函数式程序设计入门

简明讲义

邓玉欣

2021 年 5 月 6 日

前言

提到计算机编程语言，很多人只听说过流行的语言例如 C，C++，Java, Python 等。目前函数式编程语言的用户数较少而且大部分是用于学术研究而非商业，因此普及程度远远不能与流行语言相比。但是，的确有一些大型商业项目的开发基于函数式编程语言[1]。例如，Jane Street 是一家从事金融量化交易的跨国公司，拥有 400 多名 OCaml 程序员和超过 1 千 5 百万行 OCaml 代码，以支撑每天数十亿美元的交易。另一方面，近年来形式化验证方法逐渐受到关注，函数式语言被广泛用于开发编译器、程序分析器、验证器以及定理证明器，以帮助提高软硬件系统的可信程度。

在这样的背景下，作者认为有必要在国内开展函数式程序设计相关的教学工作。深入讲解函数式编程需要比较多的学时，而目前不少大学都在主推通识教育，希望压缩专业课程的学时数，因此，一种可能的办法是开设一门 1 学分的本科选修课程，或者是在一门 3 学分的编程语言课程中预留 1 学分给函数数式编程教学模块。本讲义编写的目的是服务于这类的函数式程序设计入门课，重点是让学生有机会了解函数式程序设计的基本思想和概念，让学生了解、欣赏、进而喜欢函数式编程。至于函数式语言背后极其丰富的语义理论，更适合设置在软件理论专业的研究生课程，因此不在本讲义予以讨论。例如，在介绍 λ 演算时，我们会提到邱奇-罗索定理，但具体证明并不展开讲。

在内容选取上，本讲义只涉及 λ-演算，Coq 和 OCaml。毫无疑问，λ-演算是理解函数式编程语言的基础和出发点，因此在第一章我们介绍不带类型的 λ-演算和简单类型的 λ-演算，主要讨论语法和 β-规约语义。虽然 λ-演算适合理解函数式编程的一些核心思想，比如数据即函数，但是它的语法构造比较

[1] https://ocaml.org/learn/companies.html

原始，即使表示一个数字都要写很长的 λ 项，可读性低，更不用提编写程序。Coq 是离 λ-演算比较接近但又能用于编写一些可读性较好的计算函数的编程语言，因此在第二章我们介绍 Coq，重点是从函数式编程的角度展开讨论，内容涉及自然数函数、列表、多态数据结构、高阶函数以及柯里-霍华德关联。作者认为 Coq 是来用于讲授归纳定义和归纳证明思想的出色工具。虽然 Coq 的长处在于定理证明，但是深入讲解需要很大篇幅，因此最好留给专门的书籍，不适合在入门课程的讲义中展开。为满足适合逻辑证明的需要，Coq 只接受可终止的函数。这么强的要求决定它不可能用于日常编程。因此，在第三章我们介绍一门通用的编程语言 OCaml，除了基本的程序设计概念，我们还会讨论函子和 Monad 这样比较高级的特征。关于这三部分内容，如果读者希望了解更多，都有专门的书籍进行深入讲解，例如，λ-演算的经典教材是 Barendregt 的 [1]，Coq 的著名教材是 Pierce 等人的 [9]，OCaml 的优秀参考书是 Minsky 等人的 [6] 和陈刚老师的 [3]。讲义中选取了一些练习题，希望通过做练习加强对基本概念的理解。第四章提供了部分习题的参考答案，以方便感兴趣的读者自行学习。本讲义可作为高等院校计算机科学或软件工程专业的本科教学参考书。

由于作者水平有限，讲义中的错误和不足之处在所难免，欢迎读者及时指正并发送邮件至 yxdeng@sei.ecnu.edu.cn。

邓玉欣

2021 年 5 月　　上海

目录

第 1 章　λ-演算

1.1　λ-演算的起源

在二十世纪三十年代，可计算性（computability）问题得到一些数学家和逻辑学家的关注。通俗而言，一个自然数集合上的函数 $f : \mathbb{N} \to \mathbb{N}$ 称为可计算的，是指对任何自然数 n 我们可以给一位受过良好数学训练的人足够用的纸和笔，不管花费多长时间，最终总能手动算出函数 f 在 n 上的值 $f(n)$。为了严格地刻画可计算函数，前面这个非形式化（informal）的定义显然不方便，因此需要对可计算性给出形式化（formal）的定义。当时有三位学者在这方面作出尝试，他们是阿兰 · 图灵（Alan Turing），库特 · 哥德尔（Kurt Gödel）和阿隆佐 · 邱奇（Alonzo Church）。

- 图灵提出一种抽象的理想化的计算模型，现在我们通称图灵机（Turing machine）。他假定一个函数 f 在直觉意义上是可计算的当且仅当有一个图灵机可计算出该函数在任何给定自然数 n 上的值 $f(n)$。

- 哥德尔定义了一类一般递归函数（general recursive functions），由常函数、后继函数等基本函数在函数复合和递归等算子组合下形成。他假定一个函数在直觉意义上是可计算的当且仅当它是一般递归的。

- 邱奇定义了一个称为λ-演算（λ-calculus）的形式化语言并假定一个函数在直觉意义上是可计算的当且仅当它可用一个 λ 表达式写出来。

后来邱奇等人证明了上面三个模型是互相等价的，即它们定义的是同一类（class）可计算函数。至于它们是否等价于直觉意义上的可计算函数类，则是一个没法回答的问题，因为我们没有一个描述直觉意义上的可计算函数类。

著名的邱奇-图灵论题（Church-Turing thesis）断言上面三个模型表达的恰好正是直觉意义上的可计算函数类。

下面我们主要介绍 λ-演算。现在我们经常用不带类型的 λ-演算（untyped λ-calculus）来指代邱奇当初提出的形式化语言。后来又出现这个语言的各种更精细的版本，称为带类型的 λ-演算（typed λ-calculi）[2]。

1.2 不带类型的 λ-演算

既然是形式化语言，不带类型的 λ-演算的语法和语义可以形式化地给出。语法规定这个语言中的表达式，称为λ 项（λ-terms），是如何被构造出来的，而语义则规定 λ 项演化的方式。

1.2.1 语法

不带类型的 λ-演算中的表达式，也称为 λ 项，可以通过简单的规则构造出来。

定义 1.1. 假设给定一个可数无穷的变量集合 $\mathcal{V}$，其中的元素用 x，y，z 等表示。我们用如下巴克斯-诺尔范式（*Backus-Naur Form*，简称 *BNF*）来生成 λ 项：

$$M,\ N\ ::=\ x \mid (MN) \mid (\lambda x.M)$$

上述巴克斯-诺尔范式实际上说明我们有三条规则来生成 λ 项。换句话说，所有 λ 项的集合，记为 Λ，是由下面三条规则生成的最小集合：

1. 如果 $x \in \mathcal{V}$，那么 $x \in \Lambda$；
2. 如果 $M, N \in \Lambda$，那么 $(MN) \in \Lambda$；
3. 如果 $x \in \mathcal{V}$ 且 $M \in \Lambda$，那么 $(\lambda x.M) \in \Lambda$。

这些规则会产生三种形式的 λ 项，依次称为变量（variable），作用（application），以及λ 抽象（lambda abstraction）。比如，下面是三个例子展示就是这三种类型的 λ 项：

$$y \qquad (\lambda x.(xx))(\lambda y.(yy)) \qquad (\lambda f.(\lambda x.x))$$

注 1. 为方便读者记住上面的语法规则，我们不妨把每个 λ 项看成是一个函数，把输入值变换成输出值。变量 x 是一个基本的 λ 项，相当于一个常函数，不管输入是什么，总是输出当前 x 的值；项 (MN) 表示把函数 M 作用到它的参数 N 上；项 $(\lambda x.M)$ 表示构造一个新函数，对于输入 x，输出函数体 M 表示的值。需要注意的是，这种直观理解实际上是不准确的，比如在项 (xx) 中，一个函数 x 的参数是其自身，与常规直觉不符合，但这种直觉在很大程度上有利于理解抽象的符合。

需要注意的是，在定义 1.1中我们强制性地加入括号以便把一个项和它的子项严格区分开来。为了少些一些括号但不引入歧义，我们采用下面的约定（convention）：

- 忽略最外层的括号。比如，将 $(\lambda x.M)$ 写成 $\lambda x.M$；
- λ 项的作用满足左结合性，即 $MNOP$ 等同于 $((MN)O)P$；
- λ 抽象的主体部分，即点号右边的部分，包含右边尽可能多的项。例如，$\lambda x.MN$ 表示 $\lambda x.(MN)$ 而不是 $(\lambda x.M)N$；
- 当有连续多个 λ 抽象时，只写最左边的 λ 符号。例如，把 $\lambda f.\lambda x.f(fx)$ 写成 $\lambda fx.f(fx)$。

练习 1.1. *1.* 根据上面的约定，尽可能减少下面这些 λ 项中的括号，但不改变这些项的结构：

(a) $\lambda f.(\lambda x.x)$

(b) $\lambda f.(\lambda x.(fx))$

(c) $x(y(\lambda z.z))$

2. 尽量补全下面这些 λ 项中的括号，但不改变这些项的结构：

(a) $\lambda xy.y$

(b) $\lambda ab.(\lambda ab.b)$

(c) $\lambda nmfx.nf(mfx)$

1.2.2 α-等价

之前我们说过，λ 演算是一种形式化语言。我们可以粗糙地认为，用这种语言写出来的句子应该是给计算机识别的。对机器而言，$\lambda x.x$ 和 $\lambda y.y$ 是两个不同的字符串。但根据注 1的解释，我们希望把这两个 λ 项看成同一个函数，对输入的值不做任何改变直接输出，即恒等函数。这种想法其实不陌生，在数学上我们会写 $f(x) = x$ 或者 $f(y) = y$ 来表示恒等函数，而且不假思索地认为这两种写法没有任何区别。为了把 $\lambda x.x$ 和 $\lambda y.y$ 这样的项等同起来，在本节我们引入 α-等价的概念。在此之前，我们需要介绍自由变量和受限变量的概念。

定义 1.2. 对形如 $\lambda x.M$ 的项，我们称 λx 为一个绑定子（*binder*），子项 N 为这个绑定子的范围（*scope*）。变量 x 的一次出现（*occurrence*）如果是在 N 中，则称这次出现为受限的（*bound*），否则称为自由的（*free*）。

例 1.1. 在项 $(\lambda x.yx)(\lambda y.zy)$ 中 y 的第一次出现是自由的，第二次出现是受限的；变量 x 和 z 分别出现一次，前者是受限，后者是自由的。

下面我们形式化地定义一个项 M 中有自由出现的变量的集合，记为 $FV(M)$。

$$\begin{aligned} FV(x) &\stackrel{def}{=} \{x\} \\ FV(MN) &\stackrel{def}{=} FV(M) \cup FV(N) \\ FV(\lambda x.M) &\stackrel{def}{=} FV(M) \backslash \{x\} \end{aligned}$$

以上对函数 $FV(\cdot)$ 的定义是一种典型的归纳定义，称为结构归纳。在同一行中，如果 $FV(M)$ 出现在定义符号 $\stackrel{def}{=}$ 左边，某个 $FV(N)$ 出现在右边，那么 N 一定是 M 的子项，即 N 的结构比 M 简单，以确保 $FV(\cdot)$ 是良定义的（well defined）。

练习 1.2. 令 $V(M)$ 为 λ 项 M 中出现的所有变量的集合。利用结构归纳的方式定义函数 $V(\cdot)$。

有时候我们希望对一个项 M 中的变量换名字，或者说重命名（rename）一个变量。我们用记号 $M\{y/x\}$ 来表示把项 M 中的所有变量 x 换成变量 y

$$(R_r)\quad \frac{}{M = M} \qquad (R_{ap})\quad \frac{M = M' \qquad N = N'}{MN = M'N'}$$

$$(R_s)\quad \frac{M = N}{N = M} \qquad (R_{ab})\quad \frac{M = M'}{\lambda x.M = \lambda x.M'}$$

$$(R_t)\quad \frac{M = N \qquad N = P}{M = P} \qquad (R_\alpha)\quad \frac{y \notin V(M)}{\lambda x.M = \lambda y.(M\{y/x\})}$$

表 1.1 α-等价的定义规则

之后得到的项。同样，我们用结构归纳的方式来定义这个重命名函数。

$$\begin{aligned}
x\{y/x\} &\stackrel{def}{=} y \\
z\{y/x\} &\stackrel{def}{=} x \qquad 若x \neq z \\
(MN)\{y/x\} &\stackrel{def}{=} (M\{y/x\})(N\{y/x\}) \\
(\lambda x.M)\{y/x\} &\stackrel{def}{=} \lambda y.(M\{y/x\}) \\
(\lambda z.M)\{y/x\} &\stackrel{def}{=} \lambda z.(M\{y/x\}) \qquad 若x \neq z
\end{aligned}$$

从上面的定义可以看出，在重命名的时候我们把所有 x 的出现替换为 y 的出现，不管这些出现是自由还是受限的。

有了前面这些准备工作，我们现在可以给出α-等价（α-equivalence）的定义。

定义 1.3. *在 λ 项上满足表 1.1中所有规则的最小二元关系称为 α-等价，记为 $=_\alpha$。*

在表 1.1中共有六条规则，其中（R_r）、（R_s）、（R_t）分别表示自反、对称、传递性质，满足这三条规则的二元关系是一个等价关系；（R_{ap}）和（R_{ab}）分别表示在作用和 λ 抽象两个语法构造下的封闭性，在 λ 项上满足这两条规则的二元关系具有同余性（congruence）。最核心的规则是（R_α），其意义是说我们可以把一个受限变量 x 重命名为任何一个目前没出现过的新变量 y。定义 1.3说明 α-等价是在 λ 项上满足（R_α）的等价且同余的最小二元关系。

练习 1.3. *证明如下性质：任何一个 λ 项 M 都 α-等价于另一个项 M'，使得 M' 中任何一个受限变量都与其它的受限或自由变量不重名。*

从现在开始，我们将不再区分 α-等价的 λ 项，就像我们不希望区分数学表达式 $\int xdx$ 和 $\int ydy$ 一样。

1.2.3 替换

与重命名相近但略复杂的一个概念是替换（substitution），它允许我们把一个变量替换为一个 λ 项。我们用记号 $M[N/x]$ 来表示把项 M 中的 x 替换为 N 的结果。不过这样的替换需要满足下面两个条件：

1. 只能把自由变量替换为其它项。上一节介绍的 α-等价允许把受限变量重命名为任何新的变量。如果对受限变量替换，会破坏 α-等价。例如，$(\lambda x.x)[yy/x]$ 应该等于 $(\lambda x.x)$ 而不是 $(\lambda x.yy)$，否则，本来 $(\lambda x.x)$ 与 $(\lambda z.z)$ 是 α-等价的，替换之后将两者将不等价，这不是我们期望的结果。

2. 项 N 中的自由变量不能被 M 中的绑定子捕获（capture）。例如，假设我们有 $M \stackrel{def}{=} \lambda x.xy$，$N \stackrel{def}{=} yx$。注意变量 x 在 N 中是自由的但在 M 中是受限的。如果我们允许如下替换：

$$M[N/y] = (\lambda x.xy)[yx/y] = \lambda x.x(yx),$$

则把原来 N 中自由的 x 捕获了而变成和 M 中受限的 x 视为相同的变量。这不是我们期望的结果，因为后者可以重命名为任何一个新的变量但前者不可以。遇到这种情况，正确的做法应该是在替换之前就对 M 中的受限变量 x 重命名：

$$M[N/y] = (\lambda x'.x'y)[yx/y] = \lambda x'.x'(yx)$$

为满足上面第二个条件，我们有时需要把一个受限变量重命名为一个新鲜的（在目前考虑的项中未使用过的）变量。在定义1.1中，我们假定变量集合 $\mathcal{V}$ 是可数无穷的，这样可以保证我们在任何时候都能取到新变量。

定义 1.4. 把项 M 中自由出现的 x 改写成项 N 的免捕获（*capture-avoiding*）

替换定义如下：

$$
\begin{array}{rcll}
x[N/x] & \stackrel{def}{=} & N & \\
y[N/x] & \stackrel{def}{=} & y & 若 x \neq y \\
(MP)[N/x] & \stackrel{def}{=} & (M[N/x])(P[N/x]) & \\
(\lambda x.M)[N/x] & \stackrel{def}{=} & \lambda x.M & \\
(\lambda y.M)[N/x] & \stackrel{def}{=} & \lambda y.(M[N/x]) & 若\ x \neq y\ 且\ y \notin FV(N) \\
(\lambda y.M)[N/x] & \stackrel{def}{=} & \lambda y'.(M\{y'/y\}[N/x]) & 若\ x \neq y,\ y \in FV(N) \\
 & & & 且\ y'\ 是新鲜的
\end{array}
$$

在这个定义的最后一行，我们只说 y' 是新鲜的（fresh），但没有明确具体如何选择这样的变量。不失一般性，我们不妨假设集合 $\mathcal{V}$ 中的元素有一个枚举：$y_1, y_2, \cdots$，每次需要一个新鲜的变量，我们就选一个下标最小并且在 M 和 N 中未使用过的那个变量 y_i。

练习 1.4. 对任何变量 x，任何三个项 M，N 和 P，如果 $M =_\alpha P$ 那么 $M[N/x] =_\alpha P[N/x]$。

1.2.4 β-规约

给定两个简单的算术表达式，比如 $1+2\times 3$ 和 $3+2+2$，很容易看出它们表示相同的结果。究其原因，原来我们可以运用乘法和加法的等式公理做递等式计算，这个过程是对数学表达式的化简，最后两个等式都可以化简到数字 7。

$$
\begin{array}{rlrl}
 & 1+2\times 3 & & 3+2+2 \\
= & 1+6 & = & 5+2 \\
= & 7 & = & 7
\end{array}
$$

在上面的递等式计算过程中，我们用到好几条等式公理，比如 $2\times 3 = 6$，$1+6=7$ 等。

给定 λ-演算中的两个表达式，即两个 λ 项 M 和 N，我们也想对它们进行某种转换或者化简，看最后能否变成相同的项。在 λ-演算中，我们只需要一条化简规则，称为 β-规约，其直观想法就是把函数的参数代入函数体。一个形如 $(\lambda x.M)N$ 的项称为一个*β-redex*，它把一个 λ 抽象 $(\lambda x.M)$ 作用到另

一个项 N 上。我们把它规约（reduce）到 $M[N/x]$，后面这个项称为 *reduct*。对 λ 项的这一步转换，我们写成下面的形式：

$$(\lambda x.M)N \longrightarrow_\beta M[N/x]$$

对 λ 项规约的过程就是不断寻找 redex，换成它的 reduct 的过程。如果一个项中没有 redex，我们称这个项是一个β 范式（β-normal form）。

例 1.2. 我们对项 $(\lambda x.x((\lambda y.y)z))(\lambda x.y)$ 进行 β-规约。在每一步中，我们对即将规约的 *redex* 用下划线标示出来。

$$\begin{aligned}\underline{(\lambda x.x((\lambda y.y)z))(\lambda x.y)} &\longrightarrow_\beta (\lambda x.y)(\underline{(\lambda y.y)z}) \\ &\longrightarrow_\beta \underline{(\lambda x.y)z} \\ &\longrightarrow_\beta y\end{aligned}$$

最后这个项 y 没有 *redex*，是一个范式。实际上在第二步我们可以选择另外一个 *redex* 进行规约：

$$\begin{aligned}\underline{(\lambda x.x((\lambda y.y)z))(\lambda x.y)} &\longrightarrow_\beta \underline{(\lambda x.y)((\lambda y.y)z)} \\ &\longrightarrow_\beta y\end{aligned}$$

通过上面的例子我们可以观察到如下几点：

- 对一个 redex 进行规约可以创建出新的 redex；
- 对一个 redex 进行规约可以删除其它的 redex；
- 把一个项规约为一个范式的步骤数目可以随规约顺序的不同而变化。

从上面的例子中我们也注意到，不同的规约顺序虽然经历不同长度的规约路径，但最后都到达同一个范式 y。实际上，这并非偶然，而是一个一般结论，之后我们会介绍。

存在一些项，从它们出发进行规约，最后可能不能到达任何范式。例如，

$$(\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta (\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta \cdots$$

练习 1.5. 给出一个项 M，每规约一步就到达一个更复杂的项，从而无法最终到达一个范式。

下面我们给出 β-规约的形式化定义。

定义 1.5. 单步 β-规约，记作 $\longrightarrow_\beta$，是满足下列规则的最小关系：

$$(B_\beta)\quad \frac{}{(\lambda x.M)N \longrightarrow_\beta M[N/x]} \qquad (B_{ap1})\quad \frac{M \longrightarrow_\beta M'}{MN \longrightarrow_\beta M'N}$$

$$(B_{ab})\quad \frac{M \longrightarrow_\beta M'}{\lambda x.M \longrightarrow_\beta \lambda x.M'} \qquad (B_{ap2})\quad \frac{N \longrightarrow_\beta N'}{MN \longrightarrow_\beta MN'}$$

可以看出，存在单步规约 $M \longrightarrow_\beta M'$ 当且仅当 M' 是通过规约 M 中的一个 redex 得到的。我们用 $\longrightarrow_\beta^*$ 表示 $\longrightarrow_\beta$ 的自反传递闭包，即包含 $\longrightarrow_\beta$ 的自反和传递的最小关系。如果 $M \longrightarrow_\beta^* M'$，那么 M' 是从 M 出发通过零步或多步 β-规约得到的项。

定义 1.6. 令 $\longleftarrow_\beta$ 为 $\longrightarrow_\beta$ 的逆关系，即 $M' \longleftarrow_\beta M$ 当且仅当 $M \longrightarrow_\beta M'$。我们定义 $=_\beta$ 为 $\longrightarrow_\beta$ 及其逆关系的自反传递闭包，即 $(\longrightarrow_\beta \cup \longleftarrow_\beta)^*$。因此，$M =_\beta N$ 表示可以从 M 出发通过反复地正向或逆向使用 β-规约而得到 N。

例 1.3. 令 $M \stackrel{def}{=} (\lambda x.x((\lambda y.y)z))(\lambda x.y)$ 和 $N \stackrel{def}{=} (\lambda a.\lambda b.b)xy$。由例 *1.2*我们知道 $M \longrightarrow_\beta^* y$。从项 N 出发我们有 $N \longrightarrow_\beta (\lambda b.b)y \longrightarrow_\beta y$。于是，

$$M \longrightarrow_\beta^* y \longleftarrow_\beta^* N$$

也就是说，我们有 $M =_\beta N$。

1.2.5 表达能力

虽然 λ-演算的语法和规约语义非常简单，但是我们可以用它来编码各种数据，比如布尔值和自然数，以及操纵这些数据的常见运算。事实上，我们并不特意区分数据与数据上的运算，因为它们都可用 λ 项来表达。为了给一些项命名，下面我们会用粗体字母作为名字。

布尔值 我们定义两个 λ-项 **T** 和 **F** 来编码布尔类型中的真和假两个值：

$$\mathbf{T} \stackrel{def}{=} \lambda xy.x$$
$$\mathbf{F} \stackrel{def}{=} \lambda xy.y$$

有了布尔值之后，我们接下去定义对布尔值的运算：与、或、非。为表示“与”运算，我们令 $\mathbf{and} \stackrel{def}{=} \lambda ab.aba$，则下面四条规约性质成立：

$$\begin{aligned} \mathbf{and\ TT} &\longrightarrow_\beta^* \mathbf{T} \\ \mathbf{and\ TF} &\longrightarrow_\beta^* \mathbf{F} \\ \mathbf{and\ FT} &\longrightarrow_\beta^* \mathbf{F} \\ \mathbf{and\ FF} &\longrightarrow_\beta^* \mathbf{F} \end{aligned}$$

例 1.4. 作为一个例子，我们验证第一条性质，其它三条性质类似。

$$\begin{aligned} \boldsymbol{and\ TT} &\equiv (\lambda ab.aba)(\lambda xy.x)(\lambda xy.x) \\ &\longrightarrow_\beta (\lambda b.(\lambda xy.x)b(\lambda xy.x))(\lambda xy.x) \\ &\longrightarrow_\beta (\lambda xy.x)(\lambda xy.x)(\lambda xy.x) \\ &\longrightarrow_\beta (\lambda y.(\lambda xy.x))(\lambda xy.x) \\ &\longrightarrow_\beta (\lambda xy.x) \\ &\equiv \boldsymbol{T} \end{aligned}$$

在上面推导过程中，我们用符号 $\equiv$ 表示语法等价（*syntactic equivalence*），即如果按定义展开，该符号左右两边的表达式语法上一模一样。

由于 **T** 和 **F** 是范式，我们可以说 **and TT** 求值（evaluate）到 **T**。针对上面定义的 **T** 和 **F**，可见 **and** 能完成所需要的运算，因此是对“与”运算的一个有效编码。这里需要注意两点：

- 项 **and** 构成对“与”运算有效编码的前提是“真”、“假”值分别用 **T**、**F** 编码。如果布尔值用其它项 M、N 来表示，我们不能保证 **and** MN 求值到合适的项。
- 即使对于上面定义的 **T** 和 **F**，“与”运算的编码也不唯一。比如项 $\lambda ab.bab$ 也符合要求。

练习 1.6. 我们把“否定”、“或”、“异或”运算定义成下面的项：

$$\begin{aligned} \boldsymbol{not} &\stackrel{def}{=} \lambda a.a\boldsymbol{FT} \\ \boldsymbol{or} &\stackrel{def}{=} \lambda ab.aab \\ \boldsymbol{xor} &\stackrel{def}{=} \lambda ab.a(b\boldsymbol{FT})b \end{aligned}$$

验证这几个编码的有效性。另外，尝试为这几个运算给出和上面不一样的编码。

练习 1.7. 布尔值还用在条件判断中，对于条件判断，我们定义

$$\textbf{if\_then\_else} \stackrel{def}{=} \lambda x.x$$

验证这个项具有下面期望的行为，即对任何项 M 和 N，我们有

$$\begin{aligned} \textbf{if\_then\_else } \boldsymbol{T}MN &\longrightarrow_\beta^* M \\ \textbf{if\_then\_else } \boldsymbol{F}MN &\longrightarrow_\beta^* N \end{aligned}$$

自然数 如果 f 和 x 是两个 λ 项，n 是一个自然数，我们用记号 $f^n x$ 代表把 f 作用 n 次到 x 上所得到的项。例如 $f^0 x = x$，$f^1 x = fx$，$f^2 x = f(fx)$ 等等。对每个自然数 n，我们定义第 n 个邱奇数（Church numeral）为一个 λ 项，$\overline{n} \stackrel{def}{=} \lambda fx.f^n x$。从 0 开始的前几个邱奇数如下：

$$\begin{aligned} \overline{0} &\stackrel{def}{=} \lambda fx.x \\ \overline{1} &\stackrel{def}{=} \lambda fx.fx \\ \overline{2} &\stackrel{def}{=} \lambda fx.f(fx) \\ &\vdots \end{aligned}$$

对自然数的这种编码是邱奇提出来的。这里 $\overline{0}$ 和之前定义的 $\mathbf{F}$ 是 α-等价的。接下来我们需要编码自然数集合上的一些常用函数。我们把后继函数编码为项 **succ**，其中

$$\textbf{succ} \stackrel{def}{=} \lambda nfx.f(nfx)$$

我们把 **succ** 作用于 $\overline{n}$，进行几步 β 规约以后求值到 $\overline{n+1}$，如我们所期望的一样。

$$\begin{aligned} \textbf{succ } \overline{n} &\equiv (\lambda nfx.f(nfx))(\lambda fx.f^n x) \\ &\longrightarrow_\beta \lambda fx.f((\lambda fx.f^n x)fx) \\ &\longrightarrow_\beta \lambda fx.f((\lambda x.f^n x)x) \\ &\longrightarrow_\beta \lambda fx.f(f^n x) \\ &\equiv \lambda fx.f^{n+1}x \\ &\equiv \overline{n+1} \end{aligned}$$

对于加法、乘法和幂运算，我们定义 **add**、**mult** 和 **exp** 三个项。

$$\begin{array}{rcl} \mathbf{add} & \stackrel{def}{=} & \lambda mnfx.mf(nfx) \\ \mathbf{mult} & \stackrel{def}{=} & \lambda mnf.m(nf) \\ \mathbf{exp} & \stackrel{def}{=} & \lambda mn.nm \end{array}$$

练习 1.8. 对于任何自然数 m 和 n，证明下面性质：

$$\begin{array}{rcl} \mathbf{add}\ \overline{m}\ \overline{n} & \longrightarrow_\beta^* & \overline{m+n} \\ \mathbf{mult}\ \overline{m}\ \overline{n} & \longrightarrow_\beta^* & \overline{m \times n} \\ \mathbf{exp}\ \overline{m}\ \overline{n} & \longrightarrow_\beta^* & \overline{m^n} \end{array}$$

还有一个有用的函数是判断一个自然数是否为零，如果为零那么返回真，否则返回假。我们可以用下面的项来表示：

$$\mathbf{iszero} \stackrel{def}{=} \lambda n.n(\lambda x.\mathbf{F})\mathbf{T}$$

在本节的最后，我们考虑前继函数。令人吃惊的是，与后继函数相比，它的定义复杂得多：

$$\mathbf{pred} \stackrel{def}{=} \lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)$$

这个前继函数的计算比后继函数慢很多，因为它构造结果的时候需要从零开始反复计算后继。

练习 1.9. 验证 **iszero** 和 **pred** 具有我们期望的性质：

$$\begin{array}{rcl} \mathbf{iszero}\ \overline{0} & \longrightarrow_\beta^* & \boldsymbol{T} \\ \mathbf{iszero}\ \overline{n+1} & \longrightarrow_\beta^* & \boldsymbol{F} \\ \mathbf{pred}\ \overline{0} & \longrightarrow_\beta^* & \overline{0} \\ \mathbf{pred}\ \overline{n+1} & \longrightarrow_\beta^* & \overline{n} \end{array}$$

练习 1.10. 除了前面定义的邱奇数和后继函数 **succ**，下面给出 **pair** 和 **step** 两个项：

$$\begin{array}{rcl} \mathbf{pair} & = & \lambda xyz.zxy \\ \mathbf{step} & = & \lambda p.p(\lambda xy.\mathbf{pair}\ (\mathbf{succ}\ x)\ (fxy)) \end{array}$$

证明下面规约性质：$\mathbf{step}\ (\mathbf{pair}\ \overline{n}\ a) \longrightarrow_\beta^* \mathbf{pair}\ \overline{n+1}\ (f\ \overline{n}\ a)$。

1.2.6 不动点

假设 $f : D \to D$ 是在定义域和值域均为 D 上的一个函数，$x \in D$ 是 D 中的一个元素。如果 $f(x) = x$ 成立，我们称 x 是 f 的一个不动点（fixed point）。在数学上，有的函数没有不动点，比如 $f(x) = x - 1$；有的函数如 $f(x) = x^2 - x + 1$ 有唯一一个不动点 1；有的函数如 $f(x) = x^2 - x$ 有两个不动点：0 和 2；有的函数甚至有无穷多个不动点，比如 $f(x) = x$。

对于 λ-演算，我们也引入不动点的概念。对于两个项 F 和 M，如果 $FM =_\beta M$，我们称 M 是 F 的一个不动点。与数学函数很不一样的是，任何一个 λ 项都有不动点！

定理 1.1. *在不带类型的 λ-演算中，每个项都有一个不动点。*

证明. 我们定义一个特殊的项 Θ：

$$\Theta \stackrel{def}{=} (\lambda xy.y(xxy))(\lambda xy.y(xxy))$$

对于任何项 F，我们进行如下计算：

$$\begin{array}{rcl} \Theta F & \equiv & (\lambda xy.y(xxy))(\lambda xy.y(xxy))F \\ & \longrightarrow_\beta & (\lambda y.y((\lambda xy.y(xxy))(\lambda xy.y(xxy))y))F \\ & \longrightarrow_\beta & F((\lambda xy.y(xxy))(\lambda xy.y(xxy))F)) \\ & \equiv & F(\Theta F) \end{array}$$

因此，我们有 $F(\Theta F) =_\beta \Theta F$，即 ΘF 是项 F 的一个不动点。 □

这个特殊的项 Θ 在文献中经常被称为图灵不动点组合算子（Turing's fixed point combinator），它提供一种寻找不动点的方法。下面我们通过一个例子来介绍 Θ 的应用。

例 1.5. *我们将利用不动点组合算子来定义阶乘函数。首先，我们把阶乘函数需要满足的等式写出来：*

$$\mathbf{fact}\ n \quad = \quad \mathbf{if\_then\_else}\ (\mathbf{iszero}\ n)(\overline{1})(\mathbf{mult}\ n\ (\mathbf{fact}\ (\mathbf{pred}\ n)))$$

因为待定义的项 **fact** 出现在等式两边，所有这个等式是递归的。为了得到这个 **fact**，我们需要寻找这个方程的解。为此，我们对等式做两步变换。

$$\begin{aligned}\textbf{fact} &= \lambda n.\textbf{if\_then\_else}\ (\textbf{iszero}\ n)(\overline{1})(\textbf{mult}\ n\ (\textbf{fact}\ (\textbf{pred}\ n)))\\ &= (\lambda f.\lambda n.\textbf{if\_then\_else}\ (\textbf{iszero}\ n)(\overline{1})(\textbf{mult}\ n\ (f(\textbf{pred}\ n))))\ \textbf{fact}\end{aligned}$$

令 $F \stackrel{def}{=} \lambda f.\lambda n.\textbf{if\_then\_else}\ (\textbf{iszero}\ n)(\overline{1})(\textbf{mult}\ n\ (f(\textbf{pred}\ n)))$。上面的方程变成 $\textbf{fact} = F\ \textbf{fact}$，即我们要找的 **fact** 是项 F 的一个不动点。根据定理 *1.1*的证明中给出的求不动点的方法，我们可以得到 **fact** 的一个定义：

$$\begin{aligned}\textbf{fact} &\stackrel{def}{=} \Theta F\\ &= \Theta(\lambda f.\lambda n.\textbf{if\_then\_else}\ (\textbf{iszero}\ n)(\overline{1})(\textbf{mult}\ n\ (f(\textbf{pred}\ n))))\end{aligned}$$

练习 1.11. 用数学归纳法证明 $\textbf{fact} \longrightarrow_\beta^* \overline{n!}$ 对任何自然数 n 都成立。

练习 1.12. 令 $\mathbf{Y} \stackrel{def}{=} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$。证明 $\mathbf{Y}$ 是一个不动点组合算子，即对任何项 F，它的一个不动点是 $\mathbf{Y}F$。在文献中，项 $\mathbf{Y}$ 被称为柯里不动点组合算子（*Curry's fixed point combinator*）。

1.2.7 其它数据类型

除了布尔类型和自然数类型，我们还可以在 λ-演算中表示其它数据类型，例如二元组、多元组、列表、树等等。

二元组 如果 M 和 N 是两个项，由它们组成的二元组（pair）$\langle M, N\rangle$ 可以用项 $\lambda z.zMN$ 来表示。同时我们给出两个投影（projection）函数 $\pi_1 \stackrel{def}{=} \lambda p.p(\lambda xy.x)$ $\pi_2 \stackrel{def}{=} \lambda p.p(\lambda xy.y)$。很容易验证下面两条性质：

$$\begin{aligned}\pi_1\langle M, N\rangle &\longrightarrow_\beta^* M\\ \pi_2\langle M, N\rangle &\longrightarrow_\beta^* N\end{aligned}$$

多元组 我们可以把二元组扩展到任意n-元组（n-tuples）。假设给定 n 个项 $M_1, ..., M_n$，我们用项 $\lambda z.zM_1...M_n$ 来表示 n-元组 $\langle M_1, ..., M_n\rangle$，同时定义第 i 个投影函数 $\pi_i^n \stackrel{def}{=} \lambda p.p(\lambda x_1...x_n.x_i)$。可以验证下面性质：

$$\pi_i^n\langle M_1, ..., M_n\rangle \longrightarrow_\beta^* M_i \qquad \text{其中} 1 \le i \le n.$$

列表 与多元组不同，列表（list）的长度是不固定的。一个列表可以是空的，也可以是头（head）元素后面跟上另一个尾（tail）列表。我们用 **nil** 表示空列表，用 $H :: T$ 表示一个非空列表，它的头和尾分别是 H 和 T。例如，$9 :: 7 :: 5 :: \mathbf{nil}$ 表示一个列表，前三个元素一次是 9，7 和 5。

在 λ-演算中，我们定义 $\mathbf{nil} \stackrel{def}{=} \lambda xy.y$ 和 $H :: T \stackrel{def}{=} \lambda xy.xHT$。基于这样的定义，我们可以表示操纵列表的一些函数。例如，如果列表 l 中存储的是自然数，为对表中所有数字求和，我们希望有一个满足下面等式的项 **sumlist**：

$$\mathbf{sumlist}\ l\ =\ l(\lambda ht.\ \mathbf{add}\ h(\mathbf{sumlist}\ t))(\overline{0}). \tag{1.2.1}$$

根据第 1.2.6节介绍的方法，我们可以利用不动点算子显式地构造出项 **sumlist**。

练习 1.13. *从（1.2.1）出发给出一个 λ 项以实现* **sumlist**，*并验证下面的规约成立：*

$$\mathbf{sumlist}\ (\overline{9} :: \overline{7} :: \overline{5} :: \mathbf{nil})\ \longrightarrow_\beta^*\ \overline{21}.$$

二叉树 一棵二叉树只有两种形式：要么是一个叶子节点，用一个自然数作为标号，要么是一个内部节点，从它出发有左右两棵子树。我们用 $\mathbf{leaf}\ n$ 表示标号为 n 的叶子节点，用 $\mathbf{node}(L, R)$ 表示有左右子树分别为 L 和 R 的内部节点。我们可用 λ 项对它们进行编码。

$$\mathbf{leaf}(n) \stackrel{def}{=} \lambda xy.xn, \qquad\qquad \mathbf{node}(L, R) \stackrel{def}{=} \lambda xy.yLR.$$

我们可以定义一个函数 **sumtree**，对树上所有叶子节点的标号进行求和。它满足如下等式：

$$\mathbf{sumtree}\ t\ =\ t(\lambda n.n)(\lambda lr.\ \mathbf{add}\ (\mathbf{sumtree}\ l)(\mathbf{sumtree}\ r)).$$

练习 1.14. *写出一个 λ 项来实现函数* **sumtree**。

1.2.8 邱奇-罗索定理

在不带类型的 λ-演算中，我们一直把 λ 项看作函数，可以作用于任何其它项，然后我们引入 β-等价的概念来比较两个项。如果我们回顾数学中函数

等价的含义，会得到不一样的启发。对于任意两个数学函数 f 和 g，我们说它们相等，通常是指它们的定义域相同，并且对于任何定义域上的值 x，我们总是有 $f(x) = g(x)$，这是函数的外延性质。现在把这个观点用到 λ-演算中，我们考察项 $\lambda x.Mx$ 和 M，其中变量 x 假设不在 M 中自由出现。因为 $(\lambda x.Mx)N =_\beta MN$，我们似乎没有理由区分它们。但是我们注意到 $\lambda x.Mx$ 和 M 这两个项不是 β-等价的，为了把这两个项等同起来，我们需要加入其它规则。

如果把定义 1.5中的（B_β）改成下面的规则

$$(B_\eta) \qquad \lambda x.Mx \longrightarrow_\eta M \qquad \text{其中} x \notin FV(M)$$

我们得到 η-规约的定义。令 $\longrightarrow_{\beta\eta} \stackrel{def}{=} \longrightarrow_\beta \cup \longrightarrow_\eta$，也就说，$M \longrightarrow_{\beta\eta} M'$ 当且仅当 $M \longrightarrow_\beta M'$ 或者 $M \longrightarrow_\eta M'$。多步的 $\beta\eta$-规约，记为 $\longrightarrow^*_{\beta\eta}$，$\beta\eta$-等价，记为 $=_{\beta\eta}$，以及 $\beta\eta$-范式的概念都可以类比第 1.2.4节中的有关概念而定义出来。

定理 1.2. *令 $\longrightarrow^*$ 表示 $\longrightarrow^*_\beta$ 或者 $\longrightarrow^*_{\beta\eta}$。假设有 M，N 和 P 三个 λ 项使得 $M \longrightarrow^* N$ 和 $M \longrightarrow^* P$ 都成立。那么必定存在一个项 Z 满足 $N \longrightarrow^* Z$ 和 $P \longrightarrow^* Z$。*

这个定理说明，如果从项 M 出发规约到两个项 N 和 P，那可以继续规约最终合流到一个共同的项 Z。这个性质称为*邱奇-罗索性质*（Church-Rosser property）或者*合流*（confluence）。具体证明可以参见 [2, 10]，这里我们只介绍它的几个推论。

推论 1.1. *1. 如果 $M =_\beta N$，那么存在项 Z 满足 $N \longrightarrow^*_\beta Z$ 和 $P \longrightarrow^*_\beta Z$。*

2. 如果 N 是一个 β-范式而且 $N =_\beta M$，那么 $M \longrightarrow^*_\beta N$。

3. 如果 M 和 N 是 β-范式且 $M =_\beta N$，那么 $M =_\alpha N$。

4. 如果 $M =_\beta N$，那么或者两个项都有一个 β-范式，或者两个项都没有 β-范式。

如果把上面所有的 β 改成 $\beta\eta$，这四条性质依然成立。

1.3 简单类型的 λ-演算

在介绍不带类型的 λ-演算时，我们讨论函数而没有提及它们的定义域和值域。对任何 λ 项表示的函数，定义域和值域都是所有 λ 项组成的集合。现在我们为 λ-演算引入类型（types），并显式表示函数的定义域和值域。注意类型和集合的区别：类型是语法实体（syntactic objects），我们可以讨论一种类型而不关心具有这种类型的元素，并可以把类型大致理解为集合的名字。

1.3.1 简单类型的项

对于布尔类型、自然数类型这样不可拆分的类型，我们称为基本类型。假定一个基本类型的集合，用希腊字母 ι 来代表一个基本类型。

定义 1.7. *简单类型可以用下面的 BNF 来定义：*

$$A, B ::= \iota \mid A \to B \mid A \times B \mid 1$$

我们可以看出，简单类型是从基本类型和 1 类型构造出来的，其中 1 类型只有一个元素，可以视为普通编程语言中的空类型（void type）或者单元类型（unit type）。如果一个函数没有返回值，我们可以说这个函数计算结果的类型为空类型或者单元类型。类型 $A \to B$ 是一类函数的类型，其输入值类型为 A，输出值类型为 B。类型 $A \times B$ 是二元组 $\langle x, y\rangle$ 的类型，其中 x 的类型为 A，y 的类型为 B。

我们现在有 $\to$ 和 $\times$ 两个类型构造子（constructor），其中后者的优先级比前者高，且 $\to$ 是右结合的。例如，$A \times B \to C \to D$ 表示 $(A \times B) \to (C \to D)$。

定义 1.8. *原始的（raw）带类型 λ 项可以用下面的 BNF 来定义：*

$$M, N ::= x \mid MN \mid \lambda x^A.M \mid \langle M, N\rangle \mid \pi_1 M \mid \pi_2 M \mid *$$

与定义 1.1相比，现在有些变化。对于 λ 抽象，我们写 $\lambda x^A.M$ 来说明变量 x 的类型是 A。有时候我们不关心 x 的类型，忽略上标 A 而与以前一样写作 $\lambda x.M$。另外，我们增加了四个语法构造子：$\langle M, N\rangle$ 表示由 M 和 N 组成的二元组，$\pi_i M$ 是一个满足 $\pi_i\langle M_1, M_2\rangle = M_i$（$i = 1, 2$）的投影，项 $*$ 是

$$(T_v)\quad \frac{}{\Gamma, x:A \vdash x:A}$$

$$(T_{ap})\quad \frac{\Gamma \vdash M:A \to B \qquad \Gamma \vdash N:A}{\Gamma \vdash MN:B}$$

$$(T_{ab})\quad \frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash \lambda x^A.M:A \to B}$$

$$(T_p)\quad \frac{\Gamma \vdash M:A \qquad \Gamma \vdash N:B}{\Gamma \vdash \langle M,N\rangle : A \times B}$$

$$(T_{pj1})\quad \frac{\Gamma \vdash M:A \times B}{\Gamma \vdash \pi_1 M:A}$$

$$(T_{pj2})\quad \frac{\Gamma \vdash M:A \times B}{\Gamma \vdash \pi_2 M:B}$$

$$(T_s)\quad \frac{}{\Gamma \vdash *:1}$$

表 1.2 简单类型 λ-演算的类型规则

类型 1 的唯一一个元素。自由和受限变量的概念与不带类型的 λ-演算中相同，并且我们不区分 α-等价的项。

上面定义的项之所以被称为原始的项，是因为我们没有对它们增加类型约束。为了避免一些无意义的项，比如 $\pi_2(\lambda x.M)$，我们引入一套*类型系统*（type system），它是通过表 1.2中的*类型规则*（typing rules）定义出来的。

我们用记号 $M:A$ 来表示项 M 的类型是 A。类型规则是通过*类型判断*（typing judgments）来表述的。一个类型判断是下面形式的表达式：

$$x_1:A_1, x_2:A_2, ..., x_n:A_n \vdash M:A.$$

其意义是说，对于 1 和 n 之间的任何 i，假设 x_i 的类型是 A_i，那么项 M 是一个良类型的（well typed）项，它的类型是 A。项 M 中出现的自由变量应该被包含在 $x_1, ..., x_n$ 中。也就是说，在决定项 M 的类型之前，我们必须知道它的所有自由变量的类型。

在类型判断的左边是形如 $x_1:A_1, ..., x_n:A_n$ 的假设，称为*类型上下文*（typing context），其中的每个 x_i 都与其它变量不同。我们用希腊字母 Γ 代表任意的类型上下文，用 Γ, Γ' 和 $\Gamma, x:A$ 表示类型上下文的拼接，其中涉及的变量名互相不同。如果一个类型上下文 Γ 是空的，其中没有任何变量，我们把 $\Gamma \vdash M:A$ 简写为 $\vdash M:A$。

表 1.2列出简单类型 λ-演算的类型规则。规则（T_v）不依赖于任何假设，它是一条公理。如果在类型上下文中假设 x 的类型是 A，那么项 x 的类型就

是 A。规则（T_{ap}）说明可以把一个类型为 $A \to B$ 的函数作用到类型为 A 的参数上，得到的结果具有类型 B。规则（T_{ab}）告诉我们，如果项 M 的类型是 B，其中可能有 A 类型的自由变量 x，那么 $\lambda x^A.M$ 是一个类型为 $A \to B$ 的函数。规则（T_p）说明把类型分别为 A 和 B 的两个项组合成一个二元组以后，结果的类型为 $A \times B$。与之对应的是规则（T_{pj1}）和（T_{pj2}）：对类型为 $A \times B$ 的项进行左投影，得到的项类型为 A，进行右投影得到类型为 B 的项。规则（T_s）说明 $*$ 是类型为 1 的项。

例 1.6. 对于项 $\lambda x^{A\to A}.x((\lambda y^A.xy)z)$，一个可能的类型是 $(A \to A) \to A$，具体的类型推导过程可反复利用表 *1.2*中的规则建立下面的推导树，其中的类型上下文 Γ 表示 $z : A, x : A \to A$。

$$\dfrac{\dfrac{}{\Gamma \vdash x : A \to A} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, y : A \vdash x : A \to A} \quad \dfrac{}{\Gamma, y : A \vdash y : A}}{\Gamma, y : A \vdash xy : A}}{\Gamma \vdash \lambda y^A.xy : A \to A} \quad \dfrac{}{\Gamma \vdash z : A}}{\Gamma \vdash (\lambda y^A.xy)z : A}}{\Gamma \vdash x((\lambda y^A.xy)z) : A}}{z : A \vdash \lambda x^{A\to A}.x((\lambda y^A.xy)z) : (A \to A) \to A}$$

注意观察表 1.2中的类型规则，对每种类型的 λ 项，我们有且只有一条规则。在构造类型推导树的时候，我们采用自底向上的方式，根据当前被考虑的 λ 项的类型，选择并应用唯一一条规则，然后考虑规则前提中的 λ 项，继续往上推导。如果最后运用的是无前提的规则（即公理），例如（T_v）和（T_s），那么对当前项的类型推导结束，说明我们到达推导树的一个叶子节点。如果自底向上所有分支都到达叶子节点，则整个推导过程结束，完成这棵推导树的构造。

练习 1.15. 为类型判断 $\vdash \lambda x^{A\times B}.\lambda f^{B\times A\to A}.f(\langle \pi_2 x, \pi_1 x\rangle) : A$ 构造一棵推导树。

1.3.2 规约

对于简单类型的 λ-演算，我们需要扩展不带类型 λ-演算中的 β-规约和 η-规约的规则，以处理二元组和投影形式的项。

定义 1.9. 我们为简单类型的 λ-演算引入下面这些 β-规约和 η-规约的规则：

$$\begin{array}{rcll}
(\lambda x^A.M)N & \longrightarrow_\beta & M[N/x] & \\
\pi_1\langle M,N\rangle & \longrightarrow_\beta & M & \\
\pi_2\langle M,N\rangle & \longrightarrow_\beta & N & \\
\lambda x^A.Mx & \longrightarrow_\eta & M & \text{其中} x \notin FV(M) \\
\langle \pi_1 M, \pi_2 M\rangle & \longrightarrow_\eta & M & \\
M & \longrightarrow_\eta & * & \text{若} \vdash M : 1
\end{array}$$

以及下面几条新的同余规则：

$$\frac{M \longrightarrow_{\beta\eta} M'}{\langle M,N\rangle \longrightarrow_{\beta\eta} \langle M',N\rangle} \qquad \frac{N \longrightarrow_{\beta\eta} N'}{\langle M,N\rangle \longrightarrow_{\beta\eta} \langle M,N'\rangle}$$

$$\frac{M \longrightarrow_{\beta\eta} M'}{\pi_1 M \longrightarrow_{\beta\eta} \pi_1 M'} \qquad \frac{M \longrightarrow_{\beta\eta} M'}{\pi_2 M \longrightarrow_{\beta\eta} \pi_2 M'}$$

关于规约的一条重要性质是主体规约（subject reduction），其涵义是说良类型的项只会规约到良类型项，而且保持类型不变。这在程序设计中有明显的应用，例如，一个返回类型是整型的程序，执行结束之后返回的值应该是一个整数，而不是一个浮点数。

定理 1.3 (主体规约)**.** 如果 $\Gamma \vdash M : A$ 并且 $M \longrightarrow_{\beta\eta} M'$，那么 $\Gamma \vdash M' : A$。

我们用 $\lambda^{\to,\times,1}$-演算指代上面介绍的简单类型的 λ-演算，用 $\lambda^{\to,\times}$-演算指代它的子语言，其中去掉了类型 1 和项 $*$。

定理 1.4. 在 $\lambda^{\to,\times,1}$-演算中，邱奇-罗索性质对 β-规约成立；在 $\lambda^{\to,\times}$-演算中，邱奇-罗索性质对 $\beta\eta$-规约成立。

在 $\lambda^{\to,\times,1}$-演算中，邱奇-罗索性质对 $\beta\eta$-规约不成立，主要原因在于定义 1.9中的最后一条 η-规约规则。假设 x 是一个类型为 1×1 的变量，我们有下面的规约：

$$\langle \pi_1 x, \pi_2 x\rangle \longrightarrow_\eta x$$
$$\langle \pi_1 x, \pi_2 x\rangle \longrightarrow_\eta \langle *, \pi_2 x\rangle \longrightarrow_\eta \langle *, *\rangle$$

我们从项 $\langle \pi_1 x, \pi_2 x\rangle$ 出发，得到两个不同的 $\beta\eta$-范式：x 和 $\langle *, *\rangle$。

1.3.3 正规化

有时候我们会关心这样一个问题：从给定的一个 λ 项出发进行 β-规约，是否总能到达一个范式?

定义 1.10. 给定一个项 M，如果存在一个有限长的规约序列 $M \longrightarrow M_1 \longrightarrow ... \longrightarrow M_n$ 使得 M_n 是一个范式，那我们称 M 是弱正规化的（*weakly normalizing*）。如果从 M 出发的每个规约序列都是有限长度的，或者说从 M 出发不存在无限的规约序列，那我们称 M 是强正规化的（*strongly normalizing*）。

根据定义我们知道一个强正规化的项必定是弱正规化的，反之不一定成立。

下面看几个不带类型的 λ-演算中的项。

1. 我们曾见过这个项 $(\lambda x.xx)(\lambda x.xx)$，做一步 β-规约以后回到自身，不能到达其它项，因此只能产生无限的规约序列。这个项不是弱正规化的，当然更不是强正规化的。

2. 考虑项 $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$。它可以规约到范式 $\lambda y.y$，但也有一个无限的规约序列，因此是弱正规化的，而不是强正规化的。

3. 考虑项 $(\lambda xy.y)((\lambda x.xx)(\lambda x.x))$。它有不同的规约序列，但都是有限的，且规约到共同的范式 $\lambda y.y$，因此是强正规化的。

上面的例子都是不带类型的 λ-演算中的项。对于简单类型的 λ-演算，情况会不一样。类似 xx 形式的项都不是良类型的。

定理 1.5. 在简单类型的 λ-演算中，所有良类型的项都是强正规化的。

关于这个定理的证明，可以参见 [5]。

练习 1.16. 下面的 λ 项哪些是弱正规化的？哪些是强正规化的？

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

2. $(\lambda x.xx)(\lambda ab.bbb)$

3. $(\lambda xy.x)((\lambda x.xxx)(\lambda x.xxx))$

4. $(\lambda xy.x)(\lambda x.xxx)(\lambda x.xxx)$

第 2 章　Coq

Coq是一个交互式的定理证明器（theorem prover），可以表达数学断言（assertion），检查对断言的证明，或通过与用户交互，找到形式化的证明。Coq 的工作原理基于归纳构造演算（calculus of inductive constructions）[8]。作为一种编程语言，Coq 实现了一个依赖类型的（dependently typed）函数式编程语言；作为一个逻辑系统，它实现了一个高阶类型理论。自从 1989 年发布以来，Coq 一直由法国 INRIA 研究所提供支持和更新，在 2013 年获得 ACM 系统软件奖。Coq 提供了一个规范说明语言，称为 Gallina，本章主要介绍它的用法。为练习使用 Coq，建议先从官网`https://coq.inria.fr`下载最新版本的 Coq 集成开发环境 CoqIDE。

2.1　基本的函数式编程

我们先定义一些简单的数据类型，例如枚举类型和自然数类型。

枚举类型　在 Coq 中，很多类型都可以归纳定义出来，这其中最简单的可能就是枚举类型。例如，一年中四个季节是确定的，那么我们可以定义一个 `season` 类型，包含 `spring`，`summer`，`autumn` 和 `winter` 四个数据值。具体的 Coq 定义如下：

```
Inductive season : Type :=
  | spring
  | summer
  | autumn
  | winter.
```

这个定义以关键字 Inductive 开头，说明是一个归纳定义，而紧跟其后的 season : Type 声明 season 是我们新定义的一个类型。竖线隔开四种情况，spring、summer 等是类型构造子（type constructor），因为没带任何参数，可以简单理解为四个元素，它们组成一个集合，即名为 season 的类型所表示的集合。

有了数据类型 season，我们可以定义一些函数来处理这种类型的数据。下面定义一个名为 opposite_season 的函数，对输入的一个季节，返回与它相反的季节，因此输入和输出的类型均为 season，换句话说，这个函数自身的类型是 season → season。

```
Definition opposite_eason (s : season) : season :=
  match s with
  | spring => autumn
  | summer => winter
  | autumn => spring
  | winter => summer
  end.
```

在具体定义中我们以模式匹配（pattern matching）的方式，区分 s 的不同形式。如果 s 是 spring，则返回 autumn；如果是 summer，则返回 winter，等等。

为测试刚才定义的函数，我们可以用 Compute 命令对一个复杂的表达式求值。

```
Compute (opposite_season spring).
```

在 CoqIDE 中执行这句命令后返回结果为 autumn : season,即返回值 autumn 和它的类型 season 都被打印出来。注意这里 (opposite_season spring) 的意思是把函数 opposite_season 作用到参数 spring 上，写法与 λ-演算中一样。我们再测试一个更复杂表达式的求值。

```
Compute (opposite_season (opposite_season spring)).
```

执行这句命令后返回结果为 spring : season，如我们期望的一样。

布尔类型 布尔类型实际上是一个特殊的枚举类型，它只包含两个数据值，具体名称其实不重要，在这里我们不妨称为 `true` 和 `false`。

```
Inductive bool : Type :=
  | true
  | false.
```

对布尔类型数据的常用操作如“与”、“或”、“非”等都容易定义。例如，我们定义函数 `andb` 来表示“与”操作。这个函数输入两个参数 `b1` 和 `b2`，如果 `b1` 的值为 `true`，直接返回 `b2` 的值，否则返回 `false`。

```
Definition andb (b1 : bool) (b2 : bool) : bool :=
  match b1 with
  | true => b2
  | false => false
  end.
```

上面这个函数的两个输入参数类型相同，在 Coq 中可以写得紧凑一点，依次列出两个参数，中间用空格隔开，最后写上类型，如下面函数 `andb'` 的定义所示。

```
Definition andb' (b1 b2 : bool) : bool :=
  match b1 with
  | true => b2
  | false => false
  end.
```

另外，我们还可以用 Coq 中的 `Notation` 命令来定义新的记号。

```
Notation "x && y" := (andb x y).
```

有个这个记号以后，将来如果我们写表达式 `(b1 && b2)`，那么代表的意思就是 `(andb b1 b2)`，这类似有些编程语言中的宏定义。

练习 2.1. 把下面两个函数的定义补充完整，其中 `negb` 和 `orb` 分别表示“非”和“或”操作。

```
Definition orb (b1 b2 : bool) : bool
Definition negb (b : bool) : bool
```

自然数类型 在枚举类型中我们需要逐个列出该类型的数据值，这就意味着数据值的个数必须是有限的。由于自然数的个数是无限的，我们不可能都显式地枚举出来，因此需要找到一种有限表示。归纳的思想可帮助我们用有限的规则表示无限的数据。为了不覆盖 Coq 标准库中对自然数类型的定义，我们把 nat 的定义放到一个名为 Playground 的模块（module）中。

```
Module Playground.
Inductive nat : Type :=
  | O
  | S (n : nat).
End Playground.
```

上面这个定义给出自然数的一进制表示（unary representation），其中用到两个构造子：O 表示自然数 0；S 表示后继函数，如果 n 是一个自然数，则 S n 表示它的下一个自然数。换句话说，类型 nat 包含的数据值形式如下：

$$\texttt{O},\ \texttt{S O},\ \texttt{S (S O)},\ \texttt{S (S (S O))}, ...$$

对应通常的自然数 $0, 1, 2, 3, ...$。

```
Check S (S (S O)).
```

执行这条命令后，CoqIDE 输出 3 : nat，因为它自动把 S (S (S O)) 转化成十进制数 3。

对自然数上一些简单的函数，我们用模式匹配的方式来定义。例如，下面的 pred2 函数取比当且输入数 n 小 2 的数，除了特殊情况 $n = 0, 1$，那时返回值为 0。

```
Definition pred2 (n : nat) : nat :=
  match n with
  | O => O
  | S O => O
```

```
  | S (S n') => n'
  end.
```

为定义复杂一些的函数，仅用模式匹配还不够，我们需要递归定义，反映到语法层面就是定义的开头用关键字 `Fixpoint` 而不是 `Definition`。下面的函数 `oddb` 可判断当 `n` 取 0 或者 1 时，我们直接判断出结果，否则，`n` 是不是奇数这件事情依赖于 `n` 之前第二个数字的奇偶性。

```
Fixpoint oddb (n : nat) : bool :=
  match n with
  | O => false
  | S O => true
  | S (S n') => oddb n'
end.
```

如果一个函数的参数不止一个，那么用于模式匹配的参数选取可能不唯一。下面给出加法函数的两种定义，都可以被 Coq 接受。

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.


Fixpoint plus' (n m : nat) : nat :=
  match m with
  | O => n
  | S m' => S (plus n m')
  end.
```

有时候我们可以同时对多个参数做模式匹配，不同参数之间用逗号隔开。下面定义的减法函数演示了这一点，其中不重要的参数可以不写，改用占位符“_”代替。

```
Fixpoint minus (n m : nat) : nat :=
```

```
  match n, m with
  | O , _ => O
  | S _ , O => n
  | S n', S m' => minus n' m'
  end.
```

当然，我们也可以把对多个参数的同时模式匹配改为嵌套的模式匹配。下面定义的函数 minus' 与 minus 是等效的。

```
Fixpoint minus' (n m : nat) : nat :=
  match n with
  | O  => O
  | S n' => match m with
            | O => n
            | S m' => minus' n' m'
            end
  end.
```

练习 2.2. 定义函数 div3，使得 div3 n 返回 true 当且仅当 n 是 *3* 的倍数。

练习 2.3. 定义函数 div2021，使得 div2021 n 返回 true 当且仅当 n 是 *2021* 的倍数。

练习 2.4. 定义乘法、幂和阶乘函数，使得 mult n m、exp n m 和 fact n 的值分别为 $n+m$，$n \times m$ 和 $n!$。

练习 2.5. 定义两个自然数上的比较函数，使得 eqb n m 返回布尔值 true 当且仅当 $n = m$，leb n m 返回布尔值 true 当且仅当 $n \leq m$。对每个函数，分别用同时模式匹配和嵌套模式匹配两种方式定义。

结构归纳 重新回顾函数 plus 的定义，它对第一个参数 n 做归纳。Coq 在检查这个定义的合法性时，需要确保:= 右边出现的 plus 的第一个参数在变小。事实上，如果 n 匹配到的表达式是 S n'，那么 n' 是一个子表达式，结构变小了，因此这个定义是可以接受的。之所以要求递归定义中的参数变小，是为了保证所定义的函数对任何输入值的计算最后都会终止。

练习 2.6. *考虑阿克曼函数（The Ackermann function），其定义如下。这个函数是否对任何输入都终止？尝试写出一个定义，看能否被 Coq 接受。如果不接受，为什么？*

$$A(m,n) = \begin{cases} n+1 & 若 m = 0 \\ A(m-1,1) & 若\ m > 0\ 且\ n = 0 \\ A(m-1, A(m,n-1)) & 若\ m > 0\ 且\ n > 0 \end{cases}$$

2.2 列表

二元组 上一节介绍的布尔类型和自然数类型都是基本数据类型。现在我们从基本类型出发来构造复杂一些的类型。我们先定义二元组（pair）类型，它包含的每一个元素是一对自然数。

```
Inductive natpair : Type :=
  | pair (n1 n2 : nat).
```

类型 `natpair` 只有一个构造子 `pair`，说明构造一对自然数的唯一方法是把 `pair` 作用到两个自然数上。我们可以用 `Check` 命令查看元素 `(pair 1 2)` 的类型。

```
Check (pair 1 2) : natpair.
```

我们最好引入熟悉的记号 `(x,y)` 来表示 `pair x y`。

```
Notation "( x , y )" := (pair x y).
```

练习 2.7. *类型 `natpair` 中的元素是一对自然数 `(x,y)`，即 `x` 和 `y` 同为自然数。请定义一个新的二元组类型，使得 `x` 是一个自然数但 `y` 是一个布尔值。*

下面我们定义两个投影（projection）函数，把给定的一对自然数投影到第一或第二个组成数字上。

```
Definition proj1 (p : natpair) : nat :=
  match p with
  | (x,y) => x
  end.
```

```
Definition proj2 (p : natpair) : nat :=
  match p with
  | (x,y) => y
  end.
```

例如，下面的命令测试如何把一对数字 (1, 2) 变成 (2, 1)。

```
Compute (proj2 (1, 2), proj1 (1, 2)).
```

列表 通过推广二元组类型，我们可以构建三元组、四元组等等，更一般情况就是列表，其长度可以变化。简单而言，由自然数组成的一个列表或者是空表，或者是由一个自然数和另一个列表组成的二元组。我们在模块 NatList 中定义自然数列表类型。

```
Module NatList.
Inductive natlist : Type :=
  | nil
  | cons (n : nat) (l : natlist).
```

这个定义中用到两个构造子：nil 表示空表；cons 把自然数 n 和列表 l 组合成一个更长的列表 cons n l。下面是一个长度为 3 的列表：

```
Definition alist := cons 1 (cons 3 (cons 5 nil)).
```

我们引入中缀操作符记号 :: 和方括号来表示列表，其中细节暂时不用关心。

```
Notation "x :: l" := (cons x l) (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
```

利用这些记号，我们定义 3 个列表，实际上描述的是同一个列表。

```
Definition list1 := 1 :: (3 :: (5 :: nil)).
Definition list2 := 1 :: 3 :: 5 :: nil.
Definition list3 := [1;3;5].
```

下面我们定义一些有用的函数，方便对列表进行操纵。函数 repeat 有两个输入参数 n 和 count，它返回一个长度为 count 的列表，其中每个元素都是 n。

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O => nil
  | S count' => n :: (repeat n count')
  end.
```

函数 length 可用于计算一个列表的长度。

```
Fixpoint length (l:natlist) : nat :=
  match l with
  | nil => O
  | h :: t => S (length t)
  end.
```

函数 hd 用于取一个列表的头元素，即第一个元素。由于空列表没有头元素，我们需要显式地提供一个头元素的缺省值。函数 tl 用于取一个列表的尾列表，即除去头元素之外的剩余部分。

```
Definition hd (default : nat) (l : natlist) : nat :=
  match l with
  | nil => default
  | h :: t => h
  end.
Definition tl (l : natlist) : natlist :=
  match l with
  | nil => nil
  | h :: t => t
  end.
```

测试下面 3 条命令，得到期望的结果。注释 (* = 1 : nat *) 表明执行它前一条命令输出的值为 1，类型是 nat。

```
Compute hd 0 list3.
(* = 1 : nat *)
Compute hd 0 [].
(* = 0 : nat *)
Compute tl list3.
(* = [3;5] : natlist *)
```

函数 app 用于拼接（concatenate）两个列表。

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil => l2
  | h :: t => h :: (app t l2)
  end.
```

拼接操作有个常用的记号是"++"。

```
Notation "x ++ y" := (app x y) (right associativity, at level 60).
Compute list1 ++ list2.
(* = [1;3;5;1;3;5] : natlist *)
Compute [] ++ list1.
(* = [1;3;5] : natlist *)
```

函数 rev 的作用是对一个列表做倒置操作。

```
Fixpoint rev (l:natlist) : natlist :=
  match l with
  | nil => nil
  | h :: t => rev t ++ [h]
  end.
Compute rev list1.
(* = [5;3;1] : natlist *)
End NatList.
```

练习 2.8. 感兴趣的读者不妨在命令式语言如 *C++* 中实现一个数组或者一个单链表倒置操作，与函数 rev 作对比，领略函数式编程的代码优雅之处。

练习 2.9. 定义一个函数 createList 使得对于输入参数 n 返回一个列表记录从 *1* 到 n 的自然数。例如，(createList 5) 的返回结果为 [1;2;3;4;5]。

练习 2.10. 定义一个函数 createList' 使得对于输入参数 n 返回一个列表，形如 $[1;2;...;(n-1);n;(n-1);...;2;1]$。

练习 2.11. 定义函数 fib 用于计算斐波那契数列。例如，(fib 8) 的计算结果为 [0;1;1;2;3;5;8;13;21]。

练习 2.12. 给定一个自然数列表L，定义函数 (swap L)使得L的首尾元素互换，其它元素不变。例如，

```
Example test_swap : swap [1;2;3;4;5] = [5;2;3;4;1].
Proof. reflexivity. Qed.
```

这里前一行说明我们需要的性质，后一行告诉 *Coq* 我们将提供一个证明，所用的证明方法是等式的自反性（实际也包含对两边表达式的化简），然后 Qed 表明一个证明结束。

练习 2.13. 设计一个排序算法，把一个存储自然数的列表按升序次序进行排序。

练习 2.14. 定义类型 btree 表示这样的二叉树：其叶子节点不存储任何数，每个内部节点存储一个自然数。补充完全下面的定义，使得 (occur n t) 返回 true 当且仅当自然数 n 在二叉树 t 中出现；(countEven t) 返回这棵树上偶数出现的次数。(sum t) 返回这棵树上所有数字之和。

```
Fixpoint occur (n: nat)(t: btree) : bool
Fixpoint countEven (t : btree) : nat
Fixpoint sum (t: btree) : nat
```

2.3 规则归纳

回顾上两节的内容，我们发现像布尔类型、自然数类型、列表类型等很多数据类型的定义中都用到不同的构造子。如果现在有一种新的类型需要定义，如何寻找合适的构造子呢？为回答这个问题，我们有必要了解规则归纳的思想。

我们都知道，自然数集合 $\mathbb{N}$ 包含无限多个元素，不可能直接把它们一一列出来。为此有一种间接的方式是我们设计两条规则：

$$(O)\ \frac{}{O \in \mathbb{N}} \qquad (S)\ \frac{n \in \mathbb{N}}{S\ n \in \mathbb{N}}$$

规则 (O) 没有前提，实际上是一条公理，说明 O 属于自然数集合；规则 (S) 则说明在已知 n 是自然数的前提下，我们可以得知 $S\ n$ 也是自然数。这两条规则给出用一进制方式归纳表示自然数的方法，其中规则 (O) 是归纳的基础，规则 (S) 是归纳步骤。从前一条规则出发，反复应用第二条规则，我们可以把任何一个想要的自然数都表示出来。这种利用若干条规则来归纳定义一个数学对象的方法称为*规则归纳*（rule induction）[12]。一旦有了规则，我们很容易在 Coq 中写出归纳定义。例如，上面两条规则 (O) 和 (S) 的结论分别对应第2.1节 `nat` 类型定义中的两个构造子。

对于布尔类型，因为它只包含两个常量，所以用规则定义时只需下面两条公理作为归纳基础，无须归纳步骤。

$$(T)\ \frac{}{true \in \mathbb{B}} \qquad (F)\ \frac{}{false \in \mathbb{B}}$$

这里符号 $\mathbb{B}$ 代表集合 $\{true, false\}$。根据这两条公理在 Coq 中写出的布尔类型定义中，我们有 `true` 和 `false` 两个构造子。我们注意到，这两个构造子都不带参数，和 `nat` 类型中的 `O` 一样，而构造子 `S` 需要类型为 `nat` 的参数 `n`，因为规则 (S) 代表归纳步骤，根据自然数 n 构造下一个自然数 $S\ n$。

同理，任何一个存储自然数的长度有限的列表可由下面两条规则生成出来。

$$(N)\ \frac{}{nil \in \mathbb{L}} \qquad (C)\ \frac{n \in \mathbb{N} \quad l \in \mathbb{L}}{cons\ n\ l \in \mathbb{L}}$$

最后这条规则比之前碰到的情况略微复杂一点，因为其结论中出现 n 和 l 两个参数，而且类型不同，但我们依然可以对应写出第2.2节中 `natlist` 类型的第二个构造子 `cons`。

需要注意的是，对规则的描述有各种方式。例如，定义 1.1用 BNF 来生成 λ 项，不过我们可以等效地写出如下三条规则：

$$\frac{x \in \mathcal{V}}{x \in \Lambda} \qquad \frac{M, N \in \Lambda}{(MN) \in \Lambda} \qquad \frac{x \in \mathcal{V} \quad M \in \Lambda}{(\lambda x.M) \in \Lambda}$$

其中第一条规则是归纳的基础，它依赖于变量集合 $\mathcal{V}$；后面两条规则表示有两种情况的归纳步骤。

练习 2.15. 根据上面三条规则，在 *Coq* 中定义 `LambdaTerms` 类型，使得所有合法的 λ 项都具有这种类型。

练习 2.16. 用规则归纳的方法定义这样的二叉树的集合：叶子节点不存储任何数，而每个内部节点存储一个自然数。与练习*2.14*中定义的类型 `btree` 做比较。

2.4 多态列表

在第2.2节我们讨论了自然数列表，但如果我们希望构造一个存储布尔值的列表，则需要定义一个新的类型，比如下面的 `boolist`。

```
Inductive boollist : Type :=
  | bnil
  | bcons (b : bool) (l : boollist).
```

对这个新定义的类型，相应的操纵布尔值列表的函数如 `hd`、`tl`、`length` 等都需要重新定义。为避免这样重复性的工作，Coq 支持多态类型（polymorphic type）。例如，多态列表类型可以定义如下。

```
Inductive list (X:Type) : Type :=
  | nil
  | cons (x : X) (l : list X).
```

我们发现上述定义和 `natlist` 的定义很类似，区别在于现在参数中有一个类型变量 `X`，同时构造子 `cons` 的参数类型中 `nat` 和 `natlist` 分别被 `X` 和 `list X` 替代。不妨检查一下 `list` 的类型。Coq 告诉我们这是一个函数类型 `Type -> Type`。对于任何类型 `X`，类型 `list X` 包含那些元素类型都为 `X` 的列表。

```
Check list.
(* list : Type -> Type *)
```

两个构造子 nil 和 cons 现在也是多态的。为使用它们，我们必须提供一个类型参数，代表所建列表中元素的类型。

```
Check (nil nat).
(* nil nat : list nat *)
Check (cons bool true (nil bool)).
(* cons bool true (nil bool) : list bool *)
```

检查一下 nil 和 cons 的类型，Coq 返回的结果是 forall X : Type, list X 和 forall X : Type, X -> list X -> list X 这两种类型。

```
Check nil.
(* nil : forall X : Type, list X *)
Check cons.
(* cons : forall X : Type, X -> list X -> list X *)
```

为把之前定义的 repeat 函数改造成一个适合多态列表的多态函数，我们需要提供一个类型参数。

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=
  match count with
  | 0 => nil X
  | S count' => cons X x (repeat X x count')
  end.
```

为使用 repeat 构造一个元素为特定类型的列表，我们需要把元素的具体类型作为第一个参数提供给 repeat。

```
Compute repeat bool true 1.
(* = cons bool true (nil bool) : list bool *)
```

每次使用多态函数都需要提供类型参数，这样当然很麻烦。幸运的是，Coq 能够自动合成（synthesize）很多类型参数，为用户省去提供类型参数的工作。Coq 中的 Arguments 命令声明函数（构造子也是函数）的名称和它的参数名，用花括号把可以隐式（不必用户显式提供）的参数括起来。声明好之后，我们无需再提供类型参数。

```
Arguments nil {X}.
Arguments cons {X}.
Arguments repeat {X}.
Check cons 1 (cons 3 nil).
(* cons 1 (cons 3 nil) : list nat *)
Compute repeat true 2.
(* = cons true (cons true nil) : list bool *)
```

另一种办法是不用 Arguments，而在定义多态函数的时候直接声明某些参数为隐式的，用花括号包围起来。

```
Fixpoint repeat' {X : Type} (x : X) (count : nat) : list X :=
match count with
| 0 => nil
| S count' => cons x (repeat' x count')
end.
Compute repeat' 1 3.
(* = cons 1 (cons 1 (cons 1 nil)) : list nat *)
```

练习 2.17. 对上一节定义的函数 length、app 和 rev，重新给出多态版本的定义。

既然我们已经声明构造子 nil 和 cons 中的类型参数为隐式的，那么就可以方便地定义多态列表的一些记号，Coq 会自动推导出类型参数。

```
Notation "x :: l" := (cons x l) (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
```

练习 2.18. 定义一个多态函数 rotate，使得对输入列表 l，返回一个新列表 l'，其中 l 的最后一个元素成为 l' 的第一个元素，其它元素的位置依次后移。例如，(rotate [1;2;3;4;5]) 的计算结果为 [5;1;2;3;4]。

练习 2.19. 假设一个列表中的元素互不相同，这样可以表示一个集合。给定两个列表表示的集合L1和 L2，定义函数(product L1 L2)求它们的笛卡尔积，依旧用列表表示。例如，

```
Example product_test : product [1;2;3] [4;5]
                     = [(1,4); (1,5); (2,4); (2,5); (3,4); (3,5)].
Proof. reflexivity. Qed.
```

练习 2.20. 假设一个列表 s 中的元素互不相同，这样可以表示一个集合。定义函数 powerset 使得 (powerset s) 返回 s 的幂集，即 s 的所有子集构成的集合。

2.5 高阶函数

如果一个函数输入参数的类型不全是布尔类型、自然数类型这样的基本数据类型，而可能包含函数类型，那么这个被定义的函数被称为高阶函数（higher-order function），因为它可以操纵其它函数。

```
Definition twice {X : Type} (f : X -> X) (n : X) : X := f (f n).
```

上面定义了一个多态函数 twice，它输入一个函数 f 和另一个参数 n，然后把 f 作用两次到 n 上。由于 twice 的定义中有隐式参数 X，为查看函数 twice 的类型，我们需要在前面加"@" 符号把隐式参变成显式的。

```
Check @twice.
Compute twice S 2.
(* = 4 : nat *)
```

一个非常有用的高阶函数是过滤函数 filter。它的参数包括一个元素类型为 X 的列表和一个 X 上的谓词（predicate），即类型为 X -> bool 的函数。过滤函数的作用就是把列表中满足谓词条件的元素留下，其它不满足条件的元素被滤去了。

```
Fixpoint filter {X:Type} (test: X->bool) (l:list X) : (list X) :=
  match l with
  | [] => []
  | h :: t =>
    if test h then h :: (filter test t)
    else filter test t
end.
```

例如，如果把 `filter` 作用到谓词 `oddb` 和一个自然数列表上，返回结果中将只包含列表中的奇数。

```
Compute filter oddb [1;2;3;4;5;6].
(* = [1;3;5] : list nat *)
```

高阶函数经常和*匿名函数*（anonymous function）配合起来使用。匿名函数只在使用的时候构建，无需事先定义或给它一个名字。

```
Compute twice (fun n => n + n) 3.
(* = 12 : nat *)
```

这里出现的表达式 `(fun n => n + n)` 表示一个匿名函数，它对任何输入 n 返回结果 $n+n$。把这个函数作用两次到自然数 3，我们得到的结果为 12。

另一个非常有用的高阶函数是*映射函数*（map function），

```
Fixpoint map {X Y: Type} (f:X->Y) (l:list X) : (list Y) :=
  match l with
  | [] => []
  | h :: t => (f h) :: (map f t)
  end.
```

这个高阶函数输入两个参数：一个函数 f 和一个列表 $l = [n_1, n_2, ..., n_k]$，然后返回新的列表 $[f(n_1), f(n_2), ..., f(n_k)]$，即函数 f 被作用到列表 l 中的每一个元素上。

```
Compute map (fun n => n + 5) [1;3;5].
(* = [6;8;10] : list nat *)
```

还有一个功能强大的高阶函数是*折叠函数*（fold function），可用来对一个列表中所有元素做聚合操作。具体而言，折叠函数输入一个操作函数 `f`，一个列表 `l` 和一个缺省值 `b`，从缺省值开始利用操作 `f` 逐个遍历并操作 `l` 中的元素。例如，下面例子中把 `f` 取为求和函数 `plus`，把列表中每个元素累加到缺省值 0 上，实现对整个列表求和的功能。

```
Fixpoint fold {X Y: Type} (f: X->Y->Y) (l: list X) (b: Y): Y :=
```

```
  match l with
  | nil => b
  | h :: t => f h (fold f t b)
  end.
Compute fold plus [1;3;5] 0.
(* = 9 : nat *)
```

练习 2.21. 定义函数 partition，把输入的一个自然数列表划分为一对子列表，第一个子列表仅包含原列表中所有可被 *3* 整除的元素，第二个列表包含原列表中不能被 *3* 整除的元素。

2.6 柯里-霍华德关联

Coq 有一个预先定义的类型 Prop，表示命题的类型。例如，恒真命题 True 和恒假命题 False 都具有该类型。我们可以利用 Prop 来定义谓词，如下面例子所示。

```
Inductive odd : nat -> Prop :=
  | odd1 : odd 1
  | oddSS ( n : nat) (H : odd n) : odd (S (S n)).
```

这里定义的 odd 是一个函数，具有函数类型 nat -> Prop。它实际上是一个一元谓词，说明一个给定的自然数是否为奇数。定义中出现的归纳基础说 1 是奇数，归纳步骤说如果 n 是一个奇数，那么 S (S n) 也是一个奇数。注意这里两个构造子的类型不同并显式给出了，而且自然数作为被定义对象 odd 的参数出现在类型中。按照第2.3节所讲的规则归纳，上面的定义利用了下面两条规则：

$$(odd1)\ \frac{}{odd\ 1} \qquad (oddSS)\ \frac{odd\ n}{odd\ S\ (S\ n)}$$

对于一个关于给定自然数的命题，比如 $odd\ 5$，如果成立则可以构造一棵证明树。

$$\dfrac{\dfrac{\dfrac{}{odd\ 1}}{odd\ 3}}{odd\ 5}$$

这个例子很简单，构造的树是特殊的线性结构，但一般的证明树可以有分支结构。这棵证明树自底向上构造，依次用了两次 ($oddSS$) 规则和一次 ($odd1$) 规则。Coq 作为一个定理证明助手，可以把命题 $odd\ 5$ 写成一个性质，并按照上面证明树的思想给出一个证明过程。

```
Theorem odd5 : odd 5.
Proof. apply oddSS. apply oddSS. apply odd1. Qed.
```

上面这个性质以关键字 `Theorem` 开始，表示我们定义的一个定理，其名称为 `odd5`，证明分别以关键字 `Proof` 和 `Qed` 开始和结尾。具体证明中我们用了名为 `apply` 的证明策略（tactic），即应用已有的性质和函数等。这里应用了两次构造子 `oddSS` 和一次 `odd1`，与上面建立证明树时应用规则的过程一致。我们还可以用函数作用的语法，把证明写得更简洁一些。

```
Theorem odd5' : odd 5.
Proof. apply (oddSS 3 (oddSS 1 odd1)). Qed.
```

上面的例子说明，构造一个命题的证明对应着建立一棵证明树。由于树是一种数据结构，一棵具体的树则可视为一个数据值。既然一个证明是一个数据值，那命题又是什么呢？我们所用的记号 ":" 实际上暗含的意思是也把一个命题当作一个类型。例如，我们既可以把 "`odd1 : odd 1`" 读成为 "`odd1` 是命题 `odd 1` 的一个证明"，也可以读作 "`odd1` 是类型为 `odd 1` 的一个数据值"。这样以来，我们把逻辑和计算联系起来了，得到柯里-霍华德关联（Curry-Howard correspondence）。

证明	:	命题
数据值	:	类型

在函数式编程语言中，数据也是函数，是由程序表示的，因此柯里-霍华德关联的一种解释是：

- 类型对应于逻辑公式，即命题；
- 程序对应于逻辑证明；
- 程序求值（evaluation）对应于证明简化（simplification）。

更详细的讨论可以参见 [11]。

本节开始定义的 odd 是一元谓词，当然我们也可以用类似的归纳思想定义多元谓词。比如自然数上的小于等于关系（≤）是一个二元谓词，在 Coq 中可定义如下：

```
Inductive le : nat -> nat -> Prop :=
  | len (n : nat) : le n n
  | leS (n m : nat) (H : le n m) : le n (S m).
```

下面证明 3 小于等于 5 这样一个简单性质。

```
Theorem le3 : le 3 5.
Proof. apply leS. apply leS. apply len. Qed.
```

练习 2.22. 定义谓词 gt 表示严格大于关系，即 gt n m 成立当且仅当 n 严格大于 m。

练习 2.23. 定义谓词 subseq 表示子序列关系，即 subseq l1 l2 成立当且仅当列表 l1 中的元素依次按原来的顺序出现在列表 l2 中，但 l2 中可能穿插了其它元素。例如，[1;2;3] 是 [1;1;4;6;2;7;2;3;9] 的子序列。

2.7 归纳证明

Coq 的主要用途不在编程，而在于对一些数学或逻辑性质进行机器辅助证明。例如，著名的四色定理已经在 Coq 中得到了形式化证明。作为一个定理证明器，Coq 最擅长的是归纳证明。

举一个简单的例子，假设我们希望证明对于所有自然数 n，等式 $n*0=n$ 成立。我们把这个性质写成一个名为 mult_0_r 的定理，然后用数学归纳法证明。当 $n=0$ 时，等式左边为 $0*0$，可以简化为 0，于是等式两边相等，可用 $=$ 关系的自反性证明 $0=0$。假设当 $n=n'$ 时等式成立，即 $n'*0=0$。我们给这个条件取一个名字 IHn'。对于待证目标 $Sn'*0=0$，我们根据乘法的定义把等号左边简化为 $n'*0$，再利用条件 IHn' 从左往右重写，把待证目标转化为 $0=0$，最后用 $=$ 关系的自反性证明。把这个证明过程用 Coq 写出来，成为从 Proof 到 Qed 之间的内容，由一个个证明策略（tactic）顺序串联起来，每个证明策略就像一个 Coq 可识别的命令。

```
Theorem mult_0_r : forall n:nat,
  n * 0 = 0.
Proof.
  induction n as [| n' IHn'].
  - (* n = 0*)
    simpl. reflexivity.
  - (* n = S n' *)
    simpl. rewrite -> IHn'. reflexivity.
Qed.
```

上面的证明策略 `induction n as [| n' IHn']` 告诉 Coq 我们将对 n 做数学归纳，`IHn'` 这个条件表示当 $n = n'$ 时的归纳假设。其它策略 `simpl`、`rewrite`、`reflexivity` 分别表示表达式化简、等式重写和自反性质。

除了自然数上的数学归纳法，Coq 对所有归纳定义的类型都提供了一个归纳证明方法。例如，列表类型的定义和自然数类型非常相像，因此，我们也可以对一个列表进行结构归纳。下面的例子中列表 `l` 可以为空，也可以不为空，由表头 `h` 和表尾 `t` 拼接而成。证明部分与定理 `mult_0_r` 的证明非常类似，所以不再赘述。

```
Theorem app_nil_r : forall l : natlist,
  l ++ [] = l.
Proof. intro l. induction l as [ | h t IHt].
 - reflexivity.
 - simpl. rewrite IHt; reflexivity.
Qed.
```

练习 2.24. 完成对下面两个性质的证明：

```
Theorem plus_n_Sm : forall n m : nat,
  S (n + m) = n + (S m).


Theorem app_assoc : forall l1 l2 l3 : natlist,
  (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
```

第 3 章　OCaml

在二十世纪 70 年代，计算机科学家罗宾 · 米勒（Robin Milner）等人在英国爱丁堡大学提出通用函数式编程语言 ML（Meta Language）。在二十世纪 80 年代，多位法国计算机科学家共同努力实现了基于 ML 的语言 Caml (Categorical Abstract Machine Language)。之后又融入面向对象的编程风格，产生了 Objective Caml，即 OCaml。因此，OCaml 是完全包含面向对象编程和 ML 风格的静态类型的一门语言。2005 年微软开发的 F# 语言是 OCaml 的一个变种。

3.1　安装和使用 OCaml

为安装 OCaml，我们可以使用它的包管理器 OPAM。建议先通过 OPAM 安装好 OCaml，然后再安装 UTop，这是使用 OCaml 的一个界面。具体帮助信息请参见 OCaml 官方网站 `https://www.ocaml.org`。

我们先创建一个名为 “hello.ml” 的文件，其中只包含一行命令：

```
print_endline "Hello World!"
```

下面介绍几种编译和运行 OCaml 文件的方式。

方式一： 输入下面命令来使用解释器直接执行刚创建的文件。

```
$ ocaml hello.ml
```

结果为输出字符串 “Hello World!”。

方式二： 先把程序编译成字节码（bytecode），然后通过字节码解释器来执行。

```
$ ocamlc -o hello hello.ml
$ ocamlrun hello
```

在大部分系统中，字节码可以直接运行。

```
$ ocamlc -o hello hello.ml
$ ./hello
```

方式三： 先编译成本机可执行文件，然后运行。

```
$ ocamlopt -o hello hello.ml
$ ./hello
```

方式四： 使用 OCaml 的交互运行环境。注意在交互运行环境中编写代码时，每句命令以符号";;" 结束。

```
$ ocaml
# print_endline "Hello World!" ;;
# #use "hello.ml" ;;
# #quit;;
$
```

3.2 数据类型与函数

表达式求值 我们先进入交互式运行环境。如果输入一个表达式，例如 `1+1;;`，回车后立刻可以看到这个表达式的类型 `int` 和求值结果 `2`。

```
$ ocaml
# 1+1;;
(* - : int = 2 *)
# "Hello" ^ " World!" ;;
(* - : string = "Hello World!" *)
```

基本类型 OCaml 中用到的基本类型主要包含整型 `int`、浮点型 `float`、布尔类型 `bool`、字符 `char`、字符串 `string`、单元类型 (unit)。如果一个表达式没有一个有意义的值，比方说赋值或者循环命令，那么这个表达式就可以赋予单元类型。这个类型有一个唯一的值，记为“`()`”。在条件分支语句中，如果只有 `if ... then ...` 而没有 `else` 分支，那么类型就是 `unit`。例如，

```
if !x>0 then x := 0
```

是一条正确的语句，但

```
2 + (if !x > 0 then 1)
```

不是合法的表达式。

在 OCaml 中，类型的转换，例如整型到浮点数的类型转换，需要显式地给出，不然会编译出错。

```
# 1 + 2.5 ;;
(* Error: This expression has type float but an expression was
expected of type int *)
# 1 +. 2.5 ;;
(* Error: This expression has type int but an expression was
expected of type float *)
# (float_of_int 1) +. 2.5 ;;
(* - : float = 3.5 *)
```

这个例子中的函数 `float_of_int` 把整数 1 转换成浮点数 1.。

列表 OCaml 中的列表和 Coq 中的列表很类似，用的是相同的记号。

```
# [1; 2; 3];;
(* - : int list = [1; 2; 3] *)
# 1 :: [2; 3] ;;
(* - : int list = [1; 2; 3] *)
```

定义函数 我们可用 let 语句来定义一个函数。

```
# let average a b = (a +. b) /. 2.0;;
(* val average : float -> float -> float = <fun> *)
```

OCaml 中用记号 “+” 表示整数的加法，对于浮点数的加法，用的记号是 “+.”。类似的记号如 “-.”、“*.”、“/.” 分别表示浮点数的减法、乘法和除法运算。

```
# average 3. 4.;;
(* - : float = 3.5 *)
```

在函数 average 中，两个形式参数紧跟函数名之后。另一种写法是把形式参数放到等号右边，作为一个匿名函数的参数。

```
# let plus = fun x y -> x + y ;;
(* val plus : int -> int -> int = <fun> *)
# plus 2 3;;
```

匿名函数的使用和普通函数没有区别，直接把它作用于参数上。

```
# (fun a -> a + 5) 3 ;;
(* - : int = 8 *)
```

如果一个函数不带参数，我们定义的将是一个常函数。

```
# let v = 1 ;;
(* val v : int = 1 *)
```

注意这里的 let 用法看起来有点像命令式语言如 C 语言中的变量赋值，但实际上是不同的。直观上，上面这句命令是给 1 取了一个永久的名字 v。为详细说明这一点，考虑下面的例子。

```
# let f x = x + v ;;
(* val f : int -> int = <fun> *)
# f 2 ;;
(* - : int = 3 *)
# let v = 10 ;;
```

```
(* val v : int = 10 *)
# v ;;
(* - : int = 10 *)
# f 2 ;;
(* - : int = 3 *)
```

之前我们给 1 取名字 v，后来我们给 10 也取名字 v，因此后来 v 对应的值是 10。但是函数 f 用的是前一个名字 v，之后对 v 的改变不会影响函数 f。

局部变量 我们经常用 “let ... in ...” 结构来声明一个局部变量，而且允许嵌套使用。

```
# let a = 3 in a + 5 ;;
(* - : int = 8 *)
# let average a b =
   let sum = a +. b in
   sum /. 2.0;;
(* val average : float -> float -> float = <fun> *)
```

使用局部变量的一个重要目的是把一个复杂表达式中重复的部分用一个变量替代。下面的例子中，表达式 a +. b 出现两次。通用引入局部变量 x 来替换这个表达式，我们只需把 x 写两次，使得代码更精简。这里出现的记号 x**2 表示幂运算 x^2。

```
# let f a b = (a +. b) +. (a +. b) ** 2. ;;
(* val f : float -> float -> float = <fun> *)
# let f a b =
  let x = a +. b in
  x +. x ** 2. ;;
(* val f : float -> float -> float = <fun> *)
```

数组 与列表不同的是，我们可以直接访问并更改数组（array）中某个单元的数据值。

```
# let a = [| 1; 3; 5 |] ;;
(* val a : int array = [|1; 3; 5|] *)
# a.(0) ;;
(* - : int = 1 *)
# a.(2) <- 7 ;;
(* - : unit = () *)
# a ;;
(* - : int array = [|1; 3; 7|] *)
# let a = Array.make 5 "a" ;;
(* val a : string array = [|"a"; "a"; "a"; "a"; "a"|] *)
```

在上面的例子中，我们先创建一个数组，它包含 3 个单元。第一个单元的值为 a.(0)。命令 a.(2) <- 7 把第 3 个单元的值更改为 7。库 Array 中的函数 make 根据给定的初始值创建一个新数组。

结构或记录 同一个数组中每个元素的类型必须相同，但在一个结构(structure)或记录 (record) 中，不同元素的类型不一定相同。在下面的例子中，我们定义了 pair_of_ints 和 complex 两个结构类型。数据 x 的类型为 complex 结构，它有两个数据域 (field)，其中 x.im 表示它的第二个域。

```
# type pair_of_ints = { a: int; b : int };;
(* type pair_of_ints = { a : int; b : int; } *)
# {a=3; b=5};;
(* - : pair_of_ints = {a = 3; b = 5} *)
# type complex = {re : float; im : float} ;;
(* type complex = { re : float; im : float; } *)
# let x = { re = 1.0; im = -1.0} ;;
(* val x : complex = {re = 1.; im = -1.} *)
# x.im ;;
(* - : float = -1. *)
```

如果在一个结构的域名前面加关键字 mutable，则声明这个域是可修改的。下面例子中 person 类型的第二个域 age 是可修改的，结构 p 的类型是

person，命令 p.age <- p.age + 1 让 p.age 的值增加 1。

```
# type person = {name : string; mutable age : int} ;;
(* type person = { name : string; mutable age : int; } *)
# let p = { name = "John"; age = 20};;
(* val p : person = {name = "John"; age = 20} *)
# p.age <- p.age + 1;;
(* - : unit = () *)
# p.age;;
(* - : int = 21 *)
```

变体 如果一个类型的数据值有多种形式，用变体（variant）类型是比较合适的。下面定义 foo 为一个变体类型，它的数据值有四种情况。以第三种情况的声明 Pair of int * int 为例，它表示对应的元素由构造子 Pair 后面跟一对自然数，如 Pair (3, 4)。

```
# type foo =
  | Nothing
  | Int of int
  | Pair of int * int
  | String of string;;
(* type foo = Nothing | Int of int | Pair of int * int
          | String of string *)
# Pair (3, 4);;
(* - : foo = Pair (3, 4) *)
```

递归变体 在变体类型 foo 的定义中，等式右边用到其它类型，如 int 和 string。如果我们在等式右边允许被定义的类型出现，那这个等式将是一个递归方程。OCaml 的确允许这样定义的变体类型，称为*递归变体*（recursive variant）。下面的例子利用变体类型定义二叉树类型。一棵二叉树只有两种形式：或者只有一个叶子节点，其中存有一个自然数，或者由左右两棵子树组成。

```
# type binary_tree =
```

```
| Leaf of int
| Tree of binary_tree * binary_tree;;
(* type binary_tree = Leaf of int | Tree of binary_tree * binary_tree *)
# Leaf 3;;
(* - : binary_tree = Leaf 3 *)
# Tree (Tree (Leaf 3, Leaf 4), Leaf 5);;
(* - : binary_tree = Tree (Tree (Leaf 3, Leaf 4), Leaf 5) *)
```

变体类型实质上对应着 Coq 中归纳定义的类型，语法定义也比较相像，每个构造子代表变体的一种情况，相互之间用竖线符号隔开。

参数化的变体 目前定义的二叉树所有叶子节点上存的是自然数，当然我们可以定义一棵类似的二叉树使得叶子节点上存储字符串或者其它类型。如果不希望每次都重复类似的定义，我们可以使用*参数化的变体*（parameterized variant），在被定义的类型 `binary_tree` 前面插入一个类型变量`'a`，同时把构造子 `Leaf` 后面的类型 `int` 改成`'a`。

```
# type 'a binary_tree =
  | Leaf of 'a
  | Tree of 'a binary_tree * 'a binary_tree;;
(* type 'a binary_tree =
Leaf of 'a | Tree of 'a binary_tree * 'a binary_tree *)
# Tree (Leaf "hello", Tree (Leaf "John", Leaf "Smith"));;
(* - : string binary_tree =
Tree (Leaf "hello", Tree (Leaf "John", Leaf "Smith")) *)
# Leaf 3.1;;
(* - : float binary_tree = Leaf 3.1 *)
```

上面定义了两个二叉树的例子，其中一个为 `string binary_tree`，在叶子节点上存储字符串；另一个为 `float binary_tree`，在叶子节点上存储浮点数。换一句话说，`binary_tree` 是多态二叉树类型。

我们再举一个例子，用参数化的变体定义多态列表类型 `list`，并给出两个列表数据值，具体类型分别为 `int list` 和 `float list`。

```
# type 'a list =
  | Nil
  | Cons of 'a * 'a list;;
(* type 'a list = Nil | Cons of 'a * 'a list *)
# Nil;;
(* - : 'a list = Nil *)
# Cons (1, Nil);;
(* - : int list = Cons (1, Nil) *)
# Cons (1.1, Cons(2.1, Nil));;
(* - : float list = Cons (1.1, Cons (2.1, Nil)) *)
```

多态函数 下面定义的函数 f 是恒等函数，对任何输入 x，返回结构仍然是 x，没有对参数 x 的类型做任何规定。这是一个多态函数（polymorphic function），可以看出其类型在 OCaml 中显示为'a -> 'a，这里'a 代表一个类型变量。这个函数可以作用于一个整数值、一个布尔值、甚至是一个函数。

```
# let f x = x ;;
(* val f : 'a -> 'a = <fun> *)
# f 3 ;;
(* - : int = 3 *)
# f true ;;
(* - : bool = true *)
# f print_int ;;
(* - : int -> unit = <fun> *)
# f print_int 1 ;;
(* 1- : unit = () *)
```

如上面定义的 f 函数，如果我们不指定参数的类型，OCaml 总是尽可能自动推导出最一般的类型。在下面的例子中，compose 函数的类型比较复杂，用到'a、'b 和'c 三个类型变量。

```
# let compose f g = fun x -> f (g x) ;;
(* val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun> *)
```

引用 在 OCaml 中用“let”或者“let ... in ...”结构声明的一个变量并不是普通命令式语言中的变量，因为它的值只可读不能写。真正的变量是允许被修改的。为实现这一目的，OCaml 中用*引用*（reference）。

```
# ref 0;;
(* - : int ref = {contents = 0} *)
# let my_ref = ref 0;;
(* val my_ref : int ref = {contents = 0} *)
# my_ref := 100;;
(* - : unit = () *)
# !my_ref;;
(* - : int = 100 *)
```

在上面的例子中，变量 `my_ref` 的类型不是 `int`，而是 `int ref`，它的初始值为 0，被重新赋值为 100。每次读取这个变量的值，我们都用记号`!my_ref`。需要注意的是，在 OCaml 内部实现时，引用不是一种新类型，而是预先定义好的一个记录类型，只包含一个可变的域，即

```
type 'a ref = { mutable contents : 'a }
```

而 `ref`，`!` 和`:=` 都是*语法糖*（syntactic sugar）。

多元组 *多元组*（tuples）是非常常用的一种数据类型，不同组员之间的类型可以相同或不同。

```
# (1,2,3);;
(* - : int * int * int = (1, 2, 3) *)
# let v = (0, false, "window", 'a') ;;
(* val v : int * bool * string * char = (0, false, "window", 'a' ) *)
```

为访问一个多元组的不同组员，我们可用“let”构造。

```
# let (a, b, c, d) = v ;;
(* val a : int = 0
val b : bool = false
```

```
val c : string = "window"
val d : char = 'a'  *)
# print_string c;;
(* window- : unit = () *)
```

在作为函数参数的时候，一个多元组只算作一个参数。例如，在下面的例子中，虽然二元组 (x,y) 中有两个变量，但函数 f 只把 (x,y) 作为一个类型为 int * int 的参数。

```
# let f (x,y) = x + y ;;
(* val f : int * int -> int = <fun> *)
# f (1,2);;
(* - : int = 3 *)
```

多元组的一个重要用途是方便在函数中同时返回多个值。下面的例子中用“let rec”构造定义一个递归函数，(division n m) 的返回值为二元组 (q, r)，表示 n 被 m 除所得到的商 q 和余数 r。

```
# let rec division n m =
    if n < m then (0, n)
    else let (q, r) = division (n-m) m in
      (q + 1, r) ;;
(* val division : int -> int -> int * int = <fun> *)
```

输出和输入 OCaml 提供一些简单的打印输出函数，例如，常用的 print_int、print_string 以及 print_newline () 分别用于打印整数、字符串和回车。在下面的函数 print_pair 中，我们串行组合四个输出函数，中间用分号隔开，实现分两行打印两个数字的功能。

```
# print_int 100;;
(* 100- : unit = () *)
# let print_pair (x, y) =
print_int x; print_newline (); print_int y; print_newline () ;;
(* val print_pair : int * int -> unit = <fun> *)
```

```
# print_pair (1,2);;
(* 1
2
-: unit = () *)
```

OCaml 也提供一些从终端读输入的函数，例如 `read_in` 和 `read_line` 分别读入一个整数和一行字符。

```
# let a = read_int ()
  in print_int a;;
# let name = read_line ()
  in print_string name ;;
```

模式匹配 因为变体类型的数据通常有不同形式，在使用的时候很自然会用模式匹配来针对不同情况分别处理，这一点和 Coq 类似。举一个例子，我们先定义算术表达式类型 `expr`。一个表达式可以是字符串表示的变量，或者是由加、减、乘、除四种运算组合的表达式。

```
# type expr =
  | Plus of expr * expr (* 表示 a + b *)
  | Minus of expr * expr (* 表示 a - b *)
  | Times of expr * expr (* 表示 a * b *)
  | Divide of expr * expr (* 表示 a / b *)
  | Value of string (* "x", "y", "n", etc. *);;
```

我们接下去定义一个转换函数 `rec_to_string`，把前缀表达式转换成中缀形式，以字符串的形式显示。

```
# let rec to_string e =
  match e with
  | Plus (left, right) ->
    "(" ^ to_string left ^ " + " ^ to_string right ^ ")"
  | Minus (left, right) ->
    "(" ^ to_string left ^ " - " ^ to_string right ^ ")"
```

```
  | Times (left, right) ->
    "(" ^ to_string left ^ " * " ^ to_string right ^ ")"
  | Divide (left, right) ->
    "(" ^ to_string left ^ " / " ^ to_string right ^ ")"
  | Value v -> v;;
(* val to_string : expr -> string = <fun> *)
```

这里向上的箭头符号用于连接两个字符串。下面我们测试 `to_string` 的使用，把 `expr` 类型的表达式转换成字符串然后输出给终端。

```
# let print_expr e = print_endline (to_string e);;
(* val print_expr : expr -> unit = <fun> *)
# print_expr (Times (Value "n", Plus (Value "x", Value "y")));;
(* (n * (x + y))
- : unit = () *)
```

如果一个函数在定义中直接对最后一个参数进行模式匹配，我们有一种简单的写法。

```
# let rec length l =
    match l with
    | [] -> 0
    | h :: t -> 1 + length t ;;
```

上面这个函数可以改写为下面的形式：

```
# let rec length = function
    | [] -> 0
    | h :: t -> 1 + length t ;;
```

这里的 `function` 是一个关键字，之后不用写 `match`。

占位符 在模式匹配中，可以用下划线做占位符，匹配任何我们不关心或者缺省的情况。下面我们定义函数 `xor`，用二元组取一对布尔值，做异或运算。

```
# let xor p = match p
  with (true, false) | (false, true) -> true
  | _ -> false;;
(* val xor : bool * bool -> bool = <fun> *)
# xor (true, true);;
(* - : bool = false *)
```

选择类型 有时候我们会碰到一些函数，在大多数情况下返回某种正常类型的值，但是在特殊情况下没有返回值。例如，假设函数 `hd` 的功能是取一个列表的头元素。对于非空列表 `l`，则 `hd l` 总能返回一个值。如果 `l` 是空列表怎么办？解决方案之一是提供一个缺省的值。OCaml 给我们提供另一种方案，即利用选择类型（option type）。我们可用 `Some 20` 表示一个正常返回值，而用 `None` 表示非正常返回值。

```
# Some 20 ;;
- : int option = Some 20
# None ;;
- : 'a option = None
```

利用选择类型，我们可以定义一个可读性更好的 `hd` 函数：

```
# let hd = function
    | [] -> None
    | h :: t -> Some h ;;
val hd : 'a list -> 'a option = <fun>
```

3.3 控制结构

条件分支 在上一节中我们已经用到了条件分支结构“if ... then ... else”或者“if ... then ...”。下面再举一个例子，其中定义的函数 `max` 取两个输入参数中较大的那个。注意，因为比较运算符“>”是多态的，可以比较两个整数、浮点数、或者字符串，那么定义出来的函数 `max` 也是多态的。

```
# let max a b =
```

```
 if a > b then a else b;;
(* val max : 'a -> 'a -> 'a = <fun> *)
# max 2 3;;
(* - : int = 3 *)
# max 2.1 5.4;;
(* - : float = 5.4 *)
# max "a" "b";;
(* - : string = "b" *)
```

for 循环 OCaml 语言中的 for 循环与其它语言如 C 语言中的定义类似。下面的例子结合 for 循环和引用类型，实现对从 1 到 10 的数字求和。

```
# let sum = ref 0 in
  for i = 1 to 10 do
    sum := !sum + i
  done ;
print_int !sum;;
(* 55- : unit = () *)
```

while 循环 同样，while 循环也与一般命令式语言中的定义类似。下面的程序片段演示 while 循环和引用类型的使用，只要用户输入的字符串不以字符 'y' 开始，该程序中的循环便不结束。

```
# let flag = ref false in
  while not !flag do
    print_string "Terminate? (y/n)";
    let str = read_line () in
    if str.[0] = 'y' then
      flag := true
 done;;
(* Terminate? (y/n)n
Terminate? (y/n)abc
```

```
Terminate? (y/n)y
- : unit = () *)
```

我们再看一个例子。为判断一个数值 x 是否出现在数组 a 中，我们先用 while 循环实现这一功能。

```
# let array_mem x a =
  let len = Array.length a in
  let flag = ref false in
  let i = ref 0 in
    while !flag = false && !i < len do
      if a.(!i) = x then
        flag := true;
      i := !i + 1
    done;
    !flag ;;
(* val array_mem : 'a -> 'a array -> bool = <fun> *)
# array_mem 1 [| 3; 1; 6; 7 |];;
(* - : bool = true *)
```

上面的循环结构也可用 for 循环实现，两种实现方式的代码长度差不多。

```
# let array_mem' x a =
  let flag = ref false in
    for i = 0 to Array.length a - 1 do
      if a.(i) = x then
        flag := true
    done;
    !flag ;;
(* val array_mem' : 'a -> 'a array -> bool = <fun> *)
# array_mem' 1 [| 3; 5; 1; 7 |];;
(* - : bool = true *)
# array_mem 7 [| 3; 5; 1; 6|];;
(* - : bool = false *)
```

递归函数 函数式编程语言的重要特征之一是递归函数，用 “let rec ...” 构造。下面的函数（range a b）实现的功能是把从 a 到 b 之间的所有数字以升序次序列出，放入一个列表中；如果 a 比 b 大，则返回空表。

```
# let rec range a b =
  if a > b then []
  else a :: range (a+1) b;;
(* val range : int -> int -> int list = <fun> *)
# range 1 5;;
(* - : int list = [1; 2; 3; 4; 5] *)
```

我们可以改写这个函数，以*尾递归*（tail recursion）的方式实现。尾递归是指一个函数中最后执行的操作是递归调用。OCaml 针对尾递归有特殊优化措施，可以提高执行效率。

```
# let rec range2 a b accum =
  if b < a then accum
  else range2 a (b-1) (b :: accum);;
(* val range2 : int -> int -> int list -> int list = <fun> *)
# let range a b = range2 a b [];;
(* val range : int -> int -> int list = <fun> *)
```

下面我们用递归函数实现阿克曼函数。

```
# let rec ack m n =
  if m = 0 then n + 1
  else if n = 0 then ack (m-1) 1
  else if m > 0 && n > 0 then ack (m-1) (ack m (n-1))
  else failwith "Negative parameters" ;;
(* val ack : int -> int -> int = <fun> *)
# ack 3 3;;
(* - : int = 61 *)
```

练习 3.1. 定义递归函数 fact，求输入自然数 n 的阶乘 $n!$。

练习 3.2. 用这一节开始部分定义的 max 函数,实现新的函数 (maxlist list),求列表 list 中的最大元素。

练习 3.3. 定义一个二叉树类型，使得所有叶子节点不存储数字，而每个内部节点存储一个自然数。

1. 定义函数 size 计算一棵树的规模，即所有节点的数目；
2. 定义函数 total 计算一棵树所有节点上存储的数字总和；
3. 定义函数 maxdepth 计算一棵树的高度；
4. 定义函数 list_of_tree，中序遍历一棵树，把得到的数字序列存到一张列表中。

相互递归 在 OCaml 中定义函数时可以使用*相互递归*（mutual recursion），即同时定义多个函数且互相调用。下面的例子中我们同时定义 fact 和 fact1 两个函数，前者的定义用到后者，反之亦然。

```
# let rec fact n =
    if n = 0 then 1
    else n * fact1 n
and fact1 = fact (n-1);;
(* val fact : int -> int = <fun>
val fact1 : int -> int = <fun> *)
# fact 5;;
(* - : int = 120 *)
```

练习 3.4. 用 *for* 循环实现阶乘函数 fact，并与用递归函数定义的阶乘函数做对比。

3.4 高阶函数

在第2.5中我们介绍了 Coq 使用的高阶函数。在 OCaml 中，高阶函数的定义和使用与 Coq 中很类似，这里通过四个例子说明高阶函数的用法：映射函数、管道函数、求积分和求高阶导数。

```
# let rec map f l =
    match l with
    [] -> []
   | h :: t -> f h :: map f t ;;
let halve x = x / 2 ;;
# map halve [10; 20; 30];;
(* - : int list = [5; 10; 15] *)
```

管道操作（pipeline operator）可由高阶函数实现。

```
# let pipeline x f = f x
let (|>) = pipeline
let x = 5 |> fun x -> 2 * x ;;
```

下面定义的积分函数 integral 是一个高阶函数，实现的功能就是求某个输入函数在 [0,1] 区间的积分。

```
# let integral f =
    let n = 100 in
    let s = ref 0.0 in
    for i = 0 to n-1 do
      let x = float i /. float n in
        s := !s +. f x
    done;
    !s /. float n ;;
(* val integral : (float -> float) -> float = <fun> *)
# integral sin;;
(* - : float = 0.455486508387318301 *)
# integral (fun x -> x *. x) ;;
(* - : float = 0.328350000000000031 *)
```

下面的例子比较复杂，它尝试求一个函数的高阶导数。这里需要的准备工作包括定义一次求导函数 derivative 和把一个函数多次作用的幂运算 power，而后者的定义用到相互递归，因为它涉及函数复合的运算 compose。这些函数都定义好之后，我们可以实现 sin 函数的三阶导数。

```
# let derivative dx f = fun x -> (f (x +. dx) -. f x) /. dx ;;
(* val derivative : float -> (float -> float) -> float -> float = <fun> *)

# let rec power f n =
    if n = 0 then fun x -> x
    else compose f (power f (n-1))
and compose f g = fun x -> f (g x) ;;
(* val power : ('a -> 'a) -> int -> 'a -> 'a = <fun>
val compose : ('a -> 'a) -> ('a -> 'a) -> 'a -> 'a = <fun> *)

# let sin''' = power (derivative 1e-5) 3 sin;;
(* val sin''' : float -> float = <fun> *)
```

3.5 异常

OCaml 自身有一部分预先定义好的异常，另外还允许用户自定义异常。为抛出一个异常，可用关键字 “raise”；为捕获异常，我们用 “try ... with e_1 -> n_1 | e_2 -> n_2” 的构造，即遇到异常 e_1 便返回结果 n_1，遇到 e_2 便返回结果 n_2。

```
# 1 / 0;;
(* Exception: Division_by_zero. *)
# try
1 / 0
with Division_by_zero -> 13;;
(* - : int = 13 *)
# exception My_exception ;;
(* exception My_exception *)
# try
 if true then
  raise My_exception
 else 0
with My_exception -> 13;;
```

```
(* - : int = 13 *)
```

用户自定义的异常可以带参数，以便返回更多错误信息。

```
# exception Exception1 of string;;
# exception Exception2 of int * string;;
# let except b =
  try
    if b then
      raise (Exception1 "aaa")
    else
      raise (Exception2 (13, "  bbb"))
  with Exception1 s -> "Exception1: "^s
   |Exception2 (n, s) -> "Exception2 "^string_of_int n ^ s ;;
(* val except : bool -> string = <fun> *)
# except true;;
(* - : string = "Exception1: aaa" *)
# except false;;
(* - : string = "Exception2 13 bbb" *)
```

3.6 排序

为把一个列表中的元素进行排序，我们介绍两种排序方式：插入排序和快速排序。

插入排序 下面的函数 `insertion_sort` 是一个多态函数，可以把给定列表中的元素进行升序排列。

```
# let rec insertion_sort = function
    | [] -> []
    | x :: l -> insert x (insertion_sort l)
and insert a = function
     | [] -> [a]
```

```
    | x :: l -> if a < x then a :: x :: l
                else x :: insert a l ;;
(* val insertion_sort : 'a list -> 'a list = <fun>
val insert : 'a -> 'a list -> 'a list = <fun> *)

# insertion_sort [2; 1; 3; 0];;
(* - : int list = [0; 1; 2; 3] *)

# insertion_sort ["how"; "are"; "you"] ;;
(* - : string list = ["are"; "how"; "you"] *)
```

快速排序 下面定义函数 quicksort 来实现快速排序，注意这里嵌套使用 “let rec ...” 和 “let ...” 构造。局部定义的函数 partition 起着重要作用，把列表 list 中所有小于 h 的元素放到左边，其余元素放到元素 h 的右边，然后对这两个子列表递归进行快速排序再拼接起来。

```
# let rec quicksort l =
    match l with
    | [] -> []
    | [x] -> [x]
    | h::tl ->
      let rec partition list =
        match list with
        | [] -> [], []
        | a::list' ->
          let (left, right) = partition list' in
            if a < h then. (a :: left, right)
            else (left, a :: right)
      in let (l, r) = partition tl in
           (quicksort l) @ (h :: quicksort r);;
(* val quicksort : 'a list -> 'a list = <fun> *)
# quicksort [2; 5; 1; 7; 3; 9; 3; 0; 10];;
```

```
(* - : int list = [0; 1; 2; 3; 3; 5; 7; 9; 10] *)
```

3.7 队列

队列（queue）是一种常用的数据结构。我们目前接触到与之最接近的数据结构是列表，事实上，如果有两个列表的话，就足以实现一个队列。例如，如果列表 `i = [1;2;3]`，列表 `o = [6;5;4]`，我们假想它们尾部相连，则表示一个队列，其中元素为 `[1;2;3;4;5;6]`。每次入队的元素都放在列表 `i` 做头元素，出队的元素取列表 `o` 的头元素。只要 `o` 非空，出队就不用做任何特殊操作；如果 `o` 为空，我们需要把 `i` 中的元素倒置后作为新的 `o`，同时新的 `i` 为空。只要当 `i` 和 `o` 均为空，我们的列表才为空，不能完成出队操作。

```
# let rec rev l =
    match l with
     | [] -> []
     | h :: t -> rev t @ [h] ;;
(* val rev : 'a list -> 'a list = <fun> *)

# let create () = [], [];;
(* val create : unit -> 'a list * 'b list = <fun> *)

# let push x (i, o) = (x :: i, o) ;;
(* val push : 'a -> 'a list * 'b -> 'a list * 'b = <fun> *)

# let pop q =
  let (i, o) = q in
    match o with
    | x :: o' -> x, (i, o')
    | [] -> match rev i with
            | x :: i' -> x, ([], i')
            | [] -> raise Empty ;;
(* val pop : 'a list * 'a list -> 'a * ('a list * 'a list) = <fun> *)
```

```
# push 2 ([1],[]);;
(* - : int list * 'a list = ([2; 1], []) *)
# pop ([2; 1], []);;
(* - : int * (int list * int list) = (1, ([], [2])) *)
```

练习 3.5. 给定两个表示集合的列表l1和l2，定义函数(subset l1 l2)表示子集包含关系，即该函数返回true当且仅当l1是l2的子集。例如，

```
# subset [1;2;3] [3;4;5;2;6;1];;
- : bool = true
```

练习 3.6. 定义函数(noredundancy l)来移去列表 l 中所有冗余的元素。例如，

```
# noredundancy [1;2;1;3;4;3];;
- : int list = [2; 1; 4; 3]
```

练习 3.7. 定义一个多态类型'a tree 表示一棵树，其形式有三种：

- 一棵空树；
- 一个叶子节点，有一类型为'a 的标号；
- 一个内部节点，标号为'a 类型，且有两个类型为'a tree 的儿子。

这个类型声明应该允许下面这棵树被构造出来：

```
# let tree1 = Node (5, Node (4, Leaf 3, Empty),
                       Node (8, Node (7, Leaf 6, Empty),
                                Node (9, Empty, Leaf 10))) ;;
```

练习 3.8. 定义函数 labels，使得 (labels t) 把类型为'a tree 的树 t 上叶子和内部节点的标号列出来，次序依照这棵树中序遍历的次序。例如，

```
# labels tree1;;
(* - : int list = [3; 4; 5; 6; 7; 8; 9; 10] *)
```

练习 3.9. 定义函数 replace x y t，其中 t 是类型为'a tree 的树，x 和 y 是类型为'a 的值。该函数返回一棵与 t 相同的树，但是所有标号 x 被替换为标号 y。例如，

```
# let tree2 = replace 4 40 tree1;;
 (* val tree2 : int tree =
  Node (5, Node (40, Leaf 3, Empty),
           Node (8, Node (7, Leaf 6, Empty),
                    Node (9, Empty, Leaf 10))) *)
# labels tree2;;
 (* - : int list = [3; 40; 5; 6; 7; 8; 9; 10] *)
```

练习 3.10. 定义函数 replaceEmpty y t，其中 y 和 t 都是类型为'a tree 的树。该函数返回一棵与 t 相同的树，但是每课 Empty 子树被 y 替代。例如，

```
# let tree3 = replaceEmpty (Node (12, Leaf 11, Leaf 13)) tree1 ;;
(* val tree3 : int tree =
   Node (5, Node (4, Leaf 3, Node (12, Leaf 11, Leaf 13)),
    Node (8, Node (7, Leaf 6, Node (12, Leaf 11, Leaf 13)),
     Node (9, Node (12, Leaf 11, Leaf 13), Leaf 10))) *)
# labels tree3;;
(* - : int list =
[3; 4; 11; 12; 13; 5; 6; 7; 11; 12; 13; 8; 11; 12; 13; 9; 10] *)
```

练习 3.11. 定义函数 mapTree f t，其中 t 是类型为'a tree 的树，f 是一个函数，作用于 t 上每一个节点 Node，Leaf 和 Empty，得到的最终结果为另一棵树。例如，假如函数 increment 定义如下：

```
let increment t =
  match t with
  | Empty -> Leaf 0
  | Leaf a -> Leaf (a+1)
  | Node (a, l, r) -> Node (a+1, l, r) ;;
```

那么我们有

```
# let tree4 = mapTree increment tree1;;
(* val tree4 : int tree =
  Node (6, Node (5, Leaf 4, Leaf 0),
   Node (9, Node (8, Leaf 7, Leaf 0), Node (10, Leaf 0, Leaf 11))) *)
# labels tree4;;
(* - : int list = [4; 5; 0; 6; 7; 8; 0; 9; 0; 10; 11] *)
```

练习 3.12. 定义一个多态函数 sortTree，给定一个'a list tree（每个节点的标号为一个元素类型为'a 的列表），返回一棵与原来相同的树，但是每个节点的标号已经排好序。请在 sortTree 中应用 mapTree 函数。例如，我们先创建一棵 int list tree 然后调用 sortTree。

```
# let tree5 = Node ([1;5;6;8], Leaf [1;2;3;4],
  Node ([12;4;16;13], Empty, Leaf [0;2;5;7])) ;;
(* val tree5 : int list tree =
  Node ([1; 5; 6; 8], Leaf [1; 2; 3; 4],
    Node ([12; 4; 16; 13], Empty, Leaf [0; 2; 5; 7])) *)
# labels tree5;;
(* - : int list list =
[[1; 2; 3; 4]; [1; 5; 6; 8]; [12; 4; 16; 13]; [0; 2; 5; 7]] *)
# let tree6 = sortTree tree5;;
(* val tree6 : int list tree =
  Node ([1; 5; 6; 8], Leaf [1; 2; 3; 4],
    Node ([4; 12; 13; 16], Empty, Leaf [0; 2; 5; 7])) *)
# labels tree6;;
(* - : int list list =
[[1; 2; 3; 4]; [1; 5; 6; 8]; [4; 12; 13; 16]; [0; 2; 5; 7]] *)
```

3.8 模块

为了便于管理大段代码，引入模块是有必要的。下面的例子定义了一个名为 M 的模块，其中有一个值为 1 的变量 x。为了在模块外访问这个变量，我们用

M.x。如果用 open M 把这个模块打开，则可以直接访问变量 x。

```
# module M = struct let x = 1 end ;;
module M : sig val x : int end
# M.x ;;
- : int = 1
# open M;;
# x;;
- : int = 1
```

为描述一组相关的模块，我们可以用模块类型（module type）。下面的定义给出栈的模块类型：

```
# module type Stack = sig
  type 'a stack
  val empty    : 'a stack
  val is_empty : 'a stack -> bool
  val push     : 'a -> 'a stack -> 'a stack
  val peek     : 'a stack -> 'a
  val pop      : 'a stack -> 'a stack
end ;;
```

这里定义的模块类型的名字是 Stack。等号右边部分从 sig 到 end 是一个签名（signature），由一系列的声明组成。下面给出一个用列表实现栈的具体模块 ListStack。等号右边从 struct 到 end 匹配 Stack 的签名，为签名中所有声明的（甚至更多）名字提供定义。

```
# module ListStack : Stack = struct
  type 'a stack = 'a list
  let empty = []
  let is_empty s = (s = [])

  let push x s = x :: s
```

```
  let peek = function
    | [] -> failwith "Empty"
    | x::_ -> x

  let pop = function
    | [] -> failwith "Empty"
    | _::xs -> xs
end ;;
```

模块类型 Stack 的签名中声明的类型'a stack 是抽象的（abstract），即任何实现 Stack 类型的模块都有一个名为'a stack 的类型，但在模块内具体被定义成什么类型在模块外是不可见的。例如，如果在 UTop 中查看 ListStack.empty 的类型，我们只能看到 <abstr>，表明这个类型的值已经被抽象。

```
# ListStack.empty;;
- : 'a ListStack.stack = <abstr>
```

下面这个模块 VariantStack 用变体类型而不是列表来实现栈。

```
# module VariantStack : Stack = struct
  type 'a stack =
  | Empty
  | Elem of 'a * 'a stack

  let empty = Empty
  let is_empty s = s = Empty
  let push x s = Elem (x, s)
  let peek = function
    | Empty -> failwith "Empty"
    | Elem(x,_) -> x
  let pop = function
    | Empty -> failwith "Empty"
```

```
    | Elem(_,s) -> s
end ;;
```

练习 3.13. 给定下面表示集合操作的模块类型 Set，提供基于列表的两个模块实现，其中一个允许列表中有重复元素，另一个不允许重复元素。

```
# module type Set = sig
  type 'a t
  val empty : 'a t
  val mem   : 'a -> 'a t -> bool
  val add   : 'a -> 'a t -> 'a t
end ;;
```

3.9 函子

函子（functor）是带参数的模块，可看作一个从模块到模块的函数。下面定义的函子 Pushone 把一个 Stack 类型的模块转换成另一个模块类型为 sig val x : int end 的模块。

```
# module Pushone (S : Stack) = struct
  let x = S.peek (S.push 1 S.empty)
end ;;
module Pushone : functor (S : Stack) -> sig val x : int end
```

一种等效的写法是用关键字 functor 创建一个匿名函子，就像用关键字 fun 创建一个匿名函数。

```
# module Pushone = functor (S : Stack) ->
  struct let x = S.peek (S.push 1 S.empty) end ;;
```

有了函子 Pushone,我们接下来把它作用到参数 ListStack 和 VariantStack 上，很轻松地创建 A 和 B 两个新模块。

```
# module A = Pushone (ListStack);;
# module B = Pushone (VariantStack);;
# A.x;;
```

```
- : int = 1
# B.x;;
- : int = 1
```

OCaml 提供一个语言特征称为 include，可以进行代码复用。比如，它可以使得一个模块包含另一个模块所定义的所有值。下面的例子结合 include 与函子，目的是把给定的任何 Stack 类型的模块转换为一个扩展模块，增加一个保存栈元素的值 x。

```
# module Addone (S : Stack) = struct
  include S
  let x = S.push 1 S.empty
end ;;
module Addone :
  functor (S : Stack) ->
    sig
      type 'a stack = 'a S.stack
      val empty : 'a stack
      val is_empty : 'a stack -> bool
      val push : 'a -> 'a stack -> 'a stack
      val peek : 'a stack -> 'a
      val pop : 'a stack -> 'a stack
      val x : int stack
    end
```

我们利用 Addone 创建模块 A 和 B，它们都具有各自原始模块中的值 peek 以及新增加的值 x。

```
# module A = Addone(ListStack);;
# module B = Addone(VariantStack);;
# A.peek A.x;;
- : int = 1
# B.peek B.x;;
- : int = 1
```

3.10 单子

单子（Monad）是来自范畴论的一个概念，意大利学者优吉尼奥 · 莫及（Eugenio Moggi）把它引入到函数式程序设计 [7]，用于模拟计算。通常，一个计算可看作一个数学函数，把输入映射到输出，同时可能引起副作用，例如打印输出到屏幕。单子为副作用提供一种抽象处理方法，使得副作用只按照受控制的次序发生。

在 OCaml 中，一个单子是满足两个性质的一个模块。首先，它的签名如下：

```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
end
```

其次，它遵循三条定律，我们放在本节最后讨论。

上面的签名中有两个操作，其中 `return` 的直观意义是把一个值放进一个盒子，这可以从它的类型看出来，把`'a` 类型的输入变成`'a t` 类型的输出。另一个操作 `bind` 需要用到两个输入参数：一个类型为`'a t` 的盒子，以及一个能把类型为`'a` 不带盒子的值转换成一个类型为`'b t` 的盒子的函数。`bind` 先把第一个参数中的值从盒子中取出，然后把第二个参数作用上去，返回最终结果。从计算的角度看，`bind` 操作把副作用一个接一个串联起来。以打印输出为例，`bind` 操作确保一个接一个的字符串以正确的次序输出。通常我们把 `bind` 写成中缀运算符 `>>=`，即单子的签名定义可改写如下：

```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
end
```

下面我们以除法运算为例讨论单子的使用。OCaml 内置的整数除法运算 `(/) : int -> int -> int` 定义在标准库中。如果第二个输入参数为零，则

产生一个错误。现在我们考虑另一种解决方案，运用第3.2节介绍的选择类型，修改除法运算函数的类型为 (/) : int -> int -> int option，目的是当除数为零时返回结果 None。对单个除法运算而言，第二种方案的确比原先的方案好，但是如果和其它加法、减法、乘法等整数运算放在一起使用会引起一个问题。例如，表达式 1 + (2/1) 将不能通过类型检查，原因在于我们不能把类型为 int 的数值加到类型为 int option 的值上去。为解决这个新问题，我们可以把所有整数运算都改成输入输出均为 int option 类型的运算。于是我们把整数运算“升级”(upgrade) 为一个 int option 类型的整数运算，处理的值可能为整数 (形式为 Some n，其中 n 为一个整数)，也可能为 None。形象地说，这些值好比一个个盒子，每个盒子可能装有一个整数，也可能是空的。下面我们用单子的思想实现对 int option 类型的运算。

第一个单子操作是 return，实现很容易。

```
let return (n : int) : int option =
  Some n
```

第二个单子操作是 bind，对于输入为 int option 类型的值，我们分两种情况处理：如果是 Some n 的形式，我们对 n 做正常的整数运算；如果是 None，我们的返回结果仍然是 None。

```
let bind (x : int option) (op : int -> int option) : int option =
  match x with
  | None -> None
  | Some n -> op n


let (>>=) = bind
```

下面定义 upgrade 函数，把一个类型为 int -> int -> int option 的运算升级为一个类型为 int option -> int option -> int option 的运算。

```
let upgrade (op : int -> int -> int option)
            (x : int option) (y : int option) =
  x >>= fun n ->
  y >>= fun m ->
```

```
  op n m
```

最后三行的直观意思是：先从盒子 x 中取一个数值 n，再从盒子 y 中取一个数值 m，然后对 n 和 m 进行 op 运算，把结果作为返回值。

基于前面的准备工作，我们可以把整数四则运算升级到 int option 类型的数值上。

```
let return2 op n m =
  return (op n m)
let ( + ) = upgrade (return2 Stdlib.( + ))
let ( - ) = upgrade (return2 Stdlib.( - ))
let ( * ) = upgrade (return2 Stdlib.( * ))

let div (x : int) (y : int) : int option =
  if y = 0 then None
  else return (Stdlib.( / ) x y )

let ( / ) = upgrade div
```

让我们看两个 int option 类型表达式的求值。

```
# Some 1 + Some 2 * (Some 2 / Some 1);;
- : int option = Some 5

# Some 2 - Some 2 / Some 0 ;;
- : int option = None
```

如果我们忽略掉 return 和 bind 操作中的类型标注，可以得到对 Monad 签名的一个实现模块。

```
module Maybe : Monad = struct
  type 'a t = 'a option

  let return n = Some n
```

```
  let (>>=) x op =
    match x with
    | None -> None
    | Some n -> op n
end
```

另外，我们下面三条 Monad 定律需要被满足：

- 定律 1: `return x >>= f` 与 `f x` 行为等价；
- 定律 2: `m >>= return` 与 `m` 行为等价；
- 定律 3: `(m >>= f) >>= g` 与 `m >>= (fun x -> f x >>= g)` 行为等价。

这里的行为等价意思是指两个表达式求值的最终结果相同，否则两者都产生无限循环或者都报相同的错误。定律 1 说把绑定（bind）一个有副作用的值给一个函数等效于直接把那个值传给函数。定律 2 说把绑定一个有副作用的值给 `return` 不产生任何效果。定律 3 大致是说绑定操作具有顺序复合性质。

练习 3.14. 假设我们想记录函数的调用情况，每一次调用除了返回正常的函数输出值，还要记录一条消息。已知下面两个函数：

```
let double x = 2 * x

let log (name : string) (f : int -> int) : int -> int * string =
  fun x -> (f x, Printf.sprintf "Called %s on %i; " name x)
```

(1) 实现一个 `bind` 操作，类型为 `int * string -> (int -> int * string) -> (int * string -> int * string)`

(2) 利用上面的 `log` 和 `bind` 两个函数实现另一个函数 `loggable`；如果把 `(loggable "double" double)` 作用两次到 `(1, "")` 上会产生输出 `(4, "Called double on 1; Called double on 2;")`。

(3) 实现一个类型为 `Monad` 的模块。

第 4 章　部分习题参考答案

4.1　第1章练习题

练习*1.9.*

$$
\begin{array}{rcl}
\textbf{\textit{pred}}\ \bar{0} & \equiv & (\lambda nfx.\ n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u))\bar{0} \\
& \to_\beta & \lambda fx.\bar{0}(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u) \\
& \to_\beta & \lambda fx.\ (\lambda x.x)(\lambda u.x)(\lambda u.u) \\
& \to_\beta & \lambda fx.\ (\lambda u.x)(\lambda u.u) \\
& \to_\beta & \lambda fx.\ x \\
& \equiv & \bar{0}
\end{array}
$$

我们先用数学归纳法证明：

$$
(\lambda gh.h(gf))^n(\lambda h.hx) \longrightarrow_\beta^* \lambda h.h(f^n x) \qquad \text{对所有 } n \geq 0 \text{ 都成立。} \qquad (4.1.1)
$$

对于 $n = 0$ 的情况，显然有上述性质。假设对于 $k \geq 0$ 该性质成立，我们考虑 $n = k + 1$ 的情况。

$$
\begin{array}{rcll}
(\lambda gh.h(gf))^{k+1}(\lambda h.hx) & \equiv & (\lambda gh.h(gf))((\lambda gh.h(gf))^k(\lambda h.hx)) & \\
& \to_\beta & (\lambda gh.h(gf))(\lambda h.h(f^k x)) & \text{据归纳假设} \\
& \to_\beta & \lambda h.h((\lambda h.h(f^k x))f) & \\
& \to_\beta & \lambda h.h(f(f^k x)) & \\
& \equiv & \lambda h.h(f^{k+1}x) &
\end{array}
$$

现在我们可以证明

$$\begin{aligned}
\textbf{\textit{pred}}\ \overline{n+1} &\equiv (\lambda nfx.\ n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u))\overline{n+1}\\
&\to_\beta \lambda fx.\overline{n+1}(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)\\
&\to_\beta \lambda fx.\ (\lambda x.(\lambda gh.h(gf))^{n+1}x)(\lambda u.x)(\lambda u.u)\\
&\to_\beta \lambda fx.\ (\lambda gh.h(gf))^{n+1}(\lambda u.x)(\lambda u.u)\\
&\equiv \lambda fx.\ (\lambda gh.h(gf))^{n}((\lambda gh.h(gf))(\lambda u.x))(\lambda u.u)\\
&\to_\beta \lambda fx.\ (\lambda gh.h(gf))^{n}(\lambda h.h((\lambda u.x)f))(\lambda u.u)\\
&\to_\beta \lambda fx.\ (\lambda gh.h(gf))^{n}(\lambda h.hx)(\lambda u.u)\\
&\to_\beta^* \lambda fx.(\lambda h.h(f^nx)(\lambda u.u)) \qquad \textit{by (4.1.1)}\\
&\to_\beta \lambda fx.(\lambda u.u)(f^nx)\\
&\to_\beta \lambda fx.f^nx\\
&\equiv \bar{n}
\end{aligned}$$

□

练习*1.12.*

$$\begin{aligned}
YF &\equiv (\lambda f.\ ((\lambda x.\ (f(xx)))(\lambda x.\ (f(xx)))))F\\
&\to_\beta (\lambda x.\ (F(xx)))(\lambda x.\ (F(xx)))\\
&\to_\beta F((\lambda x.\ (F(xx)))(\lambda x.\ (F(xx))))\\
&\leftarrow_\beta F((\lambda f.\ ((\lambda x.\ (f(xx)))(\lambda x.\ (f(xx)))))F)\\
&\equiv F(YF)
\end{aligned}$$

□

4.2 第2章练习题

练习*2.3.*

```
Fixpoint div2021 (n : nat ) : bool :=
  match n with
  | O => true
  | S n' => if leb n 2020 then false
            else div2021 (n' - 2020)
  end.
```

```
Example div2021_test1: div2021 4042 = true.
Proof. reflexivity. Qed.

Example div2021_test2: div2021 2027 = false.
Proof. reflexivity. Qed.
```

练习*2.6.* 下面这个定义可以被 *Coq* 接受。

```
Fixpoint Ackermann (m n: nat) {struct m} : nat :=
  match m with
    | O => S n
    | S m' =>  let fix Ackermann' (n : nat) {struct n} : nat :=
                 match n with
                  | O => Ackermann m' 1
                  | S n' =>  Ackermann m' (Ackermann' n')
                 end
               in Ackermann' n
  end.

Example testAck : Ackermann 2 10 = 23.
Proof. simpl. reflexivity. Qed.
```

练习*2.12.*

```
Definition swap (l : list nat) : list nat :=
  match l with
  | [] => []
  | h :: t => match rev t with
              | [] => [h]
              | h' :: t' => h' :: rev t' ++ [h]
              end
  end.
```

练习*2.13.*

```
Fixpoint sort (L : list nat) : list nat :=
  match L with
  | [] => []
  | h :: tl =>
         let fix insert (x : nat)(l : list nat) : list nat :=
           match l with
           | [] => [x]
           | a :: l' => if (leb x a) then x :: l
                        else a :: insert x l'
           end
         in insert h (sort tl)
  end.

Example test_sort : sort [2;4;1;6;9;6;4;1;3;5;10] =
                         [1;1;2;3;4;4;5;6;6;9;10].
Proof. reflexivity. Qed.
```

练习*2.20.*

```
Fixpoint insert_list (x : nat) (l : list (list nat))
                                  : list (list nat) :=
  match l with
  | [] => []
  | h :: tl => (x :: h) :: insert_list x tl
  end.

Fixpoint powerset (L : list nat) : list (list nat) :=
  match L with
  | [] => [[]]
  | h :: tl => let pl := powerset tl in
               pl ++ insert_list h pl
```

```
  end.

Example test_powerset1: powerset [1;2;3] = [[ ]; [3]; [2]; [2; 3];
       [1]; [1; 3]; [1; 2]; [1; 2; 3]].
Proof. reflexivity. Qed.

Example test_powerset2: powerset [1;2;3;4] = [[ ]; [4]; [3]; [3; 4];
       [2]; [2; 4]; [2; 3]; [2; 3; 4];
       [1]; [1; 4]; [1; 3]; [1; 3; 4];
       [1; 2]; [1; 2; 4]; [1; 2; 3];
       [1; 2; 3; 4]].
Proof. reflexivity. Qed.
```

练习*2.23.*

```
Inductive subseq : list nat -> list nat -> Prop :=
  | sub1 l : subseq nil l
  | sub2 l1 l2 a (H : subseq l1 l2) : subseq l1 (a :: l2)
  | sub3 l1 l2 a (H : subseq l1 l2) : subseq (a :: l1) (a :: l2).
```

4.3 第3章练习题

练习*3.5.*

```
# let rec subset l1 l2 =
   match l1 with
   | [] -> true
   | h :: t -> let rec member l =
                 match l with
                 | [] -> false
                 | h' :: t' -> h = h' || member t'
                 in member l2 && subset t l2 ;;
```

练习*3.6.*

```
# let rec noredundancy l =
  match l with
  | [] -> []
  | h :: t -> let rec member l' =
                match l' with
                | [] -> false
                | h' :: t' -> h = h' || member t'
              in if member t then noredundancy t
                 else h :: noredundancy t ;;
```

练习*3.7.*

```
# type 'a tree =
  | Empty
  | Leaf of 'a
  | Node of 'a * 'a tree * 'a tree ;;
```

练习*3.8.*

```
# let rec labels t =
    match t with
    | Empty -> []
    | Leaf x -> [x]
    | Node (x, l, r) -> labels l @ [x] @ labels r;;
```

练习*3.9.*

```
# let rec  replace x y t =
    match t with
    | Empty -> Empty
    | Leaf a ->  if x = a then Leaf y else Leaf a
    | Node (a,l,r) -> Node ((if x = a then y else a),
                            replace x y l, replace x y r);;
```

练习*3.10.*

```
# let rec replaceEmpty y t =
    match t with
    | Empty -> y
    | Leaf a -> Leaf a
    | Node (a,l,r) -> Node (a, replaceEmpty y l, replaceEmpty y r)
```

练习*3.11.*

```
# let rec mapTree f t =
    match t with
    | Empty -> f Empty
    | Leaf a -> f (Leaf a)
    | Node (a, l, r) -> f (Node (a, mapTree f l, mapTree f r));;
```

练习*3.12.*

```
# let sortTree t =
    let sortNode t' =
      match t' with
      | Empty -> Empty
      | Leaf a -> Leaf (sort a)
      | Node (a,l,r) -> Node (sort a, l, r)
  in
    mapTree sortNode t ;;
```

练习*3.14.*

(1)
```
let bind (m : int * string) (f : int -> int * string)
             : int * string =
      let (x, s1) = m in
      let (y, s2) = f x in
      (y, s1 ^ s2)

    let (>>=) = bind
```

(2)
```
let loggable (name : string) (f : int -> int)
               : int * string -> int * string =
  fun m ->
    m >>= fun x ->
    log name f x
```

经过测试，loggable 满足要求。

```
# let (>>) f g x = g (f x)

# ((loggable "double" double)
    >> (loggable "double" double)) (1, "");;
- : int * string = (4, "Called double on 1; Called double on 2;")
```

(3)
```
module Log : Monad = struct
  type 'a t = 'a * string

  let return x = (x, "")

  let (>>=) m f =
    let (x, s1) = m in
    let (y, s2) = f x in
    (y, s1 ^ s2)
end
```

参 考 文 献

[1] Barendregt H P. The Lambda Calculus, Its Syntax and Semantics. North Holland. 1985.

[2] Barendregt H P. Lambda Calculi with Types. In Abramsky, S. (ed.). Background: Computational Structures. Handbook of Logic in Computer Science. 2. Oxford University Press. pp. 117–309. 1992.

[3] 陈钢，张静. OCaml 语言编程基础教程. 北京: 人民邮电出版社, 2018.

[4] Clarkson M R et al. Functional Programming in OCaml. `https://www.cs.cornell.edu/courses/cs3110/2019sp/textbook/`

[5] Girard J, Lafont Y, Taylor P. Proofs and Types. Cambridge University Press. 1989.

[6] Minsky Y, Madhavapeddy A, Hickey J. Real World OCaml: Functional Programming for the Masses. O'Reilly Media. 2013.

[7] Moggi E. Computational Lambda-Calculus and Monads. In Proceedings of the 4th Annual Symposium on Logic in Computer Science, pages 14-23. IEEE Computer Society, 1989.

[8] Paulin-Mohring C. Introduction to the Calculus of Inductive Constructions. In Woltzenlogel Paleo, B., Delahaye, D. (eds.). All about Proofs, Proofs for All, vol. 55, College Publications, 2015, Studies in Logic (Mathematical logic and foundations).

[9] Pierce B C et al. Software Foundations (Volume 1). `https://softwarefoundations.cis.upenn.edu/lf-current/index.html`

[10] Selinger P. Lecture Notes on the Lambda Calculus. Lulu.com. 2018.

[11] Wadler P. Propositions as types. Communications of the ACM 58 (12): 75-84, 2015.

[12] Winskel G. The Formal Semantics of Programming Languages: an Introduction. The MIT Press, Cambridge. 1993

索　引