第 1 章　引　言

1.1 基本概念

1.1.1 程序设计语言的建模

对程序设计语言进行数学分析通常是从程序设计语言的建模开始，而且模型语言的设计一般要突出感兴趣的程序构造，忽略一些无关的细节。例如，若想分析过程调用机制，可先设计一个简单的语言，它的主要构造是过程声明和调用，然后分析该简化语言。这种方法不仅对分析现行的语言有用，而且有助于新语言的设计。因为实际的程序设计语言是非常庞大和复杂的，语言的设计必须包括仔细地和分离地考虑各个子语言。从简化语言可能会得出一些错误的印象，因此在进行程序设计语言的理论分析和把理论分析用于实际情况时，不要忘记所做的简化及这些简化对结论的影响。

20 世纪 60 年代就已经知道，一个复杂的程序设计语言可以被形式化为两部分：一部分是能抓住该语言本质机制的一个非常小的核心演算—λ演算（*lambda calculus*）；另一部分是一组导出形式，它们的行为可以通过把它们翻译成λ演算来理解。通过这种方式来理解语言要深刻得多。

λ演算是 20 世纪 20 年代确立的一种形式系统，它源于可计算理论，是奠定程序设计语言中函数定义和命名约定的基本机制。在这种系统中，所有的计算都归约到函数定义和函数应用这样的基本操作。20 世纪 60 年代以来，λ演算已广泛用于规范程序设计语言的特征、类型系统、设计和实现的研究中。它的重要性在于它同时可以看成一种简单的程序设计语言（用于描述计算）和看成一种数学对象（有关它的一些严格陈述可以证明）。

本书将用类型化λ演算（*typed lambda calculus*）的框架来研究程序设计语言的各种概念。用不同方式来扩充基本的类型化λ演算，可以设计出多种模型语言，它们包含历史的、现代的、甚至将来的语言特征。盯在类型化λ演算上的一个优点是，所建立的理论具有某种程度的“模块性”。类型化λ演算的许多扩充可以组合在一起，一般来说这种组合不会出现出乎意料的叠加影响（当然也会有例外）。例如，在分别调查了多态性和记录后，很容易定义含多态性和记录的语言，并且指出它的很多性质。

本书研究程序设计语言的概念和特征的目的是透过表面的语法，对各种程序短语（表达式、命令和声明等）理解到一个适当详细的程度。 本章介绍一个非常简单的、以自然数和布尔值作为基本类型的、基于类型化λ演算的语言，介绍该语言的语法、操作语义和它在程序设计中的能力。该语言是后面几章介绍的可计算函数编程（*Programming Computable Functions*，简称 PCF）语言的一个简化版本。通过这个简要介绍，读者可以对本书将要采用的技术和方法有一个浅显的了解。

本章的主要议题如下：

（1）λ表示法和λ演算系统概述；

（2）类型和类型系统的扼要讨论；

（3）基于表达式的归纳，基于证明的归纳和良基归纳。

本书以后各章也按此风格，即每一章有一节导言，导言中包含该章的主要议题。

1.1.2 λ表示法

在描述、分析和实现程序设计语言时，λ演算已被证明是非常有用的。稍加练习，读者

很快就会熟悉这种表示法，并且可以明白，C、Pascal 和 Ada 的程序短语都是在λ表达式的基础上做了语法美化。这就使得本书描述的理论更加有用，而且使得程序设计语言的多样性更容易理解。语言 PCF 将直接使用λ表示法，就像 Lisp 语言那样；所不同的是，PCF 没有表和原子，但它有相对严格的定型原则。

λ表示法的主要特征是**λ抽象**和**λ应用**，前者用于定义函数，后者用于使用所定义的函数。用λ表示法写出的表达式叫做**λ表达式**或**λ项**。λ表示法既可用于类型化λ演算，也可用于无类型λ演算。

在类型化λ演算中，函数的论域（*domain*）由给出形式参数的类型来指定。如果变元 x 的类型是σ，M 是基于这个假定的良类型（*well typed*）表达式，那么$\lambda x{:}\sigma.M$ 定义一个函数，它把σ 类型的任意 x 映射到由 M 给定的值。一个简单的λ抽象的例子是：

$$\lambda x : nat.x$$

它表示自然数上的恒等函数。记号 $x : nat$ 说明该函数的论域是 *nat*，即自然数类型。在该表达式中，点后面的部分是函数体。因为该例的函数体也是 x，因此该函数的值域也是 *nat*。

每一种形式的类型化λ演算，都有精确的规则来说明：在变量类型给定的情况下，什么样的表达式是良类型的，以及良类型表达式的类型。例如，表达式$\lambda x : nat.x + true$ 不是良类型的，因为 *true* 和自然数相加是没有意义的。

用程序设计语言定义恒等函数，最熟悉的形式可能是

$$Id\ (\ x : nat\ ) = x$$

但是这种形式要求为每个函数起一个名字，而λ表示法给出了无须给函数命名的一种简洁方式。例如，

$$\lambda x : nat.x + 1$$

定义了自然数上的后继函数。

$$\lambda x : nat.10$$

是自然数上的一个常函数，该函数的值恒为 10。

在λ抽象中，λ是一个约束算子，这意味着在λ项$\lambda x{:}\sigma.M$ 中，变元 x 只是一个占位符，就像在谓词演算公式

$$\forall x : A.\phi$$

中的变元 x 那样。因此可以重新命名λ约束变元而不改变表达式的含义，只要所选择的新变元与其他已经使用的变元没有名字冲突便可以了。仅仅约束变元名字不同的项称为α等价的。如果 M 和 N 是α等价的，可以写成 $M =_\alpha N$。如果 x 出现在表达式 M 中，并且它出现在形式为$\lambda x{:}\sigma.N$ 的子表达式中，那么称 x 的这个出现是**约束的**（*bound*），否则是**自由的**。注意，在一个表达式中，x 可能既有约束出现也有自由出现，例如表达式$(\lambda y : nat \to nat.x)(\lambda x : nat.x)$ 中的 x（这种两个项并置的含义在下面解释）。不含自由变元的表达式称为**闭表达式**。

在λ表示法中，用项的并置来表示函数应用，并且用括号来说明运算对象的结合。例如，把恒等函数应用于 5 可写成

$$(\lambda x : nat.x)\ 5$$

这个函数应用的结果是 5，即

$$(\lambda x : nat.x)\ 5 = 5$$

下一节将看到，有几种不同的方式可用来计算λ表达式的值或证明λ表达式之间的等式。

λ表示法有两个约定，以省略λ表达式中的大量括号。第一个约定是函数应用是左结合的，即 MNP 应看成$(MN)P$。第二个约定是每个λ的约束范围尽可能地大，一直到表达式的结束或碰到不能配对的右括号为止。例如，$\lambda x{:}\sigma.MN$ 解释为$\lambda x{:}\sigma.(MN)$，而不是$(\lambda x{:}\sigma.M)N$。同样地，$\lambda x{:}\sigma.\lambda y{:}\tau.\ MN$ 是$\lambda x{:}\sigma.(\lambda y{:}\tau.(MN))$的简写。这两个约定可以一起使用。例如，多元函数应用可写成

$$(\lambda x{:}\sigma.\lambda y{:}\tau.\lambda z{:}\rho.M)NPQ$$

它是

$$(((\lambda x{:}\sigma.(\lambda y{:}\tau.(\lambda z{:}\rho.M)))N)P)Q$$

的简写。

1.1.3 记号和约定

本书用到的数学基础主要是集合论的知识，包括关系和函数，这些大家都已熟悉，本书不再重复。本书用到的各种归纳法另用一节专门介绍。

本书使用几种形式的相等符号，这些符号及它们的含义如下：

$=$ 两个表达式有相同的值；

$\equiv$ 除了约束变元的名字可能不同以外，两个表达式语法上等同；

$\triangleq$ 用 $M \triangleq N$ 来表示符号或表达式 M 被定义成等于 N；

::= 在文法中用来表示表达式的可能形式；

$\cong$ 表示（集合、代数等的）同构。

本书使用的逻辑符号如下：

$\forall$ 全称量词，公式$\forall x.\phi$可以读成“对所有的 x，ϕ为真”；

$\exists$ 存在量词，公式$\exists x.\phi$可以读成“存在一个 x 使得ϕ为真”；

$\wedge$ 合取，公式$\phi \wedge \psi$ 可以读成“ϕ 合取ψ”；

$\vee$ 析取，公式$\phi \vee \psi$ 可以读成“ϕ 析取ψ”；

$\neg$ 否定，公式$\neg\phi$ 可以读成“非ϕ”；

$\Rightarrow$ 蕴涵，公式$\phi \Rightarrow \psi$ 可以读成“ϕ 蕴涵ψ”（不使用$\rightarrow$作为蕴涵，因为$\rightarrow$用于类型表达式，并用于表达式的归约（求值））；

iff 当且仅当。

本书中使用的集合运算符号如下：

$\in$ 属于运算；

$\cup$ 并集运算；

$\cap$ 交集运算；

$\subseteq$ 子集运算；

$\times$ 笛卡儿积运算。

1.2 等式、归约和语义

历史上，λ表示法是**λ演算**的一部分，λ演算是关于λ表达式的一个推理系统。除了语法外，这个形式系统有三个主要部分。按照现代程序设计语言的术语，它们分别叫做**公理语义**、**操作语义**和**指称语义**。逻辑学家可能把前二者叫做证明系统，而把第三者称为一种模型。公理语义是推导表达式之间等式的一个形式系统。操作语义是一种将等式确定为有向规则的推理，叫做**归约**。按计算机科学的术语，归约可看成**符号求值**的一种形式。指称语义或模型，本质上类似于其他逻辑系统（如等式逻辑或一阶逻辑）的模型。一个模型是一组集合，每种类型一个集合，这个集合就是对应类型的解释域，并且每个良类型的表达式都可以解释为相应集合中的一个元素。

本节对本书常用的语义方法做一个概述，让读者有一个粗浅的认识。

1.2.1 公理语义

公理语义用逻辑系统来描述程序的性质，即它是一个证明系统，可用来推导程序及其组成部分的性质。这些性质可以是项之间的等式、给定输入下有关输出的断言、或其他性质。

现在这里给出的是一个等式公理系统，它有约束变元改名公理，还有把函数应用联系到**代换**的一个公理。为了表示这些公理，需要使用记号$[N/x]M$，它表示 M 中的自由变元 x 用表达式 N 代换的结果。代换时需要注意的是，N 中的自由变元不能代换到 M 中后成为约束变元。把 M 中的自由变元 x 用 N 代换的最简单办法是，首先将 M 中所有的约束变元改名，使得它们和 N 中的自由变元都有区别，然后再将 x 的自由出现用 N 代替。第 3 章将给出更详细的定义。使用代换，约束变元改名公理可写成

$$\lambda x{:}\sigma.M = \lambda y{:}\sigma.[y/x]M,\quad M \text{ 中无自由出现的 } y \qquad (\alpha)$$

例如，$\lambda x{:}\sigma.x = \lambda y{:}\sigma.y$。

因为项$\lambda x{:}\sigma.M$ 定义了一个函数，因此可以通过用 N 代替 x 来计算该函数应用于 N 的结果。例如，将函数$\lambda x{:}nat.x+4$ 应用于 4 的结果是

$$(\lambda x{:}nat.x+4)\ 4 = [4/x](x+4) = 4 + 4$$

更一般而言，等式公理

$$(\lambda x{:}\sigma.M)N = [N/x]M \qquad (\beta)_{eq}$$

被称为β 等价公理。本质上，β 等价是说，计算函数应用就是在函数体中用实在变元代替形式变元。除了这些公理和个别其他公理外，等式系统还包含自反性、对称性、传递性和同余性规则。同余性规则是说，相等的函数应用于相等的变元产生相等的结果，可以写成

$$\frac{M_1 = M_2, N_1 = N_2}{M_1 N_1 = M_2 N_2}$$

当然，为了完全精确，还应说明它们的类型，以保证每个项都有意义。和其他逻辑证明系统一样，类型化λ演算的等式证明规则允许推导任何一组等式前提的逻辑推论。

1.2.2 操作语义

语言的操作语义可用不同的方式给出，一种接近实际实现的方式是定义一台抽象机，它是一种理论上的计算机，然后通过一系列的机器状态变换来计算程序。这种操作语义最实际的表示就是语言的解释器。操作语义比较抽象的表示是演绎出最终结果的证明系统，或者说是通过一系列步骤变换一个表达式的证明系统。本书主要使用第二种形式的操作语义，即定义完整的、一步一步求值的证明系统。

先前所列等式公理的自左向右的单向形式给出了λ演算的归约规则。直观上讲，基本归约规则描述了单步符号计算，它们可以反复用于表达式的求值，直至得到表达式的最简形式，或因无最简形式而计算不终止。符号计算过程给出了λ演算的计算特征。归约是非对称的，箭头$\rightarrow$通常用于表示一步归约，双箭头$\twoheadrightarrow$用于表示任意步（包括零步）归约。

最核心的归约规则是$(\beta)_{eq}$的单向形式，叫做**β 归约**，写成

$$(\lambda x{:}\sigma.M)N \rightarrow_{\beta} [N/x]M \qquad (\beta)_{red}$$

例如，

$$(\lambda x{:}nat.x+4)\ 4 \rightarrow 4 + 4$$

因为在代换时约束变元可能需要改名，因此归约是定义在α等价上的。即β 归约的结果依赖于新约束变元的选择，它不是唯一确定的。但是归约产生的任何两个项仅在约束变元的名字上有区别，因此它们是α等价的。

除了$(\beta)_{eq}$ 规则外，还有一些其他的归约规则。完整的归约系统及其性质在后面章节讨论。

1.2.3 指称语义

用指称方法定义语言语义的基本思想是：先确定指称物，然后给出该语言各种构造（程

序、语句、表达式等）到相应指称物的语义映射，这个映射要满足：

（1）各语言构造的每个实例都有对应的指称；

（2）复合语言构造的实例的指称只依赖于它的子构造的指称。

在类型化λ演算的指称语义中，每个类型表达式对应到一个集合，称为该类型的值集。类型σ 的项解释为其值集上的一个元素。类型$\sigma \to \tau$ 的值集是函数集合，因此项$\lambda x{:}\sigma.M$ 解释为一个数学函数。纯类型化λ演算的语义是简单的，而它的各种扩充的语义可能会比较复杂。具有挑战性的特征有函数的递归定义、类型的递归定义和多态函数等。除了不讨论递归定义的类型的指称语义外，其他两个概念的指称语义在后面都会详细讨论。

对于熟悉无类型λ演算的读者来说，值得一提的是，无类型λ演算可以从类型化λ演算中派生出来。事实上，考虑无类型λ演算的语义的最自然方式之一是从类型化λ演算的语义开始。基于这个原因，类型化λ演算被看成是更加基本的系统，更适于作为研究的起点。

1.3 类型和类型系统

类型论是为避免集合论悖论而建立起来的数学理论，主要研究集合的分层、分类方法。在程序设计语言理论中，类型论是指类型系统的设计、分析和研究。在任何类型系统中，类型提供了全体所有可能值的一种分类：一个类型是一群有某些公共性质的值。对于不同的类型系统，类型的多少和值所属的类型可能不同。

本节扼要介绍类型系统的有关概念和应用。

1.3.1 类型和类型系统

一个程序变量在程序执行期间的值可以设想为有一个范围，这个范围的一个界叫做该变量的类型。例如，类型 *bool* 的变量 *x* 在程序每次运行时的值只能是该类型的值。如果 *x* 有类型 *bool*，那么布尔表达式 *not* (*x*)在程序每次运行时都有意义。变量都被给定（非平凡）类型的语言叫做**类型化的语言**（*typed language*）。

语言若不限制变量值的范围，则被称作**无类型语言**（*untyped language*），它们没有类型，或者说仅有一个包含所有值的泛类型。在这些语言中，一个运算可以应用到任意的运算对象，其结果可能是一个有意义的值、一个错误、一个异常或一个语言未作规定的结果。

类型化语言的**类型系统**是语言的一个组成部分，它始终监视着程序中变量的类型，通常还包括所有表达式的类型。一个类型系统主要由一组**定型规则**（*typing rule*）构成，这组规则用来给各种语言构造指派类型。

在计算机科学中，类型系统的研究有两个分支：比较实际的一支是类型系统在程序设计语言中的应用；比较抽象的一支关心“纯类型化λ演算”和各种逻辑之间的对应关系。本书主要研究前面一个分支。

为语言设计类型系统的目的是什么？简单讲，类型系统的根本目的是用来保证程序运行时不会出现不良行为，例如将整数和布尔值相加。只有那些顺从类型系统的程序才被认为是类型化语言的合法程序，其他的程序在它们运行前都应该被抛弃。

非形式地说，当用一种程序设计语言所能表示的所有合法程序在运行时都不会出现不良行为时，就称该语言是安全语言。若语言的类型系统能保证语言的安全性，则又称该语言是**类型可靠的**（*type sound*）或**类型安全的**。显然，希望通过适当的分析和证明来对语言的类型安全性做出准确的回答。这种分析和证明将基于语言的类型系统，因此类型系统的研究也需要形式方法，这是本书的一个重要内容。

1.3.2 类型化语言的优点

从工程的观点看，类型语言有下面一些优点：

（1）开发时的实惠

有了类型系统可以较早发现程序中的错误，例如整数和串相加。若类型系统被很好地设计，则通过类型检查可以发现大部分日常的程序设计错误，剩下的错误也很容易调试，因为大部分错误已经被排除。

对于大规模的软件开发来说，接口和模块有方法学上的优点，类型信息在这里可以组织到程序模块的接口中。程序员可以一起讨论要实现的接口，然后分头编写要实现的对应代码。这些代码之间的相互依赖最小，并且代码可以局部地重新安排而不用担心对全局造成影响。

程序中的类型信息还具有文档作用。程序员声明标识符和表达式的类型，也就是告诉了所期望的值的部分信息，这对阅读程序很有用。

（2）编译时的实惠

程序模块可以相互独立地编译，例如 Modula-2 和 Ada 的模块，每个模块仅依赖于相关模块的接口。这样，大系统的编译可以更有效，因为改变一个模块并不会引起其他模块的重新编译，至少在接口稳定的情况下是这样。

（3）运行时的实惠

在编译时收集类型信息，保证了在编译时就能知道数据占空间的大小，因而可得到更有效的空间安排和访问方式，提高了目标代码的运行效率。例如，像 Pascal 的记录、C++的结构和对象，其域或成员的偏移可以根据它们的类型信息静态地确定。

另外，一般来说，精确的类型信息在编译时可以保证运行时的运算都应用到适当类型的数据并且不需要昂贵的运行时测试，从而提高程序运行的效率。例如在 ML 中，精确的类型信息可以删除在指针脱引用（*dereference*）中的 *nil* 检查。

上面提到，类型信息具有文档作用，但是它和其他形式的程序标注不同。一般来说，关于程序行为的标注可以从非形式的注解一直到用于定理证明的形式规范。类型处在该范围的中间：它们比程序注解精确，比形式规范容易理解。另外，类型系统应该是透明的：程序员应该能够很容易预言一个程序是否可通过类型检查；如果它不能通过类型检查，那么其原因应该是明显的。

除了这些传统的应用外，在计算机科学和有关学科中，类型系统现在还有许多应用，这里列举其中一些。

（1）类型系统一个越来越重要的应用领域是计算机和网络安全。在网络计算中出现的安全问题，像移动代理的资源访问、信任管理，也能够依赖类型系统来解决。例如，编译时收集的类型信息附加在移动代码中，可供移动代码接受方检查该代码是否符合基本安全策略：类型安全、控制流安全和内存安全等。另一个例子是，静态定型处在 Java 安全模型的核心。

（2）除了编译器外，还有许多程序分析工具使用类型检查或类型推断算法，例如，一些别名分析工具使用类型推断技术。

（3）在自动定理证明方面，类型系统（尤其是基于依赖类型的类型系统）用来表示逻辑命题和证明，一些著名的定理证明辅助工具都是直接基于类型论的。

有些语言将类型检查推迟到程序运行的时候。例如，虽然 Lisp 程序在编译时没有类型检查，但是运行时的检查保证表操作只能用于表。因为运算对象的准确值在运行时是知道的，因此运行时的检查更加精确，而且只有那些在运行时真正会出现的错误才会被查出。举个简单的例子说明动态检查和静态检查的区别。例如，对于条件表达式

if B **then** 3 **else** 4 + “polyglot”

在运行时，若 B 的值为 true，那么它不含运行时的错误。但是大多数编译时的检查器会拒绝

这个表达式。一般而言，表达式的值在编译时难以确定，因此编译时的类型检查采取稳妥的策略，从而可能拒绝了一些没有不良行为的程序。基本递归论的一个结论是：预测运行时类型错误是程序的一个不可判定性质。

本书将集中在可静态确定类型的语言上，即每个程序短语必须有一个类型。除了极端情况外，类型可以从其语法形式有效地确定。第 3 章讨论的语言的语法将由一组定型规则来定义，而不是用上下文无关文法。每个良形（*well formed*）短语都有一个类型，一遍扫描可确定短语的类型。其中没有局部声明的标识符的类型可以从某个符号表得到。

集中于静态类型化语言的原因是，把注意力集中到良类型的程序上，不必考虑有类型错误的程序，因而不必考虑运行时的类型检查。只考虑良类型的程序可以使程序设计语言的理论比较简单。考虑静态类型化语言的另一个理由是，它允许对表达式的值进行一定程度的推理，这对无类型语言或运行时才能检查类型的语言是不可能的。

1.4 归纳法

1.4.1 表达式上的归纳

本书包含许多归纳证明。最常用的是基于表达式结构的归纳和基于证明的长度或结构的归纳。本节介绍一些归纳法，对于大家已经熟悉的两种自然数归纳法，下面直接给出定义。

自然数归纳（形式 1）为了证明对每个自然数 n，$P(n)$为真，只要完成下面两步证明就足够了。

（1）证明 $P(0)$为真；

（2）对任何自然数 m，证明，若 $P(m)$为真，则 $P(m+1)$也为真。

自然数归纳（形式 2）为了证明对每个自然数 n，$P(n)$为真，只要完成下面的证明就足够了：

对任何自然数 m，若 $P(i)$对所有的 $i < m$ 都为真，则 $P(m)$也为真。

只要有方法把每一棵树联系到一个自然数，就可以使用自然数归纳法来证明树的性质。也可以为树和一些其他数学对象类阐述独立的归纳原理，最关键的问题是要能够为一类数学对象安排一个适当的次序，使得每个对象可以从最小对象经过有限步的构造得到。直观上，这类次序允许写出一种无限证明的形式，最终覆盖所关心的每一个对象。首先考虑表达式上的归纳。

通常用文法来定义表达式集合，如：

$$e ::= 0 \mid 1 \mid v \mid e + e \mid e * e$$

假定 V 是变量符号的无穷集合，该文法中的 v 表示 V 的任何元素都是一个表达式。由这个文法产生的表达式都有一棵分析树，可以由对分析树的高度进行归纳来证明表达式的性质。更精确地说，如果 P 是表达式的一种性质，那么可以定义自然数的一种性质 Q 如下：

$$Q(n) \triangleq \forall \text{分析树 } t. \text{如果 } height(t) = n \text{ 并且 } t \text{ 是 } e \text{ 的分析树，那么 } P(e) \text{为真。}$$

注意，即使某些表达式的分析树多于一棵时，这也是一个合理的自然数性质。要想得到更加清楚的证明方式，还是应该为表达式使用一种专门的归纳形式。下面使用自然数归纳证明的本质步骤来解释这种形式的归纳证明。

假设要证明表达式上的性质 P，并且像上面那样定义了自然数上的一个性质 Q。如果使用自然数归纳法来证明$\forall n.Q(n)$，那么首先必须为高度是 0 的分析树直接证明 P。然后在高度至少为 1 的分析树中，假定对其各子树对应的表达式，P 都成立。就上面所给的文法而言，对于 0，1 和变量 v，必须直接证明 P。对于形式为（$e_1 + e_2$）和（$e_1 * e_2$）的复合表达式，可以假定 P 对子表达式 e_1 和 e_2 都成立。由此分析可以得出基于表达式结构的归纳，其形式

如下：

结构归纳（形式 1） 为了证明对某个文法产生的任意表达式 e，$P(e)$为真，只要完成下面两步证明就足够了。

（1）对每个原子表达式 e，证明 $P(e)$为真；

（2）对直接子表达式为 $e_1,\ldots, e_k$ 的任何复合表达式 e，证明，如果 $P(e_i)$（i=1,…, k）都为真，那么 $P(e)$ 也为真。

对于上面所给文法，则有下面的模板：

目标：对每个表达式 e，证明 $P(e)$。

归纳基础：证明 $P(0)$和 $P(1)$，并对任何变量 v 证明 $P(v)$。

归纳步骤：证明，对任何表达式 e_1 和 e_2，若 $P(e_1)$和 $P(e_2)$为真，则 $P(e_1 + e_2)$和 $P(e_1 * e_2)$也为真。

还有另一种形式的结构归纳，它在归纳假设中包含了所有的子表达式，只不过这种区别不像自然数归纳那样强调。

结构归纳（形式 2）为了证明对某个文法产生的任意表达式 e，$P(e)$为真，只要完成下面的证明就足够了：

对任何表达式 e，若 $P(e')$对 e 的任何子表达式 e'都成立，则 $P(e)$也成立。

形式 1 和形式 2 的区别在于，形式 2 的归纳假设包含了所有的子表达式，并非只是直接子表达式。

可以把自然数归纳的两种形式看成是结构归纳对应形式的特殊情况。请看文法

$$n ::= 0 \mid succ\ n$$

直观上说，n 的后继 $succ\ n$ 是 n +1。每个自然数都可以用该文法写出来，并且对于该文法，表达式上的两种归纳形式正好对应到自然数上的两种归纳形式。

1.4.2 证明上的归纳

基于证明结构的归纳和基于表达式结构的归纳本质上是一样的。在许多方面，两者都是树上的归纳形式。在论述证明结构上的归纳之前，先回顾一般证明系统的一些公共的基本概念。

希耳伯特风格的**证明系统**由公理和推理规则组成。证明系统的**公理**是一个公式，它被定义成可证明的。**推理规则**用来表达：若某一组公式可证，则另一个公式也可证。**证明**是一个结构化的对象，它由公式来构造，这些公式遵循由一组公理和推理规则确定的约束。下面将完整地描述证明。

公理和推理规则通常写成模板，它们中的每一个都代表某种形式的所有公式或证明步骤。例如，相等性的自反公理

$$e = e \qquad (ref)$$

叫做“带元变量 e 的模板”。这个公理模板断言，每个形式为 $e = e$ 的等式是一个公理。例如，只要 x，y 和 3 都是良形表达式，那么 $x = x$，$y = y$ 和 3 = 3 都是公理。通常，推理规则模板的形式是

$$\frac{A_1 \ \ldots \ A_n}{B}$$

其含义是，如果公式 $A_1, \ldots, A_n$ 都可证，那么公式 B 也可证。把公式 $A_1, \ldots, A_n$ 的证明合起来就形成公式 B 的证明。例如，相等性的传递规则可以写成

$$\frac{e_1 = e_2 \qquad e_2 = e_3}{e_1 = e_3} \qquad (trans)$$

这意味着如果有 3 + 5 = 8 的证明和 $8 = 2^3$ 的证明，那么把它们合起来可以形成等式 $3 + 5 = 2^3$ 的证明。横线上面的公式叫做证明规则的**前提**，横线下面的公式叫做**结论**。

形式地讲，一个证明可以定义为一个公式序列，该序列中的每个公式都是公理或者是由先前的公式通过一条推理规则得到的结论。证明的这种形式定义是非常有用的，它使得可以基于证明的长度（也就是该序列的长度），用自然数归纳法来讨论证明的性质。另一种观点显得更有见识：由于推理规则通常由一组前提和一个结论组成，因此很容易把证明看成是某种形式的树，其叶结点和内部结点由公式标记。也就是把证明所用的公理看成叶结点，把所用的推理规则

$$\frac{A_1 \ \ldots \ A_n}{B}$$

看成树的内部结点，其子树必须是 $A_1, \ldots, A_n$ 的证明。为了和推理规则通常的图示方向一致，画证明树时一般把树根放在底部。这样，如果从 $A_1, \ldots, A_n$ 的证明构造 B 的证明，其证明树的形式见图 1.1。在图 1.1 中，在横线上的每个尖角图形代表一棵证明树，它的结论是本证明规则的一个前提。把证明看成树的一个好处是，它提示了一种归纳形式，这种归纳形式本质上同树的结构归纳是一样的。

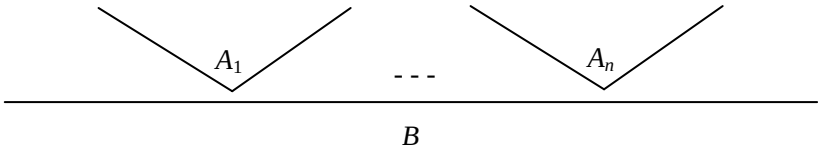

图 1.1 证明树示意图

如果基于证明树的高度进行归纳，那么归纳基础是：每个公理具备某性质；归纳步骤是：假定任何较短的证明都有这个性质，然后证明，结束于各个推理规则的证明也都具有这个性质。这就形成了证明上的结构归纳，其形式如下：

证明上的结构归纳 在某个证明系统中，为了证明对每个证明π，$P(\pi)$为真，只要完成下面两步就足够了：

（1）对该证明系统中的每个公理，证明 P 成立；

（2）假定对证明$\pi_1, \ldots, \pi_k$，P 都成立，证明 $P(\pi)$也成立。π是这样的证明，它结束于使用一个推理规则，并且是从证明$\pi_1, \ldots, \pi_k$ 延伸出来的一个证明。

例 1.1 本例用一个简单的证明系统来介绍证明结构上的归纳，它是表达式小于等于关系（$e \le e'$）的一个简单证明系统，其中 e 和 e'由下面的文法产生：

$$e ::= 0 \mid 1 \mid v \mid e + e \mid e * e$$

该证明系统有两个公理，其一是说≤是自反的：

$$e \le e \qquad (ref)$$

另一个是说 0 小于等于任何表达式：

$$0 \le e \qquad (0 \le)$$

该系统有一个传递性推理规则，另有两条规则表示加和乘的单调性。

$$\frac{e \le e' \quad e' \le e''}{e \le e''} \qquad (trans)$$

$$\frac{e_1 \le e_2 \quad e_3 \le e_4}{e_1 + e_3 \le e_2 + e_4} \qquad (+mon)$$

$$\frac{e_1 \le e_2 \quad e_3 \le e_4}{e_1 \times e_3 \le e_2 \times e_4} \qquad (\times mon)$$

对于自然数上通常的序，这是一个相对弱的证明系统，但是对于举例说明基本概念来讲是足够了。

对一个证明系统来说，需要确立的一个重要性质是，在公式的某种特定解释下，每个可证的公式都为真。这叫做证明系统的**可靠性**（*soundness*）。现在以证明本例的证明系统对≤可靠为例，来说明证明结构上的归纳，本例文法的算术表达式按通常的方式在自然数集合上解释。具体说，需要证明下面的性质对任何证明π都成立：

$P(\pi)$≜ 如果π是 $e \le e'$的一个证明，那么 e 的值≤ e'的值，不管其中的变量怎样取值。

归纳基础是为所有的公理证明该性质。这是非常容易的，不管 e 中的变量怎样取值，e 都是解释到某个自然数 n，因此总有 $n \le n$ 和 $0 \le n$。

本证明的归纳有三步，这里证明(+*mon*)和(×*mon*)两种情况，而把(*trans*)情况留给读者。假定可以证明 $e_1 \le e_2$ 和 $e_3 \le e_4$。任意选择 $e_1, \ldots, e_4$ 中变量的值，并把这四个表达式的值叫做 $n_1, \ldots, n_4$。由归纳假设，有 $n_1 \le n_2$ 和 $n_3 \le n_4$。很容易看出 $n_1 + n_3 \le n_2 + n_4$，同样有 $n_1 \times n_3 \le n_2 \times n_4$。该推理可用于变量所有可能的取值，因此该性质对终止于(+*mon*)和(×*mon*)的证明都成立。(*trans*)的情况由读者自己给出，该归纳证明结束。 □

1.4.3 良基归纳

前面所介绍的归纳都是更一般的基于良基关系的归纳的实例。良基关系在计算机科学中是重要的，在很多地方要用到它，例如它和程序终止的证明方法有联系。

集合 A 的**良基关系**（*well-founded relation*）是 A 上的一个二元关系$\prec$，它具有这样的性质：A 上不存在无穷递减序列 $a_0 \succ a_1 \succ a_2 \succ \ldots$（在此定义 $b \succ a$ 当且仅当 $a \prec b$）。例如，若 $j = i + 1$，则 $i \prec j$，那么这是自然数上的一个良基关系。从该例得知，良基关系不一定有传递性。也很容易看出，每个良基关系都是非自反的，即对任何 $a \in A$，$a \prec a$ 不成立。其理由是，如果 $a \prec a$，那么存在无穷递减序列 $a \succ a \succ a \succ \ldots$。

一个等价的定义是，A 上的二元关系$\prec$是良基的，当且仅当 A 的每个非空子集 B 有一个极小元（$a \in B$ 是极小元，如果不存在 $a' \in B$ 使得 $a' \prec a$）。它的证明在下面的引理中。

引理 1.1 如果$\prec$是集合 A 上的二元关系，那么$\prec$是良基的当且仅当 A 的每个非空子集都有一个极小元。

证明 首先，假定$\prec$是集合 A 上的良基关系，令 $B \subseteq A$ 是任意非空子集。用反证法证明 B 有极小元。如果 B 无极小元，那么对每个 $a \in B$，可以找到某个 $a' \in B$ 使得 $a' \prec a$。这样，可以从任意的 $a_0 \in B$ 开始，构造一个无穷递减序列 $a_0 \succ a_1 \succ a_2 \succ \ldots$。这就证明了该引理的前一半。

反过来，假定每个子集有一个极小元，那么不可能存在无穷递减序列 $a_0 \succ a_1 \succ a_2 \succ \ldots$，因为这样的序列给出了无极小元的集合$\{a_0, a_1, a_2, \ldots\}$。这就完成了整个证明。 □

命题 1.2（良基归纳） 令$\prec$是集合 A 上的一个良基关系，令 P 是 A 上的某个性质。若每当所有的 $P(b)$ $(b \prec a)$为真则 $P(a)$为真，则对所有的 $a \in A$，$P(a)$为真。

证明 假定对每个 $a \in A$ 有，若所有的 $P(b)$ $(b \prec a)$为真则 $P(a)$为真（用符号表示的话，即假定$\forall a.\ (\forall b.(\ b \prec a \Rightarrow P(b)) \Rightarrow P(a))$）。用反证法来证明对所有的 $a \in A$，$P(a)$为真。

如果存在某个 $x \in A$ 使得$\neg P(x)$成立，那么集合

$$B = \{\ a \in A \mid \neg P(a)\}$$

非空。由引理 1.1，B 一定有极小元 $a \in B$。但是，对所有的 $b \prec a$，$P(b)$一定成立（否则 a 不是 B 的极小元），这就和假定$\forall b.(\ b \prec a \Rightarrow P(b)) \Rightarrow P(a)$矛盾。所以该命题成立。 □

表 1.1 列出了同本书中常用归纳形式相联系的良基关系。良基关系的一个重要性质是，良基关系的传递闭包也是良基关系。这对于理解表 1.1 中两种不同形式的自然数归纳和两种不同形式的结构归纳的相似性是有帮助的。在这两种情况下，和第二种归纳形式相联系的良基关系正是和第一种归纳形式相联系的良基关系的传递闭包（不难看出，表 1.1 所列的关系都是良基关系）。

表 1.1　常用归纳形式的良基关系

归纳形式	良基关系
自然数归纳（形式 1）	$m \prec n$，如果 $m+1=n$
自然数归纳（形式 2）	$m \prec n$，如果 $m<n$
结构归纳（形式 1）	$e \prec e'$，如果 e 是 e'的直接子表达式
结构归纳（形式 2）	$e \prec e'$，如果 e 是 e'的子表达式
基于证明的归纳	$\pi \prec \pi'$，如果π是证明π′的最后一步推导规则的某个前提的证明

下面写出表 1.1 所说的自然数归纳的两种形式，读者自己不难写出结构归纳的两种形式。

自然数归纳（形式 1）：为证明对所有自然数 n，$P(n)$为真，只需证明 $P(0)$以及对任何自然数 m，如果 $P(m)$为真，则 $P(m+1)$必定为真。

自然数归纳（形式 2）：为证明对所有自然数 n，$P(n)$为真，只需证明 $P(0)$以及对任何自然数 $m(m>0)$，若所有的 $P(i)$ $(i<m)$为真，则 $P(m)$必定为真。

一类十分有用且更加复杂的良基关系是**词典序**，它们本质上是从基于某个有序集合得出的有限序列集合上的像词典一样的序。为简单起见，仅考虑自然数序列上的序。

自然数序对上的一个自然序是

$$\langle n, m\rangle \prec \langle n', m'\rangle \quad \text{iff} \quad n<n' \text{ 或者 } n=n' \text{且 } m<m'$$

如果仅考虑单数字的数 0, …, 9，并且把序对$\langle n, m\rangle$看成两位数 nm，那么这个序就是数值序。可以把序对上的这个序加以推广，给长度为 3, 4, … 的序列排序。还可以把这个序推广到不同长度的序列上。把自然数写成序列$\langle n_1, n_2, \ldots, n_k\rangle$这样的形式，可以定义更一般的序为：

$$\langle n_1, n_2, \ldots, n_k\rangle \prec \langle m_1, m_2, \ldots, m_l\rangle \quad \text{iff}$$

$$k<l \text{ 或者 } k=l \text{ 并且存在一个 } i \le k, \text{使得对所有的 } j<i \text{ 有 } n_j=m_j \text{ 并且 } n_i<m_i$$

习　　题

1.1 用∗表示自然数乘法，为平方函数写一个λ表达式。

1.2 在表达式$(\lambda y : nat.\lambda z : nat.y+z)$ 5 3 中插入括号，使得它是不省略括号的λ表达式。

1.3 在表达式$\lambda y : nat.\lambda z : nat.y+z$ 中，把约束变元 y 重新命名为 x 不会改变该表达式定义的函数。为什么把 y 重新命名为 z 会改变此表达式的含义？可以把$\lambda y : nat.y+z$ 中的 y 重新命名为 z 吗？

1.4 使用 $(\beta)_{\text{red}}$ 两次，把λ表达式$(\lambda f : nat \to nat.f\ 10)$ $(\lambda x : nat.x+x)$化简到两个自然数的和。

1.5 如果把树看成是由下面的文法产生的:

$$t ::= nil \mid leaf \mid node(t, t)$$

使用树上的结构归纳来证明，一棵二叉树的大小最多是 2^h-1 个结点，其中 h 是树的高度。

1.6 字母表Σ上的串 s 是Σ的元素的有穷序列。在这个习题中使用大写英文字母作为字母表，因此任何串 s 可以写成 $s = a_1a_2\ldots a_k$，其中每个 a_i 是 26 个大写字母中的一个。一种特别的情况是空串，写成ε。下面在串上定义了三个关系，两个是良基的，另一个不是。对于良基关系，简单解释它为什么是良基的；对于那个非良基关系，举出一个无穷的递减序列。

(*a*) 第一个关系$\prec_1$是串上的普通字母序，它由下面两个公理刻画:

$\varepsilon \prec_1 s$ iff $s \neq \varepsilon$

$a_1 s \prec_1 a_2 t$ iff 在字母表中 a_1 在 a_2 的前面或者 $a_1 = a_2$ 且 $s \prec_1 t$

(*b*) 第二个关系$\prec_2$由串的长度定义:

$a_1 \ldots a_k \prec_2 b_1 \ldots b_l$ iff $k < l$

(c) 第三个关系$\prec_3$组合前面两个关系，由$\prec_2$来定义不同长度的串的关系，而由$\prec_1$来定义相同长度的串的关系。

$$a_1 \ldots a_k \prec_3 b_1 \ldots b_l \quad \text{iff} \quad k < l \text{ 或者 } k = l \text{ 且 } a_1 \ldots a_k \prec_1 b_1 \ldots b_l$$