第35卷 第6期 西南大学学报(社会科学版) 2009年11月
Vol. 35 No. 6 Journal of Southwest University (Social Sciences Edition) Nov. ,2009

*

# 基于谓词逻辑的 Prolog 程序设计

李 娜,王 湘 云
(南开大学 哲学系,天津市 300071)

**摘 要:**一阶谓词逻辑下的 Horn 逻辑是人工智能程序语言 Prolog 的理论基础,利用 Prolog 在计算机上可实现机械化,从而使自动化求解问题和定理证明具备可行性。本文从 Horn 逻辑和 Prolog 的基础理论出发,使用 Horn 子句、SLD－归结、搜索和回溯等原理讨论了如何在计算机中实现数学函数、定理证明等自动推理的一些应用。

**关键词:**谓词逻辑;Prolog;Horn 逻辑;SLD－归结

**中图分类号:**B81 **文献标识码:**A **文章编号:**1673-9841(2009)06-0048-05

Prolog(Programming in Logic)是面向逻辑、面向问题,描述逻辑关系和抽象概念,处理对象是知识(确切地说是符号)的一种逻辑型人工智能程序设计语言。Prolog 是陈述性语言而不是过程性语言,在 Prolog 程序中不需要告诉计算机“怎么做”,只需告诉计算机“做什么”,即只要给出所需的事实和规则,Prolog 使用演绎推理的方法就可自动地对问题进行求解。Prolog 是一个典型的符号逻辑形式系统,并且是以一阶谓词逻辑为基础设计的,其目标就是处理逻辑推理。它是以一阶谓词逻辑的 Horn 子句集为语法,以归结原理为工具,加上深度优先搜索的控制策略而形成的逻辑程序。由于其程序子句的形式和一阶谓词逻辑演算表示事物的方式十分相似,因而很适合表示人的思维和推理规则。此外,Prolog 本身就是一个演绎推理机,具有匹配(即合一)、回溯、递归等功能,并具有正向推理和反向推理策略,从而可在计算机上完成自动推理。

## 一、Prolog 的逻辑理论基础

### (一)Horn 逻辑

Prolog 的逻辑理论基础是 Horn 逻辑。Horn 逻辑是对事物及其相互关系进行推理的形式系统,它是由 Horn 子句组成的一阶谓词逻辑的子部分

**定义 1** (1)一个 Horn 子句是至多包含一个正文字的子句。

(2)一个程序子句是只包含一个正文字的子句。一般形式为:$A$ :- $B_1, B_2, \cdots, B_n$。

(3)若一个程序子句包含有负文字,则称它为一个规则。即,在(2)中的一般形式中,$n>0$。

(4)一个事实(或称元子句)是只包含一个正文字的子句。一般形式为:$A$ 或者 $A$ :-。

(5)一个目标子句是不包含正文字的子句。在 Prolog 中目标子句作为一个问题来输入。一般形式为:? - $B_1, B_2, \cdots, B_n$。

(6)一个 Prolog 程序是一个只包含程序子句(规则或事实)的子句集。

由定义,Horn 子句或是程序子句或是目标子句,而程序子句或是规则或是事实。即,Horn 子句由事实、规则和目标组成。

考虑与 Horn 子句对应的逻辑形式。假设子句 $C$ 中含有正文字 $A_1, \cdots, A_m$ 及负文字 $B_1, \cdots, B_n$,则 $C$ 与 $A_1 \vee A_2 \vee \cdots \vee A_m \vee \neg B_1 \vee \neg B_2 \vee \cdots \vee \neg B_n$ 逻辑等值,也逻辑等值 $B_1 \wedge B_2 \wedge \cdots \wedge B_n \rightarrow A_1 \vee A_2 \vee \cdots \vee A_m$。若 $C$ 中至多包含一个正文字,则 $C$ 就是 Horn 子句。在 Prolog 中表示蕴涵关系的次序与谓词演算逻辑蕴涵的次序正好相反,且用“:-”代替“$\rightarrow$”,用“,”代替“$\wedge$”。因此,程序子句的一般形式 $A$ :- $B_1, B_2, \cdots, B_n$ 对应的逻辑形式为 $B_1 \wedge B_2 \wedge \cdots \wedge B_n \rightarrow A$。

### (二)Prolog 程序的解释

在 Prolog 中,事实表示的是对象的属性或者对象间的关系,对象之间的关系用谓词来定义。规则是能够从某些事实推断出新事实的凭据,即利用已知前提推出结论的子句。目标是询问系统基于已知事实和规则,提出的回答有关这些关系的问题。

**定义 2** Horn 子句在 Prolog 中的解释如下:

(1)事实{$p(X)$}只包含简单的正文字 $p(X)$,在 Prolog 中显示为:$p(X)$.

* **收稿日期:**2009-04-27

**作者简介:**李娜(1958-),女,河南开封人,南开大学哲学系,教授,博士生导师,主要研究现代逻辑。

(2)规则 $C=\{p(X),\neg q_1(X,Y),\cdots,\neg q_n(X,Y)\}$，在 Prolog 中显示为：$p(X)$ ：- $q_1(X,Y),\cdots,q_n(X,Y)$.

(3)对于(2)中的规则 $C$，我们称 $p(X)$ 是 $C$ 的目标或头；称 $q_1(X,Y),\cdots,q_n(X,Y)$ 为 $C$ 的子目标或体；连接 C 的头和体的符号"：-"叫作颈。

(4)一个(Prolog)程序是一个只含有程序子句(规则和事实)的公式(子句集)。

一个 Prolog 程序实际上就是一个由事实和规则组成的 Horn 子句集，询问就是一个目标。运行一个 Prolog 程序就是通过计算机判断由程序和目标组成的 Horn 子句集是否有模型。我们也可以把 Prolog 程序看作一个由事实和规则构成的动态数据库或知识库，Prolog 系统的任务就是根据知识库中的知识来回答询问，即证明事实和规则与目标是矛盾的，或者说程序中的子句集是不可满足的。这就是 Prolog 的说明性语义。

## 二、Prolog 的基本原理

谓词逻辑下的归结原理是机器推理或自动推理的主要方法，可实现定理的自动证明和问题的求解，是一种可在计算机上实现的逻辑推理算法。由于 Prolog 是基于 Horn 子句的逻辑程序，因而其运行机理自然就是基于归结原理的演绎推理，程序的执行过程就是从目标子句出发，并不断进行匹配(合一)、归结或回溯，直至目标被完全满足或不能满足时为止的归结演绎推理过程。

### (一)SLD－归结

Prolog 的归结演绎方法称为 SLD(Linear resolution with Selection function for Definite clause)归结。在 Prolog 程序的执行过程中，其具体实现方法是：自上而下匹配子句(即合一)；从左向右选择子目标；归结后产生的新子目标总是插入被消去的目标处(即目标序列的左边)。SLD－归结就是 Prolog 程序的运行机理，也就是所谓的 Prolog 语言的过程性语义。

**定义** 3　令 $C$ 是任一子句，$S$ 是任一公式，则

(1)满足下面三个条件的子句对序列 $\langle C_0,B_0\rangle,\cdots,\langle C_n,B_n\rangle$ 称为公式 $S$ 上的子句 $C$ 的一个线性归结：

①$C_0$ 和每个 $B_i$ 或者是 $S$ 中的元素换名替换后得到的子句，或者是 $C_i$，且 $j<i$；

②每个 $C_{i+1}(i\leqslant n)$ 是 $C_i$ 和 $B_i$ 的一个归结式；

③$C_{n+1}=C$。

(2)如果存在公式 $S$ 上的子句 $C$ 的一个线性归结(LD)，则称子句 $C$ 是从公式 $S$ 可演绎的，记作 $S\square_L C$；如果空子句□是从公式 $S$ 可演绎的，则存在一个线性归结反驳。

令 $P$ 是一程序，$G$ 是一目标子句，则：

**定义** 4　设 $\langle G_0,C_0\rangle,\cdots,\langle G_n,C_n\rangle$ 是一个线性归结证明，称 $G_{n+1}$ 是一个线性输入，若 $G_0=G$，$G_{n+1}=\square$，则称 $G_{n+1}$ 是 $P\cup\{G\}$ 的归结反驳。其中：任一 $G_i$ 是目标子句且任一 $C_i$ 是 $P$ 中的一个(换名)子句。

线性输入归结是 Prolog 程序进行归结证明的一般格式。但在运行过程中，Prolog 解释器(即解释程序)并不能检查出是否应该消除合一置换后可能重复的结果。为进一步简化结果，可以不像机器那样把子句看作文字的集合，而是把它看作有序子句，即文字的序列。

**定义** 5　(1)若 $G=\{\neg A_0,\cdots,\neg A_n\}$ 和 $C=\{B,\neg B_0,\cdots,\neg B_m\}$ 是有序子句，且 $\theta$ 是 $A_i$ 和 $B$ 的最一般合一，则可做 $G$ 和 $C$ 在文字 $A_i$ 上的有序归结。归结后得到的(有序)归结式为有序子句：$\{\neg A_0,\cdots,\neg A_{i-1},\neg B_0,\cdots,\neg B_m,\neg A_{i+1},\cdots,\neg A_n\}\theta$.

(2)若 $P\cup\{G\}$ 是已知有序子句集，则把满足下列条件的有序子句 $G_i$ 和 $C_i$ 的一个序列 $\langle G_0,C_0\rangle,\cdots,\langle G_n,C_n\rangle$ 称作 $P\cup\{G\}$ 的一个 LD－反驳：

①$G=G_0$，$G_{n+1}=\square$；

②任一 $G_i$ 是有序目标子句；

③任一 $C_i$ 是 $P$ 中的一元素换名后得到的子句，且 $P$ 中不包含 $G_j(j\leqslant i)$ 或 $C_k(k<i)$；

④任一 $G_{i+1}(0\leqslant i\leqslant n)$ 是 $G_i$ 和 $C_i$ 的一个有序归结式。如果 $C_n$ 不是□，则通常称该序列是一个 LD－归结证明。

在 LD—归结证明中，关键问题是选择 $G_i$ 中的哪一个文字来归结。在 Prolog 执行过程中使用的选择定则总是归结 $G_i$ 中最左边的文字。从 $C_i$ 推导出的归结式中的文字总是继承它们最初的顺序，并被放在所有从 $G_i$ 得到的子句的最左边。这种使用选择定则的归结方法就是 SLD－归结。

SLD－归结是可靠的和完全的。当输入一个形如"? -$A_1,\cdots,A_n$"的问题后，Prolog 解释器的工作过程，就是从程序 $P$ 和目标子句 $G=\{\neg A_1,\cdots,\neg A_n\}$ 中搜索空子句□的一个 SLD—归结证明。若不能找出这样的一个证明，则回答"no"；否则给出一个回答置换，即对 $G$ 中的变项的一个合一置换。如果解释器找出的证明是 $\langle G_0,C_0\rangle,\cdots,\langle G_n,C_n\rangle$，且其具有最一般合一 $\theta_0,\cdots,\theta_n$，则给出对 $G$ 中的变项加限定的回答置换 $\theta=\theta_0,\cdots,\theta_n$。最重要的是，这些回答置换一定是正确的，即 $(A_1\wedge\cdots\wedge A_n)\theta$ 是 $P$ 的逻辑后承。

### (二)搜索和回溯

Prolog 采用深度优先的搜索策略，且使用回溯机制来执行自动搜索及归结演绎推理的过程。尽管 SLD－归结是可靠的并且是完全的，但 Prolog 的实现却不是这样。问题有两个方面：其一在于合一算法的执行。Prolog 的实现在合一算法中省去了"occur"检查，可能使程序陷入死循环；其二，除非在 SLD－归结中要求，否则 Prolog 定理证明器不进行合一置换。这两方面破坏了系统的可靠性。

Prolog 实现的不完全性来自搜索 SLD－归结的方法。若 $P$ 是一程序，$G$ 是一目标，则逻辑程序的求解过程，就是寻找 $P\cup\{G\}$ 的 SLD—归结反驳的过程。在 SLD—归结的第 $i$ 步，需要选择 $P$ 中哪一个子句与当前目标子句 $G_i$ 最左边的项进行归结。可用 SLD－树表示

整个 SLD 推导的过程。树的根结点标记为 $G$，如果一结点标记为 $G'$，则它的后继的标记是与 $P$ 中所有可归结子句归结的结果，下一个归结式放在 $G'$ 的最左边。沿着树的每一个分支的线上，标记着程序 $P$ 中被归结子句的序号。约定：树中每个结点的后继的排列顺序为从左至右，与 $P$ 中子句的顺序一致。若一分支以□结束，则称其为成功路径，且其对应 Prolog 的正确回答。每一成功路径结束位置下是该路径所代表的证明的回答置换。若一分支结束于一个子句 $G'$，且 $P$ 中已经没有子句与 $G'$ 中最左边的项归结，则称其为失败路径。例如，设程序 $P$ 为：

$$p(X,X):-q(X,Y),r(X,Z). \quad (1)$$

$$p(X,X):-s(X). \quad (2)$$

$$q(b,a). \quad (3)$$

$$q(a,a). \quad (4)$$

$$q(X,Y):-r(a,Y). \quad (5)$$

$$r(b,Z). \quad (6)$$

$$s(X):-q(X,a). \quad (7)$$

问题 $G$ 为：? -$p(X,X)$. 以目标子句 $G=\{\neg p(X,X)\}$开始的程序 $P$ 的 SLD－树如下：

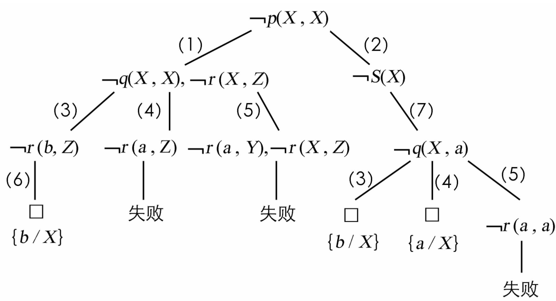


**图 1　SLD－树**

Prolog 程序的定理证明器总是首先尝试搜索最左边的项，即将当前目标 $G$ 与 $P$ 中最左边的可归结子句归结。如果定理证明器遇到一个失败的结点，则产生回溯。回溯就是沿走过的路径折回，直到找到折回路径右边某分支的一个结点。如果有多个这样的路径，则选择最左边的一个。定理证明器重复此过程，直至找到一个成功路径。

当重复向 Prolog 询问时，回溯的执行过程相同。从 $P$ 的 SLD—树可以看出，沿(1)、(3)、(6)的路径可得到一个正确的回答置换$\{b/X\}$。继续询问，即，输入“? -$p(X,X)$”或“;”，定理证明器从第一个成功路径的末结点开始产生回溯，至上一个有可选路径的结点$\neg q(X,Y)$，$\neg r(X,Z)$。然后尝试与(4)归结，失败；尝试与(5)归结，也失败。则继续向上回溯至根结点$\neg p(X,X)$，尝试分别与(2)、(7)、(3)依次归结，再次得到回答置换$\{b/X\}$。继续求解，则从成功结点又开始回溯至$\neg q(X,a)$，尝试与(4)归结，成功，并给出回答置换$\{a/X\}$。再次询问，回溯至$\neg q(X,a)$，尝试与最后剩下的(5)归结，失败，并且报告找不到其他成功路径，也没有其他可尝试的路径。如果输入问题后得到的回答是“no”，则情况也类似，这意味着 SLD—树中的所有路径都已被搜寻过且都是失败的。

上述这种搜索过程称为深度优先搜索策略。也就是在沿着树中的其他分支搜索之前，尽最大可能尝试执行到树中一条路径的末端才停止。相反，在图 1 中，若搜索过程以$\neg p(X,X)$；$\neg q(X,Y)$；$\neg r(X,Z)$；$\neg s(X)$ ¬；$\neg r(b,Z)$；$\neg r(a,Z)$；$\neg r(a,Y)$；$\neg r(X,Z)$；$\neg q(X,a)$；□；失败；失败；□；□；$\neg r(a,a)$；失败这样的顺序，则称为宽度优先搜索。显然，深度优先搜索比宽度优先搜索节省了搜索时间，但却失去了 SLD—归结方法的完全性。一般完全性定理保证，如果存在一个长度为 $n$ 的 SLD—归结证明，则使用宽度优先搜索在搜索到树的深度 $n$ 时，一定能找到一个这样的证明，即，已经搜索了长度为 $n$ 的每个可能的路径。深度优先搜索并没有这种保证，问题在于树的某些路径可以是无穷的，深度优先搜索会造成“死循环”。此外，子句的排列顺序程序实现的过程中也发挥着关键作用，但子句顺序的重新排列也不能完全保证系统的完全性。因此，尽管深度优先搜索策略是简单有效的，但实质上却是不完全的。要从根本上解决其产生的不完全性问题，还需要结合其他搜索策略或 Prolog 内部的一些控制机制。

### （三）控制：Cut

逻辑程序设计的基本公式是：算法＝逻辑＋控制，其中逻辑部分说明的是让计算机“做什么”，控制部分则说明计算机“怎么做”。一般说来，只需给出逻辑部分，控制部分可由逻辑程序系统自动处理。

Cut 是 Prolog 中控制搜索过程的一个非常重要的内部谓词。使用 Cut 可告诉 Prolog，当要回溯前面一串已满足的目标时，不必再考虑那些目标的其他可选择分支。从语法上讲，可以把 Cut 简单地看作一个特殊的文字，记作“!”，用在程序规则子句的体部或目标子句中。它没有任何说明性语义，但是却可以改变程序的运行过程。在一个规则中，“!”就像是一个目标，它将立即成功，但不能被重新满足，即当回溯再次到达这一目标时，不可能再找到另一个满足目标的成功解。换句话说，企图再满足它时则立即引起失败。

从过程性语义上来看，设目标子句 $G$ 为：? -$A_0,\cdots,A_{i-1},A_i,A_{i+1},\cdots,A_n$，即 $G=\{\neg A_0,\cdots,\neg A_{i-1},\neg A_i,\neg A_{i+1},\cdots,\neg A_n\}$，程序子句 $C$ 为：$A:=B_0,\cdots,B_j,!,B_{j+1},\cdots,B_m$，即 $C=\{A,\neg B_0,\cdots,\neg B_j,!,\neg B_{j+1},\cdots,\neg B_m\}$，假设求解 $A_0,\cdots,A_{i-1}$ 后得到的目标为 $G'$，则求解 $A_i$，若 $\theta$ 是 $A_i$ 和 $A$ 的最一般合一，经合一置换后的归结式成为新的目标子句 $G''$，即，$\{\neg A_0,\cdots,\neg A_{i-1},\neg B_0,\cdots,\neg B_j,!,\neg B_{j+1},\cdots,\neg B_m,\neg A_{i+1},\cdots,\neg A_n\}\theta$。称 $G'$ 是含有“!”的子句 $C$ 的父目标，$A_i$ 为截断点。当求解“!”时，它总是成功。但当“!”后面的子目标无解而引起回溯时，就不考虑 $A_i$ 到“!”之间的所有的其他解，Prolog 直接求解 $A_i$ 的子目标 $A_{i-1}$ 的其他解。即在 SLD－树上，“!”剪去了以它的父目标为根的那个子树中尚未搜索的分支。

例如，设程序 $P$ 为：

$t:-p,r.$ (1)

$t:-s.$ (2)

$p:-q_1,q_2,!,q_3,q_4.$ (3)

$p:-u,v.$ (4)

$q_1.$ (5)

$q_2.$ (6)

$s.$ (7)

$u.$ (8)

问题 $G=\{\neg t\}$，$P$ 的 SLD－树如下所示：

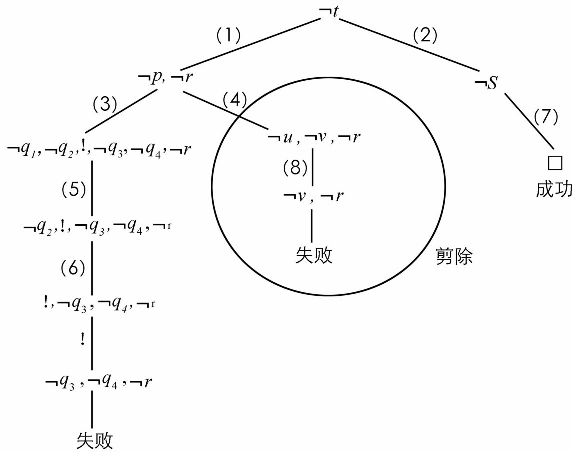


**图 2　有“!”的 SLD－树**

通常，在沿最左边的分支搜索进程中，直到遇到失败的结点，才会引起回溯。但是插入“!”后，控制信息则直接传递到标记为(s 的结点，且使得目标立刻成功。

使用“!”可以加快程序运行的速度，因为它不会浪费时间去试图满足那些能事先知道不可能导致解答的目标；也可使程序占有较少的存储空间，因为 Prolog 系统不必为它以后的搜索记录那些对求解毫无帮助的回溯点信息。但若使用不当，有时会剪去所有的成功分支或无穷分支，从而破坏 SLD－归结的完全性。

Cut 的另外一个重要用法是对否定的定义。“not”是 Prolog 的一个内部谓词，与通常逻辑中的否定(¬)含义不同。通过“!”来定义“not”，其真正的含义指的是“对于子句 $P$，如果 $P$ 失败(即，是不可证的)，则 not($P$)是可证的。”可定义为：

$not(A):-A,!,fail.$

$not(A).$

如果调用 not($A$)，Prolog 转向定义的第一个子句并调用 $A$。如果 $A$ 成功，略过“!”，遇到“fail”。“fail”也是 Prolog 的内部谓词，作为一个目标它总是失败并引起回溯。所以，如果 $A$ 成功，则 not($A$)失败；如果 $A$ 失败，则尝试定义的第二个子句 not($A$)，且 not($A$)成功。

因此，在 Prolog 程序中放置 Cut 谓词、引入“not”规则可解决使用 SLD－归结及深度优先搜索可能造成的死循环问题。

## 三、逻辑程序设计

从前述内容可以看出，Prolog 的运行机理是具有回溯控制，并使用深度优先搜索的 SLD－归结演绎的方法，其作用正好是将一阶谓词逻辑演算中的说明性语义以过程性语义解释，使得说明性的命题成了可执行的过程。所以，利用 Prolog 在计算机上可实现机械化，从而使计算机自动化求解问题和定理证明具备可行性。例如：

### (一)数学函数

例　定义产生 Fibonacci(斐波那契)数列的数学函数。

注：Fibonacci 数列：1，1，2，3，5，8，13…，即，设 $F_n$ 是数列的第 $n$ 项，且 $n=0,1,2,3\cdots$，则 $F_0=F_1=1$，$F_n=F_{n-1}+F_{n-2}=1(n>1)$。

1. 定义描叙 Fibonacci 数列项的谓词

令 fib(N,X)表示“X 是 Fibonacci 数列的第 N 项”。使得目标 fib(N,X)在 N 取某自然数值的情况下，X 被例化为 Fibonacci 数列的第 N 项。如，若输入问题：? -fib(5,X)，则 Prolog 回答：8。

2. 使用 Horn 子句定义生成 Fibonacci 数列项的规则

利用 Prolog 的内部谓词“!”实现分情况选择。即针对数列前两项等于 1 的特殊情况，定义第 0 项和第 1 项为 1 的子句：

fib(0,1) : -!. (1)

fib(1,1) : -!. (2)

谓词“!”可确定仅在子句(1)和(2)两种情况中选择一种执行。

采用递归的方法定义数列第三项开始的其余各项生成的规则：

fib(N,X) : -N1＝N-1，N2＝N－2，fib(N1,X1)，fib(N2,X2)，X＝X1＋X2.　(3)

函数 fib(N,X)是由递归法定义的，子句(1)和(2)即递归出口，也就是它的终止条件。子句(3)表示，数列第 N 项的值 X 是它前面两项值的和。

3. Prolog 算法

程序：fib(0,1) : -!.

fib(1,1) : -!.

fib(N,X) : -N1＝N－1，N2＝N－2，fib(N1,X1)，fib(N2,X2)，X＝X1＋X2.

目标：? -fib(N,X).

### (二)定理证明

例 家庭关系问题。已知事实：对所有的人 $X$、$Y$ 和 $Z$，如果 $X$ 是 $Y$ 的父亲，$Y$ 是 $Z$ 的父(母)亲，则 $X$ 是 $Z$ 的祖父；如果 $X$ 是 $Y$ 的父亲，则 $X$ 是 $Y$ 的父(母)亲；如果 $X$ 是 $Y$ 的母亲，则 $X$ 是 $Y$ 的父(母)亲。若 John 是 Mary 的父亲，Mary 是 Sue 的母亲，证明 John 是 Sue 的祖父。

(1)定义表示关系的谓词。

$g(X,Y)$：$X$ 是 $Y$ 的祖父；$p(X,Y)$：$X$ 是 $Y$ 的父(母)亲；$f(X,Y)$：$X$ 是 $Y$ 的父亲；$m(X,Y)$：$X$ 是 $Y$ 的母亲。

(2)用 Horn 子句定义已知的规则和事实。

规则子句：$g(X,Y)$ : -$f(X,Y)$，$p(Y,Z)$；$p(X,Y)$

$:- f(X,Y);p(X,Y):-m(X,Y)$

事实子句：$f$(John,Mary)；$m$(Mary,Sue)

(3)待证明的问题是 $g$(John,Sue)。即，目标子句 $G=\{\neg g(\text{John},\text{Sue})\}$。

(4)Prolog 算法：

程序 $P$：

$g(X,Y):-f(X,Y),p(Y,Z).$ (1)

$p(X,Y):-f(X,Y).$ (2)

$p(X,Y):-m(X,Y).$ (3)

$f$(John,Mary). (4)

$m$(Mary,Sue). (5)

目标 $G$：? -$g$(John,Sue)

则 Prolog 回答：yes.

Prolog 实现该问题证明的过程如图 3 所示。即 Prolog 系统运用回溯控制，使用深度优先策略从 $P$ 和 $G$ 中搜索空子句□的一个 SLD－归结证明，自动进行演绎推理的过程。

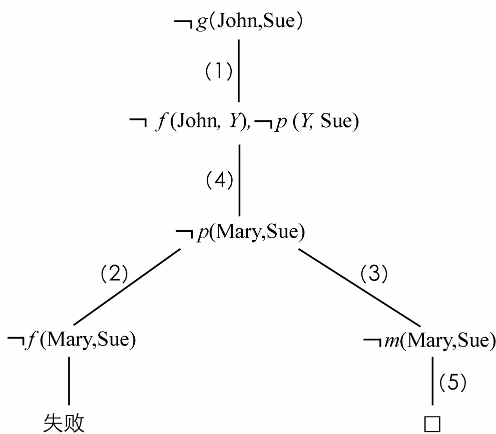


**图 3 “家庭关系问题”的 SLD－树**

## 四、结　语

严格地说，Prolog 程序并不是真正意义上的计算机程序。传统意义上的计算机程序指的是规定计算机运行步骤的一些指令和编码的集合，即机器语言。我们编写的 Prolog 程序是用特定的方法来描述一个问题，实际上是解决此问题的一个算法。要让计算机解决问题，还必须编写出计算机能自动执行的正确程序。例如，上述求解问题的具体步骤可由 Prolog 语言的解释程序 Visual Prolog 通过计算机来处理。

编写计算机程序的前提就是设计出算法。程序、算法和逻辑三者的关系用公式表示：

程序 ＝ 算法 ＋ 数据结构

算法 ＝ 逻辑 ＋ 控制

这样，现实世界中的问题只要能用谓词逻辑的形式语言表示出来，就可以对其进行逻辑程序设计，编写出 Prolog 程序，然后在计算机上实现：现实世界问题⇒谓词逻辑表示⇒Prolog 程序设计⇒计算机实现。因此，通过这种方法可使计算机自动推理得以实现，从而解决各种实际问题。

谓词逻辑作为人工智能研究中的主要形式化工具，其完备的演算系统，特别是应用于 Prolog 中的逻辑推理算法：SLD－归结法使得 Prolog 解释器成为可能、计算机自动化具备可行性。但我们也应该看到，谓词逻辑下的归结原理是一种单调性推理，因而对于知识不完全条件下的非单调推理无能为力。因此，需要加强对高阶谓词逻辑以及各种非经典逻辑系统的研究以突破经典数理逻辑在自动推理上的局限性，以适应计算语言学和智能科学发展的需要，使现代逻辑在人工智能领域中继续发挥其卓有成效的作用。

**参考文献：**


[1] 何华灿，何智涛．从逻辑学的观点看人工智能学科的发展[C]//涂序彦．人工智能 回顾与展望．北京：科学出版社，2006：77-111.

[2] 陆汝钤．人工智能(上、下册)[M]. 北京：科学出版社，1995.

[3] 史忠植．高级人工智能[M]. 北京：科学出版社，2006.

[4] 史忠植．智能科学[M]. 北京：清华大学出版社，2006.

[5] Ulf Nilsson，Jan Maluszyński. Logic，Programming and Prolog，Second Edition[M]. John Wiley & Sons Ltd，1995.

[6] George F. Luger. Artificial Intelligence：Structures and Strategies for Complex Problem Solving，Fourth Edition[M]. 北京：机械工业出版社，2003.

[7] Anil Nerode，Richard A. Shore. Logic for Application，Second Edition[M]. 北京：机械工业出版社，2006.



责任编辑　刘荣军


## Prolog Programming Based on Predicate Logic


LI Na，WANG Xiang-yun

(*Department of Philosophy*，*Nankai University*，*Tianjin* 300071，*China*)



**Abstract**：Horn logic in first order predicate logic is the theoretical basis of artificial intelligence program language Prolog. The use of Prolog in computer can implement mechanization. This causes the problem of automatic solving and theorem proof to be viable. The paper starts with an introduction to the theoretical foundations of Horn logic and the basic principles of Prolog. Then in the next part of the paper the principles of Horn clause，SLD resolution，searching and backtracking are used to discuss how to carry out some applications of automatic inferences in computer，such as mathematical function and theorem proving.



**Key words**：predicate logic；Prolog；Horn logic；SLD resolution