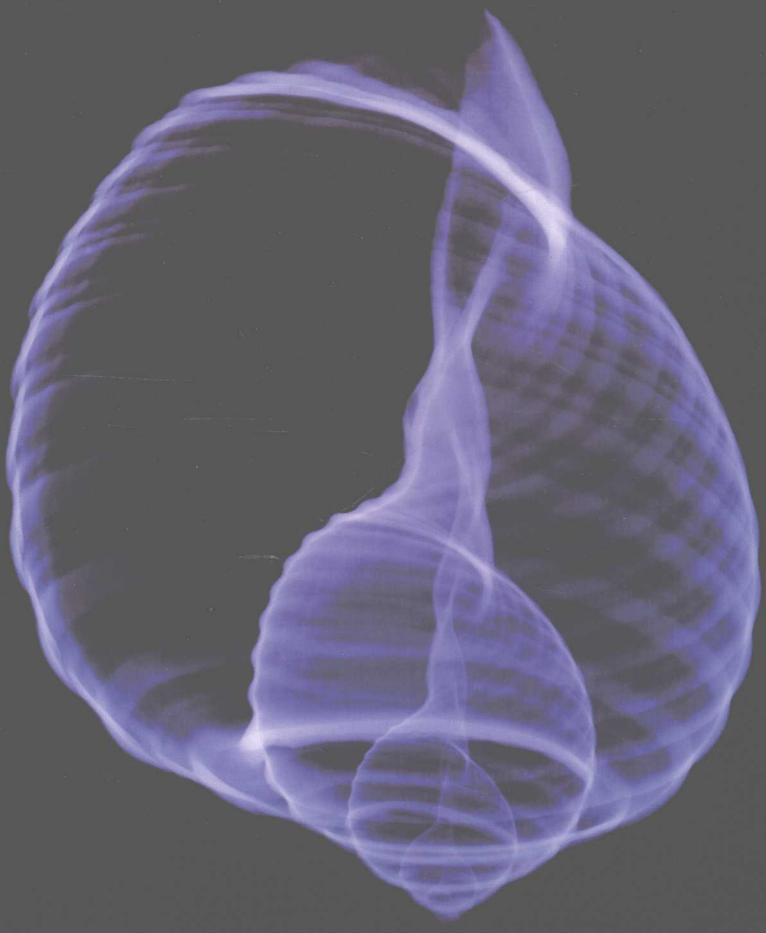
CRC Press
Taylor & Francis Group

世界著名计算机教材精选

编译器设计基础

Alexander Meduna 著
杨萍 王生原 译



ELEMENTS OF COMPILER DESIGN

清华大学出版社

世界著名计算机教材精选

编译器设计基础

Alexander Meduna 著

杨 萍 王生原 译

清华大学出版社
北 京

Elements of Compiler Design / by Alexander Meduna/ISBN: 9781420063233

图书在版编目（CIP）数据

编译器设计基础/(美)梅杜纳(Meduna, A.)著；杨萍等译. —北京：清华大学出版社，2009.4
(世界著名计算机教材精选)
书名原文：Elements of Compiler Design
ISBN 978-7-302-19334-0

Ⅰ. 编…　Ⅱ. ①梅…②杨…　Ⅲ. 编译程序－程序设计－教材　Ⅳ. TP314

中国版本图书馆 CIP 数据核字(2009)第 010409 号

责任编辑：龙啟铭
责任校对：徐俊伟
责任印制：杨　艳

出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 装 者：北京国马印刷厂
经　　销：全国新华书店
开　　本：185×260　　**印　张**：18.25　　**字　数**：434 千字
版　　次：2009 年 4 月第 1 版　　**印　次**：2009 年 4 月第 1 次印刷
印　　数：1～3000
定　　价：36.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。
联系电话：010-62770177 转 3103　　产品编号：029625-01

前　言

本书是编译器编写方面的入门教材，适用于一个学期的高年级本科课程。它坚持在这一主题的理论和实践方法之间维持一种平衡。从理论角度来看，它介绍了编译及其核心阶段的基本模型。基于这些模型，它讲解了编译器中用到的概念、方法和技术。它还简述了编译以及相关话题的数学基础，这些话题包括形式语言理论、自动机和变换机。同时，从实践的视角来看，本书描述了编译器技术是如何实现的。一个案例学习贯穿全书，它设计一种新的类 Pascal 程序设计语言，并构造其编译器；在讨论编译器各种方法的同时，这个案例学习用作其实现的实例说明。此外，本书提供了许多详细的例子和计算机程序，以强调编译算法的实际应用。书中也涵盖了核心软件工具。学完本书之后，学生应该能够掌握编译过程，编写简单的真实编译器，并可以继续学习关于该主题的更深入的书籍。

从逻辑的观点来看，本书将编译分为 6 个内聚性强的阶段。同时，它指出真正的编译器并不是以严格的顺序方式执行这些阶段；与此不同，它们的执行会有一些重叠，目的是为了尽可能加快和改进整个编译过程。因而，本书在按照逐个阶段涵盖编译过程的同时，同步地解释每一个阶段在编译期间是如何被衔接起来的。它描述这种相互衔接是如何反映到编译器构造中的，以达到整体上最好的编译效果。

就学生方面而言，并不假定他们先前有多少关于编译方面的知识。虽然这本书是自成体系的，也就是说并不需要其他额外的资源来理解这些内容，但是熟悉汇编语言和高级语言（如 Pascal 或 C）对加快理解本书是有帮助的。在介绍每一个新的概念和算法之前，都要解释它的目的，并跟随一些例子，计算机程序段以及注释，以强化对它的理解。每一复杂内容之前都有直观的解释。给出的所有应用例子都有实际意义，能够清楚地体现理论概念和它的应用之间是紧密关联的。

严格地讲，在计算机科学中，每一个算法都需要证明它可以终止并能够正确地工作。然而，就本书所给出的算法而言，终止性总是很明显的，因此其证明将全部忽略。复杂算法正确性的证明将详细给出。另一方面，在多数情况下，对于简单算法我们只是给出其要点，而将严格证明留作练习。本书将采用类 Pascal 记号描述算法，这种记号十分简单和直观，甚至不熟悉 Pascal 的学生都可以读懂它。在这种描述中，Pascal 的 repeat 循环有时用 until no change 作为结束，含义是循环重复执行直到进一步重复时所得结果不再变化为止。由于本书的编写，尤为重要的是清晰的理解，所以，算法的描述经常会加入文字以使解释更加容易和有效。

算法、约定、定义、引理以及定理在各章内部是顺序编号的，并以■结束。例子和图的组织也类似。在每一章最后，将给出一套练习，用来强化和拓广本章所涵盖的内容。标有 *Solved* 的练习题是特意选出来的，在本章结尾处有相应的解答。附录 A 包含一份 C++ 源代码，实现了一个真实编译器的重要部分。进一步的支持材料，包括课程讲稿、教学小指导、家庭作业、勘误、考试、练习解答以及编译器的实现，可以从如下网页获取：

http://www.fit.vutbr.cz/~meduna/books/eocd

致　谢

本书的基础是我在讲授关于编译器以及相关的计算机科学主题（比如自动机理论）的课程时所用过的课程讲稿，在过去的 30 多年里，这些课程在美洲、欧洲以及日本的各类大学中开设过。在日本的京都产业大学、中国台湾的国立台湾大学以及阿根廷的布宜诺斯艾利斯大学讲学时所完成的讲稿也特别的有帮助。20 世纪 90 年代的前 9 年，我在美国的密苏里-哥伦比亚大学讲授编译器的编写，并自 2000 年起在捷克共和国的布尔诺理工大学讲授这门课程。在这两所大学写的课程讲稿是这本书的基础，并且在与这两个地方的同事和学生交谈中得到了更大益处。这本书的写作受到了 GACR 201/07/0005 以及 MSM 0021630528 的资助。

我要特别感谢 Erzsebet Csuhaj-Varju，Jan Hrouzek，Masami Ito，Miroslav Novotný，Dušan Kolář，Jan Lipowski 和 Hsu-Chun Yen，与他们进行了富有成果的讨论，包括关于编译器和相关话题，如形式语言及其自动机。Ruman Lukáš 仔细阅读并验证了本书与数学相关的段落，他的评注是非常宝贵的。如果没有 Zbyněk Křivka 在准备阶段的巨大帮助，本书几乎是不可能完成的。我还要感谢 Taylor & Francis 的 Andrea Demby，Katy E. Smith，JessicaVakili 和 John Wyzalek，他们完成了卓越的编辑和制作工作。最重要的是，我要感谢妻子 Ivana 的支持与爱。

A. M.

译 者 序

编译器是计算机系统最核心最基础的支撑软件之一。与编译器设计原理与技术相关的知识体系，可以体现从计算机程序设计语言到计算机体系结构相对独立的整机概念，它又涉及形式语言与自动机、数据结构与算法等计算机学科的基础理论，不愧为联系计算机科学理论和计算机系统的典范。正如前辈 Alfred V. Aho 和 Jeffrey D.Ullman 在他们的著作中所提到的，在每一个计算机学者的职业生涯中，都会反复用到这些原理和技术。正因为如此，在大多数高等院校的计算机科学与技术专业中，编译原理与技术都作为必修的核心专业基础课程之一。

国内外关于编译原理与技术的专业教材已有不少，也有几部堪称经典之作。然而，如果你是一位想要为本科生开设一个学期有关编译原理与技术课程的教师，那么你会发现，选择适合的教材其实是相当不容易的。分析其原因，主要是因为选材非常广泛，不同理念的书籍之间内容差异较大。一些书籍的篇幅过长，它们或者在某些方面过于细节化，或者涉及许多编译高级话题，有些作者甚至根据个人的研究兴趣选择了某些专题内容，这些书籍通常需要第二个学期或者研究生阶段的学习。另有一些书籍过于技术化，涉及的原理部分不成体系，其理念主要是强调如何设计出可用的编译器。还有一些书籍认为基于编译前端的技术已经非常成熟，所以将前端所涉及的原理和技术完全弱化，主要篇幅是关于编译优化以及后端设计的内容。后两类书籍的选择固然有各自的道理，然而却忽略了这样的事实：这些以前端设计原理为主的内容真正是计算机科学理论应用于计算机系统设计的典范，是计算机学科体系中最为精髓的内容之一。本科阶段编译课程的内容不宜单纯强调技术层面，毕竟最终直接从事编译器构造这种艰巨工作的毕业生也只是少数。

由 Alexander Meduna 教授编写的 Elements of Compiler Design 或许是符合您开课理念的书籍。作者强调了这是一本编译器编写方面的入门教材，它所坚持的理念是在理论和实践方法之间维持一种平衡。在我们看来，这本书很适合作为计算机科学与技术专业本科阶段一个学期的编译原理与技术课程的教材。该书对于基本原理的讲解很到位，在系统性以及理论与实践方法之间的融合方面优于多数目前我们所能见到的教材。从本书中，读者可以深入学习基础理论如何指导实际编译器中的词法、语法及语义分析程序的设计，同时也可以轻松了解有关（中间与目标）代码生成和代码优化的整体知识框架。这些内容的融会贯通对于相关课程的学习以及从事专业工作都会有很大帮助。对于将来可能真正从事复杂编译器工作的少数学生，也可以通过本书内容的学习奠定一个坚实的起点。

在本书的理论体系中，作者采用重写系统来定义文法、有穷自动机以及下推自动机等语言处理模型。读者会体会到这样做的好处，简洁是一方面，还有就是使原理与方法容易融合起来。比如，这可以使文法与下推自动机的定义密切关联起来，前者是对语法的描述，而后者则对应语法分析乃至语法制导翻译的过程。这一点对于部分师生来说开始时可能会有一些不适应，但由于本书是自成体系的，所以应当不难克服。

限于译者水平，译文中难免会出现各种错误和疏漏，欢迎广大读者批评和指正。

目　录

第1章 导 引

这一章，我们通过描述编译过程和编译器的组成部件来介绍本书的主题。同时，定义一些数学记号和概念，以便于更清楚、更准确地讨论这些主题。

要点：本章首先回顾贯穿全书的数学记号（1.1 节），然后描述编译过程和编译器构造（1.2 节），最后介绍重写系统作为形式化编译器组成部件的基本模型（1.3 节）。

1.1 数 学 基 础

本节回顾在书中用到的大家所熟知的一些数学记号、概念和技术。例如，集合、语言、关系、翻译、图以及证明技术。

1.1.1 集合与序列

集合（set）Σ是取自一些特定**全域**（universe）的元素的汇集。如果 Σ 包含一个元素 a，记作 $a\in\Sigma$，称 a 是Σ的**成员**（member）。否则，如果 a 不属于 Σ，记作 $a\notin\Sigma$。Σ 的**基数**（cardinality）$card(\Sigma)$ 是 Σ 的元素个数。没有元素的集合称为**空集**（empty set），记作∅；注意，有 $card(\varnothing)=0$。如果 Σ 的元素个数有限，Σ 是**有穷集**（finite set）。否则，Σ是**无穷集**（infinite set）。

有穷集 Σ 通常通过列举其元素的方式来描述集合；即，$\Sigma=\{a_1, a_2, \ldots, a_n\}$，其中 $a_1 \sim a_n$ 是 Σ 的所有元素。无穷集 Ω 通常通过一个特性 π 来描述集合，使得 Ω 所包含的所有元素满足特性 π；这种描述可用符号形式化 $\Omega=\{a \mid \pi(a)\}$。以其他集合为元素的集合称为**集合家族**（families），而不把它称为集合的集合。

假设 Σ 和Ω 是两个集合，如果 Σ 的每一个元素都属于 Ω，称 Σ 是 Ω 的**子集**（subset），记作 $\Sigma\subseteq\Omega$。如果 $\Sigma\subseteq\Omega$，同时 Ω 包含一个不属于 Σ 的元素，则称 Σ 是 Ω 的**真子集**（proper subset），记作 $\Sigma\subset\Omega$。如果 $\Sigma\subseteq\Omega$，同时 $\Omega\subseteq\Sigma$，则 Σ 与 Ω **相等**（equal），记作 $\Sigma=\Omega$。Σ 的**幂集**（power set），记作 Power(Σ)，是 Σ 的所有子集的集合。

对于两个集合 Σ 和 Ω，它们的**并**（union）、**交**（intersection）和**差**（difference）分别记作 $\Sigma\cup\Omega$、$\Sigma\cap\Omega$和 $\Sigma-\Omega$，定义为 $\Sigma\cup\Omega=\{a \mid a\in\Sigma$ 或者 $a\in\Omega\}$，$\Sigma\cap\Omega=\{a \mid a\in\Sigma$ 且 $a\in\Omega\}$，以及$\Sigma-\Omega=\{a \mid a\in\Sigma$ 且 $a\notin\Omega\}$。设 Σ 是全域 U 上的一个集合，Σ 的**补**（complement）记作 *complement*(Σ)，定义为 *complement*(Σ) = U−Σ。**DeMorgan 律**（DeMorgan rules）将并、交和补运算关联起来：对于任意的集合 Σ 和 Ω，*complement* (*complement* (Σ) ∪ *complement* (Ω))= Σ ∩ Ω 以及 *complement* (*complement* (Σ) ∩ *complement* (Ω)) = Σ ∪ Ω。如果 $\Sigma\cap\Omega=\varnothing$，称 Σ 和 Ω **不相交**（disjoint）。一般地，n 个集合 Δ_1，Δ_2，…，Δ_n（$n\geqslant 2$）称为**两两互不相交**（pairwise disjoint），如果对所有 $1\leqslant i, j\leqslant n$，$\Delta_i\cap\Delta_j=\varnothing$ 时都有 $i\neq j$。

序列（sequence）是取自某个全域中的元素的列表。如果一个序列是元素的有穷列表，那么它是**有穷的**（finite）；否则，它是**无穷的**（infinite）。有穷序列 x 的**长度**（length）记作 $|x|$，是 x 中元素个数。**空序列**（empty sequence），记作 ε，是不包含任何元素的序列；即，$|\varepsilon|=0$。有穷序列通常通过列举元素的方式来描述。例如，有穷序列 x 描述为 $x=(0, 1, 0, 0)$，且有 $|x|=4$。

1.1.2 语言

字母表（alphabet）$\sum$ 是一个非空有穷集合，其成员称为**字符**（symbols）。$\sum$ 的任意一个非空子集称为 $\sum$ 的**子字母表**（subalphabet）。$\sum$ 中符号的有穷序列称为 $\sum$ 上的**字符串**（string）；特别地，ε 称为**空串**（empty string）。我们用 $\sum^*$ 表示 $\sum$ 上的所有字符串的集合；而 $\sum^+=\sum^*-\{\varepsilon\}$。设 $x\in\sum^*$，就像任何序列一样，$|x|$ 表示 x 的长度。对任意的 $a\in\sum$，$occur(x, a)$ 表示 a 在 x 中出现的次数，显然 $occur(x, a)$ 总满足 $0\leqslant occur(x, a)\leqslant|x|$。进一步，如果 $x\neq\varepsilon$，$symbol(x, i)$ 表示 x 中的第 i 个字符，其中 $i=1, \ldots, |x|$。任意的子集 $L\subseteq\sum^*$ 称为 $\sum$ 上的语言（language）。集合 $symbol(L, i)=\{a \mid a=symbol(x, i), x\in L-\{\varepsilon\}, 1\leqslant i\leqslant|x|\}$。$L$ 的任何子集称为 L 的**子语言**（sublanguage）。如果 L 表示字符串的有穷集合，则称 L 是**有穷语言**（finite language）；否则，L 是**无穷语言**（infinite language）。例如，$\sum^*$ 是一个无穷语言，称为 $\sum$ 上的**全域语言**（universal language），而 $\varnothing$ 和 $\{\varepsilon\}$ 是有穷语言。值得注意的是，$\varnothing\neq\{\varepsilon\}$，因为 $card(\varnothing)=0\neq card(\{\varepsilon\})=1$。其成员是语言的集合称为**语言家族**（language families）。

约定 1.1 对于字符串，为了简化，我们简单地并置字符，省略一对圆括号和所有逗号。即，用 $a_1 a_2 \ldots a_n$ 的形式替代 $(a_1, a_2, \ldots, a_n)$。 ■

运算（operations）。设 $x, y\in\sum^*$ 是字母表$\sum$上的两个字符串，且设 $L, K\subseteq\sum^*$ 是 $\sum$ 上的两个语言。因为语言定义为集合，因此所有的集合运算均适用于它们。具体地，L∪K，L∩K，和 L−K 分别表示语言 L、K 的并、交和差。或许更重要的一个运算是 x 与 y 的**连接**（concatenation），记作 xy，它是将 y 直接添加到 x 的尾部所构成的字符串。注意，从代数的角度来看，$\sum^*$ 和 $\sum^+$ 分别是在连接运算作用下由 $\sum$ 产生的**自由独异点**（free monoid）和**自由半群**（free semigroup）。对于每一个 $w\in\sum^*$，$w\varepsilon=\varepsilon w=w$。$L$ 和 K 的**连接**（concatenation），记作 LK，定义为 $LK=\{xy \mid x\in L, y\in K\}$。

除了二元运算，我们也定义字符串和语言上的一元操作。设 $x\in\sum^*$ 和 $L\subseteq\sum^*$。语言 L 的**补**记作 $complement(L)$，定义为 $complement(L)=\sum^*-L$。字符串 x 的**反转**（reversal），记作 $reversal(x)$，即将 x 的元素逆序排列。语言 L 的反转记作 $reversal(L)$，定义为 $reversal(L)=\{reversal(x) \mid x\in L\}$。对于所有 $i\geqslant 0$，字符串 x 的 i **次幂**，记作 x^i，递归定义为 (1)$x^0=\varepsilon$，和(2)$x^i=x\,x^{i-1}$，对于 $i\geqslant 1$。显然，这个定义基于**递归定义法**（recursive definitional method）。为了说明递归概念，我们考虑 i 次幂的串 x^i 在 $i=3$ 时的情形。根据定义中的(2)，$x^3=xx^2$。再对 x^2 使用定义中的（2），$x^2=xx^1$。对 x^1 再使用一次定义中的（2），$x^1=xx^0$。根据定义中的（1），$x^0=\varepsilon$。这样，$x^1=xx^0=x\varepsilon=x$。因此，$x^2=xx^1=xx$。最后，$x^3=xx^2=xxx$。我们经常会通过使用递归的方式给出一些新的概念，包括 L 的 i 次幂，L^i，定义为(1)$L^0=\{\varepsilon\}$和（2）$L^i=LL^{i-1}$，对于 $i\geqslant 1$。L 的**闭包**（closure），L^*，定义为 $L^*=L^0\cup L^1\cup L^2\cup\ldots$，$L$

的**正闭包**（positive closure），L^+，定义为 $L^+ = L^1 \cup L^2 \cup \ldots$。注意，我们有 $L^+ = LL^* = L^*L$ 以及 $L^* = L^+ \cup \{\varepsilon\}$。设 w，x，y，$z \in \sum^*$，如果 $xz = y$，那么称 x 是 y 的**前缀**（prefix）；进一步，如果 $x \notin \{\varepsilon, y\}$，则称 x 是 y 的**真前缀**（proper prefix）。y 的所有前缀的集合记作 *prefixes*(y)。令 $prefixes(L) = \{ x \mid x \in prefixes(y), y \in L \}$。对于 i=0，…，$|y|$，*prefix*(y, i)表示 y 的长度为 i 的前缀；特别地有，*prefix*(y, 0)= ε 和 *prefix*(y, $|y|$)= y。如果 $zx = y$，那么称 x 是 y 的**后缀**（suffix）；进一步，如果 $x \notin \{\varepsilon, y\}$，则称 x 是 y 的**真后缀**（proper suffix）。用 *suffixes*(y)表示 y 的所有后缀的集合。令 $suffixes(L) = \{ x \mid x \in suffixes(y), y \in L \}$。对于 i=0，…，$|y|$，*suffix*(y, i)表示 y 的长度为 i 的后缀。如果 $wxz = y$，称 x 是 y 的**子串**（substring）；进一步，如果 $x \notin \{\varepsilon, y\}$，则称 x 是 y 的**真子串**（proper substring）。我们用 *substrings*(y)表示 y 的所有子串的集合。不难发现，对于所有的 $v \in \sum^*$，$prefixes(v) \subseteq substrings(v)$，$suffixes(v) \subseteq substrings(v)$，以及 $\{\varepsilon, v\} \in prefixes(v) \cap suffixes(v) \cap substrings(v)$。令 $substrings(L) = \{ x \mid x \in substrings(y), y \in L \}$。

例 1.1　运算。考虑由二进制符号组成的字母表 {0，1}。ε、1 和 010 都是 {0，1}上的字符串。|ε|=0，|1|=1，|010|=3。1 和 010 的连接是 1010。1010 的 3 次幂等于 101010101010。*reversal*（1010）= 0101。串 10 和 1010 是 1010 的前缀。前者是 1010 的真前缀，而后者不是。*prefixes*（1010）={ε，1，10，101，1010}。字符串 010 和 ε 是串 1010 的后缀。010 是 1010 的真后缀，而 ε 不是。*Suffixes*(1010)= {ε，0，10，010，1010}，以及 *substrings*（1010）={ε，0，1，01，10，010，101，1010}。

设 K = {0，01}，L = {1，01}。$L \cup K$，$L \cap K$，和 $L - K$ 分别为 {0，1，01}、{01}和{0}。K 和 L 的连接 KL = {01,001,011,0101}。对于 L，$complement(L) = \sum^* - L$，所以 *complement*($L$)中包含了除 1 和 01 外的所有二进制串。*reversal*(L)= {1，10}，以及 L^2={11，101，011，0101}。$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$；$L^*$ 中由 4 个或更少的符号组成的串是 ε、1、01、11、101、011 和 0101。$L^+ = L^* - \{\varepsilon\}$。*prefixes*($L$)={ε，1，0，01}，*suffixes*($L$)={ε，1，01}，以及 *substrings*(L)={ε，0，1，01}。 ■

1.1.3　关系与翻译

对于两个对象 a 和 b，(a，b)表示由 a 和 b 组成的**序偶**（ordered pair）。设 A，B 是两个集合，A 和 B 的**笛卡尔乘积**（Cartesian product），$A \times B$，定义为 $A \times B = \{ (a, b) \mid a \in A, b \in B \}$。从 A 到 B 的**二元关系**（binary relation），简称**关系**（relation），ρ，是 $A \times B$ 的任意一个子集；即，$\rho \subseteq A \times B$。如果 ρ 表示的是有穷集，那么它就是**有穷关系**（finite relation）；否则，它就是**无穷关系**（infinite relation）。ρ的**定义域**（domain）记作 *domain*(ρ)，**值域**（range）记作 *range*(ρ)，它们定义为 $domain(\rho) = \{a \mid (a, b) \in \rho$，对某一 $b \in B\}$，$range(\rho) = \{b \mid (a, b) \in \rho$，对某一 $a \in A\}$。如果 A=B，则称 ρ 是 $\sum$ 上的关系。如果 $\sigma \subseteq \rho$，则关系 σ 是 ρ 的子关系。关系 ρ 的**逆**（inverse）记作 *inverse*(ρ)，定义为 $inverse(\rho) = \{ (b, a) \mid (a, b) \in \rho \}$。一个从 A 到 B 的**函数**（function）是满足如下条件的从 A 到 B 的关系 ϕ：对每一个 $a \in A$，$card(\{b \mid b \in B, (a, b) \in \phi\}) \leqslant 1$。如果 $domain(\phi) = A$，则 ϕ 是**全函数**（total function）；否则，ϕ 是**偏函数**（partial function）。如果对每一个 $b \in B$，$card(\{a \mid a \in A, (a, b) \in \phi\}) \leqslant 1$，则 ϕ 是**单射**（injection）。如果对每一个 $b \in B$，$card(\{a \mid a \in A, (a, b) \in \phi\}) = 1$，则

ϕ 是**满射**（surjection）。如果 ϕ 既是满射，又是单射，那么ϕ是**双射**（bijection）。

约定 1.2 $(a, b)\in\rho$ 常被记为 $b\in\rho(a)$或 $a\rho b$；换句话说，$(a, b)\in\rho$、$a\rho b$ 和 $a\in\rho(b)$ 可以互换使用。如果 ρ 是一个函数，通常记作 $\rho(a)=b$。 ■

设 A 是一个集合，且ρ 是 A 上的关系，$a, b\in A$。对于 $k\geqslant 1$，ρ 的 k **重积**（k-fold product）ρ^k 递归定义为（1）$a\rho^1 b$ 当且仅当 $a\rho b$，（2）$a\rho^k b$ 当且仅当存在 $c\in A$ 使得 $a\rho c$ 和 $c\rho^{k-1}b$ $(k\geqslant 2)$ 成立。进一步，$a\rho^0 b$ 当且仅当 $a=b$。ρ的**传递闭包**（transitive closure）ρ^+ 定义为：$a\rho^+ b$ 当且仅当存在某一 $k\geqslant 1$ 使得 $a\rho^k b$。ρ的**自反传递闭包**（reflexive and transitive closure）ρ^* 定义为：$a\rho^* b$ 当且仅当存在某一 $k\geqslant 0$ 使得 $a\rho^k b$。

设 K 和 L 分别是字母表 T 和 U 上的语言。从 K 到 L 的**翻译**（translation）σ 是一个从 T^* 到 U^* 的关系，使得 $domain(\sigma)=K$ 和 $range(\sigma)=L$。对于从 T^* 到 $Power(U^*)$ 的全函数 τ，如果对任何 $u, v\in T^*$ 都有 $\tau(uv)=\tau(u)\,\tau(v)$，则称 τ 是从 T^* 到 U^* 的一个**替换**（substitution）。根据这个定义，$\tau(\varepsilon)=\{\varepsilon\}$，$\tau(a_1a_2\ldots a_n)=\tau(a_1)\,\tau(a_2)\ldots\tau(a_n)$，其中 $a_i\in T$ $(1\leqslant i\leqslant n,\ n\geqslant 1)$，因此，通过对于每一个 $a\in T$ 定义 $\tau(a)$ 就可以将 τ 完全指定。

对于从 T^* 到 U^* 的全函数 υ，如果对任何 $u, v\in T^*$ 都有 $\upsilon(uv)=\upsilon(u)\,\upsilon(v)$，则称 υ 是从 T^* 到 U^* 的一个**同态**（homomorphism）。显然，同态是替换的特例，因此我们也可以只通过对于每一个 $a\in T$ 定义 $\upsilon(a)$ 就可完全指定 υ。如果对任何 $a, b\in T$，$\upsilon(a)=\upsilon(b)$ 蕴涵 $a=b$，则称 υ 为**单一同态**（injective homomorphism）。

例 1.2 波兰式。有一种表示形式不使用括号就可以表达通常的**中缀算术表达式**（infix arithmetic expressions）。这种表示形式称为**波兰式**（Polish notation），有两种基本形式：**后缀式**（postfix notation）和**前缀式**（prefix notation）。后缀式可递归定义如下：

设 Ω 是二元运算符集合，Σ 是运算数集合。

1. 对每一个 $a\in\Sigma$，后缀式是 a。
2. 设 AoB 是中缀表达式，其中 $o\in\Omega$，A 和 B 是中缀表达式。那么，CDo 是 AoB 的后缀式，其中 C 和 D 分别是 A 和 B 的后缀式。
3. 设 C 是中缀表达式 A 的后缀式。那么，C 是 A 的后缀式。

考虑中缀表达式 $(a+b)*c$。c 的后缀式是 c。$a+b$ 的后缀式是 $ab+$，所以，$(a+b)$ 的后缀式也是 $ab+$。因此，$(a+b)*c$ 的后缀式是 $ab+c*$。

前缀式的定义类似，只不过第二部分的定义中 o 放在 AB 的前边；细节留作练习。

为解释同态和替换，设 $\Xi=\{0, 1, \ldots, 9\}$ 和 $\Psi=(\{A, B, \ldots, Z\}\cup\{|\})$，并考虑从 Ξ^* 到 Ψ^* 的同态 h，定义：$h(0)=ZERO|$，$h(1)=ONE|$，$\ldots$，$h(9)=NINE|$。例如，h 将 91 映射到 $NINE|ONE|$。h 是一个单一同态。利用 h，定义从 Ξ^* 到 Ψ^* 的无穷替换 s 为：$s(x)=\{h(x)\}\{|\}^*$。结果有，$s(91)=\{NINE|\}\{|\}^*\{One|\}\{|\}^*$，它所包含的字符串如：$NINE|ONE|$ 以及 $NINE|||ONE||$。 ■

1.1.4 图

设 A 是一个集合，**有向图**（directed graph）或简称**图**（graph）是一个偶对 $G=(A, \rho)$，其中 ρ 是 A 上的关系。A 的成员称为顶点（node），ρ 中的序偶称为**边**（edge）。如果$(a, b)\in\rho$，那么 (a, b) 被看作是离开 a 而进入 b 的边。设 $a\in A$，那么，a 的**入度**（in-degree）

和 a 的**出度**（out-degree）分别是 $card(\{b \mid (b, a) \in \rho\})$ 和 $card(\{c \mid (a, c) \in \rho\})$。如果对所有 $1 \leqslant i \leqslant n$ $(a_{i-1}, a_i) \in \rho$，顶点序列 $\{a_0, a_1, \ldots, a_n\}$（$n \geqslant 1$）是从 a_0 到 a_n 的**长度**（length）为 n 的**路径**（path）；另外，如果 $a_0 = a_n$，那么 $\{a_0, a_1, \ldots, a_n\}$ 是长度为 n 的**回路**（cycle）。在本书中，常常会给边作标记（label）以附加某些信息。如图 1.1 所示，我们用从 a 指向 b 的箭头来表示图 $G=(A, \rho)$中的每条边 $(a, b) \in \rho$，在下一个例子中还给出了边上标记的示例。

例 1.3　图。考虑一个程序 p，其**调用图**（call graph）为 $G = (P, \rho)$，其中 P 表示 p 中子程序的集合，而 $(x, y) \in \rho$ 当且仅当子程序 x 调用子程序 y。比如，令 $P = \{a, b, c, d\}$，$\rho = \{(a, b), (a, c), (b, d), (c, d)\}$，所表示的调用关系是 a 调用 b 和 c，b 调用 d，以及 c 调用 d（见图 1.1）。a 的入度是 0，出度是 2。(a, b, d) 是 G 中长度为 2 的路径。G 中不存在回路，即，没有开始和结束于同一顶点的路径。

设想我们要研究在 4 个调用中某一全局变量的取值。比如，我们想要表达在调用 (a, b) 发生时该全局变量的取值为 0，其他调用的情况下其取值为 1。我们通过对图 G 中的边加标记来表示这种信息，如图 1.2。

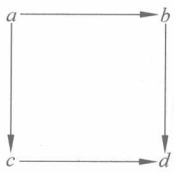


图 1.1　图

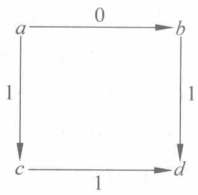


图 1.2　带标记的图

■

设 $G = (A, \rho)$是一个图。如果 G 中不包括回路，则 G 是**无圈图**（acyclic graph）。如果 $\{a_0, a_1, \ldots, a_n\}$ 是 G 中的路径，则 a_0 是 a_n 的一个**祖先**（ancestor），a_n 是 a_0 的一个**子孙**（descendent）。进一步，如果 $n=1$，则 a_0 是 a_n 的**直系祖先**（direct ancestor），a_n 是 a_0 的**直系子孙**（direct descendent）。**树**（tree）是一个无圈图 $T = (A, \rho)$，A 包含一个特定的顶点，称为 T 的**根结点**（root），记作 $root(T)$，并且每一个 $a \in A - root(T)$ 都是 $root(A)$ 的子孙，它们的入度都为 1。如果 $a \in A$ 的出度为 0，则 a 是**叶结点**（leaf）；否则，它是一个**内部结点**（interior node）。本书中，一棵树 T 总认为是**有序树**（ordered tree），T 中每个内部结点 $a \in A$ 的所有直系子孙 b_1 到 b_n（$n \geqslant 1$）从左到右排序，使得 b_1 是 a 的最左边的直系子孙，b_n 是 a 的最右边的直系子孙。此时，称 a 是**孩子**（children）b_1 到 b_n 的**双亲**（parent），所有这些顶点，包括连接它们的边，从 (a, b_1) 到 (a, b_n)，称为 T 的一个**双亲-孩子分图**（parent-children portion）。树 T 的**边界**（frontier），记作 $frontier(T)$，是 T 的叶子从左到右的序列。树 T 的**深度**（depth），记作 $depth(T)$，是 T 中最长路径的长度。称树 $S = (B, \upsilon)$ 是 T 的**子树**（subtree），如果 $\varnothing \subset B \subseteq A$，$\upsilon \subseteq \rho \cap (B \times B)$，且在 T 中，$A - B$ 内没有顶点是 B 中顶点的后代。与任意的图一样，树 T 可用二维结构表示。然而除了这种二维的表示，T 通常便于采用一维表示 $\Re(T)$ 来刻画，$\Re(T)$ 中 T 的每一个子树用出现在一对括号 < 和 > 内部的表达式来表示，而该子树的根结点出现在紧靠< 左边的位置。更精确地，根据针对树 T 的递归规则，$\Re(T)$ 定义如下：

1. 如果 T 由一个单独的顶点 a 构成，那么 $\Re(T)=a$。
2. 设从 (a, b_1) 到 (a, b_n)（$n\geqslant 1$）是 T 的一个双亲-孩子分图，$root(T)=a$，且 T_k 是以 b_k 为根结点的子树（$1\leqslant k\leqslant n$），那么，$\Re(T)=a<\Re(T_1)\Re(T_2)\ldots\Re(T_n)>$。

除了这种一维的表示，我们有时还用 T 中结点的**后序**（postorder）来表示树 T，记作 $postorder(T)$，可以从根结点开始递归应用如下过程 POSTORDER 得到。

POSTORDER：设 POSTORDER 当前作用于顶点 a。

- 如果 a 是一个内部结点，有孩子 a_1 到 a_n，从 a_1 到 a_n 递归地调用 POSTORDER，然后列出 a。
- 如果 a 是叶结点，列出 a，然后停止。

约定 1.3 画树 T 时，根结点画在最上边，所有的边方向都朝下。每个双亲的孩子根据顺序自左向右画出。这样，在画 T 时，我们总是可省去所有箭头。在实际实现 T 时，用 $\mathscr{F}T$ 表示指向 T 的根结点的指针。在 T 的一维表示中，如果 $depth(T)=0$，$\Re(T)$ 仅由一个单独的叶结点 a 组成，此时我们通常将此写作 **leaf** a，而不是简单的 a。在本书中，我们总是用 $\Re$ 作为树的一维表示。 ■

例 1.4 树。图 1.2 讨论的是无圈图。然而它不是树，因为 d 的入度为 2。通过移走边 (b, d)，得到树 $T=(P, \tau)$，其中 $P=\{a, b, c, d\}$，$\tau=\{(a, b), (a, c), (c, d)\}$。顶点 a，c 是内部结点，b，d 是叶结点。树 T 的根结点是 a。我们定义 b 和 c 分别是 a 的第一个和第二个孩子。T 的双亲-孩子分图的一个例子是 (a, b) 和 (a, c)。我们有 $frontier(T)=bd$，以及 $depth(T)=2$。T 的一维表示 $\Re(T)=a<bc<d>>$，以及 $postorder(T)=bdca$。它的子树有 $a<bc<d>>$，$c<d>$，b 和 d。为简单起见，我们通常还将一个叶子组成的子树 b 和 d 分别表示为 **leaf** b 和 **leaf** d。在图 1.3 中，我们给出 $a<bc<d>>$ 和 $c<d>$ 的图示。 ■

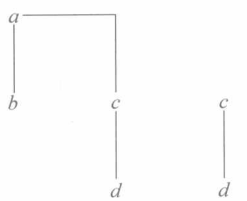


图 1.3 树和子树

1.1.5 证明

下面回顾逻辑基础知识，特别关注本书中用到的基本证明技术。

一般地，一个**形式数学系统**（formal mathematical system）S 由基本**符号**（symbols）、**形成规则**（formation rules）、**公理**（axioms）和**推理规则**（inference rules）组成。基本符号，如常数和运算符，根据形成规则构成了**命题**（statements）。公理是原始的命题，它的有效性是不经判定就被接受的。通过推理规则，一些命题推导出另外一些命题。S 中的一个命题 s 的**证明**（proof）是由命题序列 $s_1, \ldots, s_i, \ldots, s_n$ 组成，其中 $s=s_n$，并且每一个 s_i 或者是 S 中的一条公理，或者是根据推理规则由命题序列 $s_1, \ldots, s_{i-1}$ 推理出来的命题；用这种方式证明的命题可表示为 S 中的一个**定理**（theorem）。

逻辑联结词（logical connectives）将命题结合起来构成更复杂的命题。最常见的逻辑联结词是 *not*、*and*、*or*、*implies* 和 *if and only if*。这些联结词中，*not* 是一元的，其他是二元的。也就是说，如果 s 是一个命题，则 *not* s 也是一个命题。类似地，如果 s_1 和 s_2 是命题，那么 s_1 *and* s_2，s_1 *or* s_2，s_1 *implies* s_2 和 s_1 *if and only if* s_2 也是命题。我们常把 *and* 和 *or* 分别记作 $\wedge$ 和 $\vee$。下面的**真值表**（truth table）表示二元逻辑联结词的命题联结规则，

用 1 表示**真**（truth），0 表示**假**（falsehood）。对于一元联结词 *not*，如果 *s* 为真，则 *not s* 为假，而如果 *s* 为假，则 *not s* 为真。

s_1	s_1	*and*	*or*	*implies*	*if and only if*
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

图 1.4　真值表

根据真值表，如果 s_1 和 s_2 都为真，s_1 *and* s_2 为真；否则，s_1 *and* s_2 为假。同理，根据该表可解释包含其他联结词的命题成真或成假的规则。**等价**（equivalence）命题，形如 s_1 *if and only if* s_2，在本书中起关键的作用。此类命题的证明通常由两部分组成，*only-if* 部分证明 s_1 *implies* s_2 为真，而 *if* 部分证明 s_2 *implies* s_1 为真。有许多很有用的定律可用于命题蕴含关系的推理。特别地，**逆否律**（contrapositive law）表明 (s_1 *implies* s_2) 等价于 ((*not* s_2) *implies* (*not* s_1))，因此，可通过证明 (*not* s_2) *implies* (*not* s_1) 为真，来证明 s_1 *implies* s_2 为真。**反证法**（proof by contradiction）也经常使用，它的依据是如下定律：((*not* s_2) *and* s_1) *implies* 0 成真。通俗地说就是，如果假设 s_2 为假而 s_1 为真，可得到一个成假的命题，即 s_1 *implies* s_2 成真。

例 1.5　反证法。设 *P* 是全体素数的集合（自然数 *n* 是素数，如果它的正约数仅有 1 和 *n*）。下面用反证法证明 *P* 是无穷集。假设 *P* 是有穷集，令 $k = card(P)$。这样，*P* 包括 *k* 个成员 p_1，p_2，…，p_k。设 $n = p_1p_2\ldots p_k+1$，可知 *n* 不能被任何 p_i 整除（$1 \leqslant i \leqslant k$），所以，*n* 或者是一个新素数，或者是新素数之积。两种情况均说明，在 *P* 之外存在一个素数，与 *P* 包括所有素数矛盾。所以，*P* 是无穷集。 ■

归纳证明用来证明对所有 $i \geqslant b$ 的整数，命题 s_i 成真。这里，*b* 是一个非负整数。一般地，归纳证明的方式如下：

- 基础：证明 $s_b s_1$ 为真。
- 归纳前提：假设存在一个整数 *n*，$n \geqslant b$，且对所有整数 $b \leqslant m \leqslant n$，$s_m$ 为真。
- 归纳步骤：在前提和归纳前提的基础上证明 s_{n+1} 为真。

例 1.6　归纳证明。考虑对所有 $i \geqslant 1$，命题 s_i 为：

$$1 + 3 + 5 + \ldots + 2i-1 = i^2$$

换句话说，s_i 表示奇数之和是一个完全平方。该命题的一个归纳证明如下：

- 基础：$1=1^2$，s_1 为真。
- 归纳前提 ：假设对所有 $1 \leqslant m \leqslant n$，$s_m$ 为真，这里 *n* 是一个自然数。
- 归纳步骤 ：考虑

$$s_{n+1} = 1 + 3 + 5 + \ldots + (2n-1) + (2(n+1)-1) = (n+1)^2$$

根据归纳前提

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

因此

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

所以，s_{n+1} 成立，归纳证明完毕。 ■

1.2 编　　译

编译器（compiler）读入用**源语言**（source language）编写的**源程序**（source program）并将其翻译成用**目标语言**（target language）编写的**目标程序**（target program），并使得两个程序功能等价，即，它们表达了相同的所要完成的计算任务。通常，源语言是高级语言，如，Pascal 或 C，而目标语言是特定计算机上使用的机器语言或者是易翻译到该机器语言的汇编语言。翻译过程中，编译器首先分析源程序以验证它是否用源语言正确地编写。如果是，编译器生成目标程序；否则，编译器报告错误，结束不成功的翻译。

1.2.1 编译阶段

更详细一些，编译器首先对源程序进行词法、语法和语义的分析。然后，根据这三方面分析收集起来的信息，编译器生成源程序的中间代码，进行优化，生成结果目标代码。从整体上讲，**编译**（compilation）由这 6 个**编译阶段**（compilation phases）组成，每个阶段将源程序从一种内部表示转换成另一种表示：

- 词法分析
- 语法分析
- 语义分析
- 中间代码生成
- 中间代码生成优化
- 目标代码生成

词法分析（lexical analysis）将源程序分解成**单词**（lexemes），即逻辑上内聚的词法实体，例如标识符或整数。词法分析验证这些实体是良构的，产生大小一致的表示词法单位的**单词记号**（tokens），并且将这些单词记号发送至语法分析。如果需要，这些单词记号可关联一些属性以便更好地刻画其细节。词法分析通过**扫描程序**（scanner）识别每一个单词，扫描程序读取构成源程序的字符序列，识别出该序列下面一个可构成单词的部分。通过这种方式识别出单词后，词法分析将产生相应的单词记号，并把该记号发送至语法分析。

语法分析（syntax analysis）确定源程序的语法结构，该源程序由词法分析所提供，是由单词记号组成的。这一编译阶段使用了由现代数学语言学发展起来的概念和技术。实际上，源语言的语法是由**文法规则**（grammatical rules）描述的，语法分析依据这些文法规则来构造**分析过程**（parse），即产生该程序的规则序列。在语法分析中，负责构造分析过程的部分是**分析程序**（parser）。分析程序的工作方式通常可以从图的角度来解释。即，一个分析过程可以体现为一棵**分析树**（parse tree），该分析树的叶结点标记为单词记号，它的每一双亲-孩子分图构成一个**规则树**（rule tree），是规则的一种图表示形式。分析程序通过选择和组合适当的规则树来构造分析树。根据构造分析树的方式，我们将分析程序区分为两种基本类型。**自上而下分析程序**（top-down parser）从根结点开始向下到树的边界构造分