必然也是*id*：

$$j \circ i = id_{\text{NatFI}}$$

这说明NatFI和*I*是同构的。即：

$$\text{NatFI} = I$$

这恰好是上一小节介绍的不动点概念。这说明*I*是NatF的不动点。并且我们在上一小节已经知道NatF的不动点就是Nat。所以：

$$\text{NatF Nat} = \text{Nat}$$

这一结论，和1889年皮亚诺给出算术公理时的结论完全一样，任何满足皮亚诺公理的结构(*A*, [c, f])，都存在从自然数(N, [zero, succ])到这一结构的唯一同构。它将自然数n映射到f<sup>n</sup>(c) = f(f(...f(c)..))上。现在，我们可以给出**初始代数** (initial algebra) 的定义了。

**定义 4.5.5.** F-代数范畴Alg(F)中，如果存在初始对象，则这一初始对象叫做**初始代数**。

在集合全函数范畴中，许多函子，包括多态函子都存在初始代数。我们略去了初始代数的存在性证明，读者可以参考[45]。给定函子F我们可以通过它的不动点得到这一F-代数中的初始对象。



兰贝克 (Joachim Lambek) 1922 - 2014

1968年，兰贝克 (Lambek) 最早指出*i*是一个同构映射，并且称初始代数(*I*, *i*)是函子F的不动点[46]。现在这一事实被称作**兰贝克定理**。

F-代数中若存在初始代数(*I*, *i*)，则存在到任何其它代数(*A*, *f*)的唯一态射。我们把从*I*到*A*的态射记为(|f|)，使得下面的范畴图可交换：

$$\begin{array}{ccc}
 \mathsf{F}I & \xrightarrow{i} & I \\
 \mathsf{F}(\mathbb{f}) \downarrow & & \downarrow (\mathbb{f}) \\
 \mathsf{F}A & \xrightarrow{f} & A
 \end{array}$$

若  $h = (\mathbb{f})$ , 当且仅当  $h \circ i = f \circ \mathsf{F}(h)$

我们称箭头  $(\mathbb{f})$  为 **向下态射** (catamorphism, 来自希腊语  $\kappa\alpha\kappa\alpha$ , 意思是向下)。它两侧的括号 “ $(\ )$ ” 像一对香蕉, 因此被称为“香蕉括号”。

向下态射的强大之处在于, 它那能把一个非递归结构上的函数  $f$ , 转换为在递归结构上的函数  $(\mathbb{f})$ 。从而构造复杂的递归计算。我们仍然用自然数来举例子。

自然数函子  $\mathsf{NatF}$  是非递归的, 而其起始代数自然数  $\mathsf{Nat}$  的定义是递归的:

```

data NatF A = ZeroF | SuccF A

data Nat = Zero | Succ Nat

```

所以箭头  $\mathsf{NatFA} \xrightarrow{f} A$  是非递归的。一个 *cata* (向下态射) 能够从  $f$  构造出在递归的  $\mathsf{Nat}$  上进行计算的箭头  $\mathsf{Nat} \rightarrow A$ 。所以 *cata* 的类型应为:

$$(\mathsf{NatFA} \xrightarrow{f} A) \xrightarrow{\text{cata}} (\mathsf{Nat} \rightarrow A)$$

所以 (克里化的) *cata(f)* 应该能够作用于  $\mathsf{Nat}$  的两种值  $\mathsf{Zero}$  或者  $\mathsf{Succ}\ n$ 。我们可以根据这点定义出 *cata* 函数:

$$\begin{aligned}
 \text{cata } f \text{ Zero} &= f \text{ ZeroF} \\
 \text{cata } f \text{ (Succ } n\text{)} &= f(\text{SuccF} (\text{cata } f\ n))
 \end{aligned}$$

这一定义的第一行处理递归的边界情况。针对  $\mathsf{Nat}$  的零值  $\mathsf{Zero}$ , 我们转而用  $f$  对  $\mathsf{NatFA}$  的零值  $\mathsf{ZeroF}$  求值; 对于递归情况, 也就是  $\mathsf{Nat}$  的值  $\mathsf{Succ}\ n$ , 我们先递归地用  $\text{cata } f\ n$  求出类型为  $A$  的值  $a$ , 然后用  $\text{SuccF}\ a$  将其转化为  $\mathsf{NatFA}$  类型的值, 最后再把  $f$  作用于其上。这个自然数的向下态射对于任何携带对象  $A$  都成立, 它是非常通用的。我们进一步举两个具体的例子。第一个例子是将任何  $\mathsf{Nat}$  的值转换回  $\mathsf{Int}$ 。

```

toInt :: Nat → Int
toInt = cata eval where
  eval :: NatF Int → Int
  eval ZeroF = 0
  eval (SuccF x) = x + 1

```

这样  $\text{toInt Zero}$  会得到 0, 而  $\text{toInt} (\text{Succ} (\text{Succ} (\text{Succ Zero})))$  会得到 3。为了方便验证, 可以再定义一个辅助函数:

```

fromInt :: Int → Nat
fromInt 0 = Zero
fromInt n = Succ (fromInt (n-1))

```

于任何整数  $n$ , 有  $n = (\text{toInt} \circ \text{fromInt})\ n$ 。这个例子看起来很平常, 我们来看第二个关于斐波那契数列的例子。

```

toFib :: Nat → (Integer, Integer)
toFib = cata fibAlg where
  fibAlg :: NatF (Integer, Integer) → (Integer, Integer)
  fibAlg ZeroF = (1, 1)
  fibAlg (SuccF (m, n)) = (n, m + n)

```

我们特意将此前定义的函数 $fib$ 改名为 $fibAlg$ ，意思是说，从斐波那契的代数关系（非递归的），我们通过向下态射获得了递归计算斐波那契数列的能力。这样 $toFib\ Zero$ 会得到数对 $(1, 1)$ ，而 $toFib\ (Succ\ (Succ\ (Succ\ Zero)))$ 会得到数对 $(3, 5)$ 。下面的辅助函数，可以计算第 $n$ 个斐波那契数。

$$fibAt = fst \circ toFib \circ fromInt$$

事实上，对于任何满足皮亚诺公理的类自然数代数结构 $(A, c + f)$ ，我们都可以利用向下态射和初始代数 $(\text{Nat}, zero + succ)$ ，得到能够进行递归计算的 $\langle c + f \rangle$ <sup>31</sup>。我们接下来证明这一点。考虑下面的范畴图。

$$\begin{array}{ccc} \text{NatFNat} & \xrightarrow{[zero, succ]} & \text{Nat} \\ \text{NatF}(h) \downarrow & & \downarrow h \\ \text{NatFA} & \xrightarrow{[c, f]} & A \end{array}$$

由于向下态射使得这个范畴图可交换。

$$\begin{aligned} h \circ [zero, succ] &= [c, f] \circ \text{NatF}(h) && \text{可交换} \\ \Rightarrow h \circ [zero, succ] &= [c, f] \circ (id + h) && \text{多项式函子} \\ \Rightarrow h \circ [zero, succ] &= [c \circ id, f \circ h] && \text{右侧用余积的吸收律} \\ \Rightarrow [h \circ zero, h \circ succ] &= [c, f \circ h] && \text{左侧用余积的融合律} \\ \Rightarrow \begin{cases} h \circ zero = c \\ h \circ succ = f \circ h \end{cases} & & & \\ \Rightarrow \begin{cases} h Zero = c \\ h (\text{Succ } n) = f(h(n)) \end{cases} & & & \\ \Rightarrow \begin{cases} h(0) = c \\ h(n + 1) = f(h(n)) \end{cases} & & & \end{aligned}$$

这恰恰是自然数叠加的定义：

$$h = foldn(c, f)$$

也就是说，自然数上的向下态射 $\langle c + f \rangle = foldn(c, f)$ 。因此，对于斐波那契数列，我们可以用

$$\langle fibAlg \rangle = \langle start + next \rangle = foldn(start, next)$$

来进行代数计算。读到这里，大家会有一种转了一大圈又回到了第一章的感觉。这是一种认知的螺旋上升。第一章中，我们是用归纳和抽象的方法得到了这个结论，现在，我们从更高的层面，把抽象的规律应用到具体的问题上得到了同样的结论。

<sup>31</sup>我们用符号 $\langle c + f \rangle$ 是因为 $\langle [c, f] \rangle$ 看起来有太多重括号了。

### 4.5.5.3 代数数据类型

使用初始F-代数，我们可以定义更多的代数数据类型。本小节我们介绍列表和二叉树的例子。

**例 4.5.9.** 在第一章中，我们给出的列表定义为：

```
data List A = Nil | Cons A (List A)
```

而相应的非递归函子为：

```
data ListF A B = NilF | ConsF A B
```

实际上 $\text{List}$ 是函子 $\text{ListF}$ 的不动点。我们可以这样验证。令 $B' = \text{ListF } A \ B$ ，然后递归地应用到自己上无穷多次。把这个结果叫做 $\text{Fix}(\text{ListF } A)$ ：

$$\begin{aligned}\text{Fix}(\text{ListF } A) &= \text{ListF } A(\text{Fix}(\text{ListF } A)) && \text{不动点的定义} \\ &= \text{ListF } A(\text{ListF } A(\dots)) && \text{展开} \\ &= \text{NilF} | \text{ConsF } A (\text{ConsF } A(\dots)) && \text{定义} \\ &= \text{NilF} | \text{ConsF } A (\text{Fix}(\text{ListF } A)) && \text{反向用不动点}\end{aligned}$$

和 $\text{List } A$ 的定义对比，我们有：

$$\text{List } A = \text{Fix}(\text{ListF } A)$$

在 $\text{ListF}$ 这个函子下，固定 $A$ ，对于任何携带对象 $B$ 可以定义箭头

$$\text{ListF } A \ B \xrightarrow{f} B$$

这样就构成了列表的F-代数。并且我们知道初始代数就是 $(\text{List}, [\text{nil}, \text{cons}])$ 。所以就会有向下态射。

$$\begin{array}{ccc} \text{ListF } A (\text{List } A) & \xrightarrow{[\text{nil}, \text{cons}]} & (\text{List } A) \\ (\text{ListF } A)(h) \downarrow & & \downarrow h \\ \text{ListF } A \ B & \xrightarrow{[c, f]} & B \end{array}$$

如果我们有一个非递归的计算 $f$ ，就可以利用向下态射构成针对递归列表的计算。

```
cata :: (ListF a b → b) → (List a → b)
cata f Nil = f NilF
cata f (Cons x xs) = f (ConsF x (cata f xs))
```

例如，可以定义计算列表长度的代数规则：

```
len :: (List a) → Int
len = cata lenAlg where
  lenAlg :: ListF a Int → Int
  lenAlg NilF = 0
  lenAlg (ConsF _ n) = n + 1
```

这样 $\text{len Zero}$ 会得到0，而 $\text{len}(\text{Cons } 1 (\text{Cons } 1 \text{ Zero}))$ 则得到2。可以定义一个辅助函数从中括弧简记法转换为 $\text{List}$ ：

```
fromList :: [a] -> List a
fromList [] = Nil
fromList (x:xs) = Cons x (fromList xs)
```

这样就可以通过 $\text{len}(\text{fromList}[1, 1, 2, 3, 5, 8])$ 计算列表的长度。

通过改变代数规则 $f$ , 还可以得到其它针对列表的计算。下面的例子把列表中所有的元素累加起来:

```
sum :: (Num a) => (List a) -> a
sum = cata sumAlg where
  sumAlg :: (Num a) => ListF a a -> a
  sumAlg NilF = 0
  sumAlg (ConsF x y) = x + y
```

我们接下来利用列表的范畴图证明, 列表的向下态射本质上就是叠加计算 $\text{foldr}$ 。列表函子本质上是多项式函子 $\text{ListF } A \ B = 1 + A \times B$ 。由于范畴图可交换, 所以:

$$\begin{aligned} h \circ [\text{nil}, \text{cons}] &= [c, f] \circ (\text{ListF } A)(h) && \text{可交换} \\ \Rightarrow h \circ [\text{nil}, \text{cons}] &= [c, f] \circ (\text{id} + (\text{id} \times h)) && \text{多项式函子} \\ \Rightarrow h \circ [\text{nil}, \text{cons}] &= [c \circ \text{id}, f \circ (\text{id} \times h)] && \text{右侧用余积的吸收律} \\ \Rightarrow [h \circ \text{nil}, h \circ \text{cons}] &= [c, f \circ (\text{id} \times h)] && \text{左侧用余积的融合律} \\ \Rightarrow &\begin{cases} h \circ \text{nil} = c \\ h \circ \text{cons} = f \circ (\text{id} \times h) \end{cases} \\ \Rightarrow &\begin{cases} h \text{ Nil} = c \\ h (\text{Cons } a x) = f(a, h(x)) \end{cases} \end{aligned}$$

这恰恰是列表叠加的定义:

$$h = \text{foldr}(c, f)$$

也就是说, 列表F-代数的向下态射 $(c+f) = \text{foldr}(c, f)$ 。因此, 求长度的运算是 $\text{foldr}(0, (a, b) \mapsto b+1)$ , 累加的运算是 $\text{foldr}(0, +)$ 。

**例 4.5.10.** 在第二章中, 我们定义了二叉树的类型为:

```
data Tree A = Nil | Br A (Tree A) (Tree A)
```

所有类似二叉树的结构也可以通过F-代数来描述。首先定义函子:

```
data TreeF A B = NilF | BrF A B B
```

携带对象是 $B$ , 而初始代数是 $(\text{Tree } A, [\text{nil}, \text{branch}])$ 。我们把证明作为本小节的练习。这样二叉树的向下态射表示为如下范畴图:

$$\begin{array}{ccc} \text{TreeF } A (\text{Tree } A) & \xrightarrow{[\text{nil}, \text{branch}]} & (\text{Tree } A) \\ (\text{TreeF } A)(h) \downarrow & & \downarrow h \\ \text{TreeF } A B & \xrightarrow{[c, f]} & B \end{array}$$

我们可以定义二叉树的向下态射函数。它接受一个F-代数的箭头  $\text{TreeF } A B \xrightarrow{f} B$ , 返回一个从  $\text{Tree } A \rightarrow B$  的函数:

$$\begin{aligned} cata : (\text{TreeF } A B \rightarrow B) &\rightarrow (\text{Tree } A \rightarrow B) \\ cata f Nil &= f Nil F \\ cata f (Br k l r) &= f (Br F k (cata f l) (cata f r)) \end{aligned}$$

如果定义了一个将二叉树中元素累加起来的代数运算, 就可以利用向下态射递归地应用到任意二叉树上:

$$\begin{aligned} sum : \text{Tree } A &\rightarrow A \\ sum &= cata \circ sumAlg \\ \text{其中: } &\begin{cases} sumAlg : \text{TreeF } A B \rightarrow B \\ sumAlg Nil F = 0_B \\ sumAlg (Br k l r) = k + l + r \end{cases} \end{aligned}$$

接下来, 我们证明二叉树的向下态射就相当于二叉树的叠加操作  $foldt$ 。固定对象  $A$ , 二叉树的函子  $\text{TreeF } A B$  是携带对象  $B$  的一个多项式函子。即:  $\text{TreeF } A B = 1 + (A \times B \times B)$ , 对于箭头  $h$ , 为:  $(\text{TreeF } A)(h) = id + (id_A \times h \times h)$ 。

由于上面的范畴图可交换, 所以有:

$$\begin{aligned} h \circ [nil, branch] &= [c, f] \circ (\text{TreeF } A)(h) && \text{可交换} \\ \Rightarrow h \circ [nil, branch] &= [c, f] \circ (id + (id_A \times h \times h)) && \text{多项式函子} \\ \Rightarrow h \circ [nil, branch] &= [c \circ id, f \circ (id_A \times h \times h)] && \text{右侧用余积的吸收律} \\ \Rightarrow [h \circ nil, h \circ branch] &= [c, f \circ (id_A \times h)] && \text{左侧用余积的融合律} \\ \Rightarrow &\begin{cases} h \circ nil = c \\ h \circ branch = f \circ (id_A \times h \times h) \end{cases} \\ \Rightarrow &\begin{cases} h Nil = c \\ h (Br k l r) = f(k, h(l), h(r)) \end{cases} \end{aligned}$$

这恰恰是二叉树叠加的定义  $h = foldt(c, f)$ , 其中:

$$\begin{cases} foldt c h Nil = c \\ foldt c h (Br k l r) = h(k, foldt c h l, foldt c h r) \end{cases}$$

因此, 累加二叉树中元素的运算可以用叠加表示为  $foldt(0_B, (\bullet + \bullet + \bullet))$ 。

### 练习 4.9

- 对二叉树函子  $\text{TreeF } A B$ , 固定  $A$ , 利用不动点验证  $(\text{Tree } A, [nil, branch])$  是初始代数。

## 4.6 小节

本章介绍了范畴论中最基本的概念, 包括范畴、函子、自然变换。并且还介绍了积和余积、起始对象和终止对象、幂以及F-代数这些工具来构造较为复杂的代数结构。作为本章的结尾, 我们来解读最开始那段关于叠加的代码[47]:

```
foldr f z t = appEndo (foldMap (Endo . f) t) z
```

所有的这一切的背后都是为了抽象，为了使得 $foldr$ 能跳出列表，在任何可叠加的结构上进行运算。我们先看看传统的列表叠加定义：

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

其中 $f$ 是一个二元操作，如果我们把它改写为 $\oplus$ ，把 $z$ 改写为单位元 $e$ ，根据这个定义， $foldr (\oplus) e [a, b, c]$ 就可以展开成为：

$$a \oplus (b \oplus (c \oplus e))$$

这让我们联想到幺半群， $foldr$ 相当于在幺半群上重复进行二元操作。在Haskell的抽象幺半群定义中，除了单位元和二元操作外，还定义了将一列元素“累加”的操作。用我们的 $\oplus$ 和 $e$ 符号，这个定义相当于：

$$\begin{aligned} concat_M &: [M] \rightarrow M \\ concat_M &= foldr (\oplus) e \end{aligned}$$

在Haskell中这个函数的名字叫做`mconcat`。它的意思是说，对于任何幺半群 $M$ ， $concat$ 可以将一个 $M$ 中元素的列表，通过二元运算和单位元叠加计算到一起。例如字符串可以看作是幺半群，单位元是空串，而二元运算是连接。所以 $concat_M ["Hello", "String" "Monoid"]$ 就得到：

```
"Hello" ++ ("String" ++ ("Monoid" ++ "")) = "HelloStringMonoid"
```

现在我们可以对任何幺半群元素进行累加了。但能否让它变得更通用呢？如果有一列元素，它们虽然不是幺半群中的元素，但是如果我们将它们转换为幺半群，就仍然可以进行累加。将某一类型的列表映射为幺半群列表恰巧就是函子的行为，确切地说我们把箭头 $A \xrightarrow{g} M$ ，通过列表函子“举”到 $List(g)$ 。然后再执行累加：

$$\begin{array}{ccccc} [A] & \xrightarrow{fmap g} & [M] & \xrightarrow{concat} & M \\ \text{List} \uparrow & & \uparrow \text{List} & & \\ A & \xrightarrow{g} & M & & \end{array}$$

$$\begin{aligned} foldMap &: (A \rightarrow M) \rightarrow [A] \rightarrow M \\ foldMap g &= concat_M \circ fmap g \end{aligned}$$

可是美中不足的是，对于任何传入 $foldr$ 的二元组合函数 $f : A \rightarrow B \rightarrow B$ ，如果 $B$ 不是幺半群，我们仍然无法进行叠加。现在考虑 $f$ 的克里化形式 $f : A \rightarrow (B \rightarrow B)$ ，我们发现以箭头 $B \rightarrow B$ 为对象，可以组成一个幺半群。其中单位元是恒等箭头 $id$ ，而二元运算是函数组合。为此，我们把这种 $B \rightarrow B$ 的箭头通过函子封装成一个类型（集合）：

```
newtype Endo B = Endo (B -> B)
```

并且定义一个函数，从 $Endo B \xrightarrow{appEndo} B$ ：

$$appEndo(Endo a) = a$$

然后我们规定`Endo`是一个幺半群，单位元是 $id$ ，二元运算是函数组合。在Haskell中，这相当于以下代码：

```
instance Monoid Endo where
    mempty = Endo id
    Endo f `mappend` Endo g = Endo (f ∘ g)
```

现在任给一个二元组合函数 $f$ , 我们都可以把它利用 $foldMap$ 叠加到 $Endo$ 么半群上去了:

$$\begin{aligned} foldCompose : (A \rightarrow (B \rightarrow B)) &\rightarrow [A] \rightarrow Endo B \\ foldCompose f &= foldMap (Endo.f) \end{aligned}$$

这样如果计算 $foldCompose f [a, b, c]$ 则展开为:

$$\begin{aligned} &Endo(f a) \oplus (Endo(f b) \oplus (Endo(f c) \oplus Endo(id))) \\ &= Endo(f a \oplus (f b \oplus (f c \oplus id))) \end{aligned}$$

下面是一个具体的例子

$$\begin{aligned} &foldCompose (+) [1, 2, 3] \\ \Rightarrow &foldMap (Endo \circ (+)) [1, 2, 3] \\ \Rightarrow &concat_M ( fmap(Endo \circ (+))) [1, 2, 3] \\ \Rightarrow &concat_M ( fmapEndo [(+1), (+2), (+3)]) \\ \Rightarrow &concat_M [Endo (+1), Endo (+2), Endo (+3)] \\ \Rightarrow &Endo ((+1) \circ (+2) \circ (+3)) \\ \Rightarrow &Endo (+6) \end{aligned}$$

所以, 最后一步我们要把 $Endo$ 中的结果取回。具体到这个例子, 就是用 $appEndo$ 把(+6)拿出, 然后把它应用到传入 $foldr$ 的初始值 $z$ 上:

$$\begin{aligned} foldr f z xs &= appEndo (foldCompose f xs) z \\ &= appEndo (foldMap (Endo \circ f) xs) z \end{aligned}$$

这就是我们在本章开头给出的 $foldr$ 定义。在Haskell中, 还专门定义了类型 $Foldable$ , 对任意数据结构, 使用者可以选择实现 $foldMap$ 或者 $foldr$ 。具体请参见本章附录。

## 4.7 扩展阅读

限于篇幅, 我们不可能仅仅在一章中完成范畴论的介绍。本章的内容不过是冰山的一个小角。范畴论的核心精神是抽象。初始对象和终止对象, 积和余积这些对偶的概念, 它们本质上可以进一步抽象到更高层次的对偶——极限 (limits) 和余极限 (colimits)。我们没能介绍伴随 (junction) 的概念, 没能介绍米田定理 (Yoneda lemma), 也没有能够解读单子。我希望这一章除能够起到抛砖引玉的作用, 引导读者去阅读和了解更多的内容。范畴论的创始人之一麦克兰恩的编写的教材[\[48\]](#)是这个领域的经典。其目标读者是数学家, 普通读者读起来有些艰深。西蒙森的《范畴论引论》[\[39\]](#)和史密斯的[\[43\]](#)更适合初学者。对于有编程背景的读者, 迈尔维斯基的《程序员的范畴论》[\[42\]](#)是本不错的参考书。里面配有很多Haskell示例代码和一些C++的类比实现。但是这本书仅仅在集合全函数范畴 (确切地说是Hask范畴) 内讨论。伯德的《编程中的代数》[\[6\]](#)更加全面地介绍了编程中的范畴原理。

## 4.8 附录: 例子代码

函子的定义:

```
class Functor f where
  fmap      :: (a → b) → f a → f b
```

Maybe函子的定义：

```
instance Functor Maybe where
  fmap _ Nothing     = Nothing
  fmap f (Just a)   = Just (f a)
```

使用Maybe搜索二叉搜索树，并将结果转换成二进制：

```
lookup Nil _ = Nothing
lookup (Node l k r) x | x < k = lookup l x
                      | x > k = lookup r x
                      | otherwise = Just k

lookupBin = (fmap binary) ∘ lookup
```

二元函子的定义：

```
class Bifunctor f where
  bimap :: (a → c) → (b → d) → f a b → f c d
```

积函子和余积函子的定义：

```
instance Bifunctor (,) where
  bimap f g (x, y) = (f x, g y)

instance Bifunctor Either where
  bimap f _ (Left a) = Left (f a)
  bimap _ g (Right b) = Right (g b)
```

*curry*及其反向变换*uncurry*的定义：

```
curry      :: ((a, b) → c) → a → b → c
curry f x y = f (x, y)

uncurry    :: (a → b → c) → ((a, b) → c)
uncurry f (x, y) = f x y
```

幺半群的定义：

```
class Semigroup a => Monoid a where
  mempty  :: a

  mappend :: a → a → a
  mappend = (<>)

  mconcat :: [a] → a
  mconcat = foldr mappend mempty
```

可叠加类型（Foldable）的定义：

```
newtype Endo a = Endo { appEndo :: a → a }

class Foldable t where
```

```
foldr :: (a -> b -> b) -> b -> t a -> b  
foldr f z t = appEndo (foldMap (Endo . f) t) z  
  
foldMap :: Monoid m => (a -> m) -> t a -> m  
foldMap f = foldr (mappend . f) mempty
```

## 第5章 推导

3表示 $2+1$ ，4表示 $3+1$ 。所以接下来（虽然证明很长），4等于 $2+2$ 。因此，数学知识不再是神秘的。

——罗素

记得在中学数学课上，老师会在黑板上写一个有很多字母的式子，然后让同学们化简。有人会自告奋勇站到讲台上，拿起粉笔在黑板上推导。合并同类项、因式分解……各种办法都可以用。这个过程就像是变魔术，最后往往得到意想不到的简单结果。当然也有卡住或者绕圈子的时候，老师总是耐心地提示，引导我们找到思路。

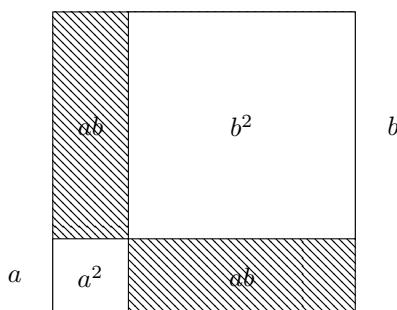
这样的经历好像就发生在眼前，一方面，满手的粉笔灰让同学们体会到老师的不易，另一方面，那种推理的神秘强力量让我感到它的强大。我总是希望知道更多的公式，这样就能在化简或者推导时派上用场。

这种推导的神奇之处在于，我们不用特别关心这些公式或者定理在当时场景下的具体含义。就像摆弄积木一样，从散落的各种零件，最后搭建起一个有趣玩具。这些公式和定理相互组装到一起，最后引向一个有趣的结

果。看到 $a^2 + 2ab + b^2$ 就会把它变换为 $(a + b)^2$ ，就像把两块积木插到一起那样自然，我们不用在推导时强迫自己回想这个公式的几何意义。



彭罗斯三角形



$$\longleftrightarrow a + b \longrightarrow$$

图 5.2:  $(a + b)^2 = a^2 + 2ab + b^2$  的几何意义

本章中，我们用两个例子说明如何进行编程中的推导。每个例子都既用直观的方法给出分析和解释，也用推导的方式给出解法。这就像 $(a + b)^2$ 的情形。一方面我们可以用几何的直观，将其理解为一大一小两个小正方形和两个相等的矩形的面积；另一方面，我们也可以用纯推导一步一步得出同样的结果。

$$\begin{aligned}
 (a + b)^2 &= (a + b)(a + b) && \text{二次方的定义} \\
 &= a(a + b) + b(a + b) && \text{乘法分配律} \\
 &= a^2 + ab + ba + b^2 && \text{再次用分配律} \\
 &= a^2 + 2ab + b^2 && \text{合并同类项} ab \text{ 和 } ba
 \end{aligned}$$

## 5.1 叠加——构建的融合

我们要举的第一个例子是叠加——构建的融合（foldr/build fusion law）。2015年Java在其1.8版本中加入了lambda表达式并且提供了一系列支持函数式编程的工具。但是有人很快发现，尽管一连串地函数调用表达能力很强，简洁优雅，但是性能会下降很多。原因之一就是这些串起来函数调用产生了大量中间结果。这些中间结果往往不是一两个简单的数值，而通常是列表、容器这样规模很大的结构。这些结构被下一个函数消费使用，然后就丢弃了。但是接下来会产生另一个同等规模的结构。这种产生——一次性消费——丢弃——再产生的过程，沿着函数调用链一环一环地重复，造成了计算的负担。

例如[49]，我们想判断一个列表中的每个元素是否都满足某个条件。可以这样进行定义：

$$all(p, xs) = and(map(p, xs))$$

传入 $all(prime, [2, 3, 5, 7, 11, 13, 17, 19, ...])$ 就可以判断是否列表中都是素数。但是这个实现的效率却不高。首先 $map(prime, xs)$ 会产生一个和 $xs$ 同样长度的列表，列表中的每个元素是一个布尔值[True, True, ...]，每个布尔值表示对应的元素是不是素数。然后这个布尔值列表传入 $and$ 函数，检查是否存在False。最后 $xs$ 和布尔值列表都被丢弃，而仅仅返回一个布尔值作为最终结果。

下面是另一种定义，它能够避免产生中间的布尔值列表：

$$\begin{aligned}
 all(p, xs) &= h(xs) \\
 \begin{cases} h(\emptyset) = \text{True} \\ h(x : xs) = p(x) \wedge h(xs) \end{cases}
 \end{aligned}$$

虽然这个实现不产生中间结果，可是和前面的 $and(map(p, xs))$ 比起来，既冗长又不直观。有没有什么办法，鱼和熊掌兼得，即不丧失直观性，又能避免低效的实现呢？我们发现有些变换满足这一要求。例如：

$$map\ sqrt\ (map\ abs\ xs) = map\ (sqrt \circ abs)\ xs$$

先把列表中每个元素取绝对值构成一列新数，然后再把这列数中的每个开方。这和把列表中每个数先取绝对值然后再立即开方后构成一列新数等价。由此我们可以得到一个转换规则：

$$map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs \tag{5.1}$$

但是这样的规则太多了，我们无法全部把它们列出。并且在千变万化的程序中，我们无法一眼就看出应该用哪一条规则优化。吉尔（Gill）、朗奇布瑞（Launchbury）和佩顿·琼斯（Peyton Jones）在1993年提出了一个方法，他们从列表最本质的构造和叠加操作入手，找到了优化的规律。

### 5.1.1 列表的叠加操作

我们在第一章就给出过列表的叠加操作，它的定义为：

$$\begin{aligned} foldr \oplus z [] &= z \\ foldr \oplus z (x : xs) &= x \oplus (foldr \oplus z xs) \end{aligned}$$

展开就是：

$$foldr \oplus z [x_1, x_2, \dots, x_n] = x_1 \oplus (x_2 \oplus (\dots (x_n \oplus z) \dots)) \quad (5.2)$$

许多列表相关的操作都可以用叠加来定义。我们接下来给出一些典型的例子：

1. 累加：

$$sum = foldr + 0$$

2. 前面提到的`and`函数，计算一个布尔值列表中的所有元素的逻辑与：

$$and = foldr \wedge True$$

这是因为：

$$and [x_1, x_2, \dots, x_n] = x_1 \wedge (x_2 \wedge (\dots (x_n \wedge True) \dots))$$

3. 在一个列表中查找某一个元素是否存在：

$$elem x xs = foldr (a b \mapsto (a = x) \vee b) False xs$$

4. 逐一映射：

$$\begin{aligned} map f xs &= foldr (x ys \mapsto f(x) : ys) [] xs \\ &= foldr ((:) \circ f) [] xs \end{aligned}$$

5. 用某一条件过滤列表中的元素：

$$filter f xs = foldr (x ys \mapsto \begin{cases} f(x) : & x : ys \\ \text{否则} : & ys \end{cases}) [] xs$$

6. 两个列表连接：

$$xs ++ ys = foldr (:) ys xs$$

这是因为：

$$[x_1, x_2, \dots, x_n] ++ ys = x_1 : (x_2 : (\dots (x_n : ys) \dots))$$

7. 多个列表连接：

$$concat xss = foldr ++ [] xss$$

叠加操作是如此基本，如果我们能把列表的叠加操作的化简规律找到，就找到了几乎所有列表操作的化简规律。

### 5.1.2 叠加——构建融合律

现在我们考虑，如果把空列表 $[]$ （即Nil）和连接操作“ $:$ ”（即Cons）进行叠加会产生什么结果。

$$\text{foldr}(:) [] [x_1, x_2, \dots, x_n] = x_1 : (x_2 : (\dots(x_n : []))\dots) \quad (5.3)$$

这回我们得到了列表本身。你也许想到了上一章介绍的不动点，我们稍后会回到这个话题。换言之，如果我们有一个运算 $g$ ，它能够从一个起始值，例如 $[]$ ，和一个二元组合运算，例如“ $:$ ”，产生一个列表。我们可以定义这个列表构造过程 $build$ ：

$$build(g) = g((:), []) \quad (5.4)$$

接着，如果用另一个起始值 $z$ 和二元组合运算 $f$ ，对这一列表进行叠加，其结果就相当于用 $z$ 替换 $[]$ ，用“ $f$ ”替换“ $(:)$ ”，然后直接调用过程 $g$ 。

$$\text{foldr}(f, z, build(g)) = g(f, z) \quad (5.5)$$

写成无参数括号的形式就是：

$$\text{foldr } f \ z \ (build \ g) = g \ f \ z \quad (5.6)$$

我们称这一结果为**叠加——构建融合定律**。

在继续深入介绍前，让我们先看一些具体的例子。考虑如何计算从 $a$ 到 $b$ 间的整数和 $sum([a, a + 1, \dots, b - 1, b])$ 。为此我们可以先产生从 $a$ 到 $b$ 之间的所有整数 $a, a + 1, a + 2, \dots, b - 1, b$ ，例如使用下面的方法：

$$range(a, b) = \begin{cases} a > b : & [] \\ \text{否则} : & a : range(a + 1, b) \end{cases}$$

这样 $range(1, 5)$ 就产生列表 $[1, 2, 3, 4, 5]$ 。现在我们只要把这个列表中的元素累加起来就得到答案了：

$$sum(range(a, b))$$

接下来关键的一步，我们把 $range$ 中的起始值 $[]$ 和二元组合运算 $(:)$ 抽出成为参数：

$$range'(a, b, \oplus, z) = \begin{cases} a > b : & z \\ \text{否则} : & a \oplus range'(a + 1, b, \oplus, z) \end{cases}$$

我们甚至可以把 $range'$ 的后两个参数克里化：

$$range' \ a \ b = f \ c \mapsto \begin{cases} a > b : & c \\ \text{否则} : & f \ a \ (range'(a + 1) \ b \ f \ c) \end{cases}$$

这样原来的 $range$ 就可以用 $range'$ 和 $build$ 表示了：

$$range(a, b) = build(range'(a, b))$$

写下来我们用融合律化简累加和的计算：

$$\begin{aligned} sum(range(a, b)) &= sum(build(range'(a, b))) && \text{代入} \\ &= \text{foldr } (+) 0 (build (range' a b)) && \text{用叠加表示累加} \\ &= range' a b (+) 0 && \text{使用融合律} \end{aligned}$$

这样就完成了化简，避免产生了中间列表。优化了算法。我们可以看一下最后算法的效果：

$$\text{range}' a b (+) 0 = \begin{cases} a > b : & 0 \\ \text{否则} : & a + \text{range}'(a + 1, b, (+), 0) \end{cases}$$

### 5.1.3 列表的构建形式

为了方便使用融合律，我们可以把常见的能够产生新列表操作都写为 $\text{build} \dots \text{foldr}$ 的形式。这样当用叠加操作和这种形式的操作组合起来时， $\text{foldr} \dots (\text{build} \dots \text{foldr})$ ，就可以使用融合律化简。

1. 首先是最简单的操作——构造空列表：

$$[] = \text{build}(f z \mapsto z)$$

我们可以代入 $\text{build}$ 的定义(5.4)来验证这个定义：

证明.

$$\begin{aligned} \text{build}(f z \mapsto z) &= (f z \mapsto z)(:) [] && \text{build的定义} \\ &= (:) [] \mapsto [] && \beta - \text{规约, 参见第二章} \\ &= [] \end{aligned}$$

□

2. 接下来是列表的链接（Cons）操作：

$$x : xs = \text{build}(f z \mapsto f x (\text{foldr } f z xs))$$

我们来验证一下：

证明.

$$\begin{aligned} &\text{build}(f z \mapsto f x (\text{foldr } f z xs)) \\ &= (f z \mapsto f x (\text{foldr } f z xs))(:) [] && \text{build的定义} \\ &= x : (\text{foldr } (:) [] xs) && \beta - \text{规约} \\ &= x : xs && \text{由(5.3), 叠加的不动点} \end{aligned}$$

□

3. 然后是列表的连接：

$$xs ++ ys = \text{build}(f z \mapsto \text{foldr } f (\text{foldr } f z ys) xs)$$

证明.

$$\begin{aligned} &\text{build}(f z \mapsto \text{foldr } f (\text{foldr } f z ys) xs) \\ &= (f z \mapsto \text{foldr } f (\text{foldr } f z ys) xs)(:) [] && \text{build的定义} \\ &= \text{foldr } (:) (\text{foldr } (:) [] ys) xs && \beta - \text{规约} \\ &= \text{foldr } (:) ys xs && \text{对内层用叠加的不动点} \\ &= xs ++ ys && \text{由(6), 列表的连接} \end{aligned}$$

□

以下操作我们只列出结果，而把它们的证明作为本小节的练习。

4. 多个列表的连接:

$$\text{concat } xss = \text{build } (f z \mapsto \text{foldr } (xs x \mapsto \text{foldr } f xxs) xss)$$

5. 一一映射产生新列表:

$$\text{map } f xs = \text{build } (\oplus z \mapsto \text{foldr } (y ys \mapsto (f y) \oplus ys) z xs)$$

6. 过滤元素产生新列表:

$$\text{filter } f xs = \text{build } (\oplus z \mapsto \text{foldr } (x xs' \mapsto \begin{cases} f(x) : & x \oplus xs' \\ \text{否则:} & xs' \end{cases}) z xs)$$

7. 重复产生同一元素的（无穷）列表:

$$\text{repeat } x = \text{build } (\oplus z \mapsto \text{let } r = x \oplus r \text{ in } r)$$

#### 5.1.4 使用融合律化简

接下来我们使用刚刚打造好的融合律来化简计算。首先是本节开始时举过的例子:  $\text{all}(p, xs) = \text{and}(\text{map}(p, xs))$

**例 5.1.1.** 我们把 $\text{and}$ 改为叠加形式, 把 $\text{map}$ 改为构建形式就可以进行化简了。

$$\begin{aligned} \text{all}(p, xs) &= \text{and}(\text{map}(p, xs)) && \text{原始定义} \\ &= \text{foldr } \wedge \text{True map}(p, xs) && \text{and的叠加形式} \\ &= \textbf{foldr } \wedge \text{True build } (\oplus z \mapsto \\ &\quad \text{foldr } (x ys \mapsto p(x) \oplus ys) z xs) && \text{map的构建形式} \\ &= (\oplus z \mapsto \text{foldr } (x ys \mapsto p(x) \oplus ys) z xs) \wedge \text{True} && \text{融合律} \\ &= \text{foldr } (x ys \mapsto p(x) \wedge ys) \text{True } xs && \beta - \text{规约} \end{aligned}$$

如果定义一个 $\text{first}$ 函数, 它将一个函数 $f$ 应用到一对值中的前一个上, 即:

$$(\text{first } f) x y = f(x) y$$

这样就可以进一步化简为:

$$\text{all } p = \text{foldr } (\wedge) \circ (\text{first } p) \text{ True}$$

**例 5.1.2.** 我们举的第二个例子是把若干词语添加上空格, 然后连接成句子。这样的文字处理过程通常叫做 $\text{join}$ , 简单起见, 我们在句子最后也添上一个空格。一个典型的定义如下:

$$\text{join}(ws) = \text{concat}(\text{map}(w \mapsto w ++ [']'), ws))$$

这个定义简单直观, 它先用逐一映射, 在每个单词的末尾添加空格, 得到一个新的单词列表, 然后再把这个列表连接成一个字符串。但是这个定义的性能不佳。在单词末尾添加空格是一个昂贵的计算, 需要先移动到每个单词的末尾, 然后再使用字符串连接操作。其次有多少单词, 逐一映射就会产生多长的新列表。最后这些中间结果都被丢弃了。接下来我们用融合律优化这一计算。

```

join(ws)
{定义}
= concat(map(w ↦ w ++ ['], ws))
{把concat写为构建形式}
= build(f z ↦
    foldr(x y ↦ foldr f y x) z map(w ↦ w ++ ['], ws))
{把map写为构建形式}
= build(f z ↦
    foldr(x y ↦ foldr f y x) z (build(f' z' ↦
        foldr(w b ↦ f'(w ↦ w ++ [']) b) z' ws)))
{融合律}
= build(f z ↦
    foldr(w b ↦ (x y ↦ foldr f y x) (w ++ [']) b) z ws)
{β - 规约x, y}
= build(f z ↦
    foldr(w b ↦ foldr f b (w ++ ['])) z ws)
{把++写为构建形式}
= build(f z ↦
    foldr(w b ↦
        foldr f b (build(f' z' ↦
            foldr f' (foldr f' z' [']) w))) z ws)
{融合律}
= build(f z ↦
    foldr(w b ↦
        foldr(foldr f b [']) w) z ws)
{据build的定义, 将(:)和[]代入}
= foldr(w b ↦ foldr(:) (foldr(:) b [']) w) [] ws
{对黑体字部分求值}
= foldr(w b ↦ foldr(:) ('' : b) w) [] ws

```

因此最后化简的结果为:

$$\text{join}(ws) = \text{foldr}(w b \mapsto \text{foldr}(:) ('' : b) w) [] ws$$

我们还可以据此把叠加操作展开, 得到一个可读性和性能都好的定义:

$$\begin{cases} \text{join} [] = [] \\ \text{join} (w : ws) = h w \end{cases}$$

其中:  $\begin{cases} h [] = '' : \text{join} ws \\ h (x : xs) = x : h xs \end{cases}$

由于 $\text{concat} \circ \text{map}(f)$ 十分常见, 很多编程环境的标准库已经提供了这样的优化实现。例如Haskell中的`concatMap`, Java和Scala中的`flatMap`。

第二个例子虽然展示了融合律的强大, 但是也暴露出了一个问题。推导过程机械繁琐、容易出错。这恰恰是机器, 而非人类擅长的事情。有些编程环境, 例如Haskell已经把融合律实现在编译器内部[49]。这样只要我们把列表的常见操作用构建——叠加形式定义好, 机器就可以替代我们利用融合律进行上述化简, 从而得到优化的程序, 避免中间结果和多余的递归<sup>1</sup>。随着时间的推移, 更多的编译器会逐渐支持这一优化工具。

<sup>1</sup>例如Haskell标准库已经用构建——叠加形式实现了大多数列表操作。

### 5.1.5 类型限制

每当我们打造出抽象的工具，就要思考它的适用范围，了解它什么情况下会失效。对于融合律也是如此。考虑下面的矛盾结果：

$$\begin{aligned} & \mathbf{foldr}\ f\ z\ (\mathbf{build}\ (c\ n \mapsto [0])) \\ = & (c\ n \mapsto [0])\ f\ z && \text{融合律} \\ = & [0] && \beta - \text{规约} \end{aligned}$$

另一方面：

$$\begin{aligned} & \mathbf{foldr}\ f\ z\ (\mathbf{build}\ (c\ n \mapsto [0])) \\ = & \mathbf{foldr}\ f\ z\ ((c\ n \mapsto [0])\ (:) \emptyset) && \mathbf{build} \text{ 的定义} \\ = & \mathbf{foldr}\ f\ z\ [0] && \beta - \text{规约} \\ = & f(0, z) && \text{叠加展开} \end{aligned}$$

显然  $f(0, z)$  不等于  $[0]$ ，连它们两个的类型都不同<sup>2</sup>。造成这一矛盾结果的原因是  $(c\ n \mapsto [0])$  不是一个真正用  $c$  和  $n$  构造结果的函数。为此我们需要对融合律  $\mathbf{foldr}\ f\ z\ (\mathbf{build}\ g) = g\ f\ z$  中  $g$  的类型做出限制。它的第一个参数既可以接受  $c$ ，也可以接受  $f$ ，换言之，它接受一个多态的二元运算  $\forall A. \forall B. A \times B \rightarrow B$ ；同理它的第二个参数也是一个多态类型  $B$  的起始值，并产生一个类型为  $B$  的结果。我们把二元运算写为克里化的形式就得到：

$$g : \forall A. (\forall B. (A \rightarrow B \rightarrow B) \rightarrow B \rightarrow B)$$

容易看出，上面反例中的类型是： $\forall A. (\forall B. (A \rightarrow B \rightarrow B) \rightarrow B \rightarrow [Int])$ 。不满足我们的类型限制。与此对应，构建函数  $\mathbf{build}$  的类型为：

$$\mathbf{build} : \forall A. (\forall B. (A \rightarrow B \rightarrow B) \rightarrow B \rightarrow B) \rightarrow \text{List } A$$

由于里面有两个多态的类型  $A$  和  $B$ ，因此它被称为二阶多态函数（rank-2 type polymorphic）。

### 5.1.6 用范畴论推导融合律

叠加——构建融合律可以用范畴理论推导出来并进行扩展。一旦把范畴论作为理论工具，人们就发现叠加——构建融合仅仅是众多融合规则中的一个。现在这些规则统一被称为“短路融合”（shortcut fusion）。它们在编译器优化，程序库优化中发挥了重要的作用。我们无法在这一章中对它们做全面的介绍。读者可以参考[51]深入了解短路融合的理论和实践方法。

在上一章中，我们介绍了 F-代数，特别是初始代数和向下态射。由于初始代数是起始对象，所以它到任何其他代数都有唯一的箭头。如下图所示：

$$\begin{array}{ccc} \mathsf{F}I & \xrightarrow{i} & I \\ \mathsf{F}(\mathbb{a}) \downarrow & & \downarrow (\mathbb{a}) \\ \mathsf{F}A & \xrightarrow{a} & A \end{array}$$

从初始代数  $(I, i)$  到另一代数  $(A, a)$  的箭头可以通过向下态射  $(\mathbb{a})$  来表示。如果还存在一个不同的 F-代数  $(B, b)$  对象，并且有从  $A$  到  $B$  的箭头  $A \xrightarrow{h} B$ ，就可以在上面的范畴图下方把  $(B, b)$  也画出：

---

<sup>2</sup>除非极特殊情况  $f = (:)$ ,  $z = \emptyset$ 。

$$\begin{array}{ccc}
 \mathsf{F}I & \xrightarrow{i} & I \\
 \mathsf{F}(a) \downarrow & & \downarrow (a) \\
 \mathsf{F}A & \xrightarrow{a} & A \\
 \mathsf{F}(h) \downarrow & & \downarrow h \\
 \mathsf{F}B & \xrightarrow{b} & B
 \end{array}$$

但由于 $(I, i)$ 是起始对象，所以它到 $(B, b)$ 也必然存在唯一的箭头。由此可知，必然存在从 $I$ 到 $B$ 的箭头，可以用向下态射 $\langle b \rangle$ 表示出来。如下图：

$$\begin{array}{ccc}
 \mathsf{F}I & \xrightarrow{i} & I \\
 \mathsf{F}(a) \downarrow & & \downarrow (a) \\
 \mathsf{F}A & \xrightarrow{a} & A \\
 \mathsf{F}(h) \downarrow & & \downarrow h \\
 \mathsf{F}B & \xrightarrow{b} & B
 \end{array}$$

$\langle b \rangle$

从这个范畴图可以看出，如果存在 $h$ ，则下面那个小正方形可交换。这等价于说从 $I$ 经由 $A$ 到 $B$ 相当于从 $I$ 直接到 $B$ 。这就是初始代数的融合律。记为：

$$A \xrightarrow{h} B \Rightarrow h \circ (a) = \langle b \rangle \iff h \circ a = b \circ \mathsf{F}(h) \quad (5.7)$$

初始代数的融合律意味这什么呢？在上一章中，我们发现向下态射可以将非递归的计算，转换成在递归结构上的叠加计算。例如，当函子 $\mathsf{F}$ 是 $\text{ListFA}$ （其中 $A$ 是给定的对象）时，箭头 $a = f + z$ （加号表示余积），而初始箭头 $i = (:) + []$ ，向下态射 $\langle a \rangle = \text{foldr}(f, z)$ 。如果记 $b = c + n$ ，则融合律可以写成：

$$h \circ \text{foldr}(f, z) = \text{foldr}(c, n)$$

它说明叠加之后再进行变换可以化简为单一的叠加。1995年，高野秋彦和梅耶（Meijer）进一步将向下态射 $\langle a \rangle$ 抽象成从 $a$ 构造某种代数结构 $g a$ ，这样融合律就可以推广为[52]：

$$A \xrightarrow{h} B \Rightarrow h \circ g a = g b \quad (5.8)$$

这一推广的融合律被称为“酸雨定律”<sup>3</sup>。另一方面，由于存在从初始代数 $I$ 到 $A$ 的箭头： $I \xrightarrow{(a)} A$ ，所以将 $\langle a \rangle$ 代入酸雨定律左侧的 $h$ ，并且用 $i$ 替换酸雨定律中的 $a$ ，用 $a$ 替换酸雨定律中的 $b$ ，我们有：

$$I \xrightarrow{\langle a \rangle} A \Rightarrow \langle a \rangle \circ g i = g a \quad (5.9)$$

<sup>3</sup>由于融合律能够消除不必要的中间结果，所以最早被称为“砍伐定律”（deforestation）。而推广的融合律被幽默地称为酸雨定律（acid rain law）。

具体到列表的例子，把 $\{a\}$ 代换为 $foldr(f, z)$ ；把初始代数 $i$ 代换为列表的初始代数 $(:) + []$ ；定义 $build(g) = g (:) []$ ，并代入酸雨定律左侧，就得到了列表的叠加——构建融合律：

$$\begin{array}{ll} \{a\} \circ g i = g a & \text{酸雨定律} \\ \Rightarrow foldr f z (g i) = g a & \text{列表的向下态射是叠加} \\ \Rightarrow foldr f z (g (:) []) = g a & \text{列表的初始代数} i \text{是} (:) [], \\ \Rightarrow foldr f z (g (:) []) = g f z & \text{用} f, z \text{替换} a \\ \Rightarrow \mathbf{foldr} f z (\mathbf{build} g) = g f z & \text{反向用} build \text{的定义} \end{array}$$

这样我们就用范畴论证明了叠加——构建融合律[51]。

### 练习 5.1

- 验证从左侧叠加也可以表示为 $foldr$ :

$$foldl f z xs = foldr (b g a \mapsto g (f a b)) id xs z$$

- 证明以下列表的构建——叠加形式:

$$\begin{aligned} concat xs &= build (f z \mapsto foldr (xs x \mapsto foldr f xxz) xss) \\ map f xs &= build (\oplus z \mapsto foldr (y ys \mapsto (f y) \oplus ys) z xs) \\ filter f xs &= build (\oplus z \mapsto foldr (x xs' \mapsto \begin{cases} f(x) : & x \oplus xs' \\ \text{否则:} & xs' \end{cases}) z xs) \\ repeat x &= build (\oplus z \mapsto let r = x \oplus r in r) \end{aligned}$$

- 利用融合律化简快速排序算法:

$$\begin{cases} qsort [] = [] \\ qsort (x : xs) = qsort [a | a \in xs, a \leq x] ++ [x] ++ qsort [a | a \in xs, x < a] \end{cases}$$

提示：将ZF表达式<sup>4</sup>转换为 $filter$ 。

- 利用范畴论验证融合律的类型限制。提示：考虑向下态射的类型。

## 5.2 巧算100

我们举的第二个例子来自伯德的《函数式算法珠玑》中的第6章[50]。高德纳在《计算机程序设计的艺术》卷4中给出了一道练习题[53]：把1到9这九个数字写成一行：1 2 3 4 5 6 7 8 9。只允许在这些数字之间添上加号和乘号，不许用括弧。如何使得最后的得数恰好是100？

例如：

$$12 + 34 + 5 \times 6 + 7 + 8 + 9 = 100$$

这看起来像是一道小学生的数学竞赛题目。如果要求找到所有可能的解法，就是一道编程趣题。最简单直接的方法，就是利用计算机进行穷举，每两个数字间一共有三种选择：1) 什么都不插入；2) 插入加号；3) 插入乘号。由于九个数字间一共有8个空，所以共有 $3^8 = 6561$ 种方法。我们只要让计算机逐一检查这6561种方案，看看哪些最终得数恰好是100就可以了。

---

<sup>4</sup>全称为“策梅罗——弗兰克尔表达式”。指集合论中 $\{f(x) | x \in X, p(x), q(x), \dots\}$ 这样的集合构建表达式。我们在下一章介绍无穷和集合论时会再次遇到它。

### 5.2.1 穷举法

我们先把一个由数字、加法、乘法组成的式子定义出来。考虑刚才的算式：

$$12 + 34 + 5 \times 6 + 7 + 8 + 9$$

由于先乘除后加减，我们可以把它看作由加号分割的若干子算式。所以上述算式等价于：

$$\text{sum}[12, 34, 5 \times 6, 7, 8, 9]$$

这样我们定义一个算式是由加号分割的若干子算式  $t_1 + t_2 + \dots + t_n$ 。即  $\text{expr} = [t_1, t_2, \dots, t_n]$ ：

```
type Expr = [Term]
```

具体到每个子算式，我们可以把它看作由乘号分割的若干因子，例如  $5 \times 6 = \text{product}[5, 6]$ 。如果是单一的数，例如 34，我们仍然可以把它看作  $34 = \text{product}[34]$ 。这样我们就可以定义由因子组成的子算式  $f_1 \times f_2 \times \dots \times f_m$ 。即  $\text{term} = [f_1, f_2, \dots, f_m]$ ：

```
type Term = [Factor]
```

最终，每个因子都可以看作若干数字组成的整数，例如由两个数字组成的 34，或仅有一个数字的 5。也就是说  $\text{factor} = [d_1, d_2, \dots, d_k]$ ：

```
type Factor = [Int]
```

从若干数字求得整数是一个叠加的过程，例如  $[1, 2, 3] \Rightarrow (((1 \times 10) + 2) \times 10) + 3$ 。我们可以将其定义为一个函数：

$$\text{dec} = \text{foldl } (n \text{ } d \mapsto n \times 10 + d) 0$$

穷举法的思路是针对每一个可能的算式，计算并检查它是否得 100。为此，我们需要定义一个函数，求出一个算式的值。根据算式的定义，我们需要递归地求出每个子算式（term）的值，然后将其累加起来；为了求每个子算式的值，需要递归地求出每个因子的值，然后再累乘起来；为了求每个因子的值，需要对每个因子使用  $\text{dec}$  函数。

$$\text{eval} = \text{sum} \circ \text{map} (\text{product} \circ (\text{map dec}))$$

明显这一定义可以用上一节介绍的融合律进行优化，我们把具体的推导过程留作练习。最终的优化结果如下：

$$\text{eval} = \text{foldr } (t \text{ } ts \mapsto (\text{foldr } ((\times) \circ \text{fork}(\text{dec}, \text{id})) 1 t) + ts) 0$$

从这一结果可以写出一个可读性与性能都更好的定义：

$$\begin{cases} \text{eval} [] = 0 \\ \text{eval}(t : ts) = \text{product} (\text{map dec } t) + \text{eval}(ts) \end{cases}$$

根据这一定义，如果算式为空，则其值为 0；否则我们取出第一个子算式，将其中的每个因子求出并乘到一起。然后再加上剩余子算式的求值结果。这样在使用穷举法时，对每个可能的算式，我们都用  $\text{eval}$  求值。留下所有等于 100 的算式。即：

$$\text{filter } (e \mapsto \text{eval}(e) == 100) es$$

其中  $es$  是所有从 1 到 9 能够组成的算式的集合。如何产生这个集合呢？我们从空集合开始，然后每次从 1 到 9 中取出一个数字，看看能扩展出哪些合法的算式。首先，从空集合和唯一的数字  $d$  只能构造出一个算式：  $[[d]]$ 。最内层是由数字  $d$  构成的唯一因子  $\text{fact} = [d]$ ；然后是由这个因子构成的

子算式 $term = [fact] = [[d]]$ ，这个唯一的子算式构成的完整算式是 $[term] = [[fact]] = [[[d]]]$ 。我们可以把这一构造过程定义为：

$$expr(d) = [[[d]]]$$

接下来，考虑一般情形。我们的策略是从右侧开始，不断取出数字9, 8, 7, ...来扩展算式。假设我们已经用数字构造了一个算式集合 $[e_1, e_2, \dots, e_n]$ ，此时如果取出下一个数字 $d$ ，如何扩展出所有的合法算式呢？我们在本节的开头说过，每两个数字间一共由三种选择：1) 什么都不插入；2) 插入加号；3) 插入乘号。我们来看看对于算式集合中的任意 $e_i$ 这三种选择的含义分别是什么。将 $e_i$ 展开表示成若干子算式的和 $e_i = t_1 + t_2 + \dots$ ，其中第一个子算式 $t_1$ 表示为若干因子的积 $t_1 = f_1 \times f_2 \times \dots$ 。

1. 什么都不插入意味着将数字 $d$ 直接添加到 $e_i$ 的第一个子算式中的第一个因子的前面。这样由 $d : f_1$ 组成一个新的因子。例如 $e_i$ 是算式 $8 + 9$ ，数字 $d$ 是7，将7写在8+9的前面而什么符号都不插入，这样就得到新算式 $78 + 9$ ；
2. 插入乘号意味着用数字 $d$ 构成一个因子 $[d]$ ，然后将它添加到 $e_i$ 中第一个子算式的前面。这样由 $[d] : t_1$ 组成一个新的子算式。具体到 $8 + 9$ 这个例子，我们把7写在它的前面，然后在7和8之间插入一个乘号，这样就得到新算式 $7 \times 8 + 9$ ；
3. 插入加号意味着用数字 $d$ 构成一个子算式 $[[d]]$ ，然后将它添加到 $e_i$ 的最前面，组成新的算式 $[[d]] : e_i$ 。具体到 $8 + 9$ 这个例子，我们把7写在它的前面，然后在7和8之间插入一个加号，这样就得到新算式 $7 + 8 + 9$ ；

把这三种情形转换为定义就得到了下面的函数：

```
add d ((ds:fs):ts) = [(d:ds):fs]:ts,
                      ([d]:ds:fs):ts,
                      [[d]]:(ds:fs):ts
```

其中算式 $e_i$ 被写成了 $(ds : fs) : ts$ 的形式， $e_i$ 中的第一个子算式是 $ds : fs$ ，第一个子算式中的第一个因子是 $ds$ 。这样，从每个已有的算式，我们都可以扩展出3个新算式。而对于算式列表 $[e_1, e_2, \dots, e_n]$ ，只要分别对每个算式扩展，然后将结果连接到一起就可以了。这恰恰就是上一节我们介绍过的`concatMap`：

$$\begin{cases} extend d [] = [expr(d)] \\ extend d es = concatMap (add d) es \end{cases}$$

这样我们就得到了穷举法的完整定义：

$$filter (e \mapsto eval(e) == 100) (foldr extend [1..9]) \quad (5.10)$$

### 5.2.2 改进

我们能改进这个穷举法么？观察用1到9这些数字从右向左穷举算式的过程。首先我们写下数字9，在添加数字8时，可能产生3个算式，分别是： $89$ ， $8 \times 9 = 72$ ，和 $8 + 9 = 17$ 。接下来添加7，算式 $789$ 显然大于100，可以立即丢弃。并且我们确信任何在789左侧扩展出的新算式也都不可能等于100，因此也可以丢弃。这样就避免了无谓的计算。同样 $7 \times 89$ 也大于100，可以丢弃并停止向左扩展。只有 $7+89$ 是可能的候选算式。接下来我们扩展出 $78 \times 9$ ，由于它大于100，因此被丢弃并停止扩展……

通过这一过程，我们发现，可以一边扩展算式，一边计算当前算式的值。只要超过100就丢弃并停止从这一候选算式向左继续扩展。整个过程就像一棵不断生长的树，只要树枝代表的算式超过100，就砍掉树枝，停止这一分支的生长。

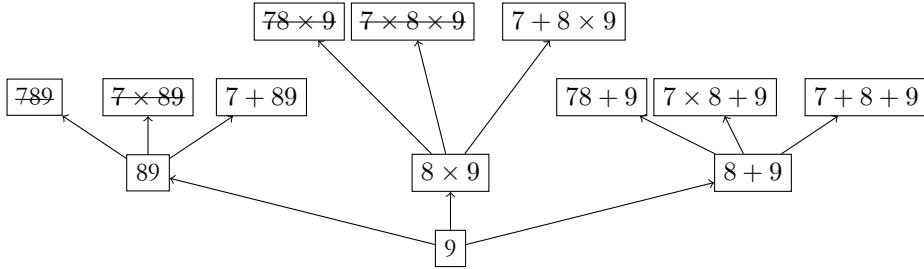


图 5.3: 穷举算式的过程类似一棵生长的树

为了能够一边扩展算式，一边快速计算当前算式的值，我们可以将算式的值分离成三个部分：第一个子算式中的第一个因子的值 $f$ ，除去 $f$ 外剩余因子的乘积 $v_{fs}$ ，以及除去第一个子算式外，剩余所有子算式的和 $v_{ts}$ 。有了这三个部分，我们就可以计算出整个算式的值 $f \times v_{fs} + v_{ts}$ 。

此时，如果继续向左扩展一个数字 $d$ ，对应三种选择，算式及其值的更新分别如下：

1. 不插入任何符号，将 $d$ 添加到第一个因子前作为最高位数字。算式的值等于： $(d \times 10^{n+1} + f) \times v_{fs} + v_{ts}$ ，其中 $n$ 是因子 $f$ 的位数。因此值的三个部分更新为： $f' = d \times 10^{n+1} + f$ ， $v_{fs}$ 和 $v_{ts}$ 保持不变。
2. 插入乘号，将 $d$ 作为第一个子算式中的第一个因子。算式的值等于： $d \times f \times v_{fs} + v_{ts}$ 。因此值的三个部分更新为： $f' = d$ ， $v_{fs'} = f \times v_{fs}$ ， $v_{ts}$ 不变。
3. 插入加号，将 $d$ 作为新的第一个子算式。算式的值等于： $d + f \times v_{fs} + v_{ts}$ 。因此值的三个部分更新为： $f' = d$ ， $v_{fs'} = 1$ ， $v_{ts'} = f \times v_{fs} + v_{ts}$ 。

为了方便第一条中 $10^{n+1}$ 的计算，我们可以把指数也记录下来作为算式值的第四个部分。这样算式的值就可以表示为一个四元组 $(e, f, v_{fs}, v_{ts})$ 。从四元组计算算式值的定义为：

$$\text{value}(e, f, v_{fs}, v_{ts}) = f \times v_{fs} + v_{ts} \quad (5.11)$$

这样在扩展算式时，我们可以一边扩展，一边更新四元组，并将结果成对放在一起：

$$\begin{aligned} \text{add } d (((ds : fs) : ts), (e, f, v_{fs}, v_{ts})) = & \\ & [(((d : ds) : fs) : ts, (10 \times e, d \times e + f, v_{fs}, v_{ts})), \\ & (([d] : ds : fs) : ts, (10, d, f \times v_{fs}, v_{ts})), \\ & ([[d]] : (ds : fs) : ts, (10, d, 1, f \times v_{fs} + v_{ts}))] \end{aligned} \quad (5.12)$$

对于每一对算式和四元组，我们利用 $\text{value}$ 函数，将四元组的值求出，如果大于100就立即丢弃，停止对其继续扩展。然后将候选算式和四元组对连接成一个列表。根据这一思路，我们将之前定义的 $\text{extend}$ 重新定义为：

$$\begin{cases} \text{expand } d [] = [(expr(d), (10, d, 1, 0))] \\ \text{expand } d \text{ evs} = concatMap ((filter ((\leq 100) \circ \text{value} \circ \text{snd})) \circ (\text{add } d)) \text{ evs} \end{cases} \quad (5.13)$$

现在我们可以对 $\text{expand}$ 进行叠加，从1到9扩展出所有小于等于100的候选算式和四元组。最后我们计算四元组的值，然后选出所有恰好等于100的算式。

$$\text{map } \text{fst} \circ \text{filter } ((= 100) \circ \text{value} \circ \text{snd}) (\text{foldr } \text{expand} [] [1, 2, \dots, 9]) \quad (5.14)$$

在本章附录中，我们给出了根据这一定义编写的程序。下面列出了等于100的所有7个算式：

```

1 : 1 × 2 × 3 + 4 + 5 + 6 + 7 + 8 × 9
2 : 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 × 9
3 : 1 × 2 × 3 × 4 + 5 + 6 + 7 × 8 + 9
4 : 12 + 3 × 4 + 5 + 6 + 7 × 8 + 9
5 : 1 + 2 × 3 + 4 + 5 + 67 + 8 + 9
6 : 1 × 2 + 34 + 5 + 6 × 7 + 8 + 9
7 : 12 + 34 + 5 × 6 + 7 + 8 + 9

```

### 练习 5.2

1. 利用融合律化简算式求值的定义  $eval = sum \circ map (product \circ (map dec))$ 。
2. 如何从左侧扩展出所有的算式？
3. 下面定义可以将算式翻译为字符串：

```
str = (join "+") ∘ (map ((join "×") ∘ (map (show ∘ dec))))
```

其中  $show$  可以将数字转换为字符串。函数  $join(c, s)$  将一组字符串  $s$  用  $c$  连接起来，例如  $join("#", ["abc", "def"]) = "abc#def"$ 。利用融合律化简  $str$  的定义。

## 5.3 小节和扩展阅读

程序推导是数学推导的一种特殊情况。通过这一手段，我们可以从直观的、未经化简或优化的定义出发，利用一套形式化的方法和定理，一步一步进行变换。从而最终得到简洁的、优化的结果。伯德在他的《函数式算法设计珠玑》[50]中给出了很多的例子。

程序推导的正确性根植于数学。为此需要一套完整的理论，能够将计算机程序形式化、数学化，而不是仅仅依赖于人的直觉。抽象代数和范畴理论正是帮助我们对计算机编程进行形式化的强大工具。叠加——构建融合律就是这样的一个典型的例子。在1993年的论文[49]中，人们发现了系统化简程序的一个工具。随着范畴理论的引入，一系列融合律被发展出来[51]，并应用于计算机程序的推导和优化。

## 5.4 附录代码

Haskell中的 `build` 和 `concatMap` 定义：

```

build :: forall a. (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) □

concatMap f xs = build (\c n -> foldr (\x b -> foldr c b (f x)) n xs)

```

用穷举法巧算100的基本定义：

```

type Expr = [Term]      -- | T1 + T2 + ... Tn
type Term = [Factor]    -- | F1 * F2 * ... Fm
type Factor = [Int]     -- | d1d2...dk

dec :: Factor -> Int
dec = foldl (\n d -> n * 10 + d) 0

expr d = [[[d]]] -- | single digit expr

```

```

eval [] = 0
eval (t:ts) = product (map dec t) + eval ts

extend :: Int -> [Expr] -> [Expr]
extend d [] = [expr d]
extend d es = concatMap (add d) es where
  add :: Int -> Expr -> [Expr]
  add d ((ds:fs):ts) = [((d:ds):fs):ts,
    ([d]:ds:fs):ts,
    [[d]]:(ds:fs):ts]

sol = filter ((==100) . eval) . foldr extend []

```

用改进的穷举法解决巧算100问题：

```

value (_, f, fs, ts) = f * fs + ts

expand d [] = [(expr d, (10, d, 1, 0))]
expand d evs = concatMap ((filter ((<= 100) . value . snd)) . (add d)) evs where
  add d (((ds:fs):ts), (e, f, vfs, vts)) =
    [(((d:ds):fs):ts, (10 * e, d * e + f, vfs, vts)),
     ([[d]]:(ds:fs):ts, (10, d, f * vfs, vts)),
     [[[d]]:(ds:fs):ts, (10, d, 1, f * vfs + vts))]

sol = map fst . filter ((==100) . value . snd) . foldr expand []

```



## 第6章 无穷

我看到了，但我不相信。

——1877年康托尔写给戴得金的信

不知在多久以前，我们的祖先仰望星空，面对浩瀚的星河，由衷地发出感叹，我们所在的世界究竟有多大？作为智慧的生命，我们的思维超越自我，超越地球，超越宇宙，不断思考着无穷的概念。我们的祖先是从具体的事物中抽象出了数的概念。例如狩猎得到的三头羊，采集得到三个果实，烧制了三个陶罐，进而得到抽象的数字三来代表任何三个东西。起初的数字大小有限，能够满足日常生活、狩猎、劳作的需要。随着文明的发展，我们开始进行贸易活动，出于记账的要求，所需要的数字逐渐变大。人们发展出种种计数系统，来掌握更大的数字。终于，我们提出问题：最大的数是什么？对于这个问题，人们分成了两种不同的态度。一种认为，这个问题没有意义，在古代，掌握千百万这样的数已经足够在生活中使用了。我们无需了解生活中用不上的大数。例如可以认为世界上沙子的数目是无穷的。在古希腊，一万曾被认为是一个十分巨大的数，人们称它为*murias*，最终变成了*myriad*一词，意为“无数”<sup>[54]</sup>。无独有偶，佛教中也用“恒河沙数”来表达大到无法计算的数。在大乘佛教经典《金刚经》中，佛陀说：“以七宝满尔所恒河沙数三千大世界，以用布施。”另一种则不这么想，阿基米德，这位古希腊伟大的数学家认为，即使是充满全宇宙的沙子数目，也可以用一个数代表。阿基米德在他的著作《数沙者》开篇中说：

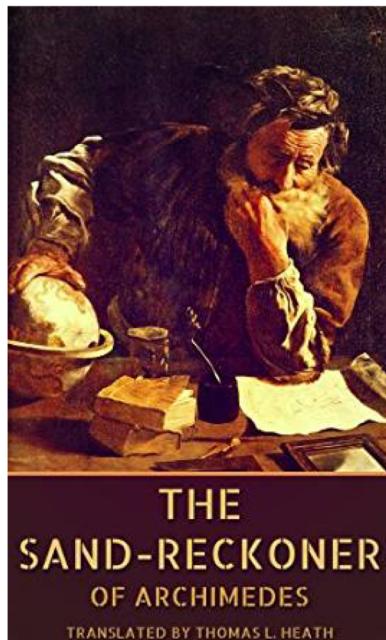
“格朗王，有人认为沙子的数量是无穷的。我所说的沙子，并不单单地指叙拉古附近和西西里岛其余地方的沙子，还包括地球上所有角落能找到的沙子，无论那里有人还是无人居住。另一些人虽然承认沙子的数量并不是无穷大的，但他们认为，我们不可能写出一个足够大的数，使它在数量上超过地球上全部沙子所代表的数量。如果想象一个和地球体积同样大的沙体，而且要从地球上的大海和谷底算起，直到最高山峰的高度都填满沙子，这些人恐怕就更加肯定，世界上不可能有如此之大的数，可以用来表示堆积起这一巨大沙体所需要的沙子的数量。但是，我将向您证明，通过一系列几何证明——您之后也可以照着做——我命名了一些数字，写在我给宙克西珀的手稿中。其中一些数字不仅超过了以我刚刚描述过的方式填充地球所需要的沙子的数量，甚至超过了填充整个宇宙所需要的沙子数量。”

阿基米德认为填满宇宙“只”需要 $10^{63}$ 粒沙子。这个宇宙的含义是指恒星天球，大约为两万倍地球的半径。今天我们知道可观测宇宙的尺寸大约为460亿光年，大约包含 $3 \times 10^{74}$ 个原子<sup>1</sup>。在古希

<sup>1</sup>—说为 $10^{80}$ 到 $10^{87}$ 个基本粒子。



埃舍尔《圆极限·4》（又名天使与恶魔）1960



《数沙者》，封面的阿基米德像是意大利画家多米尼克·费蒂1620年创作的。

腊的时代，阿基米德的想法无疑是天才的，这几乎是无穷的具体化。我们从语言中，可以看到许多表达大数单位的词语。例如下表是汉语中的大单位，从“兆”以后，每增加一万倍，就有一个对应的单位。 ([55]，第31页)

京	$10^{16}$	载	$10^{44}$
垓(gāi)	$10^{20}$	极	$10^{48}$
秭(zǐ)	$10^{24}$	恒河沙	$10^{52}$
穰(ráng)	$10^{28}$	阿僧祇(zhī)	$10^{56}$
沟	$10^{32}$	那由他	$10^{60}$
涧	$10^{36}$	不可思议	$10^{64}$
正	$10^{40}$	无量大数	$10^{68}$

可以看到，汉语中这些大单位词汇，有许多来自佛教。包括恒河沙，它表示1后面跟着52个0。英语中的大单位如下表。从一开始，每增加一千倍就有一个对应的单位。万进位和千进位的不同，也是文化上的一种差异。

thousand	$10^3$	quattuordecillion	$10^{45}$	octovigintillion	$10^{87}$
million	$10^6$	quindecillion	$10^{48}$	novemvigintillion	$10^{90}$
billion	$10^9$	sexdecillion	$10^{51}$	trigintillion	$10^{93}$
trillion	$10^{12}$	septdecillion	$10^{54}$	untrigintillion	$10^{96}$
quadrillion	$10^{15}$	octodecillion	$10^{57}$	duotrigintillion	$10^{99}$
quintillion	$10^{18}$	novemdecillion	$10^{60}$	googol	$10^{100}$
sexillion	$10^{21}$	vigintillion	$10^{63}$		
septillion	$10^{24}$	unvigintillion	$10^{66}$		
octillion	$10^{27}$	duovigintillion	$10^{69}$		
noniliion	$10^{30}$	trevigintillion	$10^{72}$		
decillion	$10^{33}$	quattuorvigintillion	$10^{75}$		
undecillion	$10^{36}$	quinvigintillion	$10^{78}$		
duodecillion	$10^{39}$	sexvigintillion	$10^{81}$		
tredecillion	$10^{42}$	seprvigintillion	$10^{84}$		

表中最后一个大单位古格尓（googol）是在1920年由9岁的米尔顿·西洛塔（Milton Sirotta）想出的名字。这个数字是1后面跟着100个零。著名的互联网公司谷歌的名字就来自它[56]。

## 6.1 无穷概念的提出

超越一切具体大数的无穷是否存在不仅是一个数学问题，还是一个哲学问题。无穷大还直接导致另一个概念——无穷小。古代中国的哲学家庄子在《天下篇》中说：“一尺之棰，日取其半，万世不竭。”，古希腊的哲学家，埃利亚学派的芝诺（Zeno of Elea）提出了著名的四个悖论，它们都与无穷有关。

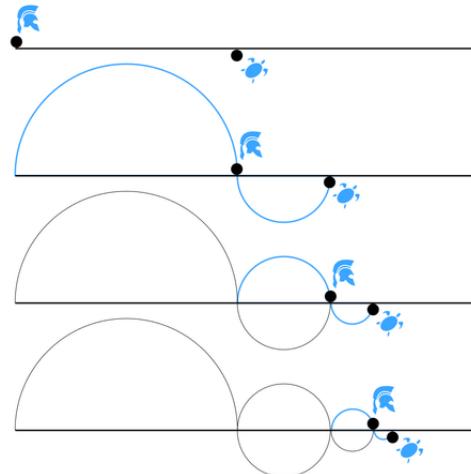


图 6.3: 阿基里斯与乌龟悖论

第一个悖论最为人们所津津乐道。名叫阿基里斯与乌龟悖论。阿基里斯是荷马史诗《伊里亚特》中的英雄，以善跑著称。这个悖论说：如果让爬得很慢的乌龟在阿基里斯前面一段路程出发，那么阿基里斯将永远追不上乌龟。这是因为，阿基里斯为了赶上乌龟，必须先到达乌龟的出发点A，但当阿基里斯到达A点时，乌龟已经在这段时间前进到了B点。但当阿基里斯到达B点

时，乌龟又已经到了前面的C点……以此类推，两者间的距离虽然越来越近，但阿基里斯永远落在乌龟的后面而追不上乌龟。如图6.3所示。但是这与我们生活中的常识是不相符的。这个悖论的推理是如此让人信服，以至于千百年来吸引了无数学者的研究。刘易斯·卡罗尔（Lewis Carroll）、侯世达（Douglas Hofstadter）甚至拿乌龟和阿基里斯作为文学作品中的主人公。

第二个悖论叫作“二分悖论”。这个悖论说，如果阿基里斯想从A到B，那么他必须先走到 $1/2$ 的位置。同样在此之前，他必须要到达 $1/4$ 的位置。而为了到达这一位置，他必须先到达 $1/8$ 的位置……以此类推。由于这样的中点有无限多个，阿基里斯永远也无法到达目的。芝诺的这个悖论实际上说明了运动根本无法发生。

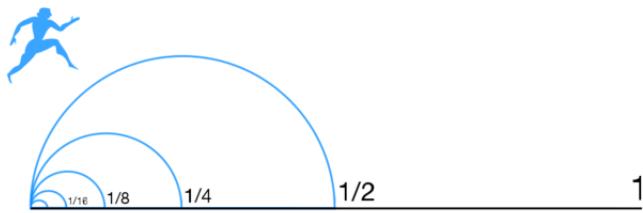


图 6.4: 二分悖论

第三个悖论叫作“飞矢不动悖论”，它从另一个角度描述无穷导致运动无法发生。芝诺指出，任何物体待在相同的位置都不叫运动，可是飞行的箭矢在任一时刻不也是待在一个地方么？这样看来，自然飞失也是不动的。如果说前两个悖论是由于分割空间导致的，则这个悖论是由对时间的分割导致的。

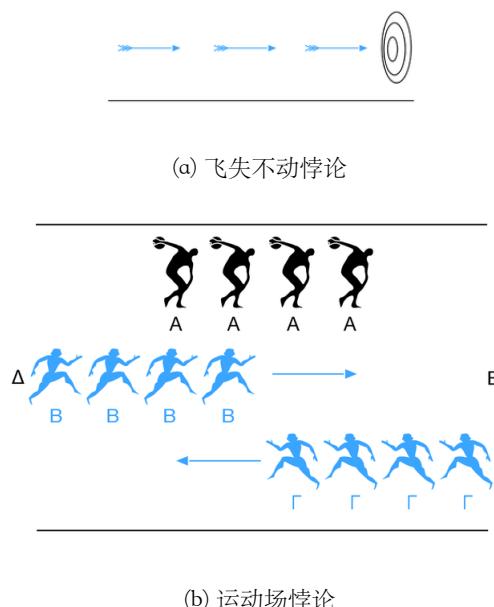
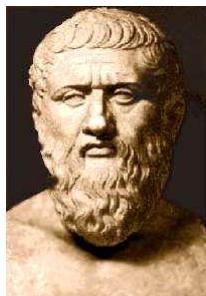


图 6.5: 飞矢不动和运动场悖论

第四个悖论叫作“运动场悖论”。这一悖论主要针对时间原子论的观点，即认为存在最小的不可分割的时间单位。如图6.5所示，运动场中有3列人。最初他们都首尾对齐。在最小的时间单元内，A列不动，B向右移一个单位，而Γ向左移动一个单位。容易得知，相对B而言，Γ其实移动了两个单位。这就意味着，应该存在这一让Γ相对于B移动一个单位的时间。而这一时间应该是最小单位时间的一半。但如果存在不可分割的“时间原子”，那么这两个时间就是相同的，即最短时间和它的一半相等。



芝诺，约490BC - 425BC

芝诺悖论并不复杂，稍加琢磨就能理解。但是导出的结果却出人意料。根据生活中的常识，运动和时间是如此真实，阿基里斯不可能赶不上乌龟。可是驳倒这些悖论却不容易，从亚里士多德到罗素，从阿基米德到赫尔曼·外尔，都对芝诺悖论提出了各种不同的解法[57]。

芝诺（约公元前490年——公元前425年），古希腊哲学家，生于意大利半岛南部的埃利亚。所以我们常称其为埃利亚的芝诺。关于他的生平，缺少可靠的文字记载。据传，他早年是一个自学成才的乡村孩子，一生经历坎坷，最终遭到一位暴君的陷害而被拘捕、拷打、直至被处死[9]。

芝诺是继毕达哥拉斯学派之后在意大利新出现的一个哲学学派——埃利亚学派的代表人物之一，这一学派的领袖是芝诺的老师巴门尼德。巴门尼德认为整个世界是个不变的整体，即“不变的一”，运动、变化与多样性都只是幻象。

芝诺以其悖论闻名。他一生曾巧妙地构想出40多个悖论，在流传下来的悖论中以关于运动的四个“无限微妙、无限深邃”的悖论最为著名。他提出这些悖论很可能是为他老师的哲学观点辩护。

芝诺悖论在当时曾给古希腊人造成深深的困惑。而芝诺悖论所涉及的对时间、空间、无限、连续、运动的看法，也都在极长的历史岁月中困扰着后来的哲学家和数学家。如何了解和认识无穷，如何使用无穷成为了摆在古希腊人面前必须解决的问题。

### 6.1.1 无穷的哲学

亚里士多德对芝诺悖论进行了深入的思考（我们今天对芝诺悖论的了解，其实是来自亚里士多德的著作《物理学》），并做出一项对后世数学发展具有深远影响的工作。他把无穷的概念区分为实无穷和潜无穷。所谓潜无穷或潜无限，是指无限在永远延伸着，是一种变化的、不断产生出来概念。它永远在构造中，永远完成不了，是潜在的，而不是实在的。自然数就是一种潜无穷，对于任何一个自然数，我们都可以找到它的后继，也就是一个更大的自然数。欧几里得几何中的直线也是一种潜无穷，我们可以按需延伸直线<sup>2</sup>。所谓实无穷是指把无限的整体本身作为一个实在的单位，是已经构造出来的东西。也就是把无限对象看作可完成的过程或无穷整体。

在做了这种区分后，亚里士多德承认存在潜无穷，但是拒绝承认实无穷的概念。他对实无穷的排斥深刻而长远地影响了日后数学的发展[9]。亚里士多德代表了当时古希腊的哲学观点，关于潜无穷和实无穷的概念区分以及争论一直影响至今。尽管对无穷的概念仍然心存疑虑，古希腊的数学家们借助潜无穷的思想取得了令人赞叹的成就。其中之一就是欧几里得证明了存在无穷多的素数。这一证明被人们认为是历史上最优美的证明之一。

**定理 6.1.1.** 《几何原本》第九卷，命题20。预先给定任意多的素数，则有比它们更多的素数[11]。

欧几里得在叙述这个命题时，小心谨慎地避免使用无穷这样的说法，通观《几何原本》一书这样的例子还有很多。欧几里得在证明这个命题时，使用了著名的反证法。我们用现代的语言来描述这一证明。

证明. 假设只存在有限多个素数， $p_1, p_2, \dots, p_n$ 。我们构造一个新数：

$$p_1 p_2 \dots p_n + 1$$

---

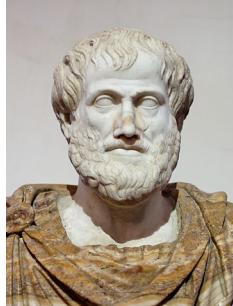
<sup>2</sup>欧几里得避免使用“无限延伸”这样的说法，而代之以按需任意延伸。这在古希腊是常见的一种处理。

也就是把这 $n$ 个素数乘起来再加一。这个数要么是素数，要么不是素数。

- 如果它是素数，明显这个数不等于 $p_1$ 到 $p_n$ 中的任何一个，这就在有限多个素数中又增加了一个新的素数；
- 如果这个数不是素数，那么它就存在一个素因子 $p$ 。但是由于 $p_1$ 到 $p_n$ 中的任何一个都不能整除我们构造的这个数，所以素数 $p$ 与任何 $p_1$ 到 $p_n$ 中的数都不同，是一个新的素数。

所以在任何情况下，我们都可以获得一个新的素数。这与有限个素数的假设矛盾，因此存在无穷多的素数。  $\square$

欧几里得用反正法得到了一种“存在性证明”，他证明了存在无穷多的素数，但却没有给出怎样得到这些素数。这在我们今天看来，是很自然的一种处理。然而在十九世纪末二十世纪初却引发了关于数学根本性的争论，我们将在下一章详细讲述这一内容。



亚里士多德，384BC - 322BC

亚里士多德是世界古代史上最伟大的哲学家、科学家和教育家之一，古希腊哲学的集大成者。他是柏拉图的学生，亚历山大大帝的老师。公元前384年，亚里士多德出生于希腊北部城市斯塔基拉(Stageira)。后人对他早年的生平所知甚少。17岁时，他赴雅典在柏拉图学园就读达20年，直到柏拉图去世后方才离开。这一时期的学习和生活对他一生产生了决定性的影响。苏格拉底是柏拉图的老师，亚里士多德又受教于柏拉图。在雅典的柏拉图学园中，亚里士多德表现得很出色，柏拉图称他是“学园之灵”。但亚里士多德可不是个只崇拜权威，在学术上唯唯诺诺而没有自己的想法的人。他努力地收集各种图书资料，勤奋钻研，甚至为自己建立了一个图书室。

公元前347年，柏拉图去世。两年后，亚里士多德离开了雅典——亚里士多德或许是带着失望的心情离开的，因为他并没有成为柏拉图思想的继承者。此后，他开始游历各地。公元前343年，亚里士多德被马其顿的国王腓力浦二世召回故乡，受国王的聘请，担任起当时年仅13岁的亚历山大大帝的老师。亚里士多德也运用了自己的影响力，对亚历山大大帝的思想形成起了重要的作用。正是在亚里士多德的影响下，亚历山大大帝始终对科学事业非常关心，对知识十分尊重。

公元前335年腓力浦去世，亚里士多德又回到雅典，并在那里建立了自己的学校。学园的名字叫做吕克昂(Lyceum)。在此期间，亚里士多德边讲课，边撰写了多部哲学著作。亚里士多德讲课时有一个习惯，边讲边漫步于走廊和花园，正是因为如此，学园的哲学被称为“逍遥派”或者是“漫步的哲学”。亚里士多德在这一期间也有很多著作，主要是关于自然科学和物理方面的自然科学和哲学，而使用的语言也要比柏拉图的《对话录》晦涩许多。他的作品很多都是以讲课的笔记为基础，有些甚至是他的课堂笔记。因此有人将亚里士多德看作是西方的第一个教科书作者。

亚历山大去世后，雅典人开始奋起反对马其顿的统治。由于和亚历山大的关系，雅典人攻击亚里士多德，并判他为不敬神罪，当年苏格拉底就是因不敬神罪而被判处死刑的。但亚里士多德最终逃出了雅典。

公元前322年，亚里士多德因身染重病离开人世，终年六十三岁。

亚里士多德重视研究现实世界，他的精神财富正在于他的实证研究方法，这成为了现代科学的基本原则。此外，现在大学里面的院系划分方式，及其所反应的知识整理方式，也直接沿承自亚里士多德所提出的分类法。

### 6.1.2 穷竭法与微积分

与欧几里得的小心谨慎不同，古希腊的另一些数学家采用了较为实用的态度对待无穷，创造并发展了一种称为“穷竭法”的有力工具，取得了惊人的成就。

穷竭法由古希腊智人学派的代表人物安提丰（Antiphon，约前480——前410）首创。为了解决古希腊三大作图难题之一的圆化方问题<sup>3</sup>，他提出用圆内接多边形逼近圆面积的方法来化圆为方。安提丰从一个圆内接正方形开始，不断将边数加倍得到正八边形、十六边形……不断重复这一过程，随着圆面积的逐渐“穷竭”，将得到一个边长越来越微小的圆内接正多边形。安提丰认为最终这一正多边形将与圆重合，这就是穷竭法的思想。后来古希腊伟大的数学家欧多克索斯对穷竭法进行了改进和严格化，使其成为解决面积、体积问题的一种有力的几何方法。为了理解这一方法，我们先要介绍著名的欧多克索斯——阿基米德公理，有时简称为阿基米德公理。

**公理 6.1.1. 阿基米德公理 (Axiom of Achimedes)** 对于任意两个量 $a$ 与 $b$ 来说，总有自然数 $n$ 使得 $a \leq nb$ 。

阿基米德公理非常基本。在第二章中，我们介绍了欧几里得最大公约数算法，但是我们没有探讨这一算法是否能够结束。利用阿基米德公理就可以给出欧几里得算法一定能够结束的证明。利用这一基本原理，欧多克索斯指出：“给定两个不相等的量，如果从较大的量减去比它大一半的量，再从所余的量减去比这个余量大一半的量，重复这一过程，必有某个余量小于给定的较小的量。”这就是穷竭法背后的逻辑。

利用穷竭法，欧多克索斯证明了棱锥体积是同底同高棱柱体积的 $1/3$ ，圆锥体积是同底同高圆柱体积的 $1/3$ 。这些成果都记录在欧几里得的《几何原本》卷12中[9]。

古希腊将穷竭法发展到最高成就的当属阿基米德，他取得了惊人的成就，计算出了圆周率 $\pi$ ，证明圆面积公式，球体、锥体的表面积和体积公式，甚至找到了计算抛物线下面积的方法。被称为古希腊的数学之神。

阿基米德（前287年——前212年），生于西西里岛的叙拉古王国。早年曾经在古希腊的学术中心亚历山大城跟随欧几里得学习。阿基米德后来回到了叙拉古，他的许多学术成果都是通过和亚历山大城的学者之间的往来信件保存下来的。虽然有关阿基米德的生平没有详细的记载，但是关于他的各种故事却广为流传、脍炙人口。

最著名的故事就是国王的王冠。叙拉古国王不知道他的王冠是否是纯金的，大臣们也一筹莫展，于是去请教阿基米德。阿基米德一直解不开这个难题，他废寝忘食，直到有一天洗澡的时候，看着浴缸里的水溢出来，突然得到了灵感。他从浴缸里一跃而出，光着身子跑到大街上，边跑边喊“尤里卡！尤里卡！”，这句希腊语Eureka的意思是“我找到了！”。阿基米德利用浮力和比重，最终发现王冠掺了假。他的这一发现就是每一个中学生都要学习的“阿基米德定律”。尤里卡后来被人们用来形容找到灵感的那一刹那。

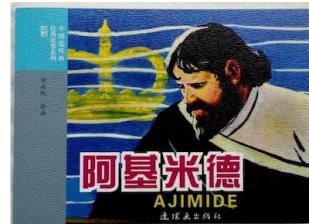
阿基米德发现球的体积是其外接圆柱体积的 $2/3$ 。他觉得这一关系无比的美妙，因此决定死后在墓碑上刻一个内接圆柱的球体。

公元前214年，第二次布匿战争爆发了。面对敌人的围城，传说阿基米德设计了巨大的抛物面镜，把阳光汇聚到帆上烧毁了敌人的战船。阿基米德还设计了巨大的机械武器，可以瞬间击毁罗马战船。后来罗马军队攻陷了叙拉古城，一个罗马士兵冲进阿基米德家里。阿基米德正专注地在沙地上画着几何图形进行思考，他说出了那句著名的画：“你挡住了我的阳光。”感到被冒犯的罗马士兵挥刀杀死了面前的这个老人。古希腊最伟大的数学家就此停止了思考。

我们现在来看一下，阿基米德是如何利用穷竭法计算圆周率的。

如图6.9所示，阿基米德在直径为1的圆内做一个内接多边形，同时在圆外做一个外切多边形。我们先看内接多边形的一条边和其上的圆弧，根据两点之间直线最短的几何公理，圆弧之长大于内接多边形的一条边长，故而圆的周长大于内接多边形的周长。同样可以得知，圆的周长小于外切多边形的周长。由于直径为1的圆，其周长恰好就是圆周率。这样我们就可以得到关系式：

<sup>3</sup>亦称为化圆为方问题，另外两个著名的难题是三分角问题和倍立方问题。任给一个圆，古希腊人希望找到只用尺规，能够作出同样面积的正方形。这个问题困扰了无数的数学家，直到19世纪后利用伽罗瓦的理论，才一举将它们题解决。三大难题被证明都是尺规不可解的。在英文中化圆为方square the circle被形容不可能做到的事情。



小儿书《阿基米德》的封面

$$C_i < \pi < C_o$$

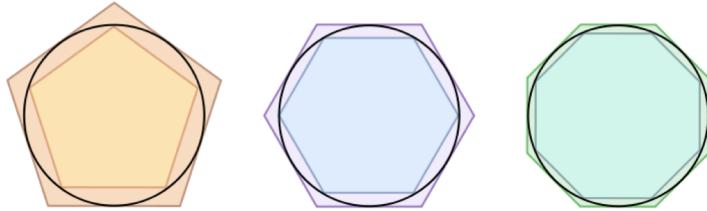


图 6.9: 用穷竭法计算圆周率

其中  $C_i$ 、 $C_o$  分别为内接多边形和外切多边形的周长。不断加大边数，就可以越来越精确地获得  $\pi$  的范围。阿基米德将边数加到 256，得出圆周率在 3.1415 和 3.1416 之间。这一记录直到南北朝时，才由我国的数学家祖冲之打破。

穷竭法的缺点是，它仍然是基于古希腊几何的一种方法，使用起来非常复杂。这部分是由于古希腊人不接受无理数，于是把几何量和“数”的概念隔离开。另一方面，古希腊整体上试图避免使用无穷大和无穷小的概念。

另一位使用穷竭法的大师是天文学家托勒密。他首次为天体运行建立了模型，地球位于宇宙的中心，月球，太阳，和其他行星则在嵌套于一起的球面上运行，最外层则是恒星天球。

亚历山大时代之后，古希腊文明的数学成果遭受了巨大的打击。罗马人征服后，迪奥多西 (Theodosius) 废除了异教，并于 392 年下令拆毁古希腊人的神庙。成千上万的希腊图书被罗马人焚毁，在公元前 47 年。罗马人纵火焚烧亚历山大港口内的船只，火势蔓延烧毁了藏书最丰富的古代图书馆。保存大量希腊著作的塞拉皮斯神庙也被摧毁，写在羊皮上的著作被刮掉而改写宗教著作。

对希腊文明的最后打击是公元 640 年新崛起的阿拉伯帝国对埃及的征服。剩余的图书被焚毁殆尽。征服者奥马尔说：“这些书的内容或许我们也有，那么我们不必读它；这些书里或许有反对我们的内容，那我们不准读它。”因此亚历山大城的浴室里接连有 6 个月用羊皮纸来烧水[63]。

每次读到此处的历史，不仅让人痛心疾首，唏嘘不已。焚书的悲剧在南美洲，在秦帝国，自古及今不断上演。埃及被占领后，一些学者迁居到东罗马帝国的首都君士坦丁堡。阿拉伯帝国后来也把一部分古希腊的数学著作翻译成阿拉伯文。巴格达的智慧宫成为了当时世界学术的中心。中世纪后，欧洲的学者又逐渐把这些阿拉伯文的著作再次译成拉丁文。伴随着文艺复兴，也是数学和哲学的复兴。接过阿基米德和穷竭法的接力棒的是德国天文学家约翰内斯·开普勒。托勒密的宇宙模型无法解释行星反向运动的现象，于是不得不引入了本轮、匀轮、偏心圆和匀速点的概念。通过分析行星的观测数据，哥白尼提出了日心说，从托勒密模型需要的 78 个圆简化到了 34 个圆。但仍然和观测有一些偏差。开普勒潜心分析德国天文学家第谷的数据 8 年，终于破解了行星运动的规律，这就是我们今天熟知的开普勒三大行星运动定律。按照开普勒定律，行星的运行轨迹并不是圆，而是一个椭圆，太阳在椭圆的一个焦点上。行星在单位时间内扫过的面积相同，因而在椭圆上有时速度快（例如在近日点附近），有时速度慢（例如在远日点附近）。分析这样复杂的数学模型需要新的工具，古希腊的穷竭法太过复杂笨重。开普勒于是进行了简化，他甚至还用自己的方法计算出了葡萄酒桶的容积。

接下来迈出重要一步的是笛卡尔和费马。通过解析几何，人们终于将数和几何结合了起来，并且在莱布尼茨和牛顿的手中发展出了微积分。微积分中的一个绕不开的概念是无限小量。并且在积分中必然要涉及无限多个量的累加。值得一提的是 1665 年，对微积分的发展做出重要贡献的学者约翰·沃利斯首次引入了现代的无穷符号  $\infty$ 。

尽管微积分的基础尚有很多争论，这一代表西方现代精神的风帆在 18 世纪乘风破浪。这是一个英雄辈出的时代，伯努利家族、欧拉、拉格朗日奔放地使用着微积分和无穷级数，解决了一个又

一个以前无法想象的难题，在天文学、力学、流体力学上开拓进取。

## 6.2 潜无穷与编程

对微积分基础的严密化直接导致了人们对实无穷的思考与激烈争论。在此之前，我们先来看一下在编程中，无穷是如何被体现的。计算机使用的是有限的资源，历史上，整数在机器内部表示为二进制形式。因为只用有限的二进制位，所以能表示的整数范围也是有限的。 $m$ 位二进制最大可以表示 $2^m - 1$ 的整数，它的二进制形式为11...1，即 $m$ 个1。例如16位整数的最大值是 $2^{16} - 1 = 65535$ 。由于这个原因，如果多个元素集合的基数用整数表示，则元素的数目也是有限的。在传统的编程中，常使用数组来容纳多个元素，为了节省内存通常要求事先确定数组大小的上限，例如下面的C语言程序定义了一个容量为10个元素的整数数组：

```
int a[10];
```

这里涉及两个概念，一个是序数，一个是基数。简单来说，序数就是我们在数数时赋予对象的一个标签。而基数是一个集合中元素的个数。我们稍后会给出它们的一般定义。早期的编程中，基数和序数都是有限的。这显然无法直接表达无穷的概念。在计算机科学发展的初期，这是完全可以理解的。那时的设备非常昂贵。人们根本不会想到使用无穷来直接解决具体的问题。然而随着时代的发展，成本逐渐降低，人们不再满足在解决问题之前，先预测出所需集合的大小。于是在一些编程环境中，开始支持动态增长的数组。人们称之为容器或者向量，可以随时按照需要向其中增加元素。现实中，即使是动态容器，其中的元素个数仍然是有限的，不能超过表示的上限。为此人们又发展出了链表结构。如本书第一章中介绍的那样，每个元素用一个节点代表，一个接一个链接起来，最后一个元素指向一个特殊的空节点表示结尾。这样我们无须知道整条链表的基数，却能从表头开始，逐一前进到表中任何位置。只要存储空间允许，链表可以任意长。这样就创造了表达潜无穷的可能。

可是链表和潜无穷终究还差了一步。我们认为自然数是无限延伸着的潜无穷，如果用链表表示自然数，不管链表有多长，例如 $n$ ，我们必须把0到 $n$ 这些数都逐一填入其中。但这只表示了序列0, 1, ...,  $n$ ，而不是自然数序列0, 1, ...,  $n$ , ...

为此，人们提出了惰性求值的概念。所谓惰性求值，就是并不立即计算一个变量的值，而是把计算推迟到需要这个值的时候。具体到自然数的例子，根据第一章介绍的皮亚诺公理，任给一个自然数 $n$ ，都存在它的后继 $n + 1$ 。而第一个自然数是0。这样自然数就可以表示为：

$$N = \text{iterate}(n \mapsto n + 1, 0)$$

其中 $\text{iterate}$ 的定义为：

$$\text{iterate}(f, x) = x : \text{iterate}(f, f(x))$$

我们来看一下自然数产生的头几步，简单起见，我们命名 $\text{succ}(n) = n \mapsto n + 1$

$$\begin{aligned} \text{iterate}(\text{succ}, 0) &= 0 : \text{iterate}(\text{succ}, \text{succ}(0)) && \text{iterate的定义} \\ &= 0 : \text{iterate}(\text{succ}, 1) && \text{succ}(0) = 0 + 1 = 1 \\ &= 0 : 1 : \text{iterate}(\text{succ}, \text{succ}(1)) \\ &= 0 : 1 : \text{iterate}(\text{succ}, 2) \\ &= 0 : 1 : 2 : \text{iterate}(\text{succ}, 3) \\ &= \dots \end{aligned}$$

如果没有惰性求值，上述过程就会一直计算下去，永不停止。这是无法用来解决实际问题的。为此我们必须把链表的链接操作实现为惰性的，其中一种方法就是利用第二章介绍的 $\lambda$ 表达式：

$$x : xs = \text{cons}(x, () \mapsto xs)$$

这种表达式 $() \mapsto exp$ 通常叫作 $delay(exp)$ ，它产生一个不带有参数函数，对这个函数求值得到结果 $exp$ 。

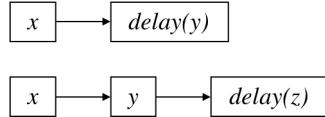


图 6.10: 链表指向的下一个节点是一个 $\lambda$ 表达式，强制求值后产生一个新节点

这样，当把 $x$ 和 $xs$ 链接到一起的时候，我们并不会求出 $xs$ 的值，而是把它放入一个 $\lambda$ 表达式中，推迟到将来再求值。这样改动后，产生自然数时就变成：

$$\begin{aligned} iterate(succ, 0) &= 0 : iterate(succ, succ(0)) && \text{iterate的定义} \\ &= cons(0, () \mapsto iterate(succ, succ(0))) && \text{惰性链接} \end{aligned}$$

计算到此为止，而不会继续进行下去。结果是一个列表，表中第一个元素是0，下一个元素是一个 $\lambda$ 表达式。如果想求出后继的元素，我们必须强制让列表求值。

$$next(cons(x, e)) = e()$$

这样，将 $next$ 应用到 $cons(0, () \mapsto iterate(succ, succ(0)))$ 就得到：

$$\begin{aligned} &next(cons(0, () \mapsto iterate(succ, succ(0)))) \\ &= iterate(succ, succ(0)) && next\text{的定义} \\ &= iterate(succ, 1) && succ\text{的定义} \\ &= 1 : iterate(succ, succ(1)) && iterate\text{的定义} \\ &= cons(1, () \mapsto iterate(succ, succ(1))) && \text{惰性链接} \end{aligned}$$

计算到这里又停了下来。不断对 $N$ 计算 $next$ ，就源源不断的获得了一个一个的自然数。人们称这种模型为“流”（Stream），用它来表示潜无穷。我们甚至可以定义一个函数从潜无穷的流中取得前 $m$ 个自然数。

$$\begin{aligned} take\ 0\ _- &= [] \\ take\ n\ cons(x, e) &= cons(x, take(n - 1, e())) \end{aligned}$$

例如 $take\ 8\ N = [0, 1, 2, 3, 4, 5, 6, 7]$ 。本章附录给出了几种语言中，用流的方法定义自然数潜无穷的一些例子。

## 练习 6.1

1. 第一章中，我们用叠加操作实现了斐波那契数列，如何用 $iterate$ 定义斐波那契数列潜无穷？
2. 用叠加操作定义 $iterate$ 。

### 6.2.1 余代数和无穷流★

本小节给出无穷流的数学解释，它需要使用第四章范畴论中介绍的余代数概念。一般的读者可以跳过两页直接阅读下一节关于实无穷的思考。首先我们回顾一下余代数和F-态射的概念。

**定义 6.2.1.** 如果  $\mathcal{C}$  是一个范畴,  $\mathcal{C} \xrightarrow{\mathsf{F}} \mathcal{C}$  是范畴  $\mathcal{C}$  上的一个自函子。对于范畴中的对象  $A$  和态射  $\alpha$ :

$$A \xrightarrow{\alpha} \mathsf{F}A$$

构成一对元组  $(A, \alpha)$  叫做  $\mathsf{F}$ -余代数。其中  $A$  叫做携带对象。

我们可以把  $\mathsf{F}$ -余代数  $(A, \alpha)$  本身看成对象。在上下文清楚的情况下, 我们通常用二元组  $(A, \alpha)$  表示对象。两个对象间的箭头定义如下:

**定义 6.2.2.**  $\mathsf{F}$ -态射是  $\mathsf{F}$ -余代数对象间的箭头:

$$(A, \alpha) \longrightarrow (B, \beta)$$

如果携带对象间的箭头  $A \xrightarrow{f} B$ , 使得下面的范畴图可交换:

$$\begin{array}{ccc} A & \xrightarrow{\alpha} & \mathsf{F}A \\ f \downarrow & & \downarrow \mathsf{F}(f) \\ B & \xrightarrow{\beta} & \mathsf{F}B \end{array}$$

即  $\beta \circ f = \mathsf{F}(f) \circ \alpha$

$\mathsf{F}$ -余代数和  $\mathsf{F}$ -态射构成了  $\mathsf{F}$ -余代数范畴  $\mathbf{CoAlg}(\mathsf{F})$ 。在  $\mathsf{F}$ -代数中, 我们关心的是初始代数。对称地, 在  $\mathsf{F}$ -余代数中, 我们关心的是终止余代数。所谓终止余代数是  $\mathsf{F}$ -余代数范畴中的终止对象, 记为  $(T, \mu)$ 。对于任何其它代数  $(A, f)$ , 存在唯一的态射  $m$  使得下面的范畴图可交换:

$$\begin{array}{ccc} \mathsf{F}T & \xleftarrow{\mathsf{F}(m)} & \mathsf{F}A \\ \mu \uparrow & & \uparrow f \\ T & \xleftarrow{m} & A \end{array}$$

使用兰贝克定理, 终止余代数是函子的不动点, 态射  $T \xrightarrow{\mu} \mathsf{F}T$  是一个同构映射, 使得  $\mathsf{F}T$  和  $T$  同构。终止余代数在编程中可用来构建无穷的数据结构。

我们使用向下态射对初始代数进行计算。对称地, 我们用向上态射(anamorphism, 词根 *an-* 的意思是向上)对终止余代数进行余计算(coevaluate)。对于任何余代数  $(A, f)$ , 它到终止余代数  $(T, \mu)$  的唯一箭头可以用向上态射表示为  $\llbracket f \rrbracket$ 。这种括号的形状不再像香蕉, 而像一对光学中的透镜符号, 所以常被叫作“透镜括号”。用范畴图表示就是:

$$\begin{array}{ccc} T & \xrightarrow{\mu} & \mathsf{F}T \\ \llbracket f \rrbracket \uparrow & & \uparrow \mathsf{F}[\![ f ]\!] \\ A & \xrightarrow{f} & \mathsf{F}A \end{array}$$

若  $m = \llbracket f \rrbracket$ , 当且仅当  $\mu \circ m = F(m) \circ f$

我们现在来看向上态射是如何构造无穷流的。向上态射接受一个余代数  $A \xrightarrow{f} FA$  和一个携带对象  $A$ , 它产生一个函子  $F$  的不动点  $\text{Fix } F$ 。其关键一点就是, 这个不动点就是终止余代数, 它具有无穷流的形式。

$$\llbracket f \rrbracket = \text{Fix} \circ F \llbracket f \rrbracket \circ f$$

我们也可以将向上态射定义为返回函子不动点的函数:

$$(A \rightarrow FA) \xrightarrow{\text{ana}} (A \rightarrow \text{Fix } F)$$

$$\text{ana } f = \text{Fix} \circ \text{fmap} (\text{ana } f) \circ f$$

举一个具体的例子, 令函子  $F$  的定义为:

```
data StreamF E A = StreamF E A
```

它的不动点为:

```
data Stream E = Stream E (Stream E)
```

$\text{StreamF } E$  是一个普通的函子, 只是我们故意把名字叫作“流”。这个函子上的余代数是这样一个函数, 它把一个类型为  $A$  的“种子”  $a$  变换为一对值, 包含  $a$ , 和下一个种子。

可以用余代数产生各种无穷流。我们给出两个例子, 第一个例子是斐波那契数列。思路是从  $(0, 1)$  开始作为起始种子。为了产生下一个种子, 我们把第二个数 1 作为新的第一个数, 把  $0 + 1$  作为新的第二个数构成一对新种子  $(1, 0 + 1)$ 。然后不断重复这一过程, 对于种子  $(m, n)$ , 我们产生下一个新种子  $(n, m + n)$ 。写成余代数就是下面的定义:

$$(Int, Int) \xrightarrow{fib} \text{StreamF } Int (Int, Int)$$

$$fib(m, n) = \text{StreamF } m (n, m + n)$$

在这个定义中, 携带对象  $A$  是一对整数。有了余代数, 我们就可以利用向上态射构造斐波那契数列的无穷流了。对于  $\text{StreamF } E$  函子, 向上态射的类型为:

$$(A \rightarrow \text{StreamF } E A) \xrightarrow{\text{ana}} (A \rightarrow \text{Stream } E)$$

我们可以将其具体定义为:

$$\text{ana } f = fix \circ f$$

其中:  $fix (\text{StreamF } e a) = \text{Stream } e (\text{ana } f a)$

将向上态射作用于余代数  $fib$  和起始对象  $(0, 1)$  就可以产生斐波那契数列的无穷流:

$$\text{ana } fib (0, 1)$$

为了从无穷流中获取前  $n$  个元素, 可以定义一个辅助函数:

```
take 0 _ = []
take n (Stream e s) = e : take (n - 1) s
```

接下来再展示一个用埃拉托斯尼筛法产生全体素数无穷流的例子。我们使用去掉1的自然数无穷列表作为携带对象 $2, 3, 4, \dots$ 从这个种子开始，我们接下来把2的所有倍数去掉就获得了下一个种子，它是从3开始的列表 $3, 5, 7, \dots$ 接下来，我们再把3的倍数都去掉，并不断重复。把这一过程写成余代数就是下面的定义：

$$\begin{aligned} [Int] &\xrightarrow{\text{era}} \text{StreamF } Int [Int] \\ \text{era}(p : ns) &= \text{StreamF } p \{n \mid p \nmid n, n \in ns\} \end{aligned}$$

然后使用向上态射我们就获得了全体素数的无穷流：

```
primes = ana era [2...]
```

特别地，对于列表的向上态射被称为展开（反折叠unfold）。向上态射和向下态射是互逆的。我们可以用向下态射把无穷流重新转换为列表。

### 练习 6.2

1. 利用第四章中介绍的不动点定义，证明 $Stream$ 是 $StreamF$ 的不动点。
2. 试定义反折叠unfold

## 6.3 实无穷的思考

亚里士多德的影响是深远的。在两千多年的时间里，数学家和哲学家们不断思考无穷的本质，大多数人能够接受潜无穷的观念。但是对于实无穷，却产生了严重的分歧。在很长一段时间，人们认为实无穷就是无所不能的上帝，或者只有上帝才能掌握实无穷。一些对实无穷的尝试带来的是令人困惑的矛盾结果。例如，假设全体自然数是一个实无穷。由于自然数从头开始，间隔着一个偶数一个奇数，人们自然认为全体偶数是全体自然数的一半。可是任何自然数乘以2，就得到了一个偶数，反之，任何偶数除以2，也对应这个一个自然数。这样看来全体自然数和全体偶数存在一对对应关系。于是这两个实无穷是同样多的。究竟是自然数多还是偶数多呢？

现代科学之父伽利略在1636年的著作《论两种新科学及其数学演化》中提出一个类似的悖论。如果将每个自然数平方，得到的新序列 $1, 4, 9, 16, 25, \dots$ 和自然数存在一一对应关系。这样看来完全平方数和自然数一样多。可是常识却告诉我们，平方数很稀疏，自然数要比平方数多得多。这一矛盾通常称为“伽利略悖论”。

不仅在算数上，在几何上人们同样发现了类似的悖论。人们注意到每条半径都把两个同心圆上的点连接起来。大圆上任意一点都一一对应到小圆上的一点。这样看来两个圆上的点同样多，可是常识通常认为大圆上的点更多。

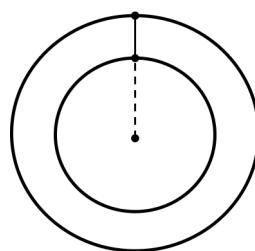
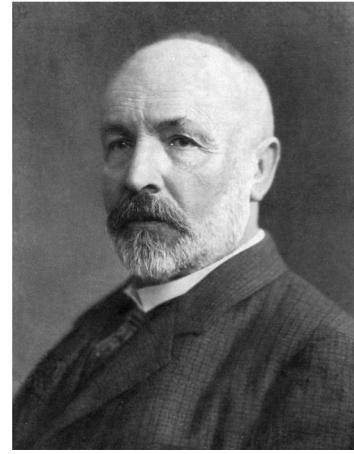


图 6.11: 同心大圆上任意一点都可通过半径对应到小圆上的唯一点

由于这些悖论的出现，人们接受了亚里士多德的观点，采取回避的态度。伽利略发现无法解释自然数和平方数孰多孰少后说：因此我们不能说自然数构成一个集合。人们拒绝“全体自然数”这类说法，否认实无限的存在。

终于，在伽利略身后两百年，德国数学家康托尔带领人们闯入了无穷王国。康托尔重新思考了陷入矛盾的一系列悖论，看起来问题的关键在于人们的常识——即整体一定大于部分。这一定是对的么？近代科学的发展一再告诉我们，人们根深蒂固用于具体事物的常识有时并不适用于更加抽象的概念。相对论挑战着人们的时空观常识——我们的所在的空间并不一定是熟悉的欧几里得空间，量子力学挑战着人们的因果观常识——随机性主导着量子世界。常识一旦突破，就会打开一片从未见过的新天地，从而带来巨大的认知进步。康托尔大胆地思考，如果我们接受“部分能够等于整体”的观点，通向无穷的大门就打开了。他提出“可以通过一一对应的方法来比较两个集合的大小，实无限是确实存在的概念。”

为了比较两个集合的大小，康托尔定义：如果能够建立集合 $M$ 和集合 $N$ 中元素的一一对应关系，那么这两个集合具有相同的基数（cardinal number），或者说它们是“等势”的<sup>4</sup>。对于有限集，显然这一结论是正确的，推广到无限集，根据这一定义，全体偶数和全体自然数一样多！完全平方数和自然数一样多，小圆上的点和大圆上的点也是一样多……甚至康托尔的挚友戴德金干脆这样定义无穷集合：如果一个集合的部分和整体可以具有相同的基数，那么这个集合是无穷集合。



格奥尔格·康托尔 (1845-1918)

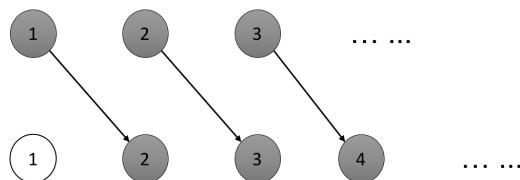
### 6.3.1 无穷王国的花园

让我们来欣赏一下康托尔为我们打开的无穷王国中的花园。数学家希尔伯特为了帮助人们理解康托尔的无穷集合概念，在1924年的国际数学家大会上，讲了这样的一个故事。

有一个神奇的旅馆，拥有无穷多的房间。在旅游旺季的时候，旅馆住满了客人，已经满员了。这一天晚上，又来了一名客人。要是一般的旅馆，就只能拒绝这个新客人入住，让他另找一家了。旅店经理希尔伯特说：“没问题，住得下。”他于是指挥客人，让1号房间的客人搬到2号房间去，2号房间的客人搬到3号房间去，3号房间的客人搬到4号房间去……每个房间的客人都搬到下一个房间去。这样就空出了1号房间给新来的这位客人。

故事没有结束。第二天，旅店迎来了一个神奇的旅游团，不同于普通的旅游团，它有无穷多个游客。酒店经理希尔伯特又说：“没问题，住得下。”他指挥房间里的客人。让昨天住进1号房间的那位新客人搬到2号房间去，让2号房间的客人搬到4号房间去，让3号房间的客人搬到6号房间去……每个房间的客人都搬到房间号2倍的那个房间去。由于酒店有无穷多的房间，所以这样一搬，2, 4, 6, …这些偶数房间住着原来的客人。而1, 3, 5, …这些奇数房间空出来，有无穷多间。刚好可以让旅游团的人一人一间住下。

故事更加曲折了。到了第三天，旅馆门前车水马龙，来了无穷多的神奇旅游团，每个旅游团都有无穷多个游客。希尔伯特的无穷旅馆还能住下么？在揭晓答案之前，我们先回顾一下前两天的故事。



如图6.13所示，在第一天的故事中，我们让每个客人搬到下一个房间去，从而空出1号房间。实际上，这建立了图中上下两行灰色圆形之间的一种对应关系 $n \leftrightarrow n + 1$ 。这告诉我们一个有趣的事，无穷加上1还是无穷。不仅如此，即使来了有限多名 $k$ 位新客人，我们可以重复这一“腾笼换鸟”的过程 $k$ 次，仍然能够让满员的宾馆住下他们。这相当于：

$$\begin{aligned}\infty + 1 &= \infty \\ \infty + k &= \infty\end{aligned}$$

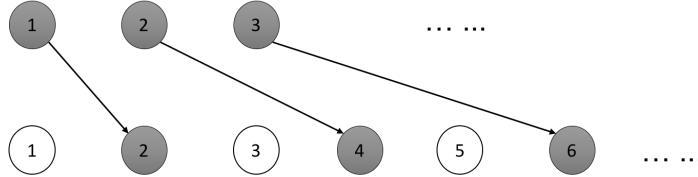


图 6.14: 希尔伯特无穷旅馆的第二天

第二天的故事如图6.14所示，我们实际上建立了自然数到偶数间的一一映射，从而空出了无穷多间奇数房间，而这些空房间又和旅游团的无穷多位客人之间建立了一一映射，这恰好是奇数到自然数间的一一映射。第二天的故事告诉我们，无穷加上无穷仍然是无穷。

$$\infty + \infty = \infty$$

尽管符号相同，我们还是不禁会问，等号左边的无穷和等号右边的无穷是相同的么？无穷之间还能再比较大小么？我们稍后会看到，正是这个问题，导致了康托尔的进一步研究。在希尔伯特旅馆的问题中，这些无穷之间都可以建立一一对应关系，因此它们是“相等”的。我们把和自然数一一对应的无穷叫作可数无穷。

要想解决希尔伯特旅馆第三天的问题，我们需要思考能否在无穷个无穷旅游团和无穷个房间之间建立一一对应。有人说，可以先让第一个旅游团依次入住1, 2, ...号房间，然后让第二个旅游团的第一个客人住在 $\infty + 1$ 号房间，第二个客人住 $\infty + 2$ 号房间……以此类推。但是这样的思路是行不通的，我们事先并不知道究竟是房间多，还是客人多。考虑安排客人的过程，第二个旅游团的第一个客人永远不知道第一个旅游团何时入住完，从而确定自己应该搬入的房间。与之相反，在第一天的故事中，一旦1号房间的客人搬入2号房间，新客人就可以立即入住了。尽管每个客人都依次搬入下一个房间是一个无穷无尽的过程。第二天的故事中也是如此，原1号房间的客人搬到2号房间的同时，旅游团中的第1个客人就可以搬入空出来的1号房间了，接下来原2号房间的客人搬到4号房间，原3号房间的客人搬到6号房间，此时旅游团中的第2个客人就可以搬入3号房间了……

图6.15给出了一种编号方案。为了方便，我们让每个旅游团的第一个客人编号为0，第二个客人编号为1，第三个客人编号为2……，我们把已经在旅馆中入住的客人编号为0号旅游团，新来的第一个旅游团编号为1号团，第二个旅游团编号为2号团……在这个图中，每个客人就对应无穷伸展的方格子中的一个点。同样我们让旅馆的房间号也从0开始。

现在我们按照这个顺序安排入住：第0号团的第0号客人入住0号房间，第0号团的第1号客人入住1号房间，第1号团的第0号客人入住第2号房间，第2号团的第0号客人入住3号房间，第1号团的第1号客人入住4号房间……这样按照图中往返画“之”字形，就可以逐一、并且毫无遗漏地安排每个客人入住。我们在无穷个旅游团的每个客人和无穷个房间之间建立了一一映射。希尔伯特的旅馆神奇地容纳了“二维”的无穷。

### 练习 6.3

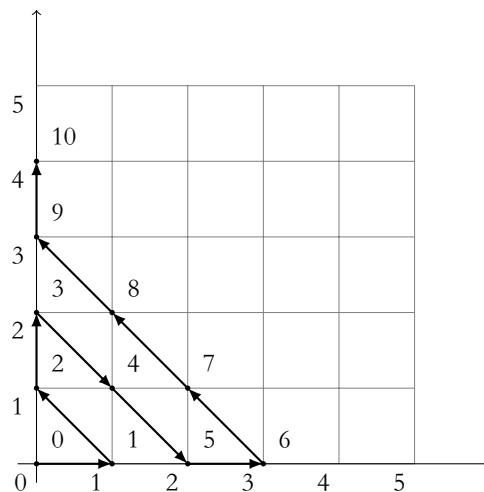


图 6.15: 对无穷个无穷的一种编号方案

1. 我们用图6.15建立了房间和任意旅游团的客人间的一一映射。第*i*号旅游团的第*j*号客人应该入住几号房间？第*k*个房间里住了哪号旅游团的哪位客人？
2. 希尔伯特旅馆第三天的故事的解法并不唯一，图6.16是《无需语言的证明》一书的封面。试根据此图给出另一种编号方案？

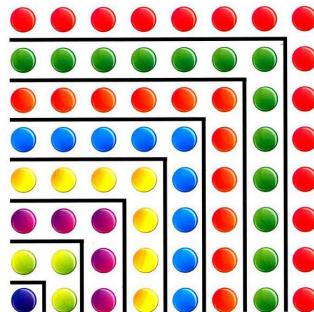


图 6.16: 《无需语言的证明》封面局部

### 6.3.2 一一对应与无穷集合

从希尔伯特旅馆的故事中，我们看到一一对应的概念对于研究无穷的重要性。如果能建立两个集合间的一一对应关系，则它们具有相同的基数。进一步我们可以用一一对应对集合进行分类。我们曾经在第三章介绍过相应的概念。具体说就是在两个集合  $A$  和  $B$  之间建立一个映射  $A \xrightarrow{f} B$ ，使得  $A$  中的每个元素  $x$ ，都可以通过映射  $x \mapsto y = f(x)$ ，与  $B$  中的元素  $y$  关联起来。对于集合，我们也称映射  $f$  为函数。 $y$  叫作  $x$  的像，而  $x$  叫作原像。如果原像是唯一的，我们称这样的映射为单射；如果  $B$  中的任何  $y$  都存在原像，我们称这样的映射为满射。既是单射又是满射的映射称为一一映射。图6.17描述了两个有限集合间的一个一一映射。

希尔伯特的无穷旅馆系列故事既让人感到神奇和震惊，也让人困惑。自然数不仅和它的无穷子集奇数和偶数同样多，并且一维的自然数  $n$  和二维的自然数对  $(m, n)$  也同样多。这意味着什么呢？

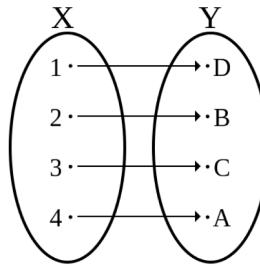


图 6.17: 集合间的一一对应

康托尔发现，他可以从自然数开始，利用一一对应，扩展出一系列的无穷集合。我们来看一下这些例子。

1. **整数**。如果我们建立这样的一一对应：

$$\begin{array}{ccccccc}
 0 & 1 & -1 & 2 & -2 & \dots & n & -n & \dots \\
 \downarrow & \uparrow & \downarrow & \uparrow & \downarrow & & \downarrow & \uparrow & \\
 0 & 1 & 2 & 3 & 4 & \dots & 2n-1 & 2n & \dots
 \end{array}$$

这样就把整数和自然数对应起来。换一种角度，我们实际上用奇数对应了全部正整数，用偶数对应了零和全部负整数。这说明全体整数和自然数一样多。换言之，我们可以用自然数构造整数。

2. **有理数**。我们知道，有理数又叫作可比数，可以写成  $p/q$  的形式，其中  $q \neq 0$ 。回想希尔伯特无穷旅馆第三天的故事，用同样的方法，我们可以在任何数偶  $(p, q)$  和自然数之间建立一一映射。我们只要稍作修改就可以建立有理数  $p/q$  和自然数间的一一对应。先不考虑负数的情形，每当遇到第二个数  $q$  等于 0 就跳过，如果  $p/q$  不是既约分数（含有公因子）就跳过。然后我们复用扩展整数的方法，再把负有理数包含进来，这样就从自然数构造出了全体有理数。例如下表给出了前几个自然数到有理数的对应关系：

$$\begin{array}{cccccccccc}
 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \dots \\
 \downarrow & \uparrow & \downarrow & \uparrow & \downarrow & \uparrow & \downarrow & \uparrow & \downarrow & \\
 0 & 1 & \frac{1}{2} & -\frac{1}{2} & -1 & -2 & -\frac{2}{3} & -\frac{1}{3} & \frac{1}{3} & \dots
 \end{array}$$

这说明自然数和有理数同样多。

3. **代数数**。所谓代数数，是指可以通过代数方程求解出的数。简单说，就是可以通过有限次加减乘除、乘方开方得到的数。因此  $\sqrt{2}$  和  $1 \pm \sqrt{3}i$  是代数数，而  $\pi, e$  都不是代数数。因此，任意给定一个整系数代数方程：

$$a_0x^n + a_1x^{n-1} + \dots + a_n = 0$$

其中  $a_0, a_1, \dots, a_n$  都是整数，且  $a_0 \neq 0$ 。它的所有根都是代数数。我们构造一个正整数：

$$h = n + |a_0| + |a_1| + \dots + |a_n|$$

也就是次数和方程系数绝对值的和。我们称  $h$  为方程的高。因此，对于任意的代数方程，高  $h$  总是一个确定的自然数。反之，给定一个自然数  $h$ ，对应的方程却不止一个。例如： $x - 3 = 0, x^3 + 1 = 0, x^3 - 1 = 0, x^2 + x + 1 = 0, x^2 - x + 1 = 0$  的高都是 4。但是对于固定的  $h$ ，以它为高的代数方程只有有限多个。因此我们可以把所有的代数方程枚举出来，先枚举  $h = 1$  的代数方程，在枚举  $h = 2$  的代数方程……如此下去，就可以把所有的代数方程都枚举出来。

举出来。当然，高同样的方程可以按任意顺序排列。根据高斯证明的代数基本定理，方程根的个数等于它的次数 $n$ 。如果考虑重根，则不同的根不超过 $n$ 。于是高为 $h$ 的代数方程的根只有有限个。现在我们就可以枚举所有代数数了。

首先把高 $h = 1$ 的代数方程（只有一个 $x = 0$ ）的根枚举出，就是0，再把高为2的方程的所有根枚举出。注意，不同方程的某个根可能是相等的，如果我们遇到某个根，在此前已经枚举过了，我们就将它跳过。这样就把代数数同自然数一一对应起来了。因此，自然数和代数数同样多。换言之，我们可以通过自然数扩展出全部代数数。

接下来我们遇到了难题，我们能否从自然数扩展到实数？不仅包括普通的无理数，还包括 $\pi$ ,  $e$ 这样的超越数？解决这个难题的是康托尔和戴德金。再进一步介绍这些伟大的成果之前，让我们先了解一下这两位数学家。

### 6.3.2.1 康托尔与戴德金

康托尔是德国数学家，集合论的创始人。1845年3月3日生于俄罗斯圣彼得堡，他一生的大部分时间是在德国度过的。康托尔的父亲是具有犹太血统的丹麦商人，母亲出身艺术世家。1856年举家迁居德国的法兰克福。尽管康托尔较早就显示出了数学才能，他的父亲却希望他成为一名工程师，因为这样更容易求职和谋生。直到17岁时，康托尔的父亲才逐渐意识到不应该把自己的意见强加给儿子。他得到父亲的允许，进入大学学习数学。康托尔给父亲回信道：“你自己也能体会到你的信使我多么高兴。这封信确定了我的未来……现在我很幸福，因为我看到如果我按照自己的感情选择，不会使你不高兴。我希望你能活到在我身上找到乐趣，亲爱的父亲；从此以后我的灵魂，我整个人，都为我的天职活着；一个人渴望做什么，凡是他的内心强制他去做的，他都会成功！”[9]

1862年，康托尔进入苏黎世大学，1863年转入柏林大学攻读数学和神学，受教于大数学家库默尔、魏尔斯特拉斯、和克罗内克。1866年曾去哥廷根学习一个学期。

1867年他在库默尔的指导下以解决整系数不定方程的论文获得博士学位。当时的数学界正在魏尔斯特拉斯的领导下进行着重建微积分严密基础的运动，康托尔也很快转入这一研究方向。在工作中，他意识到必须研究作为微积分基础的实数点集，这成为了集合论研究的开端。

1872年康托尔在瑞士旅游中偶遇了数学家戴德金。两人后来成为亲密的朋友，彼此通过信件交流，互相支持。1874年29岁的康托尔发表了关于集合论的第一篇革命性论文。康托尔开始显示他非凡的独创力。在随后的十几年中，他几乎独自一人把集合论推向深入，引领了数学中无穷的革命。在他最伟大的、最具创见的时期，康托尔却没有能得到应有的认可。他没能取得柏林大学的教授职位。他的大部分研究时光是在哈雷大学度过的。这是一所不大出名、薪金微薄的二流学院。他的成果在当时很难被理解。由于太过颠覆传统，加之无穷集合引发了一些悖论（我们将在下一章详细介绍罗素悖论），人们对集合论的基础和可靠性产生了严重的怀疑。康托尔遭到了许多人的反对，其中反对最激烈的是他的老师，柏林学派的代表人物克罗内克。克罗内克有一句名言：“上帝创造了自然数，其余都是人的工作。”他批判康托尔的无穷集合和超限数理论不是数学而是神秘主义，是一类危险的数学疯狂。他认为数学在康托尔的领导下正在走向疯人院。除了克罗内克之外，还有一些著名的数学家也对集合论发表了反对意见，包括法国著名数学家——被称为最后一个数学通才的庞加莱，他说：“我个人，而且还不只我一个人，认为重点在于，切勿引进一些不能用有限个文字去完全定义好的东西。”他把集合论当作一个有趣的“病理学的情形”来谈，并且预测说：“后一代将把集合论当作一种疾病，而人们已经从中恢复过来了”。德国数学家赫尔



康托尔摄于1870年的照片

曼·外尔认为，康托尔关于基数的等级观点是“雾上之雾”。克莱因也不赞成集合论的思想。施瓦兹原来是康托尔的好友，但他由于反对集合论而同康托尔断交。集合论的悖论出现之后，一些数学家开始认为集合论根本是一种病态，他们以不同的方式发展为经验主义、直觉主义、构造主义等学派，在数学的基础大战中，构成反康托尔的阵营。

于是悲剧的结局不是集合论进入了疯人院，而是康托尔进入了疯人院。1884年5月，他支持不住了，第一次精神崩溃。在他一生的随后岁月中，这种崩溃以不同强度反复发生，把他从社会上赶进精神病院这个避难所。1904年在两个女儿的陪同下，他出席了第三届国际数学家大会。会上他的精神受到严重刺激，立即被送进医院。在他生命的最后十年，他大都处于严重的抑郁状态中，并在哈雷大学的精神病诊所度过了漫长的岁月。他最后一次住进精神病院是1917年5月，直至1918年1月去世。

康托尔的集合论得到公开的承认和热情的称赞应该说首先在瑞士苏黎世召开的第一届国际数学家大会上表现出来。瑞士苏黎世理工大学教授胡尔维茨（Hurwitz, Adolf, 1859–1919）在他的综合报告中，明确地阐述康托尔集合论对函数论的进展所起的巨大推动作用，这破天荒第一次向国际数学界显示康托尔的集合论不是可有可无的哲学，而是真正对数学发展起作用的理论工具。在分组会上，法国数学家阿达玛也报告康托尔对他的工作的重要作用。随着时间的推移，人们逐渐认识到集合论的重要性。希尔伯特高度赞誉康托尔的集合论“是数学天才最优秀的作品”，“是人类纯粹智力活动的最高成就之一”，“是这个时代所能夸耀的最巨大的工作”。在1900年第二届国际数学家大会上，希尔伯特高度评价了康托尔工作的重要性，并把康托尔的连续统假设列入20世纪初有待解决的23个重要数学问题之首。当康托尔的朴素集合论出现一系列悖论时，直觉主义学派的代表布劳威尔等人借此再次发难，希尔伯特用坚定的语言向他的同代人宣布：“没有任何人能将我们从康托尔所创造的伊甸园中驱赶出来”。



理查德·戴德金（1831-1916）

戴德金是德国数学家、教育家。1831年10月6日生于德国的不伦瑞克镇的一个知识分子家庭。这里也是著名数学家高斯的故乡。戴德金的父亲为法学教授，母亲也是一位教授的女儿。1850年戴德金到哥廷根大学师从著名的数学大师高斯研究最小二乘法和高等测量，向斯特恩学习数论基础，向韦伯学习物理，他还选修过天文学。1852年他在高斯的指导下获得博士学位，博士论文是《关于欧拉积分的理论》。在哥廷根求学期间，他还结识了黎曼、狄利克雷。在于这些世界级的数学家交流中，他获益匪浅，并逐渐萌生了用算术性质来重新定义无理数的想法。1854年戴德金留校任代课教师。如第三章所说，他很早就在教学中讲授伽罗瓦理论并引入了域的概念。

戴德金为人谦逊。他的许多成果并不为当时的人所知。例如在狄利克雷去世之后，戴德金根据当时听讲的笔记整理出了《数论讲义》这一名著。尽管戴德金在后

继版本中添加了很多结论，但他却谦逊地将这本书记在了狄利克雷名下。令人遗憾的是，这种做法对其职业生涯造成了影响。他没有能获得哥廷根大学的终身教席，而是去了一家比较小的技术大学（[8]第119页）——在他家乡不伦瑞克的高等工业学院。

事物总是两面性的，在不伦瑞克，戴德金感觉自己有从事数学研究的充分时间与足够的自由。1888年，戴德金提出了算术公理的完整系统，其中包括完全数学归纳法原理的准确表达方式。戴德金的主要成就是在代数理论方面。他研究过任意域、环、群、结构及模等问题，并在授课时率先引入了环（域）的概念，并给理想子环下了一般定义，提出了能和自己的真子集建立一一对应的集合是无穷集的思想。在研究理想子环理论过程中，他将序集（置换群）的概念用抽象群的概念来取代，并且用一种比较普通的公式（戴德金分割概念）表示出来，直接影响了后来皮亚诺的自然数公理的诞生。是最早对实数理论提出了许多论据的数学家之一。现今数学上的许多命题和术语，如群、环、域、结构、模、理想、函数、定理等，都是与他的名字联系在一起的。

戴德金于1916年2月12日去世。说到他的去世，有一则趣闻。有一天，戴德金发现托博纳写的

《数学家传记》中，赫然写到：1897年9月4日戴德金去世。出于纠正这个错误的想法，他给传记的编辑写去了一封信：“根据我本人的日记，我在这一天非常健康，而且与我的午餐客人，尊敬的朋友康托尔一起谈论着一些趣事，过得非常愉快。”<sup>[9]</sup>

即使在当代，数学界对于康托尔和戴德金学派的观点仍存在分歧。迪厄多内在1980年代仍然认为戴德金的工作引起了不必要的混乱。更不用说二十世纪之初的激烈分歧和争执了。我们今天看到的大多数传记和评论往往过于批判责备克罗内克、布劳威尔及其代表的直觉主义，而同情康托尔的不幸遭遇，并热情赞颂无穷集合和超限数的革命性创造。作为今天理性的读者，我们建议大家独立思考，而不要被一边倒的观点左右。克罗内克和康托尔都是虔诚的宗教信徒，克罗内克对数学哲学有着强烈的信念，试图将一切数学(从代数学到分析)算术化是他的最高愿望。他写道：“有一天人们将成功地将所有数学算术化，就是说将数学建立在最狭义的数概念的单一基础上。”他相信人们在各种数学分支中能够也必须以这种方式限定一个定义，即人们可用有限步验证它是否适用于任意已知量。同样，一个量的存在性证明只有当它包含一种方法，通过它可以实际地发现要证明存在的量时，才可被认为是完全严格的。这些正是后来重要的数学哲学流派——直觉主义学派所坚持的信念。因此，克罗内克被认为是直觉主义学派的先驱。他的这些原则也正是现代数学的重要领域——构造性数学研究的起点。他的怀疑精神对人们重新批判地检查数学的基础起了鼓舞作用。它导致了数学中两种有建设性的批判运动：有限步构造性证明与存在性证明，以及从数学中驱除不能以有限个词明确表述的定义。这些有利于人们更清楚地认识数学的本质。庞加莱的观点也是值得我们深入思考的。他在《科学与假设》这本书中阐述了他的哲学思想。他认为公理是一种约定。在思考物理学的基本定义，如力的定义时他说，如果一个定义不能让我们去测量它，这个定义就没有任何用处。庞加莱的思想直接影响了爱因斯坦和相对论。

### 6.3.2.2 利用（可数）无穷定义斐波那契数列和哈明数列

有些编程环境，所有的求值默认都是惰性的，在这样的环境中，我们甚至可以直接拿潜无穷流进行复杂的计算。例如下面是另一种自然数的定义方法：

$$N = 0 : map(succ, N)$$

它的含义是， $N$ 是一个潜无穷，代表自然数。第一个自然数是0，从第二个自然数开始，后面每个自然数，都等于前一个自然数的后继。如同下面的表格：

	$N:$	0	1	2	...
	$map(succ, N):$	$succ(0)$	$succ(1)$	$succ(2)$	...
$0 : map(succ, N):$		0	1	2	3

例如下面的代码先定义了自然数的潜无穷，然后取出其中的前10个自然数：

```
nat = 0 : map (+1) nat
take 10 nat
[0,1,2,3,4,5,6,7,8,9]
```

类似地，我们也可以用潜无穷定义斐波那契数列。假设 $F$ 是全体斐波那契数列，我们知道它的第一个元素是0，第二个是1，后面每个斐波那契数都是前面两个的和。我们可以仿照自然数的例子列出下面的表格：

	$F:$	0	1	1	2	3	5	8	...
	$F':$	1	1	2	3	5	8	13	...
0	1	1	2	3	5	8	13	21	...

其中第一行为全体斐波那契数。第二行为去掉一个元素后的全体斐波那契数。也可以这样想：把第一行向左移动一格就得到了第二行。第三行最有趣，把前两行每列加起来，我们就又得到了全体斐波那契数，只不过缺了最开始的两个数0和1。所以把它们两个补在第三行的最左边。把第三行翻译过来就是斐波那契数的潜无穷流定义：

$$F = \{0, 1\} \cup \{x + y \mid x \in F, y \in F'\}$$

或者写成代码：

我们再给出一个稍微复杂些的例子，数学上把只含有2、3、5这三个因子组成的自然数叫作正规数。在计算机科学中，也常常叫作哈明数以纪念美国数学家，图灵奖获得者理查德·哈明

写一个计算机程序来产生哈明数并不简单，然而利用无穷流，我们可以得到一个直观高效的解法。假设全体哈明数的潜无穷流为 $H$ 。我们知道第一个哈明数是1，后面的哈明数可以这样构造，我们把 $H$ 中的每个数都乘以2，仍然是哈明数，记作 $H_2 = \{2x|x \in H\}$ ，类似地我们可以定义 $H_3 = \{3x|x \in H\}$ 和 $H_5 = \{5x|x \in H\}$ ，如果把这三个新序列中的数，从小到大合并起来，去掉重复的，并且在前面补上1，就又得到了全体哈明数。

$$\begin{aligned} H &= \{1\} \cup H_2 \cup H_3 \cup H_5 \\ &= \{1\} \sqcup \{2r \mid r \in H\} \sqcup \{3r \mid r \in H\} \sqcup \{5r \mid r \in H\} \end{aligned}$$

其中 $\cup$ 的含义就是从小到大、去掉重复、合并两个无穷序列  $X = \{x_1, x_2, \dots\}$  和  $Y = \{y_1, y_2, \dots\}$ ；

$$X \cup Y = \begin{cases} x_1 < y_1 : & \{x_1, X' \cup Y\} \\ x_1 = y_1 : & \{x_1, X' \cup Y'\} \\ x_1 > y_1 : & \{y_1, X \cup Y'\} \end{cases}$$

写成代码就是：

### 6.3.3 可数无穷与不可数无穷

迄今为止，我们用自然数构造了整数、有理数、和包含部分无理数的代数数。它们都和自然数之间存在一一映射，或者说和自然数一样多。我们把和自然数等势的无穷集合叫作可数无穷。是不是所有的无穷集合都是可数的呢？是否存在更大的无穷呢？1873年11月29日康托尔给戴德金的一封信中提到了这个问题：“取所有的自然数集合，记为 $N$ ，然后考虑所有实数的集合，记为 $R$ 。简单来说，问题就是两者是否能够对应起来，使得一个集中的每一个体只对应另一集中一个唯一的个体？乍一看，我们可以说答案是否定的，这种对应不可能，因为前者由离散的部分组成，而后者则构成一个连续统。但从这种说法里我们什么也得不到。虽然我非常倾向认为这两者不能有这样的一个一一对应，但是我找不出理由，我对这事极为关注，也许这理由非常简单。”

一个星期后的12月7日，在写给戴德金的信中，康托尔自己回答了这个问题，他发现实数集合不能和自然数集合构成一一对应。这一天可以看作是集合论的诞生日。康托尔曾经给出过两个证明，其中第二个证明最为脍炙人口，就是大名鼎鼎的“对角线证明”。

康托尔首先使用反证法，假设区间(0, 1)上的全部实数是可数的，可以和自然数一一对应。那么就可以把这个区间里的全部实数列出来，形成一个序列  $a_0, a_1, a_2, \dots, a_n, \dots$ 。现在将这个序列中的每个实数都表示成小数形式。如果是无理数，它的小数形式是无限不循环的；如果是除不尽的分数，则其小数形式是无限循环的，例如  $\frac{1}{3} = 0.\overline{333\dots}$ ；如果能够除尽，我们就在后面补无穷多个零，例如  $\frac{1}{2} = 0.5000\dots$ 。于是实数区间(0, 1]中的所有实数可以排成下面的序列：

$$\begin{aligned} a_0 &= 0.a_{00}a_{01}a_{02}a_{03}\dots \\ a_1 &= 0.a_{10}a_{11}a_{12}a_{13}\dots \\ a_2 &= 0.a_{20}a_{21}a_{22}a_{23}\dots \\ a_3 &= 0.a_{30}a_{31}a_{32}a_{33}\dots \\ &\dots \\ a_n &= 0.a_{n0}a_{n1}a_{n2}a_{n3}\dots \\ &\dots \end{aligned}$$

有一点我要提醒一下读者： $a_0, a_1, a_2, \dots$ 并不一定按照大小次序排列的。现在构造一个数  $b = 0.b_0b_1b_2b_3\dots b_n\dots$ ，使它的第  $n$  位数字  $b_n \neq a_{nn}$ 。为了做到这一点，我们可以规定一个很简单的规则，例如若  $a_{nn} \neq 5$ ，就让  $b_n = 5$ ，否则就让  $b_n = 6$ ，即：

$$b_n = \begin{cases} 5 & : a_{nn} \neq 5 \\ 6 & : a_{nn} = 5 \end{cases}$$

这样构造出来的数  $b$  一定不等于上述序列中的任何一个数。因为至少它们的第  $n$  位数字不相同。也就是对角线上的数字至少不同。我们把对角线上的数字写成黑体，这样很明显了。

$$\begin{aligned} a_0 &= 0.\mathbf{a}_{00}a_{01}a_{02}a_{03}\dots \\ a_1 &= 0.a_{10}\mathbf{a}_{11}a_{12}a_{13}\dots \\ a_2 &= 0.a_{20}a_{21}\mathbf{a}_{22}a_{23}\dots \\ a_3 &= 0.a_{30}a_{31}a_{32}\mathbf{a}_{33}\dots \\ &\dots \\ a_n &= 0.a_{n0}a_{n1}a_{n2}a_{n3}\dots\mathbf{a}_{nn}\dots \\ &\dots \end{aligned}$$

我们此前假设(0, 1)间的所有实数都被逐一列出了，无一遗漏。 $b$  显然属于这一区间，但它却不同于任何一个  $a_i$ 。这说明我们假设的一一映射遗漏了  $b$ ，导致了矛盾，所以假设不成立，我们无法把这一区间的所有实数和自然数之间构造一一映射。由于利用了对角线上的数字都不相等的这一事实，这一证法称作康托尔对角线证明。

有人说，把  $b$  加进  $a_0, a_1, a_2, \dots$  中去不就可以了么？假设  $b$  加进去后，处于第  $m$  个位置，我们仍然可以再次构造一个新数  $c$ ，只要让它的第  $m$  位不等于  $b_m$  就又出现了一个没有包含的数。

这一证明简单、直观。它揭示了一个惊人的事实：(0, 1)间的实数集是不可数的！这是我们发现的第一个比自然数集更大的无穷集合。接下来，我们构造一个一一映射： $y = \pi x - \frac{\pi}{2}$ 。它把区间(0, 1)中的每个实数，映射到区间 $(-\frac{\pi}{2}, \frac{\pi}{2})$ 中，无一遗漏。因此我们立即得知这一区间内的实数集是不可数的。接下来，我们压上最后一根稻草。再构造一个一一映射： $y = \tan(x)$ 。它把区间 $-\frac{\pi}{2}$ 到 $\frac{\pi}{2}$ 中的每一个实数，无一遗漏地映射到了全体实数集上。康托尔得到了他的重要结论：实数集不再是可数集，它是比可数集更高等级的无穷。康托尔称之为不可数集，记作  $C$ 。

这自然让人联想到线段上的点。在欧几里得的《几何原本》中，点被定义为没有大小的部分，而直线被认为由点组成。根据希帕索斯的发现，我们知道直线上存在无理数。或者说有理数不能够填充直线。而实数是可以填充线段的。我们在下一节中，还会介绍戴德金分割，从而给出实数的严密化定义。上面的证明告诉我们，单位长度线段上的点，和任何长度线段上的点，以及无限

长的直线上的点，也就是数轴上稠密的点是一样多的。都是不可数集。同心圆上的点也是同样多的，它们也都是不可数集。

同样令人吃惊的是无理数与有理数多少的比较。直观上思考，任何两个有理数之间，存在着无穷多的无理数；任何两个无理数之间，也存在着无穷多的有理数，我们会觉得无理数和有理数应该一样多。然而康托尔的结论告诉我们，有理数是可数的，而无理数是不可数的。这说明无理数远远多于有理数。再进一步，我们前面证明了代数数是可数的，这说明像 $\pi, e$ 这样的超越数是不可数的，它们要远远多于代数数。

在希尔伯特无穷旅馆第三天的故事中，我们发现一维的可数无穷可以和二维的无穷多格子点对应起来。并以此为基础证明了有理数是可数无穷。但是一维线段上代表实数的点和二维平面上的点谁多谁少？还是同样多呢？1874年1月，康托尔在写给戴德金的信中提出了这个问题。他几乎肯定的觉得，二维正方形比一维的线段包含更多的点。但是却没能给出证明。时光匆匆过了四年之后，康托尔惊奇地发现，他此前结论是错误的，并且找到了一个有趣的一一对应。1877年6月他写信给戴德金，请审查他的证明。在信中，康托尔说出了本章开头那句著名的话：“我看到了，但我不相信。”

我们接下来看看这个一一对应是如何展示“一沙一世界”的奇观的。我们面对的两个无穷点集一个是单位正方形

$$E = \{(x, y) | 0 < x < 1, 0 < y < 1\}$$

另一个是单位线段 $(0, 1)$ 。取单位正方形内的任意一个点 $(x, y)$ ，然后把 $x$ 和 $y$ 都表示成无穷小数（如果是有限小数，比如0.5，就写成0.4999...，请参考本节习题）。现在把 $x, y$ 的小数部分分成一组一组的，每组都终止在第一个非0数字上。例如：

$$\begin{array}{ccccccc} x = & 0.3 & 02 & 4 & 005 & 6 & \dots \\ y = & 0.01 & 7 & 06 & 8 & 04 & \dots \end{array}$$

然后我们构造一个数字 $z = 0.3\ 01\ 02\ 7\ 4\ 06\ 005\ 8\ 6\ 04\dots$ 。这个构造过程交错地从 $x$ 和 $y$ 的各组中取数字，无一遗漏。也就是先写下0和小数点，然后取 $x$ 的第一组，也就是3，然后取 $y$ 第一组，也就是数字01，然后取 $x$ 的第二组02，接下来取 $y$ 的第二组7……， $z$ 显然在单位线段内。对于单位正方形内不同的点，其小数表示 $x$ 或 $y$ 上必有不同的数字。因此对应的 $z$ 也是不同的。这说明 $(x, y) \mapsto z$ 是单射。反过来，对于单位线段上任意点 $z$ ，我们也可以把 $z$ 的无穷小数形式像上面那样分组，然后把奇数组取出放在0和小数点后构成 $x$ ，把偶数组取出构成 $y$ 。则 $(x, y)$ 是单位正方形内的点。这说明 $(x, y) \mapsto z$ 是满射。因而这个映射是一一映射。于是我们证明了单位长度线段内的点和二维平面上的点具有相同的基数，它们同样多，都是不可数无穷。

仿照此方法，我们接下来可以证明不仅线段和平面上的点同样多，它和三维空间中的点也同样多。甚至一般的 $n$ 维空间中的点也和线段上的点同样多。

在康托尔以前，尽管有争议，但是人们只能区分出有限和无穷。没有人想过无穷之中还有区别。康托尔第一个向我们揭示，无穷也是可以分类的。存在着可数无穷和不可数无穷。康托尔并没有止步，接下来的问题是：存在比不可数无穷更大的无穷么？我们在寻找无穷等级的道路上走到终点了么？在展示进一步的结论前，我们先了解一下戴德金关于实数别出心裁的定义。

## 练习 6.4

- 令 $x = 0.9999\dots$ ，则 $10x = 9.9999\dots$ ，做减法得 $10x - x = 9$ ，解方程得 $x = 1$ 。因此得到结论 $1 = 0.9999\dots$ 。这一证明正确么？

### 6.3.4 戴得金分割

为了解决微积分基础的严密性问题，十九世纪的数学家们开始重新思考牛顿、莱布尼茨、雅可比、欧拉使用的一些令人困惑的概念。包括无穷小量和级数等等。经过柯西和魏尔斯特拉斯等人的工作，终于给出了严格的极限，收敛等概念。但有一个本质问题始终未得到圆满的解决。那就

是实数的概念。微积分是建立在实数连续性上的。可人们仍然没有一个关于实数满意的定义。最早人们把有理数比作直线，结果发现有理数间充满了间隙，它是不完备、不连续的。而我们则把直线看作没有间隙的、完备的和连续的。直线的连续性究竟是什么意思？人们迫切需要连续性的一个精确定义。

戴德金经过多年的思考，终于在1872年想出了一个方法——即著名的戴德金分割。戴德金指出，有理数具有稠密性，即任意两个有理数间，不管多么靠近，总存在着另外的有理数。但是有理数却不连续。连续的直线究竟意味着什么呢？此时让我们想象一把最锋利的“思想之刀”，在天衣无缝的直线上切下去，将它分割成两截（[9]，第196页）。

由于直线是连续的，天衣无缝的，不管多么锋利，这一刀一定落在某一点上，而不是在两点的缝隙间。（如果是有理数而非实数，这一刀可能落在某个有理数代表的点上，也可能落在两个有理数之间，比如恰好落在 $\sqrt{2}$ 代表的位置上。）假定从点A的位置上把直线切开，则A不在左边，就在右边。二者必居其一，不会两边都有，也不会两边都没有。这是因为点不可分割、也不会消失掉。换言之，直线的连续性意味着，不管从何处将直线切成两段，总是有一段是带有端点的，而另一段没有端点。

由此，戴德金定义了这样一个分割 $(A_1, A_2)$ ， $A_1$ 和 $A_2$ 分别叫作下类和上类。其中下类 $A_1$ 中的每个数都小于上类 $A_2$ 中的任意一个数。也就是说 $A_1$ 对应分割的左半段直线，而 $A_2$ 对应着右半段直线。对于这样的分割，要么 $A_1$ 中存在着一个最大数，要么 $A_2$ 中存在着一个最小数。二者必居其一，且仅居其一。

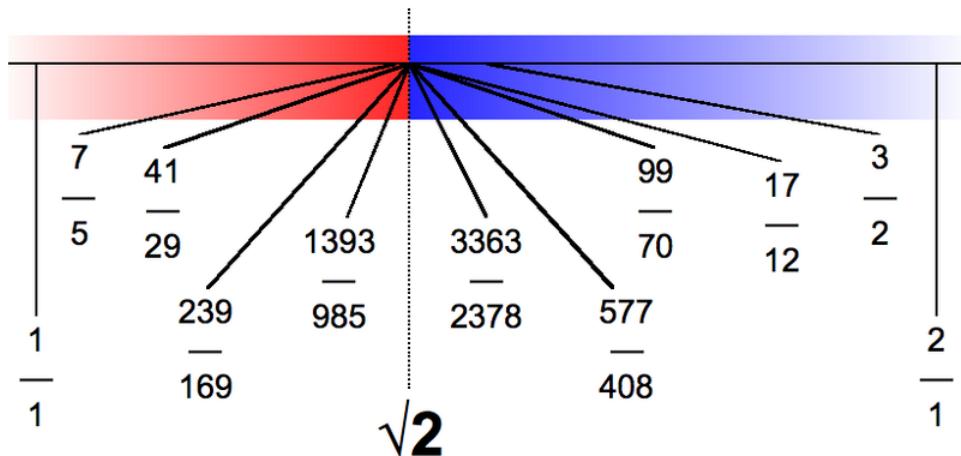


图 6.20: 用戴德金分割构造的 $\sqrt{2}$

用戴德金分割来分析全体有理数，会发现有理数是不连续的。如果 $A_1$ 包含了所有小于等于2的有理数， $A_2$ 包含了所有大于2的有理数，这一分割就定义了有理数2。但是考虑这样一个反例：下类 $A_1$ 包含所有负有理数，以及非负的，但平方小于2的有理数；上类 $A_2$ 包含剩余的有理数。不难发现在这一分割中，下类没有最大数，同时上类没有最小数。这说明有理数间存在着缝隙，从这里砍下去，这一刀就会落空。这个划分实际上确定了一个新数 $\sqrt{2}$ ，但它不是有理数。如图6.20所示。

于是戴德金得到了他的结论：有理数的每一个分割就叫作一个实数。带缝隙的分割（ $A_1$ 没有最大数， $A_2$ 没有最小数）叫作无理数；不带缝隙的分割（ $A_1$ 存在最大数，或 $A_2$ 存在最小数）叫作有理数。而实数正好包含有理数和无理数。由此戴德金分割定义了实数，直线上的每个点可以表示一个实数。戴德金的分割概念也给出了实数连续性的依据。

从毕达哥拉斯学派的希帕索斯发现无理数到戴德金最终给出实数的定义，时光经历了两千多

年<sup>5</sup>。在戴德金分割中，总是把数分成两个完成了的无穷整体，即无穷集合。这是对实无穷概念的运用和发展。

### 6.3.5 超限数和连续统假设

为了寻找更大的无穷，康托尔首先考虑了幂集。所谓幂集，就是一个集合的所有子集构成的集合。例如集合 $A = \{a, b\}$ ，则它的幂集包含 $\{\emptyset, \{a\}, \{b\}, \{a, b\}\}$ 一共四个元素。而有3个元素集合的幂集包含8个子集。一个集合中每个元素都可以被选中或者跳过以构造子集，这样具有 $n$ 个元素的集合的幂集大小为 $2^n$ 。显然有限集的幂集元素个数大于原集合。

康托尔在1891年证明了，当推广到无限集时，这一结论也成立。所以幂集的基数总是大于原集合的基数。这一定理现在被人们称作**康托尔定理**。这一证明并不困难。读者可以参考本章的附录，了解其详细过程。有了这一定理后，就打开了寻找更大的无穷的通路了。康托尔把自然数集等可数无穷的基数称作阿列夫零，记为 $\aleph_0$ ，阿列夫是希伯来文字母表的第一个字母。可数无穷的幂集基数记作 $2^{\aleph_0}$ ，并且由康托尔定理有 $\aleph_0 < 2^{\aleph_0}$ ，并且我们还可以通过幂集的幂集不断产生更大的无穷。

$$\aleph_0, 2^{\aleph_0}, 2^{2^{\aleph_0}}, \dots \quad (6.1)$$

#### 6.3.5.1 超限数

康托尔把存在等级的无穷基数序列叫作超限基数。所以希尔伯特神奇旅馆中揭示的超限数计算法则就是 $\aleph_0 + 1 = \aleph_0, \aleph_0 + k = \aleph_0, \aleph_0 + \aleph_0 = \aleph_0, \dots$

除了用幂集来产生更高等级的无穷外，康托尔还发现了另外一种方法。为此，我们需要引入序数的归纳定义：

1. 0是序数；
2. 如果 $a$ 是序数，则 $a \cup \{a\}$ 是序数，记作 $a + 1$ ，称作 $a$ 的后继；
3. 如果 $S$ 是序数的集合，也就是 $S$ 的元素都是序数，则 $\bigcup S$ 是序数；
4. 任何序数，都是通过上述1到3步获得的。

根据这个定义，从0开始的前几个序数如下：

$$\begin{aligned} 0 \\ 1 &= 0 \cup \{0\} \\ 2 &= 1 \cup \{1\} = 0 \cup \{0\} \cup \{0 \cup \{0\}\} \\ 3 &= 2 \cup \{2\} = 1 \cup \{1\} \cup \{1 \cup \{1\}\} = \dots \\ &\dots \end{aligned}$$

这里 $\bigcup S$ 称为集合 $S$ 的广义并。它是由 $S$ 的所有元素的元素组成的集合。根据序数定义的前两条，我们发现自然数 $0, 1, 2, 3, \dots, n, \dots$ 都是序数。令 $\omega$ 是自然数集合，由于自然数都是序数，所以 $\omega$ 也是一个序数的集合。我们考虑它的广义并：

$$\bigcup \omega = \{0, 1, 2, \dots\} = \omega$$

根据序数的第三条定义，说明 $\omega$ 也是一个序数，它是一个极限序数，并且是最小的无穷序数。我们把它添加到自然数的末尾就得到了一个新序列：

---

<sup>5</sup>同一年，魏尔斯特拉斯通过有界单调序列理论，康托尔通过有理数序列理论也都从有理数出发定义出无理数，从而构筑起了实数理论。可谓殊途同归。

$$0, 1, 2, \dots, \omega$$

从 $\omega$ 开始，再次重复使用序数的第二条定义，又可以得到序数列：

$$\omega + 1, \omega + 2, \omega + 3, \dots, \omega + n, \dots$$

将上面两个序列合并到一起组成一个集合，记作 $\omega \cdot 2$ 。不难发现其广义并 $\cup \omega \cdot 2 = \omega \cdot 2$ 。所以 $\omega \cdot 2$ 也是一个序数，并且是极限序数。从 $\omega \cdot 2$ 开始，继续重复上述过程，我们就得到了无限伸展的无穷序数列：

$$\begin{aligned}
& 0, 1, 2, \dots, n, \dots \\
& \omega, \omega + 1, \omega + 2, \dots, \omega + n, \dots \\
& \omega \cdot 2, \omega \cdot 2 + 1, \omega \cdot 2 + 2, \dots, \omega \cdot 2 + n, \dots \\
& \dots \\
& \omega \cdot k, \omega \cdot k + 1, \omega \cdot k + 2, \dots, \omega \cdot k + n, \dots \\
& \dots \\
& \omega^2, \omega^2 + 1, \omega^2 + 2, \dots, \omega^2 + n, \dots \\
& \dots \\
& \omega^3, \omega^3 + 1, \omega^3 + 2, \dots, \omega^3 + n, \dots \\
& \dots \\
& \omega^\omega, \omega^\omega + 1, \omega^\omega + 2, \dots, \omega^\omega + n, \dots \\
& \dots
\end{aligned} \tag{6.2}$$

除了第一行是自然数外，其它都是无穷序数，并且每行的第一个是极限序数。用这种方式得到的序数，已经远远超出人们所能想象的范围，把自然数扩展成一个无穷无尽的序数王国。但是，可以证明，这些序数都是可数序数，作为集合，竟然能够与自然数集构成一一对应。我们即将看到，还存在着不可数序数，甚至存在着一个比一个更大的无穷序数列。

我们列出的这些序数中，如果挑选一个作为可数集合的基数，用哪一个最好呢？自然会想到最小的一个极限序数 $\omega$ 。于是这引出了基数的一般定义：

**定义 6.3.1.** 设 $a$ 是一个序数，如果对于任一序数 $b$ ，当 $b < a$ 时，有 $b$ 的势小于 $a$ 的势，则称序数 $a$ 为基数。

有这个定义，我们立即得出结论：所有自然数 $n$ 都是基数，并且 $\omega$ 是基数。序数 $\omega$ 当作基数使用时，记作 $\aleph_0$ 。即 $\aleph_0 = \omega$ ，前面我们已经用 $\aleph_0$ 表示可数集合的基数。

除了 $\omega$ 外，序列(6.2)中其余的无穷序数都比 $\omega$ 大，但其势却与 $\omega$ 的势相等（都等于可数无穷）。所以根据定义，它们都不是基数。

为了获取更大的基数，为此我们将此前序列(6.2)中的所有序数汇集在一起组成一个集合，记作 $\omega_1$ 。

$$\omega_1 = \{a | a \text{ 是序数，且 } |a| \leq \aleph_0\}$$

其中 $|a|$ 表示 $a$ 的势<sup>6</sup>。可以证明 $\omega_1$ 是序数，并且是第一个不可数序数。然后我们仿照前面的方法，从 $\omega_1$ 之后扩展无穷序数列：

$$\omega_1, \omega_1 + 1, \dots, \omega_1 \cdot 2, \dots, \omega_1^2, \dots, \omega_1^\omega, \dots$$

这里枚举的无穷序列都是等势的，其中最小的一个是 $\omega_1$ ，它还满足基数的条件，这样我们就得到了第二个无穷基数 $\aleph_1 = \omega_1$ 。仿照 $\omega_1$ 的构造过程，我们可以再构造一个集合：

---

<sup>6</sup>严格来说， $A$ 的势应使用符号 $\overline{\#} A$ ，或 $\# A, card(A), n(A)$ 。

$$\omega_2 = \{a | a \text{是序数, 且} |a| \leq \aleph_1\}$$

这样就获得了第三个无穷基数 $\aleph_2 = \omega_2$ 。继续进行下去, 我们可以得到一系列无穷基数。概括来说, 对任一序数 $a$ , 当定义了无穷基数 $\aleph_a$ 之后, 我们可以再次构造集合:

$$\omega_{a+1} = \{b | b \text{是序数, 且} |b| \leq \aleph_a\}$$

由此得到比 $\aleph_a$ 大的无穷基数 $\aleph_{a+1} = \omega_{a+1}$ 。总之对于任一序数 $a$ , 相应都有一个无穷基数 $\aleph_a$ , 由此得到无穷基数组成的无穷序列:

$$\aleph_0, \aleph_1, \aleph_2, \dots, \aleph_n, \dots, \aleph_\omega, \dots \quad (6.3)$$

它们是从小到大排列的, 并且相邻的两个阿列夫之间不再有其它的无穷基数。无穷序数和无穷基数也称为超限序数和超限基数, 统称为超限数。这些越来越巨大的超限数最终归于何处呢? 康托尔认为那将是上帝。

超限数一经面试, 立即引发了激烈的反应。有人赞叹这是康托尔惊人的创举, 开辟了前所未见的新视野。也有人认为超限数是“雾上之雾”, 康托尔正在创造病态的数学。尽管存在巨大的争议, 超限数是十九世纪最惊人的思想成就之一。

### 6.3.5.2 连续统假设

康托尔发现了两种无穷基数序列, 一个是幂集, 另一个是超限基数:

$$\aleph_0, 2^{\aleph_0}, 2^{2^{\aleph_0}}, \dots$$

和

$$\aleph_0, \aleph_1, \aleph_2, \dots$$

根据上面的分析, 我们知道 $\aleph_1$ 是紧跟着可数无穷基数 $\aleph_0$ 之后的下一个超限基数。然而根据幂集的性质, 我们只知道 $2^{\aleph_0}$ 比可数无穷基数 $\aleph_0$ 大。但我们不知道它和 $\aleph_1$ 的大小关系。康托尔猜测 $2^{\aleph_0} = \aleph_1$ , 也就是在 $\aleph_0$ 和 $2^{\aleph_0}$ 之间不存在其它无限基数。 $2^{\aleph_0}$ 是第一个比可数集大的超限基数。

康托尔在1847年证明了 $2^{\aleph_0} = C$ , 也就是说, 自然数的所有子集所具有的元素数正好等于实数集的元素数。因此康托尔的猜测等价于说在可数集 $\aleph_0$ 与不可数集 $C$ 之间不存在其它无限基数。由于通常称实数集为连续统, 因此这一猜想被称为连续统假设, 英文为: Continuum Hypothesis, 简记为CH。

连续统假设还可以进一步推广。即考虑对于任一序数 $a$ ,  $2^{\aleph_a} = \aleph_{a+1}$ 是否成立。这一假设被称为广义连续统假设, 简记为GCH。

康托尔在1878年的一篇论文提出了连续统假设。他一开始对证明这一猜想是比较乐观的。据说康托尔曾经说他已经成功解决了这一难题, 并即将公布他的证明。但直到他1918年去世, 也没有把证明公之于众。大概是发现了证明中的问题而未公开发表。康托尔晚年为此投入了大量的精力, 但是长时间未能突破连续统假设, 加之其它原因最终导致他陷入了抑郁, 并在哈雷大学的精神病院中逝世。

1900年夏天, 著名的数学家希尔伯特在巴黎召开的第二届国际数学家大会上, 作了题为《数学问题》的演说。提出了23个未解决的问题, 向二十世纪的数学家提出挑战。其中第一个问题就是“证明连续统假设”。可见他对这一问题的重视。

连续统问题是数学来源于几何、力学、和物理等方面现实问题的一个范例。希尔伯特认为, 连续统问题来自外部世界, 纯数学需要从外部世界汲取新材料, 外部世界是数学的源泉。正因为连续统问题是数学中一个最基本的问题, 或者说它是数学基础的问题, 长期以来一直是数理逻辑和公理集合论的一个中心问题。一百多年来, 虽然经过许多著名数学家的精心钻研。取得了一些重大进展, 但还没有完全解决。1938年哥德尔证明了, 从公理集合论的ZFC系统(策梅罗——弗

兰克尔系统加上选择公理的简称，选择公理是说我们能从任一集合，包括无穷集合中选出若干元素。我们将在下一章详细介绍）推不出CH的否定。即连续统假设与ZFC系统是相容的。在哥德尔的结果之后，人们希望能够从ZF系统（ZF系统是不带有选择公理的集合论系统）内证明连续统假设。

1963年7月，美国的年轻数学家科恩创造了威力极大的力破法，解决了相反的问题，他证明了从ZFC推不出CH。这就说明了连续统假设和ZFC系统是相对独立的。有一则插曲说，科恩完成了证明后，并不能确信自己的证明（[9]，第280页）。他来到普林斯顿敲响了哥德尔的家门。当时的哥德尔正在同妄想症斗争，他仅仅打来了一条门缝，让科恩把证明塞进去。科恩则被关在了门外。两天后，哥德尔邀请科恩进屋喝茶，大师终于认可了他的证明。

综合哥德尔和科恩的结果，也就是说连续统假设在ZFC系统中是不可判定的。我们在下一章会深入介绍不可判定性。连续统假设与ZF系统的公理无关。类似的结论还发生在集合论中的选择公理上。哥德尔和科恩的结论同时也说明，选择公理在ZF系统中是不可判定的。这说明在ZF公理集合论系统中，承认选择公理可以得到一种数学；否定选择公理可以得到另一种数学。两者都是无矛盾的。同样，加上选择公理后，承认或者否定连续统假设也都可以各自发展出无矛盾的数学。这就是100多年来人们在选择公理与连续统假设的研究中获得的主要成果[58]。

## 6.4 无穷与艺术



[荷]梵高《星空》1889，原作收藏于纽约现代艺术博物馆

伴随着对无穷的思考与探索，也不断催生了关于无穷的艺术创作。人们仰视浩瀚苍穹，远眺无垠的大海，感叹自然的神秘和伟大。

在无数描绘广袤天空的作品中，荷兰后印象派艺术大师梵·高创作的《星空》可谓让人印象深刻。在这幅画中，梵高用夸张的手法，生动地描绘了充满运动和变化的星空。整个画面被一股汹涌、动荡的蓝绿色激流所吞噬，旋转、躁动、卷曲的星云使夜空变得异常活跃，脱离现实的景象反映出梵·高躁动不安的情感和疯狂的幻觉世界。这幅画创作于1889年，当时梵·高正在阿尔勒圣雷米的一家精神病院治疗，在那驻留了108天。在入住精神病院期间，梵高创作了大量的绘画作品，共计一百五十多幅油画和一百多幅素描。而作品《星空》所描述的风景也正是精神病院所在地圣雷米。

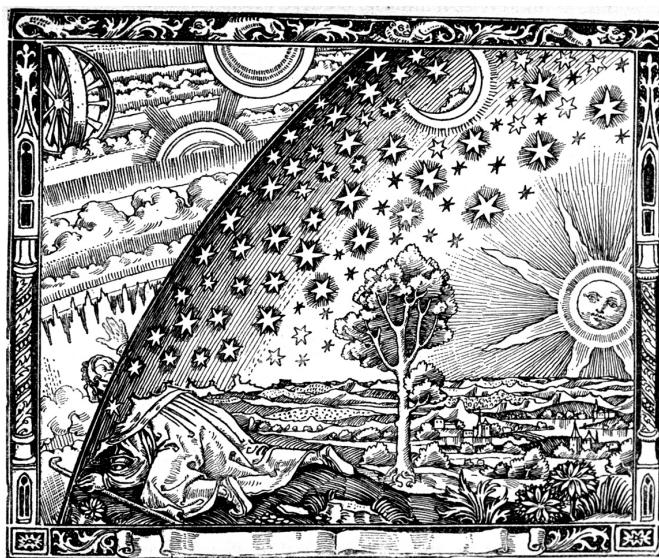
英国画家透纳，在创作《暴风雪：汽船驶离港口》时，为了充分体验大海无边的威力，他让水手把自己捆在船桅上。他后来写道：“为了观察大海，我让水手们把我捆在船桅上。我那样过了四个小时，没指望能活下来。”然而，评论家们却对这幅画表示失望和怀疑，因为在这幅画中，形体和戏剧性的场面都消失了。透纳解释说，他画这幅画是为了告诉自己和别人，在惊涛骇浪的海上，暴风雪是什么样子。虽然描画的是在漩涡风暴中航行的船，但是所呈现的却是船只与风暴融



透纳笔下的大海和风暴，1842。原作藏于英国泰特美术馆

为一体的画面。透纳大胆地以抽象手法表现船的形式，自由运用色彩，成功的表现出“风暴的气氛”。透纳因此被称为印象派的先驱。

对无穷的思考很快脱离了自然界中的具体事物，而上升到哲学和宗教。在托勒密的宇宙模型中，行星是嵌套在一起的同心球，最外层存在一个有界的恒星天球。到了中世纪，基督教在很大程度上吸收了亚里士多德和托勒密的学说，认为上帝创造的地球是宇宙的中心，而恒星天球是有界的。



1888年在巴黎出版的《弗拉马利翁》木刻版画，第163页，作者为无名氏

1888年在巴黎出版的一幅木刻版画反映了当时人们对于有穷世界边沿的思考。一个人如果站在天球的边沿，是否可以伸出自己的手臂或者举起一根手杖？如果不可以，这显然是难以理解的，如果可以，那么处于物质世界外围的空间是什么？这就是宇宙边缘悖论。为了解决这个难题，中世纪的基督教重塑了亚里士多德的学说，提出了一种渐进边缘理论。还有人为，如果在宇宙的边缘向外抛出一支矛，就会推动宇宙扩大。物质世界是有界的，但界被无尽的虚空包围。

文艺复兴时期，数学逐渐被当时艺术大师们引入到作品中。达芬奇不仅深谙解剖和透视，还有意识地在作品中使用引发美感的比例。这一时期的德国画家丢勒仔细研究了人体的各种比例，甚至利用坐标格点来描述透视的关系。丢勒的《量度四书》既介绍了绘画理论，也对几何原理和透视原理进行了研究。随后开普勒和笛沙格独立发展出了摄影几何中的无穷远点概念。笛沙格概括消失点的用途，纳入无穷远时的情形，发展出建构透视图的另一种方法。他让平行线确实平行的欧氏几何成为所有可能的几何系统都会有的特例。

真正从本质上改变了艺术家的视角，使得人们能够直接表现无穷要从非欧几何的诞生说起。长期以来，欧几里得几何被人们认为是完美的公理系统和演绎推理的典范。但是追求尽善尽美的数学家对于欧几里得第五公设颇有微词。前几条公设简单直观，符合直觉，例如说两点之间能够画一条直线，所有直角都相等。而第五公设描述却比较复杂。它说如果某条直线与两直线相交，且同侧两个内角和小于两个直角，那么两条直线无限延长后就会在该侧相交。第五公设也叫作平行公设，它等价于说，在平面内过直线外一点有且仅有一条平行线。人们感觉第五公设能从前四条公设里推导出，并且实际上欧几里得在《几何原本》前面相当大的部分也都没有使用第五公设。在其后的两千多年里，很多人试图证明第五公设，但都失败了。于是意大利数学家撒凯里（Saccheri）尝试用反证法来证明，他假定第五公设不成立，然后导出一整套几何系统和奇怪的结论，接下来他宣称这些结果太过荒谬，从而说明第五公设是必定是正确的。

19世纪，德国数学家高斯、俄国数学家罗巴切夫斯基、匈牙利数学家波尔约等人各自独立地认识到这种证明是不可能的。也就是说，平行公理是独立于其他公理的，并且可以用不同的“平行公理”来替代它。高斯关于非欧几何的信件和笔记在他生前一直没有公开发表，只是在他1855年去世后出版时才引起人们的注意。罗巴切夫斯基和波尔约分别在1830年前后发表了他们关于非欧几何的理论。在这种几何里，罗巴切夫斯基平行公理替代了欧几里得平行公理，即在一个平面上，过已知直线外一点至少有两条直线与该直线不相交。由此可演绎出一系列全无矛盾的结论，并且可以得出三角形的内角和小于两直角。罗氏几何中有许多不同于欧氏几何的定理。

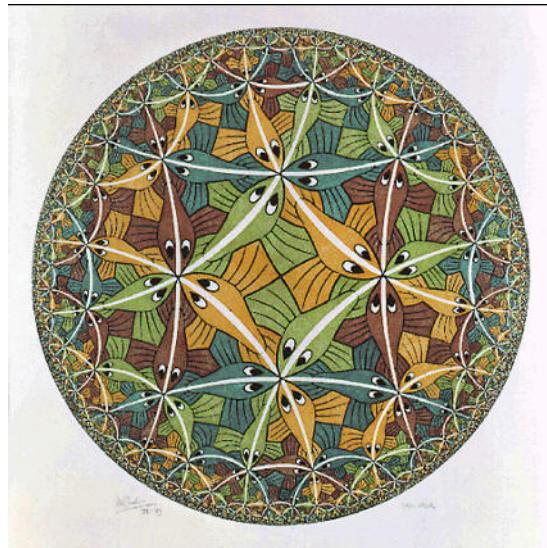
继罗氏几何后，德国数学家黎曼在1854年又提出了既不是欧氏几何也不是罗氏几何的新的非欧几何。这种几何采用如下公理替代欧几里得平行公理：同一平面上的任何两直线一定相交。同时，还对欧氏几何的其他公理做了部分改动。在这种几何里，三角形的内角和大于两直角。人们把这种几何称为椭圆几何。

直到1866年，意大利数学家贝尔特拉米在他出版的《非欧几何解释的尝试》中，证明了非欧平面几何可以局部地在欧氏空间中实现。1871年，德国数学家克莱因认识到从射影几何中可以推导度量几何，并建立了非欧几何模型。这样，非欧几何的相容性问题就归结为欧氏几何的相容性问题，由此非欧几何得到了普遍的承认。

非欧几何向我们揭示这样一种可能性，即无穷的空间可能是有界的。法国数学家庞加莱在他的科普读物《科学与假设》中介绍了这样一种有趣的世界。整个世界被一个大小有限的球包围起来。中心的温度很高，随着远离中心，温度成比例地减小。当接近包围这个世界的球面时，温度降到绝对零度。如果大球的半径是 $R$ ，某点到球心的距离是 $r$ ，则温度与 $R^2 - r^2$ 成比例。这个世界中，由于热胀冷缩，物体的大小和温度成比例，越接近世界的边沿，物体越小。于是就出现这样的一个奇观，当这个世界的居民接近球面时，温度越来越低，他们越来越小，步伐也越来越小，他们永远也到不了世界的边沿，尽管这个世界是有限的。庞加莱描述的这个世界，实际上起作用的几何是一种称为双曲几何的非欧几里得几何学。

荷兰画家埃舍尔受到庞加莱的启发，创作了多个艺术作品来描述这种有限但无穷的世界。这一系列作品被命名为圆极限系列。不管是天使、魔鬼、还是游动的鱼，都在接近圆盘的边缘时变小，从而永远无法到达这个有界但无穷的边沿。

不仅是艺术，在音乐中也有对无穷的思索和表现。1747年5月，巴赫访问了波茨坦宫廷圣苏西宫。巴赫到了圣苏西宫后，腓特烈大帝向他展示了刚刚引进的吉尔博曼钢琴。宫廷音乐会上腓特烈大帝给了巴赫一个音乐主题，老巴赫当场即兴对其进行了一个三声部的赋格变奏。这是事前毫无演练的即兴演奏。这不仅使大帝十分满意，在场众人也无不瞠目结舌。巴赫本人觉得这首曲子的主题非常美丽，于是打算将来写成一首赋格曲，以供出版。回到莱比锡后，巴赫重新对国王主题进行变奏创作，将整个曲子按两首赋格曲，四乐章三重奏鸣曲和十首卡农的构成完成了整个曲子。巴赫在献词上所署名的时间正好是7月7日。这就是巴赫的经典名作《音乐的奉献》，乐曲



埃舍尔《圆极限·3》1959

编号BWV1079。在其中有一首极不寻常的卡农，只标着“Canon per Tonos”这三个词。翻译过来的意思是经由种种调性的卡农。后人称之为“无穷升高的卡农”。侯世达在《哥德尔、埃舍尔、巴赫——集异璧之大成》一书中写到：



巴赫创作的无穷升高的卡农的曲谱局部

它有三个声部，最高声部是国王主题的一个变奏，下面两个声部则提供了一个建立在第二主题之上的卡农化的和声。这两个声部中较低的那个声部用C小调奏出主题，而较高的那个则在差五度之上奏出同一主题。特殊之处在于，当它结束时——或者不如说似乎要结束时——已不再是C小调而是D小调了。巴赫在听众的鼻子底下转了调。而且这一结构使得这一“结尾”很通顺地与开头联接起来。这样可以重复这一过程并在E调上回到开头。这些连续的变调带着听众不断上升到越来越远的调区，因此听了几段之后，听众会以为他要无休止地远离开始的调子了。然而在整整六次这样的变调之后，原来的C小调又魔术般地恢复了！所有的声音都恰好比原来高八度。在这里整部曲子可以以符合音乐规则的方式终止。人们猜想，这里就是巴赫的意图。但巴赫很明确地留下了一个暗示，说这一过程可以无休止地进行下去。也许这就是为什么他在边上写下了“转调升高，国王的荣耀也升高。”<sup>[5]</sup>

### 练习 6.5

- 在两个镜子中间点燃一支蜡烛，你看到了什么？这是潜无穷还是实无穷？

## 6.5 附录：例子代码

使用流定义自然数潜无穷，并取出前15个自然数。Java语言1.8中的例子：

```
IntStream.iterate(1, i -> i + 1);

IntStream.iterate(1, i -> i + 1)
    .limit(15).forEach(System.out::println);
```

Python语言版本3中的例子：

```
def naturals():
    yield 0
    for n in naturals():
        yield n + 1
```

Haskell语言中使用递归定义自然数潜无穷：

```
nat = 1 : (map (+1) nat)

take 15 nat
```

Haskell语言中使用递归定义斐波那契数列潜无穷，以及计算第1500个斐波那契数。

```
fib = 0 : 1 : zipWith (+) fib (tail fib)

take 15 fib
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]

fib !! 1500
13551125668563101951636936867148408377786010712418497242133543153221487310
87352875061225935403571726530037377881434732025769925708235655004534991410
29242495959974839822286992875272419318113250950996424476212422002092544399
20196960465321438498305345893378932585393381539093549479296194800838145996
187122583354898000
```

Haskell语言中使用余代数定义素数的潜无穷流

```
data StreamF e a = StreamF e a
data Stream e = Stream e (Stream e)

takeStream 0 _ = []
takeStream n (Stream e s) = e : takeStream (n - 1) s

era (p:ns) = StreamF p (filter (p `notdiv`) ns)
  where notdiv p n = n `mod` p /= 0

primes = ana era [2..]

takeStream 15 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

## 6.6 附录：康托尔定理的证明

**定理 6.6.1. 康托尔定理：**对于任意集合都有 $|S| < |2^S|$ ，其中 $|S|$ 表示集合 $S$ 的势， $2^S$ 表示 $S$ 的幂集，即 $S$ 的所有子集组成的集合。

证明. 我们分两步证明。首先证明 $|S| \leq |2^S|$ 。对于任一 $x$ ，令 $f(x) = \{x\}$ ，也就是仅含有 $x$ 唯一元素的集合。对于不同的元素 $x_1 \neq x_2$ ，自然有 $\{x_1\} \neq \{x_2\}$ ，即 $f(x_1) \neq f(x_2)$ 。从而映射 $S \xrightarrow{f} 2^S$ 是一单射。因此有

$$|S| \leq |2^S|$$

第二步，我们证明 $|S| \neq |2^S|$ 。采用反证法，假设等号成立。则存在一一映射 $S \xrightarrow{\phi} 2^S$ ，使得对任一 $x \in S$ ，都有 $\phi(x) \in 2^S$ 。即 $\phi(x)$ 是 $S$ 的某个子集，所以 $\phi(x) \subseteq S$ 。现在要问： $x$ 是否属于 $\phi(x)$ ？有两种可能：一种是 $x \in \phi(x)$ ，也可能是 $x \notin \phi(x)$ 。我们把所有 $x$ 不属于 $\phi(x)$ 的元素放在一起构造一个新集合 $S_0$ ：

$$S_0 = \{x|x \in S, \text{ 并且 } x \notin \phi(x)\} \quad (6.4)$$

显然 $S_0$ 是 $S$ 的子集，即 $S_0 \subseteq S$ ，因此 $S_0 \in 2^S$ 。由于 $\phi$ 是一一映射，所以必然存在一个 $x_0$ ，使得 $\phi(x_0) = S_0$ 。根据逻辑中的排中律，要么 $x_0 \in S_0$ ，要么 $x_0 \notin S_0$ ，二者必居其一，且仅有一个成立。

接下来分情况讨论。如果 $x_0 \in S_0$ 成立，根据式(6.4)中 $S_0$ 的定义，应该有 $x_0 \notin \phi(x_0)$ ，由于 $\phi(x_0) = S_0$ ，所以 $x_0 \notin S_0$ 。

如果 $x_0 \notin S_0$ ，因为 $S_0 = \phi(x_0)$ ，所以得到 $x_0 \notin \phi(x_0)$ 。这样根据 $S_0$ 的定义(6.4)，又应该有 $x_0 \in S_0$ 。

这样不论 $x_0$ 是否属于 $S_0$ ，都导致矛盾。这说明我们最初的假设 $S$ 到 $2^S$ 间存在一一映射是错误的。所以不等式 $|S| \neq |2^S|$ 成立。

由这两步的结果： $|S| \leq |2^S|$ ，并且 $|S| \neq |2^S|$ ，我们得到了康托尔定理的结论：

$$|S| < |2^S|$$

□

证明的第二部分，不由让我们联想起了著名的罗素悖论：令 $S$ 为所有不属于自己集合构成的集合，问 $S$ 是否属于自己？我们将在下一章讲述罗素悖论和哥德尔不完全性定理。

## 6.7 附录：巴赫《音乐的奉献》无限上升的卡农

## Canon a 2. (Per tonos.)

Musikalisch Opfer BWV 1079

Johann Sebastian Bach

A musical score for Johann Sebastian Bach's Canon a 2. (Per tonos.) from the Musicalische Opfer (BWV 1079). The score consists of four staves of music for three voices. The voices are represented by soprano, alto, and two bass parts. The music is written in common time, with a key signature of one sharp (F#). The notation includes various note heads, stems, and bar lines. The score is divided into measures, with measure numbers 1 through 15 visible.

A continuation of the musical score for Canon a 2. (Per tonos.) from measure 16 to 35. The score remains in common time with a key signature of one sharp (F#). The three voices (soprano, alto, and two bass parts) continue their melodic lines with complex sixteenth-note patterns. Measure numbers 16 through 35 are indicated above the staff.

1

2

A continuation of the musical score for Canon a 2. (Per tonos.) from measure 36 to 55. The score maintains its common time and F# key signature. The three voices continue their intricate sixteenth-note canon. Measure numbers 36 through 55 are indicated above the staff.

3

## 第7章 悖论

除了自己的无知，我什么都不懂。

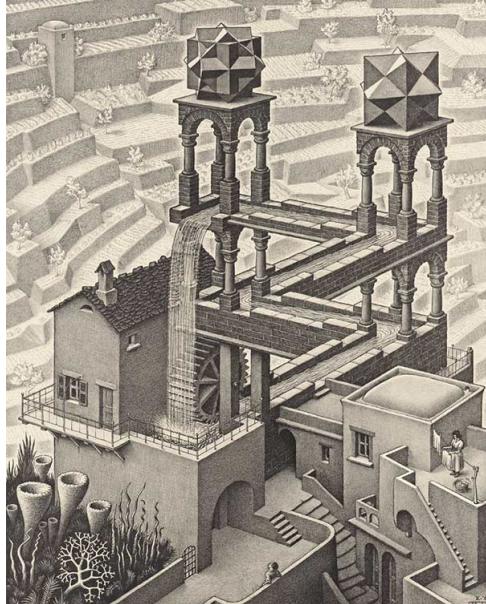
——苏格拉底

1996年，第26届国际奥林匹克运动会正在美国的亚特兰大举行。来自世界各地的选手在速度、力量、技巧上展开竞赛挑战人类的极限。与此同时，还进行着另一场有趣的竞赛。超级计算机“深蓝”与国际象棋世界冠军卡斯帕罗夫展开了对抗赛。比赛结果是深蓝2胜4负输给了人类象棋冠军。翌年，改进的深蓝再次向卡斯帕罗夫发起挑战。5月11日，计算机在正常时限的比赛中首次击败了卡斯帕罗夫。总比分2胜1负3平。深蓝计算机重1270公斤，有32个微处理器，每秒钟可以计算2亿步。为了能让“深蓝”挑战人类冠军，设计小组输入了一百多年来优秀棋手的两百多万个对局。人类用智慧创造的机器，在人类骄傲的智慧领域首次击败了人类自己——这一结局引发了关注、恐惧、和激烈的讨论。

在当时，人们普遍认为，这是人工智能的一大进步。尽管在国际象棋上取得了巨大进步，但是在围棋上，计算机和人类仍存在巨大差距。对于国际象棋来说，棋盘8行8列，32枚棋子。计算机要在 $10^{123}$ 这样巨大的博弈树中进行搜索。即使深蓝美秒能算2亿步，遍历博弈树仍需要近 $10^{107}$ 年。为此深蓝的设计小组通过计算机程序缩小了搜索空间，使得深蓝能够搜索当前棋局后面的12步棋。而一般好的人类棋手，大约只能估计到10步左右。但是围棋的棋盘有19行19列，在一共361个格点上可以放置黑色或者白色的棋子。博弈树的规模为 $10^{360}$ ，远远超越国际象棋。所以在之后的一段时间里，人们仍然不相信计算机可以挑战我们。

时光匆匆过去了20年，2016年，计算机程序Alpha-Go向人类的围棋大师展开了挑战。韩国的九段棋手李世石以1比4的总比分输掉了比赛。一年后，Alpha-Go再次以3局全胜的成绩战胜了中国棋手柯洁。被人们认为是人工智能游戏“圣杯”的围棋终于被攻破了。面对没有感情的计算机，柯洁心有不甘，潸然落泪。作为人类，我们的心情很复杂。即使是从事智力工作的程序员群体也感到了来自机器的压力——我们是否会被机器取代？

传统上我们认为，艺术文学等领域，涉及人们的文化背景、内在感情和与生俱来的性格因素，是无法被机器所替代的。2015年，德国斯图加特以南40公里的小镇图宾根大学的盖提斯、埃克、贝特格三位研究人员利用机器学习人类艺术家的风格，把图宾根镇的一张风景照片变换成了不同



埃舍尔《瀑布》1961



卡斯帕罗夫在与深蓝对弈，图片原载《科学美国人杂志》

风格的艺术画作[60]。无论是后印象派大师梵高色彩强烈夸张的画风，还是透纳那浪漫主义水天浑浊的光影效果，都被机器模仿得惟妙惟肖。犹如大师本人所作（图7.3）。

在随后的数年中，人工智能和机器学习突飞猛进地进入了各种领域。机器产生不同音乐家风格的音乐，能够演奏出紧张、舒缓等不同的情绪的旋律和节奏，而不再是呆板单调的电子琴音。机器批量翻译新闻稿和各种学术论文，和专业翻译的文笔不相上下。机器处理X片、CT、核磁共振等医学图像并给出病理诊断，并且结果在准确程度上超越人类医生，人工智能操控的无人机在街道上行驶，成功超过其它车辆并避让行人。无人值守的商店突然出现在街边，人们可以直接从货架上拿走商品，并在走出商店的一刻自动支付……作为人类的我们不禁会问：我们消灭工作岗位的速度是否会超过创造工作机会的速度？人类是否会被机器全面取代？机器是否最终会统治我们？

所有这些在本质上都可以归结到一个问题：计算的能力是否存在边界？如果有的话，计算的边界在哪里？

## 7.1 计算的边界

顾森在《思考的乐趣》一书中讲到注视着一个运行了很久的程序时的两难心情：这个程序能结束么？是应该继续等下去，还是杀掉进程强行结束？有没有什么编译器能事先告诉你的程序是否会无限运行下去？([61]，第228页)

为什么不可能呢？这个东西看上去比时光旅行机更现实一些。或许我们会在某个科幻电影中看到，一个程序员在漆黑的屏幕上输入几个数，敲了一下回车，然后屏幕上立即用高亮加粗字体显示：“警告：该输入数据会导致程序无限运行下去，确定执行？（Y/N）”如果有一天，这一切真的成为了现实。那么你能利用这个玩意儿来做什么实用、有价值的事情？如果说你能靠这玩意儿发大财的话，你相信么？……我上来就先写一个哥德巴赫猜想的验证程序。我写一个程序，让他从小到大枚举所有的偶数，看是不是有两个质数加起来等于它。如果找到了，继续枚举下一个偶数，否则输出反例并结束程序。然后编译该程序。这个编译器不是可以预先判断我这个程序能否终止吗？如果编译器说我这个程序会无限执行下去的话，我岂不是相当于证实了哥德巴赫猜想吗？或者，编译器说程序会最终终止，那哥德巴赫猜想不就直接被推翻了吗？不管怎样，我都将成为解决哥德巴赫猜想的第一人，在数学史上留下自己的名字。接下来呢？把刚才的程序代码改成孪生素数搜索器，在利用编译器检查一下，看看是不是真的有无穷多个孪生素数。梅森素数是否有无穷多个，这个也是数论中



图 7.3: 机器学习产生的不同艺术风格的画作: A, 图宾根镇的风景照片; B, 英国画家透纳1810年的原作《运输船遇难》和透纳风格的画作; C, 荷兰后印象派画家梵高1889年的原作《星空》和梵高风格的画作; D, 挪威表现主义画家爱德华·蒙克1893年的原作《呐喊》和蒙克风格的画作; E, 西班牙现代艺术家毕加索1910年的原作《坐着的裸女》和毕加索风格的画作; F, 俄罗斯抽象艺术先驱画家康定斯基1913年的原作《构成第七号》和康定斯基风格的画作。

长期以来悬而未决的难题。不过现在看来，我也能不费吹灰之力就把它解决了。还记得 $3x + 1$ 问题吗？写一个“证明程序”也只是几分钟的事情，而且还能拿走埃尔德什提供的500美元奖金呢。数学上的未解之谜多着呢。我永远不愁没事做。1984年，马丁·拉巴尔询问能否用9个不同的平方数构成 $3 \times 3$ 幻方，这个问题的奖金目前已经积累到了100美元加100欧元再加一瓶香槟。网上搜索“数学未解难题”，看看哪些问题是离散的，其中又有哪些问题是悬赏的，写几个程序就可以把它们统统解决……

1936年，计算机科学和人工智能的先驱图灵证明了一个命题：不存在可以判断任何程序是否可以停机的通用算法。证明的核心部分包含了计算机程序的数学定义——图灵机模型。后人称这一问题为图灵停机问题。

为了证明图灵停机问题，我们采用反证法，假设存在一个名叫 $halts(p)$ 的算法，能够判断任意程序 $p$ 是否停机。首先我们定义一个永不停机的程序：

$$\text{forever}() = \text{forever}()$$

这是一个无穷递归的调用。然后我们构造一个名为 $G$ 的特殊程序<sup>1</sup>，它的定义如下：

$$G() = \begin{cases} halts(G) = \text{停机} : & \text{forever}() \\ \text{否则} : & \text{停机} \end{cases}$$

在程序 $G$ 中，我们通过 $halts(G)$ 判断 $G$ 本身是否停机。如果停机，我们就调用 $\text{forever}()$ 永远运行下去。但这恰恰说明 $G$ 不会停机，所以 $halts(G)$ 应该为假，但是按照上面定义的第二行，此时我们停机。这恰恰说明 $halts(G)$ 应该为真。所以不论 $halts(G)$ 是真是假，我们都会得到矛盾的结论。因此我们最初的假设不成立，也就是说，不存在一个可以判断任意程序能否停机的通用算法。

也有一种分两步证明图灵停机问题的方法([62]，第268页)，前面都一样，但在构造 $G$ 时， $G$ 接受一个参数 $p$ ，它把 $p$ 应用到自身上并传给 $halts$ ：

```
G(p) = if halts(p(p)) then forever() else 'Halted'
```

接下来的一步中，我们把 $G$ 传给自己 $G(G)$ 看发生了什么？此时如果 $halts(G(G))$ 返回真，则接下来运行 $\text{forever}()$ ，所以 $G(G)$ 永远运行不会停机。但这恰恰说明 $halts(G(G))$ 应该返回假，所以接下来程序进入`else`分支，返回停机。但这又说明 $halts(G(G))$ 应该返回真。所以不管停机与否，都陷入了矛盾之中。

伟大的图灵停机定理清晰地给出了一个不可计算问题。击碎了我们本节中给出的那些奇思妙想。看到这里，你是否想起了上一章附录中康托尔定理的证明？我们用极为类似的方法证明了任何集合，包括无穷集合的势都小于它的幂集的势。实际上，图灵停机问题让我们联想起了一大类有趣的逻辑悖论。

## 7.2 罗素悖论

悖论从古希腊时期就被人们发现了。上一章我们介绍了关于无穷和连续的芝诺悖论，而逻辑悖论是一类从严密逻辑导出矛盾结果的有趣问题。公元前四世纪，古希腊哲学家米利都的欧步里德提出这样一个命题：“我现在说的是一句假话”，怎样判断这句话的真伪呢？

如果这句话是假话，那么它陈述的事实（正在说谎）就成了真的，因此矛盾。但如果这句话是真话，那么这句话的原话说正在说谎，因此它是假话，也产生了矛盾。不论欧步里德说的是真是假，我们都将陷入矛盾中，这一著名的令人困惑的问题被人们称为“说谎者悖论”。

说谎者悖论还有一个两段体的变形，以对话的形式出现。例如：

**阿基里斯：**乌龟是个狡猾的家伙，总爱说谎，你听，它下面的话就是假的。

**乌龟：**亲爱的阿基里斯，诚实的你总是说真话。

<sup>1</sup> 我们用字母 $G$ 是有特殊用意的， $G$ 是哥德尔的首字母，它恰好和哥德尔不完全定理中不可判定命题的名字相同。

乌龟的话到底是真是假呢？如果乌龟说了真话，也就是阿基里斯的陈述是真的。但阿基里斯说乌龟在撒谎，这就导致了矛盾。反之如果乌龟说的是假话，那么阿基里斯说的就是假的，于是乌龟说的这句就该为真。我们陷入了怪圈，无论乌龟的话是真是假，都会导致矛盾。

这种两段体式的说谎者悖论有时还以恶作剧的形式出现。你收到一张纸条，上面写着“背面是假的”，等你翻到纸条背面，却看到上面赫然写着“背面是真的”。到底哪面是真的呢？仔细分析下来，就会发现陷入了逻辑怪圈。

儿童故事中，也有不少这种悖论。有一则说狮子捉到了兔子，得意地说，如果你能猜中接下来我要干什么，我就放了你，要是猜错了，我就吃掉你。聪明的兔子说：“我猜你要吃掉我。”

如果狮子吃掉兔子，那说明兔子猜中了。这样狮子应该兑现承诺，放掉兔子。可是如果放掉兔子，这说明兔子猜错了。按道理狮子又应该吃掉兔子。狮子陷入了两难处境。既不能吃掉兔子，也不能不吃兔子。估计它只能发疯而让聪明的兔子溜走了。

传说古希腊的军队战胜了波斯，国王决心“优待俘虏”，让他们选择死亡的方式。俘虏可以说一句话，如果是真话，就被砍头，如果是假话，就被绞死。一个聪明的俘虏说：“我猜你要绞死我。”如果国王绞死了俘虏，说明他说了真话，可是这样，按照规则应该被砍头。但如果砍掉他的头，就和这个人讲的内容不符了，所以他做了假话。这样就应该被绞死。结果不论砍头还是绞死，国王的命令都没有被正确的执行。国王万般无奈，不仅释放了这个俘虏，还释放了所有其他人。



[德]埃·奥·卜劳恩《父与子》一则，1930年代

理发师这样就会陷入困境。

罗素最早在1901年发现了集合论的悖论。他归纳总结了一系列悖论，并最终将它们形式化为当时集合论本质上的问题。人们现在一般将这类悖论称为罗素悖论。在康托尔的朴素集合论中，罗素考虑了任何集合是否属于它自身的问题。有些集合属于它本身，有些集合则不属于。例如所有茶匙的集合显然不是另一个茶匙，但所有不是茶匙的东西构成的集合显然也不是一个茶匙。罗素考虑了后者这类情况全体构成的集合。他构造了集合 $R$ ，由所有不是自身元素的集合所组成。用形式化的定义表示就是：

$$R = \{x | x \notin x\}$$

罗素接着思考， $R$ 是否属于 $R$ 呢？根据逻辑中的排中律，一个元素或者属于一个集合，或者不属于一个集合。因此对于一个给定的集合，问它是否属于自己是有意义的。但是这个定义良好

塞万提斯在他的伟大作品《堂·吉诃德》中，也讲了一个有趣的悖论。有一位贵族的封地被一条大河分成了两半，河上有一座桥，桥的尽头有个绞架。这位贵族制定了一条法令：“过桥的人必须诚实声明他的目的，如果是真话就允许过桥，如果说谎，就判处绞刑，绞死在桥那边的绞架上。”结果有个人来这里发誓道，我过桥别无目的，就是想死在那个绞架上。怎样处置这个人呢？如果他说了真话，那么就应该放他过桥。可是这样这个人说的内容就不成立了，按照法令就应该绞死他。可是这样一来他说的话就成了真话，又应该放他过桥。

和说谎者悖论同样著名的是理发师悖论。这是1919年由著名数学家、逻辑学家罗素提出的。故事说村子里的理发师宣布：“他只给那些不给自己刮胡子的人刮胡子。”那么这位理发师是否给他自己刮胡子？如果他给自己刮胡子，那么按照他的规定，他就应该给自己刮胡子。而如果他不给自己刮胡子，那么他就应该向自己提供服务，也就是给自己刮胡子。理

的，看似合理的问题却陷入了两难境地。

如果 $R$ 属于 $R$ ，那么根据 $R$ 的定义，它只包含不属于自身的元素构成的集合，应该有 $R$ 不属于自己。反之，如果 $R$ 不属于 $R$ ，同样根据定义，它包含不属于自身的集合，又应该有 $R$ 属于 $R$ 。不管属于或不属于，都会导致矛盾。形式化的表达就是：

$$R \in R \iff R \notin R$$

这样罗素就明确表明了康托尔的集合论中存在悖论。

罗素1872年生于英国蒙茅茨郡的一个贵族家庭。两岁时母亲去世，三岁时父亲也去世。6岁时祖父也去世了，于是罗素和祖母生活在一起。祖母对他的童年和青少年时期的发展有过决定性的影响。她曾告诫罗素：“你不应该追随众人去做坏事”，罗素一生都努力遵循这条准则。

罗素少年时代未被送到学校去学习，而是在家接受教育。1883年开始，11岁的罗素跟随堂哥弗兰克学欧几里德几何。不久，罗素开始接触哲学思辨，并在宗教问题上，悄悄写下自己的想法在一家杂志发表。1890年罗素考入剑桥大学三一学院，大学前三年，他专攻数学，获数学荣誉学位考试的第七名。1894年，参加伦理学荣誉学位考试。完成研究论文《论几何学的基础》。在剑桥期间，他结识了当时的数学讲师怀特海等人。1895年，罗素在三一学院获得了研究员的职位。二十世纪初，他发现了著名的罗素悖论，并引发了一场关于数学基础的大讨论。其后十多年间，罗素投身于数学基础和数理逻辑的研究中。1920年，罗素应邀到中国讲学一年。足迹遍及中华南北，作了多场演讲。话题从数理逻辑到切中时弊的社会改造建议，在当时成为中国文化界的一件盛事。给我国哲学界以很大的影响。他的《西方哲学史》在我国的哲学爱好者中有着广泛的影响。

二十世纪50年代后，罗素从哲学转向国际政治。他反对核战争、主张核裁军。由于伸张民主和参加核裁军运动，罗素一生曾两次被捕入狱。其中第二次入狱时已经是89岁高龄。1950年罗素获得诺贝尔文学奖。委员会在授奖时称他为“当代理性和人道的最杰出代言人之一，西方自由言论和自由思想的无谓斗士。”

1970年2月2日，罗素在彭林德拉耶斯逝世，他的骨灰被撒在威尔士的群山之中。



伯特兰·罗素 1872-1970

### 7.2.1 罗素悖论的影响

罗素发现集合论基础的悖论后极为沮丧。他后来回忆道：“每天早晨，我面对一张白纸坐在那儿，除了短暂的午餐，我一整天都盯着那张白纸。常常在夜幕降临之际，仍是一片空白……似乎我整个余生很可能就消耗在这张白纸上。让人更烦恼的是，矛盾是平凡的。我的时间都花在这些似乎不值得考虑的事情上。”([9], 第231页) 罗素把他的发现告诉了数学家、逻辑学家弗雷格。当时弗雷格正在进行算术基础的建立工作，他的著作《算术的基本规律》已在付印中。弗雷格看到罗素悖论后非常沮丧，他写道：“一个科学家所遇到的最不合心意的事莫过于在他工作即将结束时，其基础崩溃了。罗素先生的一封信正好把我置于这个境地。”戴德金也推迟了《什么是数的本质》一书的再版。罗素悖论涉及的是集合论中最基础的部分。由于集合论逐渐被大家接受，并进入了大多数数学分支，这使得人们对于数学和逻辑学的基本原理和有效性产生了怀疑。

### 练习 7.1

1. 我们可以用语言定义数，例如“最大的两位数”定义了99。定义一个集合，是所有不能用20个以内的字描述的数字。考虑这样一个元素：“不能用20个以内的字描述的最小数”，它是否属于这个集合？
2. “这个世界上唯一不变的是变化”——这句话是否是罗素悖论？

3. 本章开头苏格拉底的话是否是罗素悖论?

## 7.3 数学基础的分歧

为了解决罗素悖论这一影响理性思维基础的问题。数学家们从1900年到1930年间持续进行讨论并各自提出了解决方案。数学在历史上长期被当作理性思维的真理，其绝对性和唯一性从未被引起怀疑和争论。在这一大讨论中，人们终于意识到，在不同的哲学观念下，可以存在不同的数学。

### 7.3.1 逻辑主义



戈特洛布·弗雷格(1848-1925)

逻辑主义的早期代表人物是弗雷格。他认为数学的基础并不是数，算术理论可以建立在逻辑的基础上。弗雷格把朴素集合论看作是逻辑的一部分。他做的第一件工作是利用逻辑来定义自然数。我们知道数具有抽象的含义。3可以代表3个人、三个鸡蛋、图形中的三角等等，这些类<sup>2</sup>都有三个元素，用哪一个来代表自然数3呢？弗雷格的意见是：全部。即所有能和上述类一一对应的类所组成的，无穷的，抽象的类来定义自然数3。弗雷格的这一定义看起来有些复杂，但是很了不起。它突破了文化背景的限制。不管你是使用何种语言，何种符号，按照弗雷格的方法，对数字3的理解都不会有歧义。因为佛雷格的定义中根本不需要任何符号。这样佛雷格就定义了数——它是所有类的类。接下来佛雷格借助这一定义和逻辑理论建立了自然数的理论，进而形成了逻辑化的算术理论。再进一步，弗雷格打算从利用逻辑发展出除几何以外的全部数学。这就是他在《算术的基本规律》一书中打算完成的事情。由于弗雷格坚信逻辑的原则是完全可靠的，这样他的工作一旦完成，数学“就被固定在一个永恒的基础上了。”

我们知道接下来发生了什么。在《算术的基本规律》正在付印时，罗素的信“及时”寄到了。罗素悖论让弗雷格陷入了困惑。他工作的基础——利用逻辑定义数的概念——恰好是关于所有类的类。这样的定义直接导致了悖论的出现。弗雷格动摇了，并最终放弃了他的逻辑主义立场。

罗素接过了逻辑主义的火炬，积极投身于数学基础的重建和悖论的解决工作中。罗素坚信数学就是逻辑，逻辑就是数学。他形成这样的思想，意大利数学家皮亚诺起到了重要的作用。罗素后来说道：“在我学术生命中最重要的一年是1900年，而这一年中最重要的事是我去巴黎参加国际哲学会议……皮亚诺和他的学生们在一切讨论中所表现出来的，为他人所没有的精确性，给了我深刻的印象。”他和怀特海两人每天讨论数学的基本概念，经过艰苦的工作终于写出了著名的《数学原理》<sup>3</sup>三卷本分别在1910年到1913年间出版。这可以说是数理逻辑的经典之作。为了解决悖论，罗素指出一切悖论都源于某种“恶性循环”，而恶性循环源于某种不合法的集体，具体的说就是集体的整体这一概念。所以要想消除悖论、避免恶性循环，凡是涉及一个集体的整体的对象，它本身不能是该集体的成员。从这一思想出发，罗素提出了“分支类型论”。

分支类型论将集合进行了层次划分，定义域中的对象个体属于第0类；个体的集合属于第1类；第1类中的个体的集合，也就是集合的集合，属于第2类……每一集合都必须从属确定的类。而命题中的对象必须从属于它所在的等级。这样做可以有效地消除悖论，但使用起来极其繁琐不便。

《数学原理》第一卷直到第363页才推出数字1的定义。庞加莱挖苦道：“这是一个可亲可佩的定义，它献给那些从来不知道1的人。”使用分支类型论，所有的工作只能在各自的等级上进行，整数在整数的等级上，有理数在有理数的等级上，我们不能把 $n/1$ 和n混为一谈。更为严重的是：“所有实数……”这样的命题不合法了，因为它涉及了集合中不同的层次。

<sup>2</sup> 弗雷格的工作在康托尔之前，他当时使用了“类”（class）一词。康托尔后来使用了德语中的“集合”。

<sup>3</sup> 罗素和怀特海希望通过书的名字向牛顿致敬。因为牛顿的著作名为《自然哲学中的数学原理》

但最有争议的地方是“无穷公理”、“选择公理”、“约化公理”的使用。为了处理自然数以及更为复杂的实数和超限数，罗素和怀特海引入公理来承认无穷的存在。他们也承认可以从非空集合，甚至无穷集合中选择元素组成新的集合。这两条有争议的公理在集合论中也存在，但最难以让其他数学家接受的是约化公理。为了支持数学归纳法，约化公理认为任何较高层次的一个命题与一个层次为0的命题等价。约化公理激起了反对，因为它显得太任意了。1909年庞加莱说：约化公理比数学归纳法更靠不住，更含糊不清。

后来连罗素自己也动摇了：“从严格的逻辑化来看，我找不出任何理由来相信约化公理是逻辑必然的，这就是说，它在所有可能的世界中都是真的。因此，在逻辑体系中，承认这个公理是个缺憾，即使从经验来看是真的。”<sup>[63]</sup>

### 7.3.2 直觉主义



布劳威尔(1881-1966)

与逻辑主义同时，另一群称为直觉主义的数学家使用了截然不同、完全相反的方法来重建数学的基础。直觉主义可以追溯到帕斯卡，其先驱是上一章介绍过的德国数学家克罗内克。鲍莱尔、勒贝格、庞加莱、外尔等一批数学巨匠都是逻辑主义的支持者。其代表人物是荷兰数学家布劳威尔。布劳威尔于1881年生于荷兰鹿特丹附近的小镇奥弗希。1897年他考入阿姆斯特丹大学攻读数学。在读大学时，他获得了关于四维空间连续运动的某些结果，并发表在阿姆斯特丹皇家科学院报告集上。在当大学生时，通过自己的刻苦钻研，更由于受到曼诺利(Mannoury)教授一系列启迪性讲座的启发，布劳威尔接触到了拓扑学和数学基础，并且终生钟爱它们。

受到希尔伯特在巴黎的第二届国际数学家大会的讲演的影响，布劳威尔从1907年到1913年进行了大量拓扑学的研究。建立布劳威尔不动点定理是他的突出贡献。这个定理表明：在二维球面上，任意映到自身的一一连续映射，必定至少有一个点是不变的。他把这一定理推广到高维球面。尤其是在 $n$ 维球内映到自身的任意连续映射至少有一个不动点。1910年，布劳威尔证明了维数的拓扑不变性。1913年，他给出了拓扑空间维数的严格定义。

由于布劳威尔在拓扑学上的出色成就，他被推选为荷兰皇家科学院院士。

在攻读博士学位时，布劳威尔以极大的热情关注着罗素和庞加莱关于数学的逻辑基础的论战<sup>4</sup>，并以此为题写成他的博士论文。总的说来，他倾向于庞加莱的观点，反对罗素和希尔伯特关于数学基础的思想。但是，他又不同意庞加莱关于数学存在性的说法。他认为，庞加莱的办法不能排除悖论。为此，他在博士论文“论数学基础”中开始建立直觉主义的数学哲学。1966年，85岁的布劳威尔不幸死于车祸。

布劳威尔的直觉主义来源于他的哲学。数学是起源和产生于头脑的人类活动，它并不存在于头脑之外，因此，它是独立于真实世界的。头脑识别基本的、清晰的直觉，这些直觉不是感觉或经验上的，而是对某些数学概念直接的确定，其中包括整数。布劳威尔认为数学思维是智力构造的一



阿尔弗雷德·怀特海(1861-1947)

<sup>4</sup> 庞加莱认为逻辑和直觉是数学与科学中不可或缺的两个重要方面。逻辑可以帮助我们严密化，直觉是发明和创造所必须的。逻辑不能完全替代直觉。直觉可能导致假象<sup>[64]</sup>。

个过程，它建造自己的天地，独立于经验，并且只受到必须建立于基本的数学直觉之上的限制。这种基本的直觉概念不应被理解为像在公理理论中的那种未定义概念，而应设想为某种东西，只要它们在数学思维中确实是有用的，用它就可以对出现在各种数学系统中的未定义概念做出直观上的理解。

布劳威尔认为“要在这个构造过程中发现数学唯一可能的基础，必须再三思考，反复斟酌。哪些论点是直觉上可接受的，头脑中所自明的；哪些不是。”是直觉而不是经验或逻辑决定了概念的正确和可接受性。当然，这一陈述并未否认经验所起的历史作用。除了自然数以外，布劳威尔坚持认为加法、乘法和数学归纳法在直觉上是清晰的。而且，当头脑已获得自然数1, 2, 3……的概念后，使用“空洞形式”无限重复的可能性，从 $n$ 到 $n+1$ 的步骤，就产生了无穷集合。然而，这种集合只是潜无穷，因为对于任一给定的有限数集，总可以加入一个更大的数。布劳维否定了康托尔的所有元素都“一下子”出现的无限集，并因此否定了超限数理论、策梅罗的选择公理以及使用了真正的无限集的那部分分析。在1912年的一次演讲中，布劳威尔甚至接受了直至 $\omega$ 的基数和可数集。

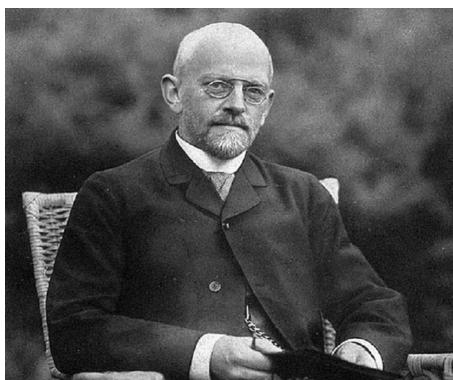
布劳威尔坚持要求可构造性，否定无限制地使用排中律，尤其在涉及无穷时。这样以来传统数学中的许多内容都必须被丢弃了。例如欧几里得关于素数有无穷多的证明，并没有依次构造出素数，而是通过排中律指出存在性。这样在直觉主义看来是不可接受的。为了让其他数学家信服，1924年，布劳威尔提出了扇形定理。这一定理表明存在这样的性质，对于有界展延的全部元素来说，可以使它要么持有这种性质要么不持有这种性质。对排中律的否定产生了一种新的可能性——不可判定的命题。对于无穷集合，直觉主义主张还有第三种状况，即可以有这样的命题，既不是可以证明的，也不是不可以证明的。

总体来说直觉主义在当时的工作中批判多于建设。直觉主义否定了一大批数学成果。包括无理数概念，函数论、康托尔的超限数。一大批推理模式包括排中律都无法使用。因此遭到了其他数学家们的强烈反对，希尔伯特说：“与现代数学的浩瀚大海相比，那点可怜的残余算什么。直觉主义者所得到的是一些不完整的、没有联系的孤立的结论。”



庞加莱(1854-1912)

### 7.3.3 形式主义



大卫·希尔伯特(1862-1943)

关于数学基础思想的第三大派系是由希尔伯特领导并风行一时的形式主义流派。希尔伯特是伟大的德国数学家，1862年生于东普鲁士的哥尼斯堡。他从小勤奋好学，对于科学特别是数学表现出浓厚的兴趣。1880年，希尔伯特进入哥尼斯堡大学学习数学。在这里结识了比他年长三岁的副教授胡尔维茨和比他高一班的闵可夫斯基。希尔伯特后来这样追忆他们的友谊：“在日复一日无数的散步时刻，我们漫游了数学和科学的每个角落。”

1895年，数学领袖克莱因邀请希尔伯特去哥廷根大学。希尔伯特在那里度过了48年直到去世。他与克莱因一起，开创了哥廷根学派的全盛时期，把哥廷根大学变成了全世界的数学中心。世界各地的学生把哥廷根看作数学的圣地：“打起背包，到哥廷根去！”

希尔伯特是二十世纪前后最伟大的数学家之一。当时唯一可以与其并驾齐驱的是法国数学家庞加莱。他在数学几乎所有领域都作出了巨大贡献。以希尔伯特命名的数学名词多如牛毛，有些连

希尔伯特本人都不知道。比如有一次，希尔伯特问系里的同事“请问什么叫做希尔伯特空间？”他去世时，美国的《自然》杂志上写道：“希尔伯特就像数学世界的亚历山大，在整个数学版图上，都留下了他那巨大显赫的名字。”[9]

1900年，在巴黎召开的第二届国际数学家大会上，希尔伯特提出了新世纪数学家应当努力解决的23个数学问题，被认为是20世纪数学的至高点，对这些问题的研究有力推动了20世纪数学的发展，在世界上产生了深远的影响。

希尔伯特帮助培养了一大批顶级的数学家，包括外尔、柯朗、埃米·诺特、冯·诺伊曼、策梅罗等等。1933年纳粹上台，驱赶了大批犹太裔师生。1943年，希尔伯特在孤独中逝世。在他的墓碑上，留下了他那极富感染力的乐观主义名言：“我们必须知道，我们必将知道。”

希尔伯特的《几何基础》（1899）是公理化思想的代表作，书中把欧几里得几何学加以整理，成为建立在一组简单公理基础上的纯粹演绎系统，并开始探讨公理之间的相互关系与研究整个演绎系统的逻辑结构。1904年，希尔伯特开始着手研究数学基础问题，经过多年酝酿，于二十年代初，提出了如何论证数论、集合论或数学分析一致性的方案。他建议从若干形式公理出发将数学形式化为符号语言系统，并从不假定实无穷的有穷观点出发，建立相应的逻辑系统。然后再研究这个形式语言系统的逻辑性质，从而创立了元数学和证明论。希尔伯特的目的是试图对某一形式语言系统的无矛盾性给出绝对的证明，以便克服悖论引起的危机，一劳永逸地消除对数学基础以及数学推理方法可靠性的怀疑。

为了实现这一主张，希尔伯特规划了他的方案，主要内容是：

1. 证明古典数学的每个分支都可在数学系统公理化意义下予以公理化。
2. 证明每一个在上述意义下被公理化了的系统都是完备的。即系统内任一可表述的命题均可在系统内得到判定。
3. 证明每一个在上述意义下的系统都是相容的。
4. 证明每个这样的系统所相应的模型都是同构的。
5. 寻找这样一种方法，借助于它，可在有限步骤内判定任一命题的可证明性。

希尔伯特为具体实施其规划而创立证明论，即元数学理论。它着眼于整个形式系统，并以“证明”本身作为研究对象。这样就区分出三种数学系统：

1. 非形式化的数学系统 $G$ ：即普通的数学系统，在其中允许使用古典逻辑推理规则，例如，在无穷集合上使用排中律等。
2. 形式化的数学系统 $H$ ：在 $H$ 中的符号、公式、公理、命题等都是形式的，在未加解释之前都是没有内容和意义的，而经解释后就是 $G$ 中相应的内容。即 $G$ 是 $H$ 的模型， $H$ 是 $G$ 的形式化。用希尔伯特的一句名言来举例：“我们必定可以用‘桌子、椅子、啤酒杯’来代替（几何中的）‘点、线、面’。”在这种处理下，原几何理论中所包含的特定意义和直观背景被完全舍弃。我们研究的只是未定义项之间的关系，而关系由公理组来体现。
3. 元数学系统 $K$ ：这是用以研究 $H$ 的元理论，而在 $K$ 中的推理规则必须保持直觉的可信性，例如不能涉及无穷，不允许在无穷集合上使用排中律等。

就在希尔伯特实施他的规划时，1931年，年轻的哥德尔发现了不完全性定理，从本质上宣告了希尔伯特计划是行不通的。我们将在后面详细介绍这一发现。

### 7.3.4 公理集合论

与逻辑主义、直觉主义、形式主义不同，集合论公理化派的成员在开始时并没有形成他们独特的哲学，但是他们逐渐获得了支持，有了明确的方案。在今天，我们可以肯定地说这个派别在数学家中所拥有的支持者是与我们前面介绍的三个派别势均力敌的。

集合论公理化的起源可以追溯到戴德金和康托尔的工作中。尽管他们主要关心的是无穷集合问题，并且也都着手于在集合概念的基础上建立整数的概念。一旦整数建立了，也就能推导出全部数学了。当罗素悖论和集合论的矛盾出现时，一些数学家相信这是由于滥用集合所致。康托尔的集合论的整个表示形式在今天通常被说成“朴素集合论”。因此集合论的公理化思想，做为一种经仔细选择的公理化基础，可以排除集合论中的悖论，正如几何和数系中的公理化可以在那些领域里解决逻辑问题一样。1908年数学家策梅罗沿着这一方向进行了一次成功的尝试。

策梅罗希望清晰明确的公理能够澄清集合的含义和集合所应具有的属性，尤其是他想要设法限制集合的大小。他没有什么哲学根据，只是力图避免矛盾。他的公理系统包含未加定义的集合的基本概念，以及一个集合被另一个集合所包含的关系。所有这些加上已定义的概念就可以满足公理中的陈述，只有公理所提供的集合的性质才能使用。在公理中，无穷集的存在性，以及像集合的并与子集的形成这一类的运算也由公理给出。策梅罗也用到了选择公理。



(a) 恩斯特·策梅罗(1871-1953)



(b) 亚伯拉罕·弗兰克尔(1891-1965)

策梅罗的公理系统在1922年由弗兰克尔改进。策梅罗没有区分集合的属性和集合本身，它们被当作同义语使用。弗兰克尔在1922年找出了它们之间的区别。这套被集合论公理化者最通常使用的公理系统叫做策梅罗-弗兰克尔系统，简称ZF系统。他们俩分别预测到了精致的、严密的数学逻辑的可行性，却没有详细说明逻辑的原理。他们认为这些都是在数学范围之外的，并且确信他们可以像1900年以前的数学家使用逻辑一样来使用这些逻辑原理。

策梅罗在1908年的论文中给出了集合论的7条公理。1930年又加入了弗兰克尔、斯克朗、冯·诺伊曼建议的两条公理。这些公理表示如下：

1. **外延公理**：如果两个集合含有相同的元素，那么它们相等。形式化就是对于集合 $A$ 和 $B$ ，若 $A \subseteq B$ 且 $B \subseteq A$ ，则 $A = B$ 。外延公理相当于逻辑上的同一律。
2. **空集**：空集存在。
3. **分离公理**：任何可用理论形式化的属性都可以用来定义一个集合。即对集合 $S$ ，命题函数 $p(x)$ 是确定的，则存在集合 $T = \{x | x \in S, p(x)\}$ 。分离公理也称作概括公理。
4. **幂集公理**：对任一集合，都可以作出其幂集；即任一给定集合中的所有子集的全体也是一个集合（这个过程可以无限次重复）。
5. **并集公理**：一组集合的并也是一个集合。

6. **选择公理**: 设  $S$  为一个由非空集合所组成的集合, 可以从每一个在  $S$  中的集合中, 都选择一个元素和其所在的集合配成有序对来组成一个新的集合。选择公理简称为 AC。
7. **无穷公理**: 存在一集合  $Z$ , 它含有空集, 对任一对象, 若  $a \in Z$ , 则  $\{a\} \in Z$ 。(这一公理保证了无限集是可构造的。)
8. **替换公理**: 这条公理是弗兰克尔1922年引入的。对于任意的函数  $f(x)$  和集合  $T$ , 当  $x \in T$  时,  $f(x)$  都有定义的前提下, 一定存在一集合  $S$ , 对于所有的  $x \in T$ , 在集合  $S$  中都有一元素  $y$ , 使  $y = f(x)$ 。也就是说, 由  $f(x)$  定义的函数其定义域在  $T$  中的时候, 它的值域可限定在  $S$  中。
9. **正则公理**: 这条公理是冯·诺伊曼1925年引入的。 $x$  不属于  $x$ 。

这样, 集合论就被抽象成一个公理化的理论。集合成了未定义概念, 它是满足上述公理的对象。通过这样的限制, 避免了“所有对象”这样的说法, 从而避免了悖论, 弥补了朴素集合论的缺陷。但是就这些公理的选择和承认仍然存在争议。其中最大的争议来自选择公理。



图 7.12: 巴拿赫-塔斯基“悖论”: 一个球可以分解和重新组合成两个大小和原来一样的球

1924年波兰数学家巴拿赫和塔斯基证明了分球定理<sup>5</sup>。这一定理指出在选择公理成立的情况下, 可以将一个三维实心球分成有限(不可测的)部分, 然后仅仅通过旋转和平移到其他地方重新组合, 就可以组成两个半径和原来相同的完整的球。巴拿赫和塔斯基提出这一定理原意是想拒绝选择公理, 但该证明很自然, 因此数学家认为这仅意味着选择公理可以导致少数令人惊讶和反直觉的结果, 有人提出不应该把它包含进来。不包含选择公理的集合论被称为ZF系统, 包含选择公理的被称为ZFC系统。我么在上一章曾介绍过选择公理和连续统假设之间的有趣关系。

## 7.4 哥德尔不完全性定理

这样, 到1930年, 四种彼此独立的、截然不同的并且或多或少有些冲突的关于数学基础的方法都已亮相。并且可以毫不夸张地说, 他们彼此的追随者也都处于对峙状态。一个人再也不能说一条数学定理是被正确地证实了, 因为到1930年, 他必须加上一句, 即依照谁的标准它被认为是正确的。除了直觉主义者认为人的直觉能保证相容性外, 数学的相容性, 这个激发了新方法的重要问题, 根本就没有得到解决。希尔伯特还在乐观地规划着证明数学是完备的和一致的。改变这一切的是年轻的数学家、逻辑学家哥德尔。

哥德尔于1906年生于奥匈帝国的布尔诺(今属捷克共和国)。8岁时突患急性风湿热, 很可能是这次疾病的后果, 哥德尔后来不断被妄想症困扰。童年时期他充满好奇心, 被家里人叫作“为什么先生”。1924年哥德尔考入了维也纳大学学习物理。接触了数论的课程后, 很快哥德尔意识到数学才是自己真正的追求。于是1926年转入数学系。哥德尔对于哲学也很有兴趣, 经常参加哲学小组的讨论, 旁听哲学教授的课程。对哲学的探索贯穿了哥德尔的一生。

1929年夏天, 23岁的哥德尔证明了“谓词演算的有效公式皆可证”, 并于1930年以此获得了博士学位。随后, 他进一步研究希尔伯特规划, 试图寻找在有限步骤内证明自然数系统的相容性和一致性的办法。但是哥德尔得到了一个意外的结果。1930年在哥尼斯堡召开的数学讨论会上, 他公布了这一结果——即伟大的哥德尔第一不完全性定理。不久他又证明了第二不完全定理。

<sup>5</sup>又称为豪斯多夫-巴拿赫-塔斯基定理, 或者“分球怪论”



库尔特·哥德尔(1906-1978)

1933年后，哥德尔一直在维也纳大学工作。希特勒上台后，纳粹开始插手奥地利的学术界。1936年，维也纳学派的物理学家和逻辑学家摩里兹·石里克被一个学生刺杀，当场死亡。哥德尔参加过石里克的讨论组，因此深受刺激，他患上了妄想症，总疑心有人要毒杀自己。二战爆发后，哥德尔接受了普林斯顿高等研究院的邀请来到美国。在这里他结识了爱因斯坦并成为了终身好友。他们经常一起在普林斯顿散步和闲谈。1955年爱因斯坦去世对哥德尔的情绪有很大打击。哥德尔晚年，美籍华裔逻辑学家王浩是他最好的朋友之一。

哥德尔的妻子阿黛尔(Adele Nimbursky)比他大六岁。哥德尔21岁两人认识时，阿黛尔已婚且在夜总会工作。他们的婚姻遭到哥德尔家人反对，但有情人终成眷属，在1938年9月20日结婚。哥德尔的晚年一直和妄想症斗争，他唯恐有人下毒，只吃妻子阿黛尔做的食物。1977年阿黛尔动手术住院后，他干脆什么都不吃了。1978年

1月，“因营养不良和身体机能衰竭”与世长辞，死时体重只有60磅。由于在逻辑方面的杰出贡献，人们把他视为自亚里士多德以来最伟大的逻辑学家。

1931年，哥德尔发表了论文“论《数学原理》及有关系统中的形式不可判定命题”，题目中的《数学原理》就是罗素和怀特海的巨著。哥德尔证明了在任何包含自然数算术的形式系统中，如果它是相容的，则必定存在一不可判定命题 $G$ ，即不能证明 $G$ ，也不能证明 $G$ 的否定。这一定理被称为哥德尔第一不完全性定理。这一定理表明，无矛盾的形式化系统是不完全的。只要系统强大到足以包含自然数公理系统，都会有超越于它的问题。人们自然会想，既然 $G$ 是不可判定的命题，如果把 $G$ 或者 $G$ 的否定作为公理加入到系统中，不就可以得到一个更为强大的系统了吗？但是不久哥德尔进一步证明了第二不完全性定理。它表明，如果一个足以包含自然数算术的公理系统是无矛盾的，那么这种无矛盾性在该系统内是不可证明的。所以不论是把 $G$ 或是 $G$ 的否定当作公理加入系统，得到的新系统仍然是不完全的。总是存在更高一层的不可判定命题。

例如在欧几里得几何中，如果把第五公设抽出，仅仅使用前四条公理的形式化系统既不能证明第五公设，也不能证否第五公设。我们后来知道承认或者否定第五公设都会导致无矛盾的几何——分别是欧几里得几何和不同的非欧几何。再比如在公理集合论ZF系统中，既不能证明也不能证否选择公理，承认选择公理得到无矛盾的ZFC系统，而否定选择公理得到另一无矛盾的系统。即使加入选择公理，在ZFC中既不能证明也不能证否连续统假设。承认连续统假设得到一种无矛盾的系统，否认连续统假设得到另外的无矛盾系统。

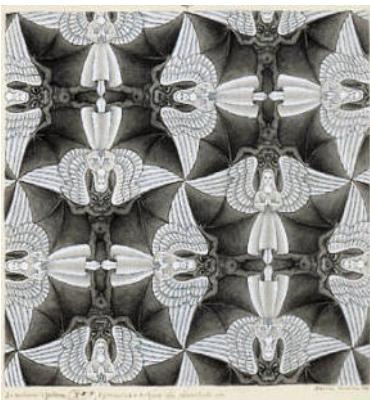


图 7.14: 埃舍尔《天使与魔鬼》1941

哥德尔第一不完全性定理和第二不完全性定理合称为“哥德尔不完全性定理”。这等于宣布了希尔伯特纲领是行不通的。即便基本算术系统是协调的，那么这种协调性也不可能在算术系统内证明。伟大的数学家安德烈·韦伊评论说：“因为数学具有一致性，所以上帝与我们同在；因为我们不能证明这一点，所以魔鬼亦与我们同在。”<sup>[9]</sup>

## 7.5 不完全性定理的证明

按照希尔伯特规划，首先是把整个数学理论组织成一个形式系统。然后在元数学中对这一形式系统进行研究。为此我们需要把古典数学的每一分支表述为这样的形式系统。其中只包含有限条公理，然后通过元数学证明这样的系统是完备的且无矛盾的。其中最基本的一个系统就是自然数算术，因为很多数学系统都同构于它。在上一章，我们看到了如何从自然数出发定义出整数，有理数，甚至实数。从实数对应到点，我们又可以利用解析几何把欧几里得几何算术化。

### 7.5.1 构建形式系统

这里我们采用侯世达在《哥德尔，埃舍尔，巴赫——集异璧之大成》中的方法和术语来简要介绍这一证明过程。哥德尔不完全性定理的证明也是从构建形式系统开始的。我们称这一系统为“印符数论”简称TNT<sup>6</sup>。恰巧这也是黄色炸药三硝基甲苯的分子式，暗示其威力大到足以——摧毁自己这座大厦。所谓印符数论，就是把用我们熟悉的自然语言表示的数论形式化为一系列印刷字符。这看起来很复杂，但是在我们第一章中介绍的皮亚诺公理的基础上，其实并不难。首先我们要定义数字，按照皮亚诺公理，零是自然数，每个自然数都有其后继，我们很自然地可以这样定义数的印符：

零	0
一	S0
二	SS0
三	SSS0
.....	.....

其中符号S代表后继，两个S表示后继的后继。一百个S和一个0表示0的一百次后继，也就是自然数100。尽管很长，但是规则异常的简单。定义了自然数，我们还要定义变量，为了是系统尽量简单，我们可以仅使用5个印符字母 $a, b, c, d, e$ 。当需要更多变量时我们就加撇号， $a', a'', a'''$ 。接下来我们需要加法符号“+”和乘法符号“·”以及辅助运算顺序的左右括弧。为了形式化命题我们需要等号“=”，表示否定的 $\neg$ ，表示蕴含的箭头 $\rightarrow$ 。这样我们就可以表示命题了，例如（先不论命题的真假）：

- 一加二等于四： $(S0 + SS0) = SSS0$
- 二加二不等于五： $\neg(SS0 + SS0) = SSSSS0$
- 如果1等于0，那么0等于1： $(S0 = 0) \rightarrow (0 = S0)$

一个命题中可以含有自由变元，例如：

$$(a + SS0) = SSS0$$

表示 $a$ 加上2等于3。显然 $a$ 的取值决定这个命题的真假，为此我们还需要引入存在量词符号 $\exists$ 和全称量词符号 $\forall$ ，以及表示量词约束关系的冒号“:”。这样命题：

$$\exists a : (a + SS0) = SSS0$$

---

<sup>6</sup>印符数论的英文Typographical Number Theory首字母的缩写。

就表示：存在 $a$ 使得 $a$ 加上2等于3。再看另一个例子：

$$\forall a : \forall b : (a + b) = (b + a)$$

这恰好就是自然数的加法交换律。如果去掉 $a$ 的量词约束，就变成：

$$\forall b : (a + b) = (b + a)$$

这是一个开公式，其中 $a$ 是自由变元。它表示未指定的 $a$ 可以与 $b$ 交换。当然这一命题的真假并没有确定。为了把命题复合起来，我们还需要析取符号 $\wedge$ ，合取符号 $\vee$ 。尽管TNT中的符号极少，可是它的表达能力却很强，我们举一些例子看：

2不是任何数的平方： $\neg \exists a : (a \cdot a) = SS0$

费马大定理在 $n$ 为3时成立： $\neg \exists a : \exists b : \exists c : ((a \cdot a) \cdot a) + ((b \cdot b) \cdot b) = ((c \cdot c) \cdot c)$

现在，我们只是有了可以表示命题的印符，为了构建TNT形式系统，我们还需要公理和推理规则。

### 7.5.1.1 公理和推理规则

仿照皮亚诺算术的公理，我们为TNT系统定义下述的公理：

1.  $\forall a : \neg Sa = 0$ ，这条公理说，没有任何自然数是零的后继；
2.  $\forall a : (a + 0) = a$ ，这条公理表示任何数加零都等于它本身；
3.  $\forall a : \forall b : (a + Sb) = S(a + b)$ ，这条公理定义出了自然数的加法；
4.  $\forall a : (a \cdot 0) = 0$ ，这条公理表示任何自然数乘以零都为零；
5.  $\forall a : \forall b : (a \cdot Sb) = ((a \cdot b) + a)$ ，这条公理定义出自然数的乘法。

接下来我们构建推理规则。比如我们希望从公理1，零不是任何自然数的后继这一普通情况，导出1不是零的后继这一特殊情况，为此我们需要引入特称规则：

**特称规则：**如果 $u$ 是出现在印符串 $x$ 中的变元。如果 $\forall u : x$ 是一个定理，则 $x$ 也是定理，并且对 $x$ 中的 $u$ 做任何替换得到的新印符串也是定理。

这里有一个限制，在替换 $u$ 时，不能包含任何 $x$ 中被量化的变元，并且替换要一致。与特称规则相反的是概括规则。这个规则允许我们把全称量词加到定理的前面。

**概括规则：**如果 $x$ 是一个定理，其中的变元 $u$ 是自由出现的。则 $\forall u : x$ 也是一个定理。

例如， $\neg S(c + S0) = 0$ ，表示不存在一个数加一的后继等于零，我们可以概括为： $\forall c : \neg S(c + S0) = 0$ 。

接下来的规则可以让我们把全称量词和存在量词互换。

**互换规则：**如果 $u$ 是一个变元，那么印符串 $\forall u : \neg$ 与 $\neg \exists u :$ 可互换。

例如公理1可以按照互换规则变成： $\neg \exists a : Sa = 0$ 。接下来的一个规则允许我们在一个印符串的前面加上存在量词。

**存在规则：**假设一个项在定理中出现一次或多次，可以用一个变元来代替这个项，并且在前面加上存在量词。

我们还用公理1来举例： $\forall a : \neg Sa = 0$ 中，我们可以把0，用变元 $b$ 来代替，并且在前面加上存在量词。这样就得到： $\exists b : \forall a : \neg Sa = b$ 。意思是说，存在一个数，使得任何自然数都不是它的后继。

接下来考虑相等的对称性和传递性，我们定义等号规则。令 $r, s, t$ 都代表任意的项。

**等号规则：**

- 对称：如果 $r = s$ 是定理，则 $s = r$ 也是定理；

- 传递：如果  $r = s$  和  $s = t$  都是定理，则  $r = t$  也是定理。

对于增、减后继符号  $S$ ，我们也定义一个规则。

**后继规则：**

- 增加：如果  $r = t$  是定理，则  $St = Sr$  也是定理；
- 去除：如果  $St = Sr$  是定理，则  $r = t$  也是定理。

现在的印符数论TNT已经很强大了，我们可以用它构造出各种比较复杂的定理。

## 练习 7.2

1. 尝试给出费马大定理的印符串。
2. 尝试用印符推理规则证明加法结合律。

### 7.5.1.2 印符系统的不完完全性

使用TNT系统中的公理和推理规则不难证明下面的一系列定理：

$$\begin{array}{ll} (0 + 0) & = 0 \\ (0 + S0) & = S0 \\ (0 + SS0) & = SS0 \\ (0 + SSS0) & = SSS0 \\ \dots & \dots \end{array}$$

第一条可以从公理2中，把  $a$  代换成  $0$  导出；第二条可以用公理3和上一条定理导出，每条定理都可以从上一条定理轻松导出。作为旁观者的我们，立刻想到这能不能概括成一条定理：

$$\forall a : (0 + a) = a$$

请注意它和公理2的区别。遗憾的是，利用迄今为止所给出的TNT规则，我们无法导出这一定理。我们也许希望添加一条规则，如果一系列这样的串都是定理，那么概括它们的全称量词串也是定理。但是这条规则只有在系统外部的人才能洞见。它不是一条可以放入形式系统的规则。

缺少这样的概括能力说明印符数论系统TNT是不完全的，确切地说，叫作  $\omega$  不完全的。这里的  $\omega$  恰恰就是上一章介绍的可数无穷基数。我们说一个系统是  $\omega$  不完全的，如果一系列无穷的串都是定理，而其全称概述却不是定理。并且更奇特地，这一印符串的否定形式：

$$\neg \forall a : (0 + a) = a$$

也不是TNT中的定理。这意味着这个串在TNT中是不可判定的。这只是说TNT的能力不足以判定这个串是否是定理。就如同利用欧几里得几何的前四条公设无法判定第五公设一样。我们或者加入第五公设构成欧几里得几何；或者加入第五公设的否定构成非欧几何。我们可以将这一印符串或者其否定加入到TNT中，构造出不同的形式系统。

如果我们选择了这一印符串的否定，这里看似奇怪的一点是，零加上任何数都不再等于这个数了。这和我们通常熟悉的自然数加法大相径庭。但这恰恰提醒我们，所谓形式系统是带有未定义项的。我们只是方便理解选择了加号和自然数的含义。

印符系统TNT的  $\omega$  不完全性提示我们，我们的系统还缺少一条重要的规则——读者也许想到了——代表数学归纳法的皮亚诺第五公设。为此我们添上最后一块拼图。

**归纳规则：**如果  $u$  是印符串  $X$  中的一个变元，记为  $Xu$ 。如果把  $u$  替换为  $0$  时成立，且有  $\forall u : Xu \rightarrow XSu$ 。也就是若  $u$  时  $X$  成立，则将  $u$  替换为  $Su$  时  $X$  也成立。那么  $\forall u : Xu$  是一个定理。

加上数学归纳法后，印符数论系统TNT终于和皮亚诺算术具有同样的能力了。

### 练习 7.3

- 利用新加入的归纳规则证明  $\forall a : (0 + a) = a$

#### 7.5.2 哥德尔配数

哥德尔证明的最关键一步是引入了哥德尔配数。TNT系统已经强大到可以反映其它形式化系统，有没有可能利用TNT系统谈论它自身呢？哥德尔想到的办法就是把推理规则进行“算术化”。为此他把所有符号分配了一个数。

符号	数	符号	数
0	666	S	123
=	111	+	112
.	236	(	362
)	323	a	262
'	163	$\wedge$	161
$\vee$	616	$\rightarrow$	633
$\neg$	223	$\exists$	333
$\forall$	626	:	636

表 7.1: 对TNT系统进行哥德尔配数的一种方法

这样公理1就可以做如下的翻译：

$$\begin{array}{ccccccc} \forall & a & : & \neg & S & a & = \\ 626 & 262 & 636 & 223 & 123 & 262 & 111 & 666 \end{array}$$

配数的方法并不唯一。经过这样的配数，TNT中的任何串都可以表示为一个数了（尽管非常大）。现在问题来了，任给一个数，我们有没有办法判断它是一个TNT定理所代表的数呢？我们知道最开始有5个数一定是TNT数。它们就是五条公理所代表的数。根据TNT的推理规则，从这5个数开始，可以构造出无穷无尽的TNT数。在此之上我们引入一个数论谓词：

$a$ 是个TNT数。

例如626,262,636,223,123,262,111,666是个TNT数（这里的逗号只是为了方便读写，和英文中每三位一分隔是同样的）它表示公理1。其否定形式是：

$\neg a$ 是个TNT数。

例如我们说123,666,111,666不是个TNT数。这意味着我们可以把 $a$ 替换成0后面跟着123666111666个S。而这个巨大的串的含义实际是： $S0 = 0$ 不是TNT的定理。这意味着，TNT的确可以谈论它自己。这不是一种巧合，而是源于任何形式系统都可以在数论 $N$ 中得到反映。于是我们形成了一个圈：形式系统TNT中的一个串有一个数论 $N$ 中的解释，而 $N$ 中的一个陈述可以有第二层含义，就是作为元语言对TNT的陈述。

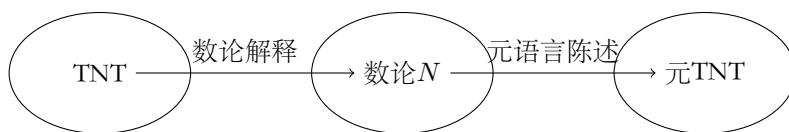


图 7.15:  $TNT \rightarrow \text{数论 } N \rightarrow \text{元TNT}$

### 7.5.3 构造自我指涉

哥德尔做的最后一步是构造一个自我指涉。找到一个TNT串，称之为 $G$ ，它是关于它自己的。其意义为：

$$G \text{不是TNT的定理}$$

接下来我们可以引爆TNT了。那么究竟 $G$ 是否是TNT中的定理呢？如果 $G$ 是一个定理，那么它表示的就是真理，即：“ $G$ 不是定理”。我们看到自指命题的威力了。由于充当了定理， $G$ 就不得不是个假理。根据我们的假定，TNT不会把假理当作定理，于是我们被迫得出结论说 $G$ 不是个定理。但是知道了 $G$ 不是个定理后，我们就得承认 $G$ 表示了一个真理。这揭示出TNT没有达到我们的期望——我们发现了一个符号串，它表示了真理，然而却不是一个定理。但考虑 $G$ 还有一个算术解释这一事实。它表示了关于自然数的某个算术性质的一个陈述。通过在TNT系统外面进行的推理，我们确定了这个陈述为真，还确定了这个串不是TNT的定理。我们问TNT这个串是否为真，TNT将既不能说“是”，也不能说“不”。

$G$ 就是那个不可判定命题。这就是哥德尔不完全定理的证明思路。

## 7.6 万能的程序与对角线证明

哥德尔不完全性定理对于编程意味着什么呢？我们有一个在编程上完全同构的问题。为此，我们从设计一个形式化的计算机语言开始。这个语言支持原始递归函数。所谓原始递归函数是一类数论函数，它们可以从自然数映射到自然数，并遵循下面5条公理：

1. 常函数：0元常函数0是原始递归的；
2. 后继函数：一元后继函数 $S(k) = k + 1$ 是原始递归的；
3. 投影函数：传入 $n$ 个数，返回其中第 $i$ 个数的函数 $P_i^n$ 是原始递归的；
4. 组合：原始递归函数的有限次组合是原始递归的；
5. 原始递归：

$$\begin{cases} h(0) = k \\ h(n+1) = g(n, h(n)) \end{cases}$$

称 $h$ 是由 $g$ 经原始递归运算得到的，可以扩展到多元的情况。

包含加、减基本运算符号、条件分支（if-then）、等于、小于判断和有界循环的编程语言被称为原始递归的编程语言。所谓有界循环指循环的次数在进入循环体前是确定的。它可以是不带跳转语句（goto）的loop，或者是在循环体中不能改变循环变量的for循环。但不能是while循环或repeat-until循环。由于这些限制，所有原始递归的程序必然是确定停机的。

原始递归函数的一个重要性质是全部原始递归程序是递归可枚举的。假设我们可以从全部原始递归程序中筛选出那些只有一个输入一个输出程序，把它们列出，放入到一个无穷大的程序库里。我们给每个程序一个编号<sup>7</sup>，从自然数0开始，1, 2, 3……一直枚举下去。我们可以把这些程序记为 $B[0], B[1], B[2], \dots$ 。对于第 $i$ 个程序，输入参数数 $n$ 时得到结果是 $B[i](n)$ 。

现在，我们构造这样一个特殊的函数 $f(n)$ ，当输入 $n$ 时，它的输出恰好是编号为 $n$ 的那个程序，当输入等于 $n$ 时的值加上1，即：

$$f(n) = B[n](n) + 1$$

---

<sup>7</sup>其中一种编号方案是把程序文字的ASCII码连接起来形成一个数，然后把这些数从小到大排列。由于没有两个程序是相同的，所以其ASCII码形成的数也不同。

这样的 $f$ 显然是可计算的。我们现在问， $f$ 是一个存储在函数库中的原始递归程序么？如果是，假设它在程序库中的编号是 $m$ ，根据我们之前的定义，第 $m$ 个程序输入 $m$ 时的结果应该是 $B[m](m)$ 。可是根据 $f$ 的定义，它的输出结果又应该是 $f(m) = B[m](m) + 1$ 。这两个结果显然不相等。这就证明了存在不是原始递归的可计算函数。

这个证明方法和上一章介绍的康托尔对角线证明法如出一辙。因此我们被迫放弃有界循环的限制来扩展编程语言的能力。我们可以引入带有跳转的loop循环，可以在循环体中改变循环变量的for循环，while循环，repeat-until循环，以及可递归的函数。这样就从原始递归函数扩展到了全递归函数。这样的语言称为图灵完备的语言。人们创造的大多数计算机语言都是图灵完备的，可以同构到自然数算术这样的形式系统中。但图灵完备的语言仍然存在漏洞，因为我们可以构造出判断停机的原始递归程序，从而证明存在着这不可计算问题。有没有可能进一步放宽限制，增强图灵完备语言，从而设计出万能的程序么？答案是否定的，图灵完备语言已经达到了最高级别，也就是形式系统的极限，没有其他限制可以去掉了。哥德尔不完全性定理告诉我们，形式系统一旦强大到足够的程度，可以包含自然数算术，那么它就必然存在不可判定命题。

## 7.7 尾声

我们的理性思维是伟大的。它可以跨越千年，与古代的先哲对话；它可以跨越宇宙，思考我们足迹无法踏上天体；它可以预见出肉眼看不见的基本粒子；它可以突破直觉，到达高维度的神奇世界。仰望天空，无论是天高云淡，还是月朗星稀，我们也会感叹自身的渺小，在浩瀚的时间长河中，我们不过是匆匆的过客，犹如沧海一粟。

本章我们探讨的问题，究其实质是人类自身的问题。我们的理性思维是否存在边界？我们是否在沿着一个怪圈吞噬着自己？在人工智能突飞猛进发展时，这是每个人都会思考的问题。人类在试图用机器同构自己，用巨大的计算资源同构我们的大脑和理性思维。这就如同埃舍尔作品中的龙，它奋力想从二维世界中挣脱出来，它自己觉得已经把画纸的中部剪开一个缺口，并把尾巴伸了出来。这条龙一口咬住自己的尾巴，拼命想把自己拉到三维的世界中去。作为旁观者的我们，却明明白白知道所有的这一切，仍然在二维的纸张上。这条龙的努力是徒劳的。所有这一切，皆是虚妄、如雾亦如电、当作如是观。

一百年前的激烈讨论和哥德尔的天才证明，在今天依然有着现实意义。作为人类，我们心怀敬畏，敬畏自然，敬畏宇宙，敬畏我们祖先，也敬畏我们自身。



图 7.16: 埃舍尔《龙》



Part I

## 附录



## .1 加法交换律的证明

为了证明加法交换律 $a + b = b + a$ 。我们首先证明三个结论。第一个是说对于任何自然数都有：

$$0 + a = a \quad (1)$$

也就是说，加法左侧的零可以消去。首先看起始情况，当 $a = 0$ 时，根据加法定义的第一条规则有：

$$0 + 0 = 0$$

其次是递推情况，设 $0 + a = a$ ，我们要推出 $0 + a' = a'$ 。

$$\begin{aligned} 0 + a' &= (0 + a)' && \text{加法定义的规则二} \\ &= a' && \text{递推假设} \end{aligned}$$

接下来我们定义0的后继为1，并证明第二个重要结论：

$$a' = a + 1 \quad (2)$$

也就是说，任何自然数的后继，等于这个自然数加一。这是因为：

$$\begin{aligned} a' &= (a + 0)' && \text{加法定义的规则一} \\ &= a + 0' && \text{加法定义的规则二} \\ &= a + 1 && \text{定义0的后继是1} \end{aligned}$$

第三个要证明的结论是交换律的一个特例：

$$a + 1 = 1 + a \quad (3)$$

首先看 $a = 0$ 的起始情况。

$$\begin{aligned} 0 + 1 &= 1 && \text{刚证明的加法左侧零可消去} \\ &= 1 + 0 && \text{加法定义的第一条规则} \end{aligned}$$

然后是递推情况，设 $a + 1 = 1 + a$ 成立，我们要推出 $a' + 1 = 1 + a'$ 。

$$\begin{aligned} a' + 1 &= a' + 0' && 1 \text{是0的后继} \\ &= (a' + 0)' && \text{加法定义的第一条规则} \\ &= ((a + 1) + 0)' && \text{根据刚刚证明的结论二} \\ &= (a + 1)' && \text{加法定义的规则一} \\ &= (1 + a)' && \text{递推假设} \\ &= 1 + a' && \text{加法定义的规则二} \end{aligned}$$

有了这三个结论，我们就可以着手证明加法交换律了。我们首先证明 $b = 0$ 时交换律成立。根据加法定义的规则一，我们有 $a + 0 = a$ ；同时根据刚才证明的结论一，又有 $0 + a = a$ 。这就证明了 $a + 0 = 0 + a$ 。然后我们证明递推情况。假设 $a + b = b + a$ 成立，我们要推出 $a + b' = b' + a$ 。

$$\begin{aligned} a + b' &= (a + b)' && \text{根据加法定义的第二条规则} \\ &= (b + a)' && \text{递推假设} \\ &= b + a' && \text{加法定义的第二条规则} \\ &= b + a + 1 && \text{刚刚证明的结论二，即(2)} \\ &= b + 1 + a && \text{刚刚证明的结论三，即(3)} \\ &= (b + 1) + a && \text{第一章证明的加法结合律} \\ &= b' + a && \text{刚刚证明的结论三，即(3)} \end{aligned}$$

这样我们就使用皮亚诺公理，完整地证明了加法的交换律[10]。

## .2 倒水趣题完整程序

求得两个瓶子倒水的次数后，就可以生成一系列倒水的步骤，将这些步骤像表(2.2)一样输出。

```
— Populate the steps
water a b c = if x > 0 then pour a x b y
               else map swap $ pour b y a x
where
(x, y) = jars a b c

— Pour from a to b, fill a for x times, and empty b for y times.
pour a x b y = steps x y [(0, 0)]
where
steps 0 0 ps = reverse ps
steps x y ps@((a', b'):_) =
| a' == 0 = steps (x - 1) y ((a, b'):ps) — fill a
| b' == b = steps x (y + 1) ((a', 0):ps) — empty b
| otherwise = steps x y ((max (a' + b' - b) 0,
                           min (a' + b') b):ps) — a to b
```

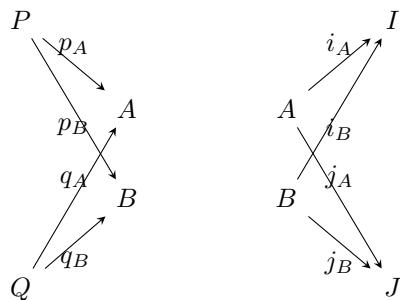
运行这一程序，输入water 9 4 6就可以得到最佳的倒水步骤：

`[(0,0),(9,0),(5,4),(5,0),(1,4),(1,0),(0,1),(9,1),(6,4),(6,0)]`

## .3 积和余积的唯一性

积和余积是唯一的么？下面的定理回答了这个问题：

**引理 .3.1.** 对于范畴 $\mathbf{C}$ 中的一对对象 $A$ 和 $B$ ，令下图中的对象和箭头



各自为一对

积              余积

的楔形。则

$P, Q$                $I, J$

是同构的楔形，存在唯一的箭头：

$$\begin{array}{ccc} P & & I \\ f \downarrow & \uparrow g & f \downarrow \\ Q & & J \end{array}$$

使得：

$$\begin{cases} p_A = q_A \circ f & p_B = q_B \circ f \\ q_A = p_A \circ g & q_B = p_B \circ g \end{cases} \quad \begin{cases} i_A = g \circ j_A & i_B = g \circ j_B \\ j_A = f \circ i_A & j_B = f \circ i_B \end{cases}$$

并且， $f$ 和 $g$ 是互逆的同构箭头对。

证明. 我们只证明左侧积的部分，右侧余积部分的证明与此类似。给定 $A, B$ ，对象 $Q$ 和两个箭头 $q_A, q_B$ 构成的楔形是一个积。我们把对象 $P$ 和一对箭头 $p_A, p_B$ 看成是另一个楔形。根据积的定义，存在唯一的媒介箭头 $f$ 满足：

$$p_A = q_A \circ f \quad \text{和} \quad p_B = q_B \circ f$$

交换 $P$ 和 $Q$ （另 $P$ 为积， $Q$ 为任意楔形），又可以得到：

$$q_A = p_A \circ g \quad \text{和} \quad q_B = p_B \circ g$$

这样就有：

$$\begin{cases} p_A \circ g \circ f = q_A \circ f = p_A \\ p_B \circ g \circ f = q_B \circ f = p_B \end{cases}$$

以及：

$$\begin{cases} q_A \circ f \circ g = p_A \circ g = q_A \\ q_B \circ f \circ g = p_B \circ g = q_B \end{cases}$$

因此：

$$g \circ f = id_P \quad f \circ g = id_Q$$

□

这一结论说明，如果两个对象存在积（或者余积）则它是唯一的。

## .4 集合的笛卡尔积和不相交并集构成积和余积的证明

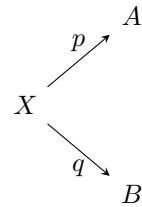
证明. 我们可以用构造法证明。首先证明左侧积的部分。 $A \times B$ 的笛卡尔积是来自两个集合的元素对的全部组合

$$\{(a, b) | a \in A, b \in B\}$$

我们定义两个特殊的箭头（函数）作为 $p_A$ 和 $p_B$

$$\begin{cases} fst(a, b) = a \\ snd(a, b) = b \end{cases}$$

考虑任意楔形



其中  $p(x) = a, q(x) = b, x \in X$ 。举个具体的例子， $X$  为  $\text{Int}$ ,  $A$  为  $\text{Int}$ ,  $B$  为  $\text{Bool}$ , 而两个函数  $p, q$  的定义为：

$$\begin{cases} p(x) = -x \\ q(x) = \text{even}(x) \end{cases}$$

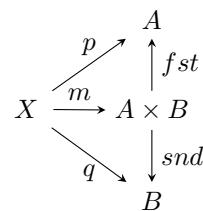
也就是说， $p$  对一个整数取相反数， $q$  判断其是否是偶数。我们定义函数  $X \xrightarrow{m} A \times B$  如下：

$$m(x) = (a, b)$$

具体到上面的例子，有：

$$m(x) = (-x, \text{even}(x))$$

这样，下面图中的箭头就可交换了：



不难验证：

$$\begin{aligned} (fst \circ m)(x) &= fst(m(x)) && \text{函数组合} \\ &= fst(a, b) && m \text{的定义} \\ &= a && fst \text{的定义} \\ &= p(x) && \text{反向用 } p \text{ 的定义} \end{aligned}$$

并且：

$$\begin{aligned} (snd \circ m)(x) &= snd(m(x)) && \text{函数组合} \\ &= snd(a, b) && m \text{的定义} \\ &= b && snd \text{的定义} \\ &= q(x) && \text{反向用 } q \text{ 的定义} \end{aligned}$$

具体到上面的例子，有：

$$\begin{cases} (fst \circ m)(x) = fst(-x, \text{even}(x)) = -x = p(x) \\ (snd \circ m)(x) = snd(-x, \text{even}(x)) = \text{even}(x) = q(x) \end{cases}$$

我们还需要证明 $m$ 是唯一的。假设存在另一个函数 $x \xrightarrow{h} A \times B$ , 也使得范畴图中的箭头可交换。即:

$$fst \circ h = p \quad \text{且} \quad snd \circ h = q$$

我们有:

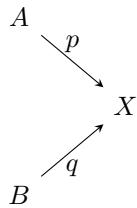
$$\begin{aligned} (a, b) &= (p(x), q(x)) && p, q \text{ 的定义} \\ &= ((fst \circ h)(x), (snd \circ h)(x)) && \text{可交换} \\ &= (fst h(x), snd h(x)) && \text{函数组合} \\ &= (fst(a, b), snd(a, b)) && \text{反向用 } fst, snd \text{ 的定义} \end{aligned}$$

所以 $h(x) = (a, b) = m(x)$ 。这样就证明了 $m$ 的唯一性。

接下来, 我们证明右侧的余积部分。不相交的并集 $A + B$ 中的元素分为两类。一类是来自 $A$ 中的元素 $(a, 0)$ , 另一类是来自 $B$ 中的元素 $(b, 1)$ 。为此, 我们定义两个特殊的箭头(函数)作为 $i_A, i_B$ :

$$\begin{cases} left(a) = (a, 0) \\ right(b) = (b, 1) \end{cases}$$

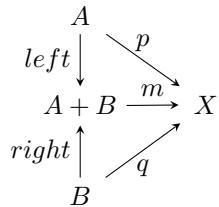
考虑任意集合 $X$ 构成的楔形:



我们定义箭头 $A + B \xrightarrow{m} X$ 如下:

$$\begin{cases} m(a, 0) = p(a) \\ m(b, 1) = q(b) \end{cases}$$

这样, 下图中的箭头就可交换了。



接下来要证明 $m$ 的唯一性, 也就是说 $m$ 是唯一使上图可交换的箭头。假设存在另一个箭头 $A + B \xrightarrow{h} X$ 也使得上图可交换。即满足:

$$h \circ left = p \quad \text{且} \quad h \circ right = q$$

我们任取 $a \in A, b \in B$ , 有:

$$\begin{cases} h(a, 0) = h(left(a)) = (h \circ left)(a) = p(a) = m(a, 0) \\ h(b, 1) = h(right(a)) = (h \circ right)(b) = q(b) = m(b, 1) \end{cases}$$

于是 $h = m$ , 这就证明了 $m$ 的唯一性。  $\square$

## 参 考 文 献

- [1] Wikipedia. “古代计数系统的历史”. [https://en.wikipedia.org/wiki/History\\_of\\_ancient\\_numeral\\_systems](https://en.wikipedia.org/wiki/History_of_ancient_numeral_systems)
- [2] [美]卡尔文·C·克劳森. “数学旅行家：漫游数王国”. 袁向东、袁钧译，上海教育出版社。ISBN: 7-5320-7883-3/G cdot 7972
- [3] Wikipedia. “古巴比伦数字”. [https://en.wikipedia.org/wiki/Babylonian\\_numerals](https://en.wikipedia.org/wiki/Babylonian_numerals)
- [4] [美] M·克莱因 著 李宏魁 译“数学：确定性的丧失”湖南科学技术出版社，2007年4月 ISBN: 978-7-5357-1857-0
- [5] [美]候世达“哥德尔、埃舍尔、巴赫——集异璧之大成”. 商务印书馆 1996. ISBN: 978-7-100-01323-9
- [6] Richard Bird, Oege de Moor. “Algebra of Programming”. University of Oxford, Prentice Hall Europe. 1997. ISBN: 0-13-507245-X.
- [7] 顾森“浴缸里的惊叹”. 人民邮电出版社. 2014, ISBN: 9787115355744
- [8] Stepanov and Rose. “数学与泛型编程”. 爱飞翔译，机械工业出版社。ISBN: 978-7-111-57658-7. 2017. pp147 - 148.
- [9] 韩雪涛“数学悖论与三次数学危机”. 人民邮电出版社. 2016, ISBN: 9787115430434
- [10] [美] 亚历山大 A·斯捷潘诺夫, 丹尼尔 E·罗斯著, 爱飞翔译. “数学与泛型编程：高效编程的奥秘”. 机械工业出版社. 2017, ISBN: 9787111576587
- [11] [古希腊] 欧几里得著, 兰纪正 朱恩宽 译, 梁宗巨 张毓新 徐伯谦 校订“几何原本”. 译林出版社. 2014, ISBN: 9787544750066
- [12] 韩雪涛“好的数学——“下金蛋”的数学问题”. 湖南科学技术出版社. 2009, ISBN: 9787535756725
- [13] Wikipedia “贝祖等式” [https://en.wikipedia.org/wiki/Bézout's\\_identity](https://en.wikipedia.org/wiki/Bézout's_identity)
- [14] 刘新宇“算法新解”人民邮电出版社. 2017, ISBN: 9787115440358
- [15] Wikipedia “艾伦·图灵” [https://en.wikipedia.org/wiki/Alan\\_Turing](https://en.wikipedia.org/wiki/Alan_Turing)
- [16] [法] 吉尔·多维克 著, 劳佳 译“计算进化史：改变数学的命运”. 人民邮电出版社. 2017, ISBN: 9787115447579
- [17] Simon L. Peyton Jones. “The implementation of functional programming language”. Prentice Hall. 1987, ISBN: 013453333X
- [18] 韩雪涛“好的数学——方程的故事”. 湖南科学技术出版社. 2012, ISBN: 9787535770066

- [19] Wikipedia “伽罗瓦理论”. [https://en.wikipedia.org/wiki/Galois\\_theory](https://en.wikipedia.org/wiki/Galois_theory)
- [20] Wikipedia “埃瓦里斯特·伽罗瓦”. [https://en.wikipedia.org/wiki/Évariste\\_Galois](https://en.wikipedia.org/wiki/Évariste_Galois)
- [21] Wikipedia “魔方群”. [https://en.wikipedia.org/wiki/Rubik's\\_Cube\\_group](https://en.wikipedia.org/wiki/Rubik's_Cube_group)
- [22] 张禾瑞“近世代数基础”. 高等教育出版社. 1978, ISBN: 9787040012224
- [23] M.A. Armstrong “群与对称（影印版）”. Springer. 1988. ISBN: 0387966757.
- [24] Wikipedia “约瑟夫·拉格朗日”. [https://en.wikipedia.org/wiki/Joseph-Louis\\_Lagrange](https://en.wikipedia.org/wiki/Joseph-Louis_Lagrange)
- [25] Wikipedia “费马小定理的证明”. [https://en.wikipedia.org/wiki/Proofs\\_of\\_Fermat's\\_little\\_theorem](https://en.wikipedia.org/wiki/Proofs_of_Fermat's_little_theorem)
- [26] Wikipedia “莱昂哈德·欧拉”. [https://en.wikipedia.org/wiki/Leonhard\\_Euler](https://en.wikipedia.org/wiki/Leonhard_Euler)
- [27] Wikipedia “卡米歇尔数”. [https://en.wikipedia.org/wiki/Carmichael\\_number](https://en.wikipedia.org/wiki/Carmichael_number)
- [28] Sanjoy Dsgupta, Christos Papadimitriou, Umesh Vazirani. 钱枫 邹恒明 注释. “算法概论（注释版）”. 机械工业出版社. 2009年1月. ISBN: 9787111253617
- [29] Wikipedia “米勒——拉宾素数检验”. [https://en.wikipedia.org/wiki/Miller-Rabin\\_primality\\_test](https://en.wikipedia.org/wiki/Miller-Rabin_primality_test)
- [30] Wikipedia “埃米·诺特”. [https://en.wikipedia.org/wiki/Emmy\\_Noether](https://en.wikipedia.org/wiki/Emmy_Noether)
- [31] 章璞. “伽罗瓦理论：天才的激情”. 高等教育出版社. 2013年5月. ISBN: 9787040372526
- [32] John Stillwell. “Galois Theory for Beginners”. The American Mathematical Monthly, Vol. 101, No. 1 (Jan., 1994), pp. 22-2
- [33] Dan Goodman. “An Introduction to Galois Theory”. <https://nrich.maths.org/1422>
- [34] Michael Artin. “代数（英文版，第二版）”. 机械工业出版社. 2011年12月. ISBN: 9787111367017
- [35] [法]让·迪厄多内 著，沈用欢 译“当代数学，为了人类心智的荣耀”. 上海教育出版社. 2000年3月. ISBN: 7532063062
- [36] Haskell Wiki. “Monad”. <https://wiki.haskell.org/Monad>
- [37] Wikipedia. “塞缪尔·艾伦伯格”. [https://en.wikipedia.org/wiki/Samuel\\_Eilenberg](https://en.wikipedia.org/wiki/Samuel_Eilenberg)
- [38] Wikipedia. “桑德斯·麦克兰恩”. [https://en.wikipedia.org/wiki/Saunders\\_Mac\\_Lane](https://en.wikipedia.org/wiki/Saunders_Mac_Lane)
- [39] Harold Simmons. “An introduction to Category Theory”. Cambridge University Press; 1 edition, 2011. ISBN: 9780521283045
- [40] Wikipedia. “Tony Hoare”. [https://en.wikipedia.org/wiki/Tony\\_Hoare](https://en.wikipedia.org/wiki/Tony_Hoare)
- [41] Wadler Philip. “Theorems for free!”. Functional Programming Languages and Computer Architecture, pp. 347-359. Asociation for Computing Machinery. 1989.
- [42] Bartosz Milewski. “Category Theory for Programmers”. <https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/>
- [43] Peter Smith. “Category Theory - A Gentle Introduction”. [http://www.academia.edu/21694792/A\\_Gentle\\_Introduction\\_to\\_Category\\_Theory\\_Jan\\_2018\\_version\\_](http://www.academia.edu/21694792/A_Gentle_Introduction_to_Category_Theory_Jan_2018_version_)

- [44] Wikipedia. “Exponential Object”. [https://en.wikipedia.org/wiki/Exponential\\_object](https://en.wikipedia.org/wiki/Exponential_object)
- [45] Manes, E. G. and Arbib, M. A. “Algebraic Approaches to Program Semantics”. Texts and Monographs in Computer Science. Springer-Verlag. 1986.
- [46] Lambek, J. “A fixpoint theorem for complete categories”. Mathematische Zeischrift, 103, pp.151-161. 1968.
- [47] Wikibooks. “Haskell/Foldable”. <https://en.wikibooks.org/wiki/Haskell/Foldable>
- [48] Mac Lane. “Categories for working mathematicians”. Springer-Verlag. 1998. ISBN: 0387984038.
- [49] Andrew Gill, John Launchbury, Simon L. Peyton Jones. “A Short Cut to Deforestation”. Functional programming languages and computer architecture. pp. 223-232. 1993.
- [50] Richard Bird. “Pearls of Functional Algorithm Design”. Cambridge University Press; 1 edition November 1, 2010. ISBN: 978-0521513388.
- [51] Ralf Hinze, Thomas Harper, Daniel W. H. James. “Theory and Practice of Fusion”. 2010, 22nd international symposium of IFL (Implementation and application of functional languages). pp.19-37.
- [52] Akihiko Takano, Erik Meijer. “Shortcut Deforestation in Calculational Form”. Functional programming languages and computer architecture. pp. 306-313. 1995.
- [53] Donald Knuth. “The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees.” Reading, MA: Addison-Wesley. ISBN: 978-0321637130. 2006.
- [54] [法] 让-皮埃尔·卢米涅, 马克·拉雪茨-雷 著, 孙展 译. 从无穷开始——科学的困惑与疆界. 人民邮电出版社. 2018. ISBN: 9787115479198
- [55] [日] 野口哲也 著, 刘慧 韩丽红 译. 数学原来可以这样学. 湖南人民出版社. 2014. ISBN: 9787556100897
- [56] Wikipedia. “Googol”. <https://en.wikipedia.org/wiki/Googol>
- [57] Wikipedia. “Zeno’s Paradoxes”. [https://en.wikipedia.org/wiki/Zeno%27s\\_paradoxes](https://en.wikipedia.org/wiki/Zeno%27s_paradoxes)
- [58] 张锦文, 王雪生“连续统假设”. 世界数学名题欣赏丛书。辽宁教育出版社 1988. ISBN: 7-5382-0436-9/G.445
- [59] [法]彭加勒著, 李醒民译“科学与假设”商务印书馆. 2006. ISBN: 978-7-100-04796-8
- [60] Leon A. Gatys, Alexander S. Ecker, Matthias Bethge. “A Neural Algorithm of Artistic Style.” 2015. arXiv:1508.06576 [cs.CV] IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2017.
- [61] 顾森《思考的乐趣——Matrix67数学笔记》人民邮电出版社, 2012年, ISBN: 9787115275868
- [62] Harold Abelson, Gerald Jay Sussman, Julie Sussman 著 裴宗燕译“计算机程序的构造和解释（原书第二版）”. 北京机械工业出版社 2004年 ISBN: 7-111-13510-5
- [63] [美] M·克莱因 著 李宏魁 译“数学：确定性的丧失”湖南科学技术出版社, 2007年4月 ISBN: 978-7-5357-1857-0
- [64] [法]彭加勒著, 李醒民译“科学的价值”商务印书馆. 2010 ISBN: 978-7-100-07045-4

# Index

- alpha变换, 35
- beta变换, 35
- eta变换, 36
- lambda抽象, 34
- lambda演算, 33
- RSA算法, 73
- Y组合子, 38
- 不动点, 38
- 不变子群, 65
- 不可公度, 31
- 丘奇-罗瑟定理, 37
- 交换环, 76
- 代数基本定理, 47
- 伽罗瓦基本定理, 81
- 伽罗瓦群, 81
- 伽罗瓦 (Galois) , 48
- 克里化 (Currying) , 33
- 函子
  - id函子, 96
  - Maybe函子, 96
  - 可能函子, 96
  - 常函子, 96
  - 恒等函子, 96
- 函数组合, 13
- 分裂域, 79
- 列表的叠加, 16
- 列表连接的结合律, 16
- 加法结合律, 11
- 勾股定理, 23
- 半环, 77
- 半群 (semigroup) , 53
- 卡尔丹 (Gerolamo Cardano) , 47
- 变换群 (transform group) , 57
- 右陪集 (right coset, 63
- 同态映射 (homomorphism) , 55
- 同构 (isomorphism) , 56
- 商群, 65
- 图灵, 32
- 域, 78
- 堆 (heap) , 52
- 多项式函子, 124
- 子群, 62
- 对称群 (symmetric group) , 58
- 左陪集, 64
- 希帕索思 (Hippasus of Metapontum) , 30
- 么半群 (monoid) , 52
- 归纳公理 (Axiom of induction) , 8
- 形数 (figurate number) , 21
- 循环群 (cyclic group) , 60
- 扩域, 79
- 扩展欧几里得算法, 26
- 拉格朗日, 66
- 拉格朗日定理, 67
- 整环, 77
- 斐波那契, 13
- 斐波那契数列, 14
- 斜堆 (skew heap) , 52
- 最大公度, 23
- 核, 65
- 根域, 79
- 欧几里得, 24
- 欧几里得算法, 24
- 欧拉, 71
- 欧拉函数, 70
- 欧拉定理, 70
- 正规子群, 65
- 毕达哥拉斯, 21
- 毕达哥拉斯定理, 23
- 环, 76
- 皮亚诺公理 (Peano Axioms) , 8
- 置换群 (permutation group) , 58
- 群元素的阶, 54
- 群的定义, 50

自同构, 80  
自同构 (automorphism) , 56  
自然数的乘法, 10  
自然数的加法, 10  
自然数的叠加, 10  
诺特 (Emmy Noether) , 75  
贝祖等式, 26  
费马小定理, 68  
费马 (Pierre de Fermat) , 69  
  
选择箭头, 116  
配对堆 (pairing heap) , 53  
链表, 15  
除环, 77  
零因子, 77  
高斯, 47