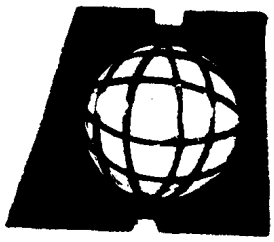
逻辑程序设计语言PROLOG
——基础、原理、实现和应用

文　迅

中国科学院H电脑公司
1985·7·北京



逻辑程序设计语言 PROLOG
——基础、原理、实现和应用
编 辑：
出 版：中国科学院H电脑公司
印 刷：北京医学院印刷厂
订 购：北京8721信箱

序

本书介绍作为第五代计算机系统的主体语言 Prolog。其内容涉及 Prolog 的基础，原理，实现和应用诸方面。

第一章是前言。第二章介绍 Prolog 的基本语法。第三章是 Prolog 的基本数据结构，第四章讨论 Prolog 的基本算法和具体实现方案。第五章说明 Prolog 和用户的接口—输入输出。第六章说明所有的标准 Prolog 所应提供的内部谓词。第七章提供一些比较简单的例子。虽然有的例子已穿插在全书各处。第八章阐述 Prolog 和逻辑之间的关系。在附录A中给出一些实用的 Prolog 程序，这包括符号差分求解，语言编辑系统，通用数据库查询系统和一个专家系统的外壳。所有这些程序都可在H电脑公司的中西文兼容的 Hprolog 解释下运行。最后，在附录B中给出了用 UNIX YACC 格式写成的一个 Prolog 的语法描述，这个描述是 Hprolog 解释系统的一部分。

本书的对象当然主要是 Prolog 语言、第五代计算机人工智能和专家系统方面的研究工作者，有关方面的大学高年级学生和研究生。对于 Prolog 只有一般兴趣或只想了解 Prolog 的使用的读者，可以略去第四章第四节和整个第八章。附录A主要希望对从事语言编译，数据库管理和专家系统等 Prolog 最重要应用领域工作的读者能有所帮助。至于其他读者，如果不是特别有兴趣也可不读。

本书是应各方面要求赶出来的一篇急就章，主要是希望能将 Prolog 的基本情况向国内读者介绍一下。既是急就章，缺点乃至错误当然会很多，希望各界朋友们批评指正。谢谢。

1985.5.30.

3.35元

目　　录

一、前　言

常规程序设计语言经过30年的发展，目前可以说已经接近登峰造极的地步。Ada 语言的出现就是一个明显的标志。但事实已经证明，在常规语言概念限制之下，用于解决“软件危机”的种种方法收效有限，运用超高级语言作为软件开发的工具已经势在必行。同时，由于集成电路和计算机硬件技术的飞速发展，超高级语言也已有可能进入实用阶段。目前已经出现或已经提出的超高级语言大致可分为如下几类：

- 逻辑型语言
- 函数型语言
- 面向目标的语言。
- 逻辑／函数混合型语言。

这些语言的产生和实现已经对程序设计技术的发展产生了极其深远的影响。同时对计算机体系结构的变化发展也必将起到极大的推动作用。

本书所要介绍的 Prolog 语言就属于逻辑型超高级语言的范畴，它又称为人工智能的高级语言（相比之下，多年来在人工智能领域中广泛使用的 LISP 语言则只能称为“汇编语言”）。

1. 发展简史

七十年代初，R. Kowalski 和 P. Hayes 提出了一整套逻辑程序设计思想，这种思想的核心是在一阶逻辑的基础上，用句型（主要是 Horn 子句）对问题进行描述，从而达到对问题进行证明或求解的目的。1972 年前后，法国 Marseille 大学的 A. Colmerauer 在这种思想的基础上设计了 Prolog 语言，并用 Algol-W 实现了第一个 Prolog 解释程序。从名字上看，Prolog 是逻辑程序设计（Programming in Logic）的缩写。因此可以想象设计者最初的设计目标只是处理逻辑推理，以便对抽象问题求解；而且直到七十年代末，Prolog 语言在计算机界仍然没有多大影响，尤其在我国则简直没有引起任何注意。八十年代初，尤其是 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之后，Prolog 才开始成为一种在计算机界众所嘱目的语言。此间，1980 年到1982年在匈牙利，美国和法国召开的几次国际会议上，Prolog 的呼声日高。1977 年，Edinburgh 大学的 D. Warren等在 DEC—10 上实现的第一个 Prolog 编译系统证明 Prolog 与其它人工智能语言一样能够得以高效实现。到目前为止，在各种大、中、小型，甚至微型计算机上实际运行的 Prolog 解释、编译系统已有数十种（参看表 1）。1984 年初，我们也在 TRS-80, Intel 86/330, Dual-68000, 和 IBMPC/XT, IBMPC/AT, M24和AT&T3B2 等微型计算机上分别实现了功能强弱不同的 Prolog 解释系统。

2. Prolog 的特点及其应用

作为一种逻辑型超高级语言，Prolog 的解释执行系统提供了一个“内部”的定理证明机制。使用这种机制，用户仅需以一阶逻辑（Horn 子句）的形式给出所求解问题的描述，而系统即可根据这种描述，自动提供模式匹配，回溯等超高级功能以求得解答。

但应注意，Prolog 并非一种万能的程序设计工具，对某些领域中的问题，Prolog 可能是不适用的或者是效率较低的。比较适于用 Prolog 求解的问题大体上可分为如下几类：

表 1　　具有代表性的 Prolog 解释系统

名　　称	使用语言	方　式	计　算　机	研制单位
DEC-10Prolog	汇　编	编译	DEC-10，20	爱丁堡大学
Prolog	Algol-W	解释	—	马赛大学
Prolog	汇　编	解释	IBM	IBM
Prolog/KR	Utilisp	解释	M-200H	东京大学
Dural	Maclisp	解释	DEC-20	日通研
Prolog/UNIX	C	解释	PDP-11	爱丁堡大学
Prolog/RT-11	汇　编	解释	LSI-11	爱丁堡大学
Micro-Prolog	BASIC	解释	TRS-80	计算所
Prolog/UNIPLUS	C	解释	Dual83/20	计算所
Polog/XENIX	C	解释	Intel86/330	计算所
Prolo	LISP	解释	IBM370/138	计算所
Hprolog	C	解释	AT&T3B2	H电脑公司
Hprolog	C	解释	IBMPC/XT，M24	H电脑公司
Hprolog	C	解释	IBMPC/AT	H电脑公司
Prolog	PASCAL	解释		

1）检索：Prolog 比较适合于描述某种目标所应满足的条件的集合，并用其模式匹配和回溯机制去检索这一目标。但应当使用 Prolog 所提供的某些特殊功能（如截断）去限制这种非确定性检索的复杂性。

2）由于 Prolog 具有较强的模式匹配功能，灵活的递归结构和易于处理的算符优先表达式文法，所以，Prolog 具有比 LISP 更强的符号处理能力。

3）Prolog 程序本身实际上就是一种表示事实和规则间关系的关系数据库，而这种数据库又因具有一定程序的智能而可称为知识库。所以，特别适合于书写查询和增删库中信息的数据／知识库管理系统。

由于 Prolog 具有上述各种特点，所以，它将在极其广阔的领域中获得应用，这包括：

1）计算机辅助设计，例如建筑，生化，电路等。

2）程序验证和综合合成问题求解，定理证明，公式处理，人工智能研究。

3）数据库/知识库管理。

4）自然语言处理。

5）专家系统。

6）超高级模拟。

7） 软件工程快速模型研究。

综上所述，Prolog 是一种与常规语言非常不同的语言，这种语言具有超高级语言的某些明显特点。由于这些特点，Prolog 将会成为第五代计算机系统在语言方面的特征之一，而且已经在极其广泛的领域中获得了应用。

二、基 本 语 法

Prolog 作为一种超高级语言，具有极强的描述和解题功能。但令人吃惊的是，Prolog 的基本语法却比任何一种常规高级语言都要简单。这种简单当然首先归功于作为理论基础的一阶逻辑表示方法的简洁和严格。这种语法上的简单同时也使得 Prolog 的直接实现也十分简单。有人做过试验，实现一个具有最基本功能的 Prolog 最小子集，只需要一个 600 行左右的 PASCAL 或 100 行左右的 LISP 程序即可。Prolog 的推广和传播较快，原因之一就在于由语法简单而导致的实现简单。

本段将以 BNF 范式对 Prolog 基本语法进行描述，并在适当的地方插入一些说明，描述以自顶向下的方式进行。首先，我们来看看如下的一段完整的 Prolog 程序：

```
1. employee ('浦心宏', 22)
2. employee ('宋瑜', 22)
3. employee ('章瀚', 23)
4. employee ('文迅', 38)
5. address ('浦心宏', '北京')
6. address ('宋瑜', '南京')
7. address ('章瀚', '成都')
8. address ('文迅', '北京')
9. select (_某人) :—address (_某人, _住地) _住地/='北京'
10. ? —select (_某人) 。
```

这段程序中的每一行都是一个 Prolog 语句，其中行号是显示格式加上去的，可以看出，前 8 个语句实际上给了一个关于职工姓名，年龄和家庭所在地的关系数据库。第九句则给出从职工中选择一个其家庭所在地不是北京的人的规则。含有这种规则的数据库又称为知识库，第 10 个语句则是要求进行选择的提问。如果对此 Prolog 程序进行解释或执行，则 Prolog 系统将给出：

X＝宋瑜

如果我们需要进一步求出满足条件的其他解，则只需在“宋瑜”之后输入“；”并回车，则 Prolog 将给出

X＝章瀚

如果我们还要求 Prolog 继续求出其他的解，则 Prolog 将响应以

no

并自动终止执行。

在下面的几小段中，我们将以此为例，说明 Prolog 程序构成的语法。

1. Prolog程序

〈程序〉::＝〈语句〉·{〈语句〉·} 可以看出，Prolog 程序由若干语句构成；

语句的结束符是句点。

2. 语句

〈语句〉::=〈目标〉｜〈目标〉:—

〈目标〉{〈分隔〉〈目标〉}｜〈目标〉〈—

〈目标〉{〈分隔〉〈目标〉}｜?—〈目标〉{

〈分隔〉〈目标〉}

〈分隔〉::=，｜;

语句可以是单个目标构成的事实（或称断语），表示条件和结论的规则或所要求解的问题这三种形式之一。例如目标

employee（'文迅'，38）

表示文迅这个职工现年38岁这一事实。符号":—"和"〈—"都表示蕴含。规则一般是一个 Horn 子句，由若干表示前提的目标的合取式（称为规则的本体）和表示结论的单一目标构成（称为规则的头部）。例如，规则：

select（X）:—address（X，Y），Y/='北京'.

表示如果职工X的所在地是Y，而且Y不是北京，则选择X。

一个问题是以符号"?—"开始的一组用逗号或分号分开的目标构成，其中逗号和分号表示这些目标之间的合取和析取关系。例如，我们可以更加简洁地用

9' ?—地址（X，Y），Y/='北京'.

来取代上例中的第9，10两个语句。

3. 目标

〈目标〉::=〈谓词〉（〈参数〉{，〈参数〉}）｜

〈参数〉〈谓词〉〈参数〉｜

〈谓词〉｜〈空〉

除可以为空外，目标有三种基本形式：

1) 独立形式：这种形式不带参数，它直接用于描述某一与其它成分没有关系的谓词是否成立。例如，Prolog 内部谓词

nl

将输出一个换行字符。

2) 前缀形式：这种形式下，谓词位于其参数表之前，参数表由用括号括起来的若干参数构成，参数间用逗号分开。例如

employee（'章瀚'，23）和

select（X）

3) 中缀形式：中缀形式的目标中，谓词位于其参数之间，这一般指二元谓词。对一元谓词，谓词一般放在参数之前，但也可由用户自行定义放在参数之后。例如

X=Y和

X/=Y及

X+Z

4. 谓词和原子

〈谓词〉::=〈原子〉

〈原子〉::=〈小写字母〉{〈尾部〉}｜

〈谓词符号〉|'任何字符串'

〈小写字母〉::=a··Z

〈尾部〉::=〈字母〉|〈数字〉|〈汉字〉

〈字母〉::=〈小写字母〉|〈大写字母〉

〈大写字母〉::=A··Z|_

〈数字〉::=0··9

〈汉字〉::系统所允许使用的任何汉字

在上述 BNF 范式中，〈谓词符号〉是用于表示谓词的某些特殊符号，例如

·，!，=，/=，+，-，*，/等

原子是构成 Prolog 的基本词法单位之一，它用于表示不变的目标的名字，谓词，函数名等。可以看出，原子只能以小写字母开头。此外在本段开始的例中，

'北京'，'南京'，'章瀚'，'文迅'

都是由引号引起的汉字字符串构成的原子。下面是用英文单词构成的原子的例子：

father，sister，own。

5. 参数和项

〈参数〉::=〈项〉

〈项〉::=〈常量〉|〈变量〉|〈复合项〉|〈表〉

〈常量〉::=〈原子〉|〈整数〉

〈整数〉::=〈数字〉{〈数字〉}

〈表〉::=[ ]|[〈项〉{，〈项〉}]|[〈项〉|〈表〉]|〈复合项〉|"任何串"

项是构成 Prolog 程序的基本语法单位之一，它一般用作谓词的参数。例如

'浦心宏'，22等

都是简单项的例子。

6. 变量

〈变量〉::=〈大写字母〉{〈尾部〉}

注意在〈大写字母〉中，包括下线字符_。变量用于表示尚未赋名（或称实例化，instantiation）或不需赋名的目标或未知数。变量和原子的区别在于它必须以大写字母或下线符号（在有些 Prolog 的实现中，用 * 、$或其它特殊字符）开头。单独一个下线符号表示无名变量。对在同一子句中的若干无名变量的赋名，可以不完全一致。在本段开始的例子中，X，Y都是变量，变量的其它例子如下：

国家 Someone，

7. 复合项

〈复合项〉::=〈函数名〉(〈项〉{,〈项〉})|

〈项〉〈算符〉〈项〉{〈算符〉〈项〉}

〈函数名〉::=〈原子〉

〈算符〉::=〈原子〉

复合项的语法与目标类似。其中各项称为此复合项的分量。例如：

2*3.14*R，gcd(X，f(X))，own(John，book)

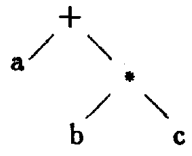
三、基本数据结构

Prolog 的主要数据结构是树和作为其一特例的表。

1. **树**

可以用复合项的概念来表示 Prolog 的树形数据结构。在此复合项中，函数名是树形结构的根，而各个分量则为其分枝。例如复合项 Parents（'杨延昭'，'佘太君'，'杨业'）表示杨六郎的母亲是佘太君而父亲是杨业。它表示了如下的树形结构：

又，表达式a+b*c（或+(a, *(b, c)))可表示为



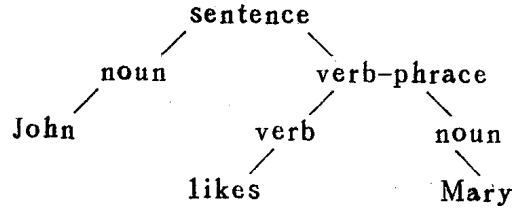
如果对英语语句

Johm likes Mary

进行分析，则可以用如下复合项表示：

Sentence (noun (John) , verb-phrase (verb (likes) , noun (Mary))) .

这与如下结构对应



由此可见，用 Prolog 的复合项可以表示任意复杂的树形结构。

2. **表**

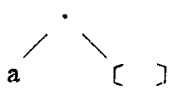
在非数值应用中，表是一种经常用到的数据结构。表是其元素的有序序列。而此序列的长度是可以任意的。表中的元素，可以是原子，项（包括变量和复合项），也可以是另一个表。表实际上可以定义符号计算中所需要的任何一种结构。在分析树，语法表示，地图，计算机程序和诸如图，公式和函数之类的数学概念中都常用到表形数据结构。著名的程序设计语言 LISP 中，就仅有两种数据结构，即常数和表。在 Prolog 中，表则以树的一种特殊情况的面目出现。

Prolog 中的表可以是空表。空表用〔 〕表示，这与 LISP 中的 nil 相对应。非空

表可以有两个分量，即表头和表尾。表头和表尾可以作为一个名为“·”（句点）的函数的两个分量出现。表的结束习惯上用一个被置为空的尾部表示。于是，只有一个元素‘a’的表可表示为：

.(a, 〔 〕)

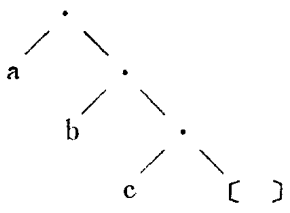
其树形表示是：



类似地，由元素a,b,c构成的表可记为

.(a, .(b, .(c, 〔 〕)))

其树形表示是：



有时，也将点函数定义为一个算符，而将上述二表记为中缀形式：

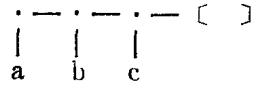
a.〔 〕和 a.(b.(c.〔 〕))

后者可进一步写成

a. b. c.〔 〕

因为算符·是右结合的。注意，表是一个有序序列，所以 a.b 和 b.a 是不相同的。

比树形表示更为清晰的表示方法是所谓“葡萄藤”式的表示方法：表从左到右增长，而头部分枝朝下，尾部分支向右。表的最右端总是〔 〕。用这种表示方法，表a.b.c.〔 〕可表示为如下一串葡萄藤



对复杂的表列而言，点表示法是不太方便的。我们可以采用另一种表表示法：整个表列的所有元素都放在一对方括号中，元素之间用逗号隔开。例如前两个例子可以分别记为

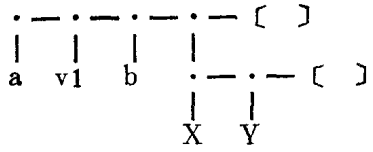
〔a〕和〔a, b, c〕

更加复杂的表可能以另一个表作为其元素，例如：

〔a, b, c, 〔d, e, f〕, g〕

〔a, v1, b, 〔X, Y〕〕

后者中的变量将按与其他结构中的变量相同方式加以处理。其葡萄藤表示为：



从此图中可清楚地看出，每条水平的藤是一个具有一定数量元素的表。上面一条藤有四个元素，其最后一个元素是一个表，与此表相应的藤上有两个元素。

在使用表表示法时，表头是表的第一个元素，而表尾是由所有余下元素构成的一个表。下面各例将说明表头，表尾在表表示法中的相应地位。

表	头　部	尾　部
[a, b, c, d]	a	[b, c, d,]
[a]	a	[ ]
[ ]	(无　头)	(无　尾)
[[the, cat] , sat]	[the, cat]	[sat]
[the, [cat, sat]]	the	[[cat, sat]]
[the, [cat,sat] ,down]	the	[[cat, sat] , down]
[X+Y, X+Y]	X+Y	[X+Y]

由于将一个表分裂为头尾两部分是十分常用的一种操作，所以 Prolog 为此提供了一种专用的运算符 | 。如果某表的头和尾分别为X和Y，则此表可记为 [X | Y] 。注意，| 运算应在表表示法中而不是在点表示法中使用。例如，如果我们有如下数据库：

```
p ([1, 2, 3])
p ([the, cat, sat, [on, the, mat]]) .
```

而问

```
? —p ([X | Y]) .
```

则 Prolog 将回答：

```
X = 1    Y = [2, 3]
```

在问到

```
? —p ( [_, _, _, [_|×]]).
```

时，Prolog 回答为：

```
X= [the, mat]
```

与 LISP 语言中 CAR, CDR 和 CONS 相比较，| 算符不仅可以一当三，而且功能更强。例如，在 LISP 中求表A的头和尾，需用

CAR A 和 CDR A

而在 Prolog 中，只需写

A＝〔X｜Y〕

则自然有，X，Y即A的头，尾了。

四、Prolog 的基本算法

Prolog 系统内部的定理证明机制主要采用两种方法完成解题任务：模式匹配(Pattern matching，或称一致化——Unification) 和回溯 (Backtracking)

1. **模式匹配**

为了清楚起见，我们来看一个阶乘问题的求解过程。Prolog 程序中的知识库如下：

1. fact (1, 1) .

2. fact (X, Y) :—X1 is X—1, fact (X1, Y1) , Y is X * Y1.

库中的第一个语句说明了 1 的阶乘为 1 这一事实。第二个语句则给出了要求X的阶乘，必须先求X— 1 的阶乘，然后将所得结果乘以X的规则。

如果我们提出问题

? —fact (2 , N)

则 Prolog 系统将给出答案

N＝2

Prolog 是怎么求出这个答案来的呢？首先，它将问题中的目标与库中的事实和规则逐一进行比较，把问题与第一件事实比较时，发现虽然fact（相同，但 1 和 2 不同，所以匹配失败。当问题与第二件事实比较时，fact（仍然相同；X和 2 虽然不同，但因为X是一个尚未实例化的变量，所以匹配仍算成功，而且X被实例化为 2；在，号匹配成功之后，遇到的是两个未实例化的变量Y和N，Prolog 对此采取的对策是让这两个变量共享存储，以后，一旦这两个变量中的任一变量被实例化，则另一变量也将被实例化为同一个值；在）匹配成功之后，Prolog 发出目标匹配成功信号。然后问题中的目标 fact（2，N）被分解为三个子目标

X1 isX—1,

fact (X_1, Y_1) ,

Y isX * Y

这个过程称为消解。

在第一个子目标中，is 和—都是内部谓词（或称算符），而X已被实例化为 2，故 X_1 被实例化为 1．从而第一个子目标成功，由于X_1已实例化为 1，因此第二个子目标变为 fact（1，Y1），然后 Prolog 重新搜索知识库，发现该子目标的前半部 fact (1, 与第一件事实，fact (1, 1) 相同，而Y1为一未实例化的变量，于是将Y1实例化为 1，在)匹配成功之后，第二个子目标亦告匹配成功。第三个子目标中的 * 也是内部谓词，故Y被实例化为 2。

由于N和Y共享存储，所以N也被实例化为2，至此，定理证明机制宣告问题求解成功，并给出正确的答案N＝2。

在阶乘问题求解过程中，我们看到Prolog解题的最重要的手段之一就是模式匹配，对于模式匹配，我们可以给出如下规则：

1） 未实例化的变量将与任何项匹配

结果该变量将代表（或被实例化为）与之匹配的项。

2） 整数或原子只能与其本身匹配。

3） 结构只能与结构相匹配，两个结构的函数名，参数个数必须相同，而且所有相应参数也必须匹配。

4） 两个未实例化的变量在匹配之后将共享存储，以后，其一变量的实例化将同时使另一个被实例化为同一值。

值得注意的是，如果数据库中有如下事实

sum（5）.

sum（3）.

sum（X＋Y）.

那么，？—sum（2＋3）将与哪个事实相匹配呢？答案应该是第3个而不是第1个，在与第3个匹配后，X和Y才分别被实例化为2和3。

2. **回溯**

阶乘问题的求解是一帆风顺的：在搜索数据库／知识库时，我们总找得到匹配的事实或规则，而且所有的子目标都能得到满足，但这样的情况实际上是少见的，所以在用单处理器或单进程解题的过程中经常会发生回溯。

在如下两种情况下，我们需要进行回溯：

1） 在满足某个子目标时，找不到与之匹配的事实或规则头部。

2） 在已经求出问题的某个解之后，我们还希望求出其它的解。

我们用下面的例子来说明回溯的过程：

```
1. colour('苹果', '红').
2. colour('桔子', '黄').
3. colour('玉米', '黄').
4. colour('草莓', '红').
5. colour('西红柿', '红').
6. colour('胡萝卜', '红').
7. fruit('桔子').
8. fruit('苹果').
9. fruit('草莓').
10. vegetable('西红柿').
11. vegetable('胡萝卜').
12. corn('玉米').
13. like('章瀚', X):—
      colour(X, '红'), vegetable(X).
14. like('文迅', X):—
      colour(X, '黄'), fruit(X).
```

上述知识库说明各种食物的颜色和类别，并分别给出了章瀚和文迅爱吃的东西的颜色和类别。

如果我们提出问题

？—like（'章瀚'，X）

Prolog将按如下步骤解决这一问题：

1） 在知识库中从头到尾搜索与like（'章瀚'，X）相匹配的事实或规则。结果找到第13条规则，将一个回溯位置标志指向第14条规则，然后Prolog将企图满足子目标

colour（X，‘红’）和 vegetable（X）

2） 首先满足 Colour（X‘红’），为此，搜索知识库，发现第一件事实就之匹配。此时，Prolog 将X实例化为苹果，而使第一个子目标得到满足，同时将第二个位置标志指向下一件即第二件事实。

3） 满足 vegetable（X），由于X已实例化为苹果，所以实际上应满足 vegetable（‘苹果’），但遍索知识库找不到任何与之匹配的事实和规则，因此，子目标 vegetable（‘苹果’）失败，此时应进行回溯。

4） 回到第一个子目标，并使X返回未实例化状况，重新满足第一个子目标 Colour（X，‘红’），回溯从第二个位置标志指向的事实的语句即第二件事实开始寻求匹配，发现第4件事实与之匹配，X被实例化为草莓，第二个位置标志指向第5件事实。

5） 满足 vegetable（‘草莓’），与3)类似，但这一企图失败，再次发生回溯。

6） 回溯从第5件事实开始寻求匹配，发现第五件事实与之匹配，故X实例化为西红柿，第二个位置指针指向第6件事实。

7） 试图满足 vegetable（西红柿），发现第10件事实与之完全匹配。

第二个子目标得到满足，至此求解完成，Prolog 给出答案：

X＝西红柿

我们看到，在此求解过程中，发生了两次回溯，如果我们对求解出的答案不够满意，还需求出另外的解，则可输入分号，则 Prolog 回到第一个子目标，试图从第6件事实开始回溯，重新满足。

colour（X‘红’）

这与第6件事实匹配，X被实例化为胡萝卜，第2个位置标志指向第7件事实。然后试图满足。

vegetable（‘胡萝卜’）

而这与第11件事实匹配，故 Prolog 给出第二个解：

X＝胡萝卜

3. 截断

仔细考察前一段给出的目标满足算法可以发现，如果我们希望求出某问题所有的解，则该算法实际上是一种穷索法：目标将被与所有 Horn 子句比较，所有各点都可以是回溯点，但这对某些问题而言，显然是多余的。为了对这些问题节省时间和空间，我们引入截断的概念。截断将搜索树的某些分枝砍掉不予处理。之所以要引入这个概念，主要有两个理由：

1） 有些目标可能对解决某一问题毫无帮助，所以不必企图去满足它们。

2） 有些点一旦通过，就不可能再从其处回溯，所以不必记录它们。

截断在 Prolog 中是用一个无参数的谓词符号“!”来表示的。！作为一个目标，它总是立即得到满足，而且不能重新被满足。也就是说，一旦通过这一目标，就不可能从其处回溯，其副作用就在于它将改变以后回溯的路径。其效果相当于去掉某些目标的位置标志，以使这些目标不能重新满足。

我们用下面这一例子来说明截断的含义：

$p:-q_1, q_2, !, r1, r2.$

$p:-s1, s2.$

一般情况下，如目标q_1或q_2失败时，就会发生回溯。如果q_1失败，则转向下一Horn子