

## 第六章 逻辑式程序设计语言

逻辑式语言基本形式：用一种**符号逻辑**作为程序设计语言来进行程序设计，通常称为逻辑程序设计语言，或声明性语言

- 程序要对数据结构实施某个算法过程，算法实现计算逻辑

算法 = 逻辑 + 控制

- 逻辑程序设计的基本观点是程序描述的是数据对象之间的关系。关系也是联系。
- 对象和对象、对象和属性的联系就是我们所说的事实。事实之间的关系以规则表述，根据规则找出合乎逻辑的事实就是推理。
- 逻辑程序设计范型是陈述事实、制定规则，程序设计就是构造证明。程序的执行就在推理。

## 6.1 谓词演算

谓词演算是符号化事实的形式逻辑系统，它也是逻辑程序设计语言的模型

- 表示命题
- 表示命题之间的关系
- 描述如何根据假设为真的命题推断出新命题

- 谓词演算诸元素

用形式方法研究论域上的对象需要一种语言，它能表达该域对象具有什么性质(properties)，以及对象间有些什么关系(relations)

描述以公式(Formulas)表达。谓词公式中各元素按一定逻辑规则变换，即谓词演算(predicate calculus)

- ① 公式：由一组约定的符号组成的序列，它包括常量、变量、逻辑连接、命题函数、谓词、量词
- ② 常量：指明论域上的对象
- ③ 变量：可束定到特定域上某个范围的对象上
- ④ 函数：表征对象具有的映射关系
- ⑤ 谓词：表征对象某种性质的符号
- ⑥ 量词：量词限定的变量名作用域是整个公式
- ⑦ 逻辑操作：and, or, not,  $\rightarrow$ (蕴含)  $\Leftrightarrow$ (全等)

当谓词应用到的变元是常量或已被束定的变量上时，就叫做句子(sentence)或命题(proposition)

谓词变元的个数称作目(arity)，有单目、N目谓词之称  
N-目谓词的例子。

谓词	目	含义
odd(X)	1	X是奇数
father(F, S)	2	F是S的父亲
divide(N, D, Q, R)	4	N除D得商Q和余数R

谓词例化	结果值
odd(2)	False
divide (23, 7, 3, 2)	Ture
father (changshan, changping)	True
divide (23, 7, 3, N)	N未例化， 不知真假

## 谓词的量化

量化谓词

结果值

$\forall X \text{odd}(X)$

False

$\exists X \text{odd}(X)$

True

$\forall X (X = 2 * Y + 1 \rightarrow \text{odd}(X))$

True

$\exists X \exists Y \text{divide}(X, 3, Y, 0)$

True, 如  $X=3, Y=1$

$\forall X \exists Y \text{divide}(X, 3, Y, 0)$

False

$\exists X \forall Y \text{divide}(X, 3, Y, 0)$

False

证明一个全称谓词是比较难的，因为最可靠的证明方法是枚举例证。于是采取反证的方法，全称量化的谓词取反

量化谓词	取反	
$\forall X \text{odd}(X)$	$\exists X \text{not odd}(X)$	[1]
$\exists X \text{odd}(X)$	$\forall X \text{not odd}(X)$	[2]
$\forall X(X=2*Y+1 \rightarrow \text{odd}(X))$	$\exists X \text{not}(X=2*Y+1 \rightarrow \text{odd}(X))$	[3]
	$\exists X \text{not}(X=2*Y+1) \text{ or } \text{odd}(X)$	[4]
	$\exists X((X=2*Y+1) \text{ and } \text{not odd}(X))$	[5]
$\forall X \exists Y \text{ divide}(X, 3, Y, 0)$	$\exists X \forall Y \text{ not divide}(X, 3, Y, 0)$	[6]
$\exists X \exists Y \text{ divide}(X, 3, Y, 0)$	$\forall X \forall Y \text{ not divide}(X, 3, Y, 0)$	[7]
$\exists X \forall Y \text{ divide}(X, 3, Y, 0)$	$\forall X \exists Y \text{ not divide}(X, 3, Y, 0)$	[8]

# 谓词演算的等价变换

一般谓词公式变换为子句的实例。‘ $\vdash$ ’号为“可推出”

[1]以 $\wedge$ ,  $\vee$ ,  $\neg$ 消除 $\rightarrow$ 、 $\Leftrightarrow$ 符号

[2]化为前束范式,消除最外的 $\neg$ 符号,否定符号内移

$$\neg(\exists X P(X)) \vdash \forall X(\neg p(X))$$

[3]用斯柯林变换消去存在量词

$$\forall X(a(X) \wedge b(X) \vee \exists Y c(X, Y))$$

$$\vdash \forall X(a(X) \wedge b(X) \vee c(X, g(X)))$$

[4] 消除前束范式的全称量词

$$\vdash a(X) \wedge b(X) \vee c(X, g(X))$$

[5]用分配率 $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$ 化成合取范式

$$\vdash (a(X) \vee c(X, g(X))) \wedge (b(X) \vee c(X, g(X)))$$

经过以上变换,任何一复合公式均可成为如下形式:

$$F = C1 \wedge C2 \wedge \dots Cn$$

且其中 $C_i$ 称为子句

若以';'代' $\vee$ '则有:

$$C_i = L1 \vee L2 \vee \dots L_v = L1;L2;\dots;L_v$$

因此,任一公式均可化为' $\vee$ '连接的子句的集合



## 6.2 自动定理证明

- 证明系统

事实即证明系统中的公理(axioms)

证明系统(proof system)是应用公理演绎出定理(theorems)的合法演绎规则的集合

演绎也叫归约(deduction)，是对证明系统中合法推理规则的一次应用

演绎从公理导出结论(conclusion)，中间可利用以这些规则演绎出的定理

证明(roof)是个语句序列，以每个语句得到证明而结束，即每个句子要么演绎成公理，要么演绎成前此导出的定理

- 一个证明若有N个语句(命题)则称N步证明
- 反驳(refutation)是一个语句的反向证明。它证明一个语句是矛盾的，即不合乎给定的公理
- 一个语句若能从公理出发推演出来，则称合法语句，任何合法语句也叫做定理(theorem)
- 从某一公理集合导出的所有定理集合称为理论(theory)

## • 模型

从公理集合中导出定理集称之为理论，有了理论我们要解释它的语义必须借助某个模型(model)。

因为形式系统只是符号抽象，借助模型我们可为每个常量、函数、谓词符号找到真理性的解释。即定义每个论域，并表明域上成员和常量公理之间的关系。

公理的谓词符号必须派定为域中对象的性质，函数派定为对域中对象的操作。

公理集合一般情况下只是定义的部分(偏)函数和谓词，是问题域的一个侧面。所以能满足该理论的模型往往不止一个。

## • 证明技术

从谓词演算具有完整性，理论上可证明按公理集合建立的任何理论。关键是效率。

如果我们从公理出发做出每一个步骤，在新的步骤上仍然要查找每一个公理，找出可能的推理。如此下去就形成一个庞大的树行公理集，每层的结点表示一个公理的语句，其深度和宽度随问题和最初给出的公理而定，一层一步骤，N层的树就是N步推理。

对于自动定理证明程序，只有穷举每条可能的证明步骤才能说它是完全的。

穷举完所有路径马上遇到组合爆炸问题，无论是深度优先还是广度优先，百步演绎可能的路径数都是天文数字。

## • 归结定理证明

J.A.Robinson1965年提出的归结法(resolution)，是命题演算中对合适公式的一种证明方法。

为了证明合适公式F为真，归结法证明 $\neg F$ 恒假来代替F永真。

把两子句合一(unification)并消去一对正逆命题，故归结也译作消解。

归结证明的过程并称之为归结演绎，其步骤如下：

[1]把前题中所有命题换成子句形式。

[2]取结论的反，并转换成子句形式，加入[1]中的子句集。

[3]在子句集中选择含有互逆命题的命题归结。用合一算法得出新子句(归结式)，再加入到子句集。

[4]重复[3]，若归结式为 $\square$ 则表示此次证明的逻辑结论是矛盾，原待证结论若不取反则恒真。命题得证。否则继续重复[3]。

例：归结证明

若有前题

p1  $Q \vee \neg P$

p2  $R \vee \neg Q$

p3  $S \vee \neg R$

p4  $\neg U \vee \neg S$

待证命题

$\neg P \vee \neg U$

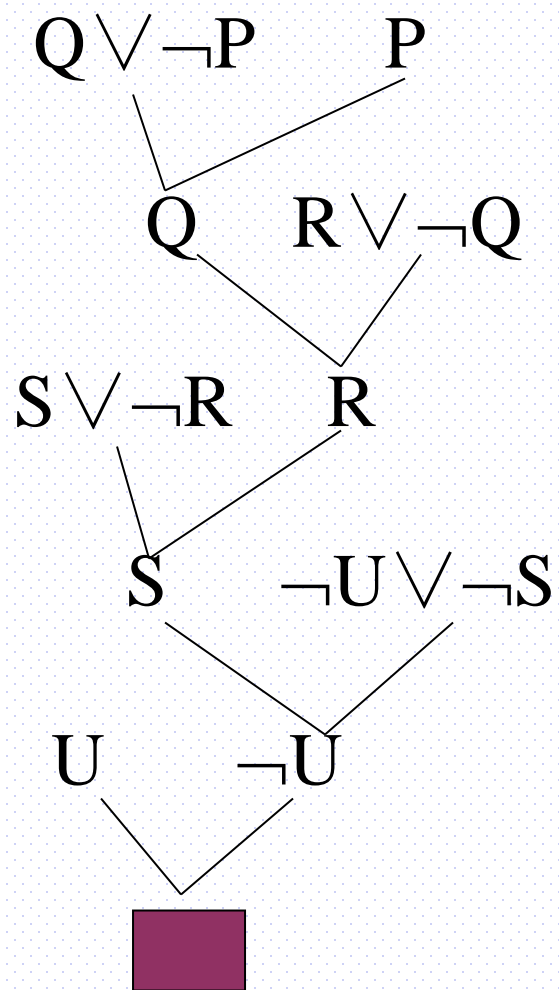
取反得新子句

p5  $P$

p6  $U$

取待证命题的反，得 $P \wedge U$ ，它是 $\wedge$ 连接的两个子句 $P$ 、 $U$ ，把它们加到前题子句集，为p5，p6。

归结演绎如下图:



p1-p5归结

再与p2归结

再与p3归结

再与p4归结

再与p6归结

矛盾

p1	$Q \vee \neg P$
p2	$R \vee \neg Q$
p3	$S \vee \neg R$
p4	$\neg U \vee \neg S$
p5	$P$
p6	$U$



由本例可以看出两个问题：

第一，归结法是由合一算法实现的。所谓合一是找出型式匹配的两子句，将它们合一为归结式，相当于代数中的化简。

第二，如果得不出矛盾，那么归结法要无休止地做下去，中间归结式出得越多，匹配查找次数越多，每一步都做长时间计算。

**Solution:** 利用切断(cut)操作，并利用对子句形式进一步限制的超级归结法(Hyperresolution)。

## HORN子句实现超归结

Horn子句是至多只有一个非负谓词符号的子句

Horn子句形式示例如下：

$$\neg P \vee \neg Q \vee S \vee \neg R \vee \neg T$$

其中只有一个非负谓词S，可作以下演算：

先将S移向右方

$$\vdash S \vee \neg P \vee \neg Q \vee \neg R \vee \neg T$$

按德·摩根定律

$$\vdash S \vee \neg (P \wedge Q \wedge R \wedge T)$$

' $\vee \neg$ '即' $\rightarrow$ '，则

$$\vdash S \rightarrow (P \wedge Q \wedge R \wedge T)$$

此条件Horn子句的意义是： if S then  $(P \wedge Q \wedge R \wedge T)$  。 若S为空， 则为无条件Horn子句， 是一个断言(事实)

## 归结练习

已知：某些病人喜欢所有的医生(A1)

没有一个病人喜欢任意一个骗子(A2)

欲证明：任意一个医生都不是骗子(B)

证明：事实表示：令

$P(x)$  :  $x$ 是病人

$D(x)$  :  $x$ 是医生

$Q(x)$  :  $x$ 是骗子

$L(x, y)$  :  $x$ 喜欢 $y$

A1:  $\exists x(P(x) \wedge \forall y(D(y) \rightarrow L(x,y)))$

A2: ?    B: ?

## 归结练习

$P(x)$  :  $x$ 是病人

$D(x)$  :  $x$ 是医生

$Q(x)$  :  $x$ 是骗子

$L(x, y)$  :  $x$ 喜欢 $y$

$A1: \exists x(P(x) \wedge \forall y(D(y) \rightarrow L(x, y)))$  //某些病人喜欢所有的医生(A1)

$A2: \forall x(P(x) \rightarrow \forall y(Q(y) \rightarrow \neg L(x, y)))$  //没有一个病人喜欢任意一个骗子(A2)

$B: \forall x(D(x) \rightarrow \neg Q(x))$  //任意一个医生都不是骗子(B)

要证明B是A1和A2的逻辑结果: //即公式  $A1 \wedge A2 \wedge \neg B$ 是不可满足的

## 归结练习

$$A1 = \exists x(P(x) \wedge \forall y(\neg D(y) \rightarrow L(x,y)))$$

$$A2: \forall x(P(x) \rightarrow \forall y(Q(y) \rightarrow \neg L(x,y)))$$

$$B: \forall x(D(x) \rightarrow \neg Q(x))$$

$$A1 = \exists x(P(x) \wedge \forall y(\neg D(y) \rightarrow L(x,y)))$$

$$= \exists x \forall y(P(x) \wedge (\neg D(y) \rightarrow L(x,y)))$$

$$\text{---} \rightarrow \forall y(P(a) \wedge (\neg D(y) \rightarrow L(a,y)))$$

$$A2 =$$

$$\neg B =$$

$A1 \wedge A2 \wedge \neg B$ 的子句集是什么

$$S =$$

## 归结练习

$$A1 = \exists x(P(x) \wedge \forall y(\neg D(y) \vee L(x,y)))$$

$$= \exists x \forall y(P(x) \wedge (\neg D(y) \vee L(x,y)))$$

$$\text{---} \rightarrow \forall y(P(a) \wedge (\neg D(y) \vee L(a,y)))$$

$$A2 = \neg \forall x(P(x) \rightarrow \forall y(\neg Q(y) \vee \neg L(x,y)))$$

$$= \forall x(\neg P(x) \vee \forall y(\neg Q(y) \vee \neg L(x,y)))$$

$$= \forall x \forall y (\neg P(x) \vee \neg Q(y) \vee \neg L(x,y))$$

$$\neg B = \neg (\forall x(D(x) \rightarrow \neg Q(x)))$$

$$= \exists x(\neg(\neg D(x) \vee \neg Q(x)))$$

$$\text{---} \rightarrow (D(b) \wedge Q(b))$$

$$S = \{P(a), \neg D(y) \rightarrow L(a,y), \neg P(x) \vee \neg Q(y) \vee \neg L(x,y), D(b), Q(b)\}$$

## 归结练习

S不可满足的归结演绎序列为：

(1)  $P(a)$ ,

(2)  $\neg D(y) \vee L(a,y)$ ,

(3)  $\neg P(x) \vee \neg Q(y) \vee \neg L(x,y)$ ,

(4)  $D(b)$

(5)  $Q(b)$

(6)  $\neg Q(y) \vee \neg L(a,y)$     // 由 (1)    (3)

(7)  $L(a,b)$                     // 由 (2)    (4)

(8)  $\neg L(a,b)$                 // 由 (5)    (6)

(9)  $\square$  由                    // (6)    (8)

## 6.3 逻辑程序的风格

第一个特点是它不描述计算过程而是描述证明过程

第二个特点是描述性

逻辑式程序处理的是关系而不是函数。用关系比函数更灵活，关系能统一处理参数和结果，也就是关系中没有计算方向的概念。

第三个特点是大量用表和递归实现重复操作

`sort (old_list, new_list)`

$\vdash \text{permute}(\text{old\_list}, \text{new\_list}) \wedge \text{sorted}(\text{new\_list})$

$\text{sorted}(\text{list}) \wedge \forall j \text{ 使得 } 1 \leq j < n, \text{list}(j) \leq \text{list}(j+1)$

\*permute是一个谓词，如果第二个参数组是第一个参数组的一个排列，就返回真



# PROLOG语言

- Prolog是一种基于一阶谓词的逻辑式语言
- Prolog是基于Horn子句的，使用归结推理，具有很强的逻辑描述能力和推理能力
- Prolog语言特点：
  - 一阶逻辑的语言形式是形式化地严格定义的
  - 一阶逻辑的语法简易易懂
  - 逻辑公式不需要重复表达，与不同应用无关
  - 事实、假设、推理、查询、视图和完整性规约条件都能以基于一阶逻辑的prolog语言表达
  - 逻辑语言Prolog可作为定义和比较其它知识表示模型的共同模型

例 求平均成绩的逻辑程序，打开一分数文件scores，  
读入分数求和并用的数N除之得平均成绩

```
average :- see(scores),  
          getinput (Sum, N),  
          seen (scores),  
          Av is Sum /N,  
          print ('Average = ', Av)  
getinput (Sum, N) :- ratom (X),  
                    not (eof),  
                    getinput (Sum1, N1),  
                    Sum is Sum1 + X,  
                    N is N1 + 1.  
getinput (0, 0) :- eof.
```

## 6.4 典型逻辑程序设计语言PROLOG

猜测并验证，查询形式：是否存在一个S，使得guess(S)且verify(S)

- Prolog要环境支持，即管理事实和规则的数据库
- Prolog的基本成分是对象(常量、变量、结构、表)、谓词、运算符、函数、规则
- 从纯语法意义上Prolog的项什么都可以表示：
- $\langle \text{项} \rangle ::= \langle \text{常量} \rangle \mid \langle \text{变量} \rangle \mid \langle \text{结构} \rangle \mid (\langle \text{项} \rangle) \mid \langle \text{表} \rangle \langle \text{后缀算符} \rangle$
- $\mid \langle \text{项} \rangle \langle \text{中缀算符} \rangle \langle \text{项} \rangle \mid \langle \text{, 项} \rangle \langle \text{前缀算符} \rangle$

- 从语义角度， 以下语法描述提供了处理时的语义概念:

<程序>  $\rightarrow$  <子句>

<子句>  $\rightarrow$  (<事实> | <规则> | <查询> )

<事实>  $\rightarrow$  <结构>

<规则>  $\rightarrow$  <头> : -<体>

<头>  $\rightarrow$  <结构>

<体>  $\rightarrow$  <目标> , <目标>

<目标>  $\rightarrow$  /\*形如p或q(T, ..., )的字面量\*/

# 与PROLOG交互

- ? - （在每轮交互开始时系统都会给出“提示符号”）

表示希望得到一个查询

? - consult (links) .

- consult结构读入包含事实和规则的文件，并将这些内容添加到当前规则数据库末尾。

- ? - link (algol60, L) , link (L, M) .

L = cpl

M = bcpl

表示：是否存在L和M，使link (algol60, L) and link (L, M) ?

输入“;”并回车，Prolog将用另一个解作为响应，或者用“no”说明已经无法在找到解。

- 规则的表示

规则就是horn子句

<term> :- <term>1, <term>2, ....., <term>k.

:- 左边的项称为头部，在:- 右边的那些项称为条件。

事实是规则的特殊形式，只有头部而没有条件。

- Path(L, L).
- Path(L, M) :- link(L, X), path(X, M).
- 其中变量X，出现在条件里面，而不在头部，表示某个满足条件的对象

# 与PROLOG交互

## ■ 合一

? -  $f(X, b) = f(a, Y)$  .

$X = a$

$Y = b$

- 得到项T的实例方法：用一些项去替换T中的一个或几个变量。同一个变量的所有出现必须用同一个项去替换。
- $f(a, b)$  是  $f(X, b)$  的实例，同理  $f(a, b)$  是  $f(a, Y)$  的实例。共同的实例是  $f(a, b)$  。
- $g(a, b)$  不是  $g(X, X)$  的实例
- 合一是在规则应用时隐含发生的。

## ■ 算术：“=” 运算符表示合一

? -  $X = 2+3$

$X = 2+3$

中缀运算符 “is” 对表达式求值：

? -  $X \text{ is } 2+3$

$X = 5$

## •Prolog程序结构

Prolog程序由子句组成，子句模型是Horn子句。

### (1) 事实与规则

Prolog程序先定义公理集

例：Prolog的规则和事实

条件子句(规则) pretty (X):-artwork(X)

pretty (X):-color(X, red), flower(X).

watchout (X):-sharp(X, \_).

无条件子句(事实) color (rose, red).

sharp (rose, stem).

sharp (holly, leaf).

flower(rose).

flower(violet)

artwork (painting (Monet, haystack\_at\_Giverny)).

## (2)查询

Prolog中查询(query)是要求Prolog证明定理。 因为提出的问題就是证明过程的目标, 所以查询也叫目标(goal)。

例: Prolog的查询

```
?- pretty (rose).
```

```
yes
```

```
?- pretty (Y).
```

```
Y=painting (Monet, haystack_at_Giverny).
```

```
Y=rose.
```

```
no
```

```
?-pretty(W), sharp(W, Z)
```

```
W=rose Z=stem
```

```
no
```



例：最大公约数的欧基里得算法

最大公约数欧基里得算法可用三条规则描述：

$\text{gcd}(A, 0, A).$

$\text{gcd}(A, B, D) :- (A > B), (B > 0), R \text{ is } A \bmod B,$   
 $\text{gcd}(B, R, D).$

$\text{gcd}(A, B, D) :- (A < B), \text{gcd}(B, A, D).$

## •函数和计算

### (1) 函子完成逻辑设计中的计算

函子以结构形式出现， 如：

中缀表示

$X+Y*Z$

$A-B/C$

前缀表示

$+(X, *(Y, Z))$

$-(A, /(B, C))$

故它不是谓词， 仅仅是一特殊的结构：

$\langle \text{函数名} \rangle (\langle \text{变元} \rangle, \dots, \langle \text{变元} \rangle)$

函数求值的的结果一般通过谓词 $\text{is}(\langle \text{变元} \rangle, \langle \text{表达式} \rangle)$ 束定到变元上

$\text{gcd}(A, B, D);-(A>B), (B>0), R \text{ is } A \bmod B, \text{gcd}(B, R, D).$

把函数改写为约束， 很容易写出prolog程序

## 例 求斐波那契数的Prolog程序

斐波那契函数以下述公式生成以下数列:

1, 1, 2, 3, 5, 8, 13, 21, ...

$\text{Fib}(0) = 1$

$\text{Fib}(1) = 1$

$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

第一、二式是事实也是公理，把结果值作为变元照写。第三式说明，若n为斐波那契数，n-1和n-2的斐波那契必须成立，且这两个数之和是n的斐波那契数， $n > 1$ ，于是有Prolog程序

$\text{Fib}(0, 1).$

$\text{Fib}(1, 1).$

$\text{Fib}(n, f) :- \text{Fib}(m, g), \text{Fib}(k, h), m \text{ is } n-1, k \text{ is } m-1,$   
 $f \text{ is } g+h, n > 1.$

当有查询  $?\text{-Fib}(5, f)$  时，f 返回8

## (2) 逻辑程序的算法表达

算法怎样用公理表达呢？ 拿一个最典型的Quicksort分类程序讨论。

quicksort(未分类表, 分类完的表):

- (从未分类表拿出第一元素, 以它为准, 分成两个表), [1]

quicksort(小表, 分类完小表), [2]

quicksort(大表, 分类完大表), [3]

append (分类完小表, 基准元素和分类完大表, 分类完总表) [4]

这样把快速分类的总目标变成了四个子目标

## 例 快速分类的Prolog代码

r1 split(\_, [], [], []).

r2 split (Pivot, [Head | Tail], [Head | Sm], Lg):-  
    Head < Pivot, split (Pivot, Tail, Sm, Lg).

r3 split (Pivot, [Head | Tail], Sm [Head | Lg]):-  
    Pivot < Head, split (Pivot, Tail, Sm, Lg).

r4 quicksort ([ ], [ ]).

r5 quicksort ([Head [ ] ], Head).

r6 quicksort ([Pivot | Unsorted] AllSorted):-  
    split (Pivot, Unsorted, Small, Large),  
    quicksort (Small, SmSorted),  
    quicksort (Large, LgSorted),  
    append (SmSorted, [Pivot | LgSorted], AllSorted).

### (3)逻辑和控制分离

Prolog无通常意义的控制结构，也就是该程序动作次序（显然也有）和计算的子句逻辑没有必然的关系。例如:把上例中r4,r5,r6写在r1,r2,r3前面并不影响本程序的执行结果。

## 封闭世界内的假设

- 如果有某个子目标查遍数据库也找不到能满足的事实，该子目标失败，但不等于整个目标的失败。
- 即使是整个目标最后失败，也不等于这个目标追求的命题是否定的，因为限于数据库存放的规则和事实有限，它是“**封闭世界假设**”之下的失败。

## •cut和not谓词

因为Prolog的归结模型只能完整地证明正命题， 是否有解无法判定。

如果明知再作没有意义，可人为截断cut。

### (1) 安全cut

非形式解释cut， 它如同一篱笆， 由程序员任意置放在规则之中， 以停止无意义的回溯。



例 安全cut示例：求1到N的整数之和

```
r1  sum_to(N, 1):-N=1, ! .
r2  sum_to (N, r):-N1 is N-1, sum_to(N1, r1),
                                     r is r1 + N.
```

无论X是否为真，  
! 左侧的都回溯  
不过去。

当有查询：

```
?-sum_to(1, X) //匹配r1
```

```
X=1;           //打 ‘;’ 号由于有! 不致无限查找第2个
```

```
no
```

```
?-sum_to(6, X) //匹配r1失败， 匹配r2连续r2
```

```
X=21;          //直至成功， 打';'号也不再找
```

```
no
```

r1 可用sum\_to(1,1).事实代

## (2) cut 实现not操作

r1 not(X):-X, ! , fail.

r2 not(\_).

其推理过程是:

- 若X为假, 匹配r1, 在未达到! 时已失败, 则匹配规则r2, 由于r2什么变元都可以且总为成功, 所以, not(X)是成功的。
- 若X为真, 匹配r1后, X为真, 控制通过! 传到fail, 则r1失败。于是回溯到! 过不去, 只好失败。由于用了!就地失败, 它不再匹配r2, 故not(X)为失败。
- 正是由于这个原因, 谓词p和not(not (p))求值结果不能保证一样, 有时not(p)和not(not (p))求值结果倒是一样的, 以下是not谓词出毛病的例子:

## 例 不可靠的not谓词

假定一规则test有以下定义:

test (S, T):-S=T.

运行以下查询时有:

?-test(3, 5).

no

?-test(5, 5)

yes

?-not( test(5, 5) )

no

?-test(X, 3), R is X+2.

X=3

R=5


?- not (not test (X, 3)), R is X+2.

! error in arithmetic expression : not a number

- r1 not(X):-X, ! , fail.

- r2 not(\_).

由于第二次not(外部的)求值时用到上例规则r1, 其中X是not(test(X, 3))的结果值, 故X+2不是数加2。这个问题原因在于子句逻辑的不可判定性



一般来说，将not应用于没有变量的项是安全的，因为这种项里不存在合一修改的变量。

### (3)不安全的cut

cut使我们处于两难的境地， 它的高效是以风险为代价得到的，如同60年代goto技巧对非结构化程序的影响。只要模型是超级归结， cut的两面性是不可以解决的。

## 6.5 PROLOG评价

- Prolog提供一种证明风格的声明式程序设计，推理清晰，概括能力强，程序和数据没有明显分离。
- Prolog程序具有自文档性
- 由于非过程性，它也成为潜在的并行程序设计语言的候选者
- 它的效率仍不及传统过程语言。由于它的声明性质，程序员在优化算法时作用有限
- 复杂的大型系统一开始很难按照证明系统开发，程序不大运算量惊人，而Prolog本身也只有局部量，天生来也不是大型软件开发的工具。因此，Prolog只能作为逻辑程序设计的独枝存在，解决大型应用多范型语言是个出路

## 逻辑式思考题

- 解决数独问题，参考快速排序算法表达式写出4\*4单元数独Prolog的算法表达式

例子：sudoku ( Puzzle, Solution ) :-

Solution = Puzzle.

```
|?- sudoku ([ _, _, 2, 3,  
            _, _, _, _,  
            _, _, _, _,  
            3, 4, _, _ ],  
            Solution)
```

Solution = [4, 1, 2, 3, 2, 3, 4, 1, 1, 2, 3, 4, 3, 4, 1, 2]