PROLOG

语言，它的应用与实现

刘椿年　曹德和　著

科学出版社

内 容 简 介

本书系统、翔实地介绍逻辑程序设计语言 PROLOG 的特性、理论基础、程序设计方法，以及 PROLOG 语言在人工智能和软件工程方面的应用与实现技术.

全书共十章. 前三章介绍 PROLOG 的特性、理论基础和程序设计方法. 第四至第八章分别讨论 PROLOG 在人工智能搜索方法、编译程序构造、自然语言理解、函数型程序设计和知识工程等方面的应用. 最后两章分别介绍 PROLOG 的解释实现和编译实现技术.

本书内容丰富、论述清晰，含有 20 多个具有实用价值的、可运行的 PROLOG 程序.

本书可作为高等院校有关专业的教材和教学参考书，也可供从事人工智能研究与开发的科技人员阅读.

PROLOG 语言，它的应用与实现

刘椿年　曹德和 著

责任编辑　那莉莉

科学出版社出版

北京东黄城根北街 16 号

邮政编码：100707

中国科学院印刷厂印刷

新华书店北京发行所发行　各地新华书店经售

*

1990 年 8 月第 一 版　开本：850×1168　1/32

1990 年 8 月第一次印刷　印张：9 3/4

印数：0001—2 550　字数：254 000

ISBN 7-03-001740-4/TP · 131

定价：12.20 元

序　言

本书系统地介绍逻辑程序设计语言 PROLOG 的主要特点、理论基础、程序设计方法、应用和实现技术.

60 年代末出现的"软件危机"(大型软件系统的复杂性、不可靠性及高昂的研制费用) 促进了程序设计语言、程序设计方法学和软件工程学的研究. 但在常规程序设计语言 (如 FORTRAN, PASCAL, ADA 等)的限制下, 似乎难以彻底解决"软件危机"问题. 其主要原因有以下两点.

首先,常规语言的程序难以论证.这些语言尽管千差万别, 但归根结底都是以对变量赋值为基本活动. 因此变量和表达式的值在不断变化,使得数学推理中最基本的等式替换法则都难以应用. 再加上某些运算(如函数调用)可能有副作用, 很难仅根据静态程序文本对其性质(如正确性)进行数学论证, 而必须动态地追踪控制流程. 从 60 年代末 Floyd 和 Hoare 的工作开始,经过十多年认真的努力, 至今未能得到对大型实用程序的正确性进行形式证明的真正实际可行的技术和系统. 因此大型程序的可靠性仍是一个问题.

其次,常规语言的程序仍嫌冗长. 与汇编语言相比, FORTRAN 的表达能力提高了 5 至10倍, 即解决同一问题所需程序量仅为汇编程序的 1/10 至 1/5. 另一方面,其他常规语言与 FORTRAN相比,在表达能力方面却相差无几.一般地,用常规语言写一个编译程序,动辄上万行、百万行的商业软件包并不鲜见. 这样大的程序量不但需要高昂的研制费用,而且也无法保证系统的可靠性.

现今常规程序设计语言已发展到顶峰, 出现了 ADA 这样集大成的语言. 为了进一步解决"软件危机", 运用抽象程度更高、表达能力更强、更易于数学论证的超高级语言作为软件开发的工

具已势在必行．其实人们早已在理论上认识到这一点，实行的主要障碍在于硬件支持．70 年代末以来，人们对超高级语言的兴趣大增，正是因为超大规模集成电路和计算机硬件技术的飞速发展为克服这一障碍展示了前景．反过来，超高级语言的研究对计算机体系结构的变化发展也必将起到巨大的推动作用．

本书打算介绍的 PROLOG 语言属于上述的超高级语言的范畴．具体地说，它是目前逻辑程序设计语言中最成熟、最具代表性的一种．而逻辑程序设计语言和函数型程序设计语言是两类最主要的超高级语言．

PROLOG 语言的主要特点是：

1. 语法简洁． 从语法上讲，PROLOG 语言是一阶谓词逻辑系统的 Horn 子集，因此具有极为简洁的语法．它的 BNF 语法描述只需一、两页的篇幅．如此简单的语法却有极强的描述和解题能力．

2. 非过程性． 逻辑程序设计思想的主要倡导者 Kowalski 把算法分成逻辑和控制这两个独立的成分． PROLOG 正是遵循这一思想设计的．它允许程序员只描写算法的逻辑方面（即做什么），而把控制方面(即怎么做)留给系统自动完成． 因此人们把 PROLOG 这样的语言称为非过程语言，而把常规语言称为过程性语言．非过程语言显然具有易读、易写、程序量小等优点．一般说来，解决同一个问题 PROLOG 的程序量仅为 PASCAL 等过程性语言的 1/10．

3. 向数学靠拢．PROLOG 以一阶谓词逻辑为理论基础，抛弃了赋值、函数调用副作用等难以进行数学论证的概念．现在已建立了 PROLOG 的完整、统一、简洁的语义模型．

4. 理论与实践的权衡． 如上所述，PROLOG 有严格的理论基础．但是严格按照理论要求(正确性和完备性等)的实现在现有硬件支持下是低效率的． 1972 年法国马赛大学实现了第一个 PROLOG 解释系统，1977 年英国爱丁堡大学实现了第一个 PROLOG 编译系统．现在各种大、中、小型计算机甚至微型机上运行的

PROLOG 系统已有数十种. 这些系统为了达到可接受的运行效率，都牺牲了某些理论要求(如完备性)，为了方便用户，又都附加了若干非逻辑的预定义成分. 在这样的权衡下，PROLOG 经编译后可达到甚至超过其他人工智能语言(如 LISP)已达到的效率，并可用于广泛领域的软件开发.

1981 年 10 月，日本公布的第五代计算机研制规划明确指定 PROLOG 为第五代计算机的核心语言. 尽管学术界对此看法不一，但却因此而使 PROLOG 成为计算机界众所瞩目的语言. 可以预料，今后 PROLOG 语言将会不断改进完善，同其他超高级语言一起在新一代计算机系统的发展中起重要的作用. PROLOG 的优点也只有在新一代计算机硬件的支持下才能充分地发挥出来. 在现有的硬件支持下，我们不能认为 PROLOG 是万能的程序设计工具，它有自己的适用范围(比如 PROLOG 不适于表达大量的数值计算).

当前比较适于用 PROLOG 求解的问题主要在人工智能、关系数据库、计算机辅助设计和软件工程中快速原型的建立等领域，但 PROLOG 的应用范围仍在迅速扩大. 随着使用这种语言的人数的增多和实现技术的改进，它将在我国得到进一步的推广和应用. 本书就是为满足广大读者的需要而编写的，它既可作为高等院校计算机专业本科生和研究生的教材和参考书，也可作为不同层次的软件工作者的参考资料.

本书第一至第三章介绍 PROLOG 语言及其程序设计方法. 由于 PROLOG 语言具有严格的数学背景，我们有可能也有必要在一定的理论框架下介绍 PROLOG 的语法和语义. 但这样对初学者可能有一定的困难，所以我们采取由浅入深的方法组织材料. 前两章采取非形式的讲法，第三章再给出形式的处理，以适应不同层次读者的需要. 第一章介绍"纯"逻辑程序设计，第二章逐步引入非逻辑的附加功能. 由于 PROLOG 的程序设计思想和风格与常规语言截然不同，又由于当前的 PROLOG 语言并非完全是非过程的，因此程序员仍需处理若干控制方面的问题，掌握好

PROLOG 程序设计方法并不是非常容易的. 特别是对只熟悉常规语言程序设计方法的人有一个"转弯"的过程. 本书前两章用大量小型例题说明 PROLOG 程序设计的特点和风格，使初学者得以入门，获得在具体应用领域中使用 PROLOG 语言的基础. 第三章给出 PROLOG 语法和语义的形式描述. 这不仅对后面讲解 PROLOG 的实现技术有帮助，也为打算在逻辑程序设计理论方面进行研究工作的读者提供了必不可少的基础知识. 当然，只关心 PROLOG 应用的读者则不必在这一章花费太多精力，特别是打＊号的各节可略去不读. 另一方面，对 PROLOG 已有了解的读者则可快速浏览前两章后，直接从第三章开始学习.

第四至第八章，介绍 PROLOG 语言在人工智能、编译构造、自然语言理解、函数型程序设计和知识工程等几个方面的应用. 这五章的相互独立性较强，每一章集中讲述一个应用领域. 一般地，每章开头对该领域都给出必要的背景知识，然后通过较大型的 PROLOG 程序具体显示 PROLOG 语言在该领域的应用实例，读者可根据自己的需要选读其中的任何一章. 由于逻辑程序设计的效力只有通过较大型的实用问题才能令人信服，而且它的方法和风格也只有通过应用实践才能真正掌握，因此读者最好选择自己比较熟悉的领域，认真读几个大程序，并亲手上机试一试. 在此基础上就可尝试用 PROLOG 解决自己感兴趣的课题了.

最后两章，即第九章和第十章分别介绍 PROLOG 的解释实现和编译实现技术. 除非仅仅是一般地了解一下 PROLOG 语言，凡打算认真学习和使用 PROLOG 语言的读者至少应了解解释实现的技术，这对于正确、流畅地进行 PROLOG 程序设计有极大的好处. 对于打算在自己工作的单位实现效率较高的 PROLOG 系统的读者，本书最后两章提供了相当多的优化技术.

本书尽量包括一些最新的研究成果，其中有些部分包含作者及其指导的研究生的工作. 本书各章都附有适量的习题. 这些习题有一部分旨在对书中的理论内容加深理解，另一部分则可作为上机实习.

当前写 PROLOG 的书有一个难题，这就是该语言没有标准文本。本书采用国内外有关 PROLOG 出版物中普遍使用的、也是国内研制的各种 PROLOG 系统普遍采用的语法形式，书中的程序均可在北京工业大学研制的 BPU-PROLOG 系统下运行。实际上各种 PROLOG 版本的差异纯粹是表面性的，再加上 PROLOG 的语法极为简单，程序量又小，程序移植既无本质的困难，也不需要多少工作量。

本书的初稿是作者在 1985 年秋编写的逻辑程序讲义。作者曾在清华大学、中国地质大学和北京工业大学用该讲义为计算机系学生和研究生试讲过五学期。根据教学反映，作者对讲义作了大量修改补充。作者感谢听课同学提出的修改意见。本书的写作得到中国科学院软件研究所唐稚松研究员、数学研究所陆汝钤研究员、上海交通大学孙永强教授和清华大学林行良教授的热情鼓励和帮助，在此对他们表示衷心的感谢。

• ▼ •

目　录

第一章　基本的 PROLOG 程序设计

PROLOG 是逻辑程序设计语言中最典型的一种，有时泛指各种逻辑程序设计语言．本书中的 PROLOG 语言是特定的，它被描述在第一章和第二章中．这两章的描述是系统的，但不是形式化的，关于逻辑程序设计的形式化描述在第三章给出．对 PROLOG 已有了解的读者在快速浏览前两章之后，可以直接从第三章开始学习．

本章首先通过引例从直观上看一看 PROLOG 是如何解决实际问题的，它的程序是什么样子以及它的运行方式是什么，然后系统地介绍 PROLOG 的语法（程序的外部形状）和非形式语义（程序的含义和运行机制），最后讨论 PROLOG 最常用的数据结构——表及其处理方法．这些内容均属于基本的 PROLOG 程序设计范畴，最后我们指出进一步深化的方向．

1.1　引　　例

例 1.1　设有某家庭成员的集合

$$\{a, b, c_1, c_2, c_3, d_1, d_2, d_3, d_4\}$$

我们已知他们之间的亲属关系为：

b 是 c_1，c_2，c_3 的母亲

c_3 是 d_4 的母亲

c_1 是 d_1，d_2，d_3 的父亲

a 是 b 的丈夫

图 1.1 形象地画出了这些亲属关系．

用通常的关系数据库的术语，我们有 3 个二元关系：

mother（X，Y）：X 是 Y 的母亲

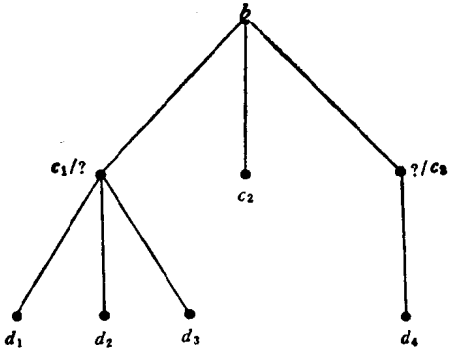

图 1.1 一个家庭关系图

father (X, Y): X是Y的父亲

husband (X, Y): X是Y的丈夫

已知的关系为:

(1) mother (b, c_1).

(2) mother (b, c_2).

(3) mother (b, c_3).

(4) mother (c_3, d_4).

(5) father (c_1, d_1).

(6) father (c_1, d_2).

(7) father (c_1, d_3).

(8) husband (a, b).

以上(1)—(8)在 PROLOG 中是语句,称为无条件子句. 它可以和关系数据库一样, 对无条件子句代表的数据库进行检索.

PROLOG 比关系数据库更强的地方在于它还能进行推理.推理先要有规则,比如要从已知的亲属关系推出谁是谁的祖辈,这就必须先给出祖辈关系的定义. 其中一种定义可以是: 对所有 X, Y, Z,

1) 若X是Y的母亲且Y是Z的母亲,则X是Z的外祖母;

2) 若X是Y的母亲且Y是Z的父亲,则X是Z的祖母;

3) 若X是Y的丈夫且Y是Z的(外)祖母，则X是Z的（外）祖父。

若采用西方习惯，以 grandmother (X,Y) 表示X是Y的祖母或外祖母，以 grandfather(X,Y) 表示X是Y的祖父或外祖父，则上面的关于祖辈关系的定义 1)—3) 在 PROLOG 中可写成：

(9) grandmother(X,Z):-mother(X,Y), mother(Y,Z).

(10) grandmother(X,Z):-mother(X,Y), father(Y,Z).

(11) grandfather(X,Z):-husband(X,Y), grandmother(Y,Z).

(9)—(11) 也是 PROLOG 的语句，称为条件子句. (1)—(11) 合起来构成了一个 PROLOG 程序P，它的含义(说明性语义或描述性语义)通过上面导出程序的过程已经很清楚了. 但为了执行这个程序，我们还要知道P的过程性语义(即其运行机制).

PROLOG 是一种交互式语言. 把程序P输入计算机后它被表示为一种内部形式，然后系统等待用户的提问，即在终端上显示提示符

?-

如果用户想知道a有无孙辈以及他的孙辈都是哪些人，他可以这样提问(提问称为目标子句)：

?- grandfather(a,X).

系统在运行一段时间后将会回答：

yes

$X = d_4$

这表示问题有解，其中一个解为 $X = d_4$，即 d_4 是 a 的一个孙辈. 然后系统等待用户的反应. 若用户满足于这一个解，他可以键入“回车”，系统就结束这次运行，再次显示提示符：

?-

等待用户的下一个提问；若用户想知道a有无其他孙辈，他可以键入“;”，系统就会继续进行搜索和推理并回答：

$X = d_1$

如此下去，直到 a 的孙辈 d_4, d_1, d_2, d_3 全部找到之后．如果用户还想继续求解，系统将回答：

no more solution

?-

即“此问题已没有其他的解了，请问下一个问题”．

PROLOG 到底是怎样解决这个问题的呢？ 这可以从两个角度来解释．若从逻辑的角度来看，我们说提问：

?- grandfather (a, X). (1.1)

代表了一个反证．为了证明 a 有孙辈，我们设其反面：

$\sim\exists_x$ grandfather (a, X) (1.2)

它就是提问 (1.1) 的逻辑原形． PROLOG 把程序 P 中的语句和 (1.2) 放在一起进行推理．若推出矛盾，就从反面证明了 a 确有孙辈：

$\exists_x$ grandfather (a, X) (1.3)

而反证过程中得到的反例 X 就是提问 (1.1) 中 X 的解．具体的反证方法称为 SLD 反驳-消解法 (3.3 节)．注意，P 中的子句若含有变量 X, Y 等，它们的作用域仅限于所在子句，且都是全称量化的，即意指“所有……”，只不过全称量词$\forall$省略不写罢了．

对于熟悉通常的 PASCAL 或 C 语言的读者来说，从过程说明和过程调用的角度来解释可能更加自然． PROLOG 程序 P 可看成对过程 mother, father, grandmother 等的说明． 过程体要么是空(无条件子句)，要么仅包含过程调用(条件子句)．我们把目标子句 ?- grandfather (a, X). 看成是对过程 grandfather 的调用，实参 a 代表输入，实参 X 代表输出．这一调用在运行时依次触发对过程 husband 和 grandmother 的调用． 后者又调用过程 mother 和 father．最后，无条件子句 (1)—(8) 提供了过程的出口．建议读者试一下这种过程性的解释，看看能不能得到 X 的解 d_1, d_2, d_3 和 d_4.

如果你真地试过了，你就会发现 PROLOG 的过程与 PASCAL 或 C 的过程有许多差别，如过程定义是多入口的，参数传递方式不

一样等等．我们在 1.3.2 小节中将系统地讨论这些差别．

例 1.2 迷宫问题．

本例和上例不同，主要用来说明 PROLOG 具有极强的表达能力．程序中有一些细节目前还不能讲得十分清楚，待学完前两章后，就可以彻底明白这些细节了．

考虑图 1.2 所示的迷宫．我们说“匪徒”和“妖怪”代表危险，问题是寻找一条路，从入口进去，找到“财宝”，避开危险，从出口归来．

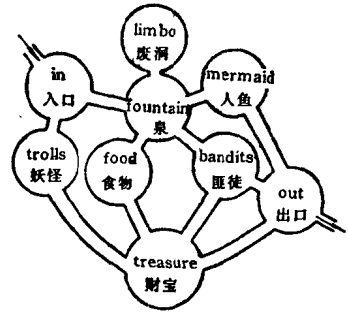

图 1.2 迷宫

假定我们所用的PROLOG可以处理汉字信息．我们将根据对问题的描述(已知事实，探险规则和所求的解) 自然导出求解的 PROLOG 程序，而不必涉及具体的搜索过程．

首先，我们可用下列 24 个无条件子句描述迷宫的结构(由于对称性可以缩减一半)：

(1) 相连(入口，泉)．

(2) 相连(入口，妖怪)．

(3) 相连(泉，食物)．

……

(24) 相连(出口，财宝)．

其次，我们来考虑探险规则．设在任一时刻，已经走到位置 *From*，从入口到此位置所走过的部分路径 (即问题的部分解) 为 *S*，它是部分路径上各位置的序列 (PROLOG 表示序列的数据结构称为表)．下一步怎么办？这里存在两种可能：

1) *From* = 出口且“财宝”在序列 *S* 中．此时应认为 *S* 已是所求之解，只要把 *S* 打印出来即可．用 PROLOG 的一个条件子句可表达为：

(25) path (出口，*S*):-member (财宝，*S*)，

打印(*S*).

这里 path(*X*,*Y*) 意为"若当前已走到位置*X*且部分解序列为*Y*，继续探索以找到全路径"；member(*X*, *S*) 意为*X*属于序列*S*，它应该有自己的定义，但我们将到1.4节再详细讲，现在你只要明白它的含义即可.

2) 在其他情况下，应从 *From* 走到与它相连的、未曾走过的、没有危险的某位置 *Next*；然后认为当前已走到 *Next*，部分解序列增加了一个位置 *Next*，(在 PROLOG 中，增加后的序列表示为扩大的表 [*Next*|*S*]，即把*Next* 插在表 *S* 的最前面所得的表). 如果从新的状态继续求解，就有可能最终找到解. 用 PROLOG 的一个条件子句来表达为:

(26) path(*From*, *S*):-相连 (*From*, *Next*),
not(member(*Next*,*S*)),
not(member(*Next*, [妖怪,匪徒])),
path(*Next*,[*Next*|*S*]).

这里 not(···) 意为…之否定(详见 2.1 节和 3.4.3 小节). 注意，通过"相连 (*From*, *Next*)" 选出的 *Next* 有可能在以后被发现是错误的，在这种情况下 PROLOG 系统能自动回到这一点，重新选取另一个新位置，这称为回溯.

子句(1)—(26)构成了一个 PROLOG 程序 *P*，它描述了问题的已知事实和规则，并且这种描述实为文字描述的自然翻版.

最后，问题所求的解用目标子句表达:

?-path (入口,[入口]).

它要求从入口开始寻找满足解条件(无危险、获得财宝和从出口归来)的序列 *S*.

PROLOG 系统将会回答:

yes

S = [出口,财宝,食物,泉,入口]

注意序列 *S* 是倒排的，因为从(26)中我们知道位置是反向插入 *S* 的.

这两个例子除了说明 PROLOG 程序的书写方式和运行机制外，特别显示了 PROLOG 语言的如下特点：

1. 语法极为简单。

2. 表达能力极强，程序极为简洁。例 1.2 若用 PASCAL 或 C 来写决不是一件轻而易举之事。你首先得设计复杂的数据结构来表示迷宫，然后你还得编写某一种无向图搜索算法来搜索迷宫。算法中含有一切控制流程的细节。整个编程工作量在 PROLOG 的 10 倍之上。

3. PROLOG 是一种说明性语言（又称描述性语言或应用式语言）。与通常的过程性语言（或称命令式语言）不同，它只需描写问题的逻辑方面（即做什么），而把解题的控制细节（怎么做）交给计算机系统去自动完成。严格地说，只有第三章描述的抽象的逻辑程序设计语言才真正具备这一特征，但 PROLOG 已基本具备说明性语言的特征，正如本节的两个例子所表明的。

1.2 PROLOG 的语法

PROLOG 的语法极为简单，它的 BNF 描述大约只占一页篇幅。麻烦在于 PROLOG 尚无标准文本，各种 PROLOG 系统总有一些小的差异。我们在这里描述的 PROLOG 语法是基于北京理工大学开发的 BPU-PROLOG 系统，它与当前世界上流行的大多数 PROLOG 系统采用相同的语法，适于教学。本书所有程序均在 BPU-PROLOG 系统下调试过。个别地方叙述的内容若超出 BPU-PROLOG 的范围，我们总是加以特别的注解。无论如何，PROLOG 的语法极为简单，PROLOG 程序一般不会太长，各种 PROLOG 系统的差异又是微小的，所以程序移植问题是极易解决的。

我们先讲词法要素，然后自底向上地给出 PROLOG 语法，最后列出其 BNF 描述。

1.2.1 词法要素

1. 变量.

变量是以大写字母打头的字母数字串，如 X, Y, Z, From Next, S, X1, Y2 等. 有一种变量,它只在子句中出现一次,它的拼写方式无关紧要,我们就用"_"代表它，称为匿名变量. 一个子句中若出现多个"_",则每一个"_"代表一个不同的匿名变量.

2. 整数.

整数如 125, 0, −23 等与通常的程序设计语言中的表示一样. 整数是逻辑常数的一种，但可用算术预定义谓词(见 2.2 节)对它进行算术运算,它还可出现在中缀形式的算术表达式中.

3. 标识符.

以小写字母打头的字母数字串统称标识符，用于表示非整数的逻辑常数、函数符或谓词符. 常数标识符既可看成 0 元函数符也可看成 0 元谓词符. 一个标识符究竟代表什么取决于上下文.

4. 保留字.

有些标识符被系统用来表示预定义的常数、预定义的函数和预定义的谓词，用户不得另作他用. BPU-PROLOG 使用 100 多个保留字,详见 2.1 节—2.8 节.

5. 正文常数.

以双引号括起来的不包括"%"号的任意字符串称为正文常数. 它的作用同用标识符表示的常数一样，但大大增强了常数的表达力. 例如要表示当前输入文件是 *a*.log，则可写成（见 2.4 节):

see ("*a*.log")

6. 运算符和分隔符.

下一小节讲述 PROLOG 语法结构时逐一引入它们.

7. 注解.

源程序中若遇到"%"，则从此直到"%"所在行的结束被认为是注解,语法分析时跳过此部分.