

# Prolog&Reasoning

-for Classical AI Problems

# 关于Prolog

Prolog 是"逻辑编程" (programming in Logic) 的意思。只要给出事实和规则，它会自动分析其中的逻辑关系，然后允许用户通过查询，完成复杂的逻辑运算。

Prolog是声明式逻辑编程语言，面向逻辑与问题，支持知识表示、描述逻辑关系和抽象概念。Prolog 语言以一阶谓词逻辑 Horn 子句集为语法，处理对象是知识，这种方式与一阶逻辑演算描述事物的方式很相似，因而，很适于表示人的思维和推理，常被用来编写解决非数值计算、推理、规划、决策等复杂问题的程序，被广泛应用于专家系统设计与自然语言处理等人工智能领域。

# 关于Prolog

2300B.C. 亚里士多德提出了著名的逻辑三段论：

- 苏格拉底是人
- 人都是要死的
- 所以苏格拉底是要死的

Prolog如何证明？

# 关于Prolog

三段论：

苏格拉底是人

人都是要死的

苏格拉底会死吗？

Prolog：

man(socrates).

mortal(X):- man(X).

?- mortal(socrates).

# 关于Prolog

事实：苏格拉底是人  
`man(socrates).`

规则：人都是要死的  
`mortal(X):- man(X).`

查询：苏格拉底会死吗？  
`?- mortal(socrates).`

# Outline

- SWI-Prolog
- 基本语法
- 地图着色问题
- 求阶乘
- 汉内塔
- Prolog列表
- 农夫过河问题

# SWI-Prolog

先安装 Prolog 的运行环境 SWI-Prolog，才能运行后面的代码。



```
sudo apt-get install swi-prolog
```

```
$ swipl  
?-
```



# Download SWI-Prolog stable versions

[HOME](#)[DOWNLOAD](#)[DOCUMENTATION](#)[TUTORIALS](#)[COMMUNITY](#)[USERS](#)[WIKI](#)

Linux versions are often available as a package for your distribution. We collect information about available packages and issues for building on specific distros [here](#). We provide a [PPA](#) for [Ubuntu](#) and [snap images](#)






Please check the [windows release notes](#) (also in the SWI-Prolog startup menu of your installed version) for details.



Examine the [ChangeLog](#).

## Binaries

	12,810,386 bytes	<a href="#">SWI-Prolog 8.2.1-1 for Microsoft Windows (64 bit)</a> Self-installing executable for Microsoft's Windows 64-bit editions. Requires at least Windows 7. See the <a href="#">reference manual</a> for deciding on whether to use the 32- or 64-bits version. This binary is linked against GMP 6.1.1 which is covered by the LGPL license. <b>SHA256:</b> 45fc082c3f39188657e21c0bd51173a0be1bdb7528f584913ac7f18b49fd0cb6
	12,460,084 bytes	<a href="#">SWI-Prolog 8.2.1-1 for Microsoft Windows (32 bit)</a> Self-installing executable for MS-Windows. Requires at least Windows 7. Installs <b>swipl-win.exe</b> and <b>swipl.exe</b> . This binary is linked against GMP 6.1.1 which is covered by the LGPL license. <b>SHA256:</b> 38537e2cd70630d65a9868708fd305e5600d654fb736d1dbc16090fe9eb2894d
	27,697,698 bytes	<a href="#">SWI-Prolog 8.2.1-1 for MacOSX 10.12 (Sierra) and later on intel</a> Installer with binaries created using <a href="#">Macports</a> . Installs /opt/local/bin/swipl. Needs <a href="#">xquartz</a> (X11) and the Developer Tools (Xcode) installed for running the <a href="#">development tools</a> <b>SHA256:</b> 1fd495fea2e523b098c7221c092fb6403cbeed7f9c99df3737cd336bb39d6b84

## Sources



# SWI-Prolog

进入 Prolog 运行环境，?-是命令提示符。下面是 Hello world 例子。

```
?- write("Hello, world").
```

```
Hello, world!
```

```
true.
```

几个地方需要注意。Prolog 所有语句的结尾都用一个"点" (.) 表示结束。write()是打印命令。命令本身就是一个表达式，输出完成以后，返回值就是true.，也会显示出来。如果想在 Hello world 之间插入一个换行，可以使用nl命令。

# SWI-Prolog

```
?- write('Hello,'), nl, write('world').
```

```
Hello,
```

```
world
```

```
true.
```

要退出 SWI-Prolog，可以使用halt命令，后面还是要加一个点。

```
?- halt.
```

# 基本语法 常量和变量

Prolog 的变量和常量规则很简单：小写字母开头的字符串，就是常量；大写字母开头的字符串，就是变量。变量可以和常量匹配/合一。

```
?- write(abc).
```

```
abc
```

```
true.
```

```
?- write(Abc).
```

```
_3386
```

```
true.
```

# Prolog程序

Prolog 程序通常由一组事实、规则和问题（目标）构成。问题是程序执行的起点，也称为程序的目标。事实与规则由一组事实和规则共同构成子句，Prolog 语言主要就是通过这些子句来实现逻辑推理的。

事实是由谓词名及用“()”括起来的一个或多个对象构成，一般表示对象的性质或关系。谓词和对象可由用户自己定义，另外，Prolog 语言也定义了许多内部函数和标准谓词，用以方便程序编写与提高程序运行效率。

# Prolog程序

事实的语法结构为：<谓词名> (<项表>).

例如，谓词 mother (a1,b2) .表示 a1 与 b2 之间有母子关系。

规则是由几个具有相互依赖性的谓词组成，用来描述事实之间的依赖关系，一般表示对象间的因果关系、蕴含关系或对应关系。

其语法结构为：

<谓词名> (<项表>) :-<谓词名> (<项表>) {,<谓词名> (<项表>)}.

例如，grandmother (X, Y):-mother (X,Z),mother (Z,Y).表示如果 X 是 Z 的母亲且 Z 是 Y 的母亲，则 X 是 Y的祖母。

# 基本语法 关系和属性

再比如，jack 的朋友是 peter，写成`friend(jack, peter).`。注意，jack 的朋友是 peter，不等于 peter 的朋友是 jack。如果两个人都认为对方是朋友，要写成下面这样。

```
friend(jack, peter).
```

```
friend(peter, jack).
```

如果括号里面只有一个参数，就表示对象拥有该属性，比如 jack 是男性，写成`male(jack).`。

# 基本语法 规则

规则是推理方法，即如何从一个论断得到另一个论断。举例来说，我们定下一条规则：所有朋友关系都是相互的，规则写成下面这样。

```
friend(X, Y) :- friend(Y,X).
```

上面代码中，X和Y都是大写，表示这是两个变量。符号:-表示推理关系，含义是只要右边的表达式friend(Y, X)为true，那么左边的表达式friend(X, Y)也为true。因此，根据这条规则，friend(jack, peter)就可以推理得到friend(peter, jack)。将变量和知识库中的信息进行合一Prolog的核心。

# 基本语法 规则

如果一条规则取决于多个条件同时为true，则条件之间使用逗号分隔。

```
mother(X, Y) :- child(Y,X), female(X).
```

上面代码中，X是Y的母亲（mother(X, Y)）取决于两个条件：Y是X的小孩，X必须是女性。只有这两个条件都为true，mother(X, Y)才为true。



# 基本语法 规则

如果一条规则取决于某个条件为false，则在条件之前加上\+表示否定。

```
onesidelove(X, Y) :- loves(X, Y), \+ loves(Y,X).
```

上面代码中，X单相思Y，取决于两个条件。第一个条件是X喜欢Y，第二个条件是Y不喜欢X。

# 基本语法 规则

Prolog 程序的执行过程完全依靠其语言内含的推理机来进行匹配、回溯搜索。此求解过程就是一个深度优先的消解过程。因此，程序员在编写程序时只要将主要精力放在描述对象间的逻辑关系上而不必过多地考虑过程的细节，编写事实和规则，并提出目标，具体的操作交由计算机自动处理求解即可。

# 基本语法 查询

Prolog 支持查询已经设定的条件。我们先写一个脚本auto.pl。

```
friend(john, julia).
```

```
friend(john, jack).
```

```
friend(julia, sam).
```

```
friend(julia, molly).
```

# 基本语法 查询

然后在 SWI-Prolog 里面加载这个脚本。

```
?- [auto].
```

```
true.
```

上面代码中，true.是返回的结果，表示加载成功。

# 基本语法 查询

然后，可以查询两个人是否为朋友。

```
?- friend(john, jack).
```

```
true.
```

```
?- friend(john, sam).
```

```
false.
```

# 基本语法 查询

listing()函数可以列出所有的朋友关系。

?- listing(friend).

friend(john, julia).

friend(john, jack).

friend(julia, sam).

friend(julia, molly).

true.

# 基本语法 查询

还可以查询john有多少个朋友。

```
?- friend(john, Who).
```

```
Who = julia ;
```

```
Who = jack.
```

上面代码中，Who是变量名。任意的变量名都可以，只要首字母为大写。

# 合一操作

unification–匹配工作三个必要条件

1. 目标谓词一致
2. 谓词参数数量一致
3. 所有参数一致



# 合一的例子

---

$A = A$  :成功(重言式)

$A = B, B = abc$  :  $A$ 和  $B$ 二者合一于原子  $abc$

$xyz = C, C = D$  :合一是对称的

$abc = abc$  :合一成功

$abc = xyz$  :合一失败，因为原子是不同的

$f(A) = f(B)$  :  $A$ 合一于  $B$

$f(A) = g(B)$  :失败，因为项的头部是不同的

$f(A) = f(B, C)$  :合一失败，因为项有不同的元数

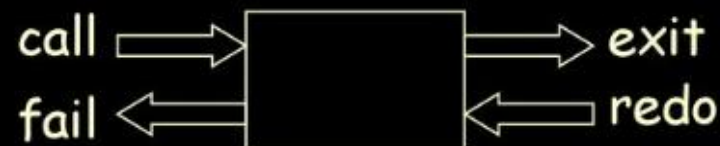
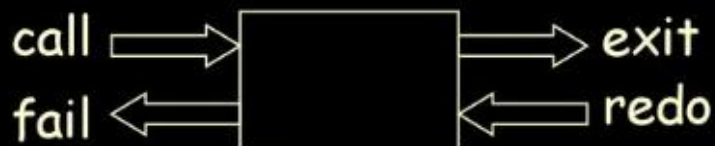
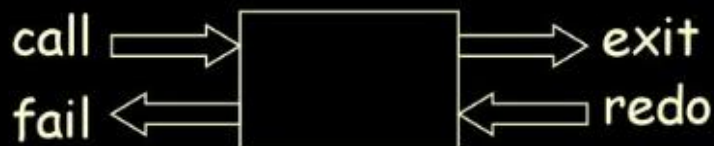
$f(g(A)) = f(B)$  :  $B$ 合一于项  $g(A)$

$f(g(A), A) = f(B, xyz)$  :  $A$ 合一于原子  $xyz$ 而  $B$ 合一于项  $g(xyz)$

# 查询端口

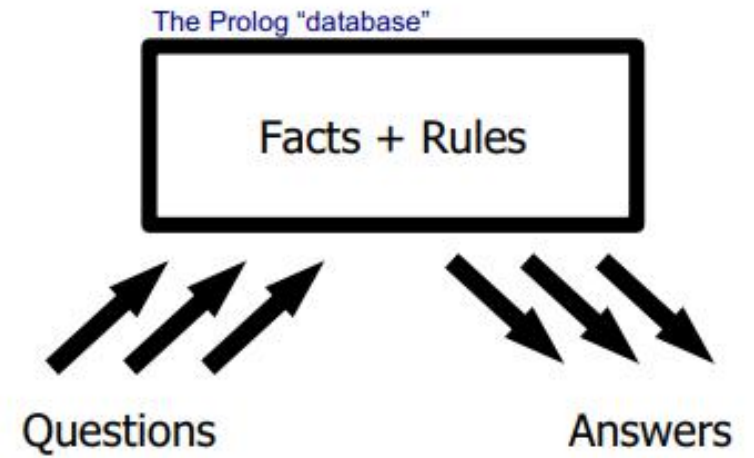
每个谓词查询有4个端口：

1. call 开始查询
2. exit 成功查询到结果，并绑定到变量
3. redo 从上次成功查询处之后继续查询，先释放变量
4. fail 查询失败退出



# Prolog vs 其他语言

数据库中的事实代替了一般语言中的数据结构。  
回溯功能能够完成一般语言中的循环操作。  
通过模式匹配能够完成一般语言中的判断操作。  
规则能够被单独地调试，它和一般语言中的模块相对应。  
规则之间的调用和一般语言中的函数的调用类似。



# 地图着色问题

地图的相邻区域不能使用同一种颜色。现在有三种颜色：红、绿、蓝。请问如何为湖南及其相邻省份地图着色？



# 地图着色问题

首先，定义三种颜色。

color(red).

color(green).

color(blue).



# 地图着色问题

然后，定义着色规则。

```
colorify(Hn,Cq,Hb,Jx,Gd,Gx,Gz) :-  
    color(Hn), color(Cq), color(Hb), color(Jx),  
    color(Gd),color(Gx),color(Gz),  
    \+ Hn=Cq, \+ Hn=Hb, \+ Hn=Jx,  
    \+ Hn=Gd, \+ Hn=Gx, \+ Hn=Gz,  
    \+ Cq=Hb, \+ Cq=Gz, \+ Hb=Jx,  
    \+ Jx=Gd, \+ Gd=Gx, \+ Gx=Gz,  
    \+ Gz=Cq.
```



# 地图着色问题

这两段代码合在一起，写入脚本auto.pl，再加载这个脚本。

```
?- [auto].  
true.
```



# 地图着色问题

执行表达式`colorify(Hn,Cq,Hb,Jx,Gd,Gx,Gz).`, SWI-Prolog 就会将三种颜色依次赋值给变量, 测试哪些组合是可能的结果。

?- `colorify(Hn,Cq,Hb,Jx,Gd,Gx,Gz).`

`Hn = red,`

`Cq = Jx, Jx = Gx, Gx = green,`

`Hb = Gd, Gd = Gz, Gz = blue .`





# 地图着色问题

执行表达式`colorify(Hn,Cq,Hb,Jx,Gd,Gx,Gz).`, SWI-Prolog 就会将三种颜色依次赋值给变量, 测试哪些组合是可能的结果。

?- `colorify(Hn,Cq,Hb,Jx,Gd,Gx,Gz).`

`Hn = red,`

`Cq = Jx, Jx = Gx, Gx = green,`

`Hb = Gd, Gd = Gz, Gz = blue .`

按 ; 键, 计算机可以给出4组解,  
即有4种可行的地图着色方法。



# 求阶乘 使用递归

$$N! = N \times N-1 \times N-2 \times \dots \times 3 \times 2 \times 1$$

?- factorial(N,X).

# 求阶乘 使用递归

factorial(0,1).

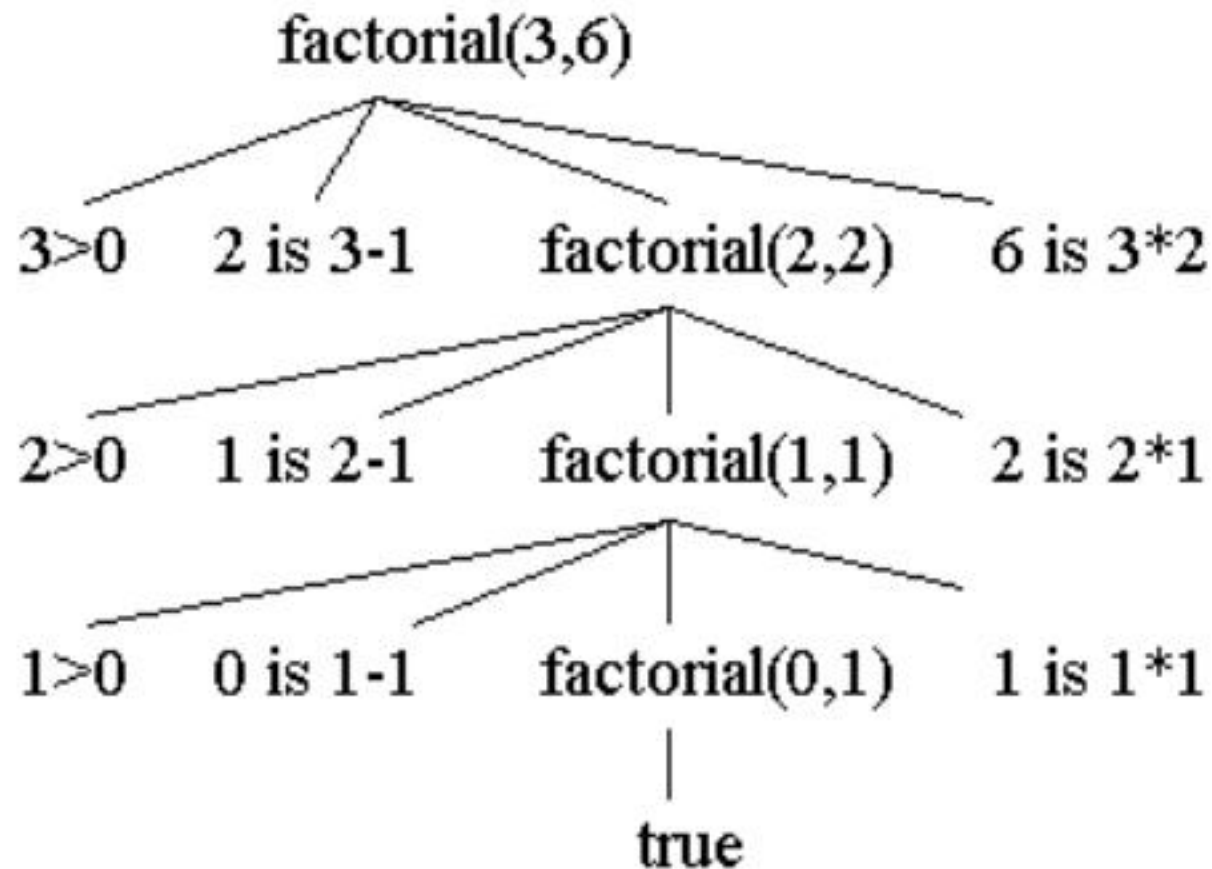
factorial(N,F) :-

N>0,

N1 is N-1,

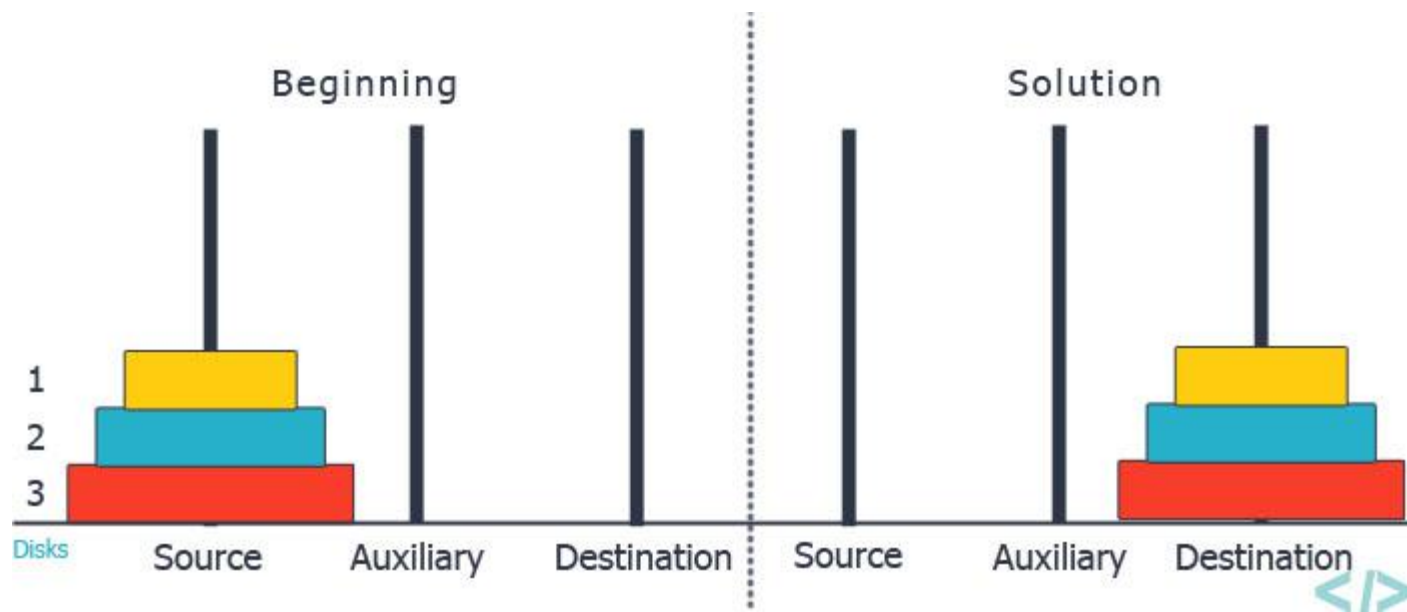
factorial(N1,F1),

F is N \* F1.



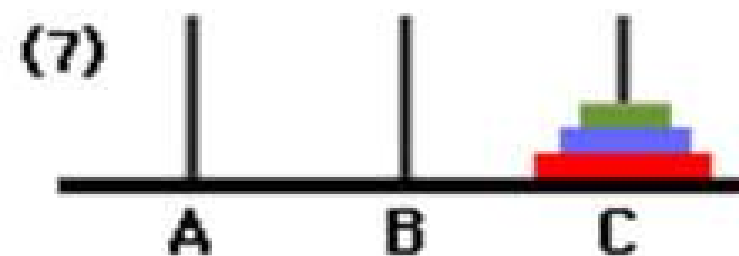
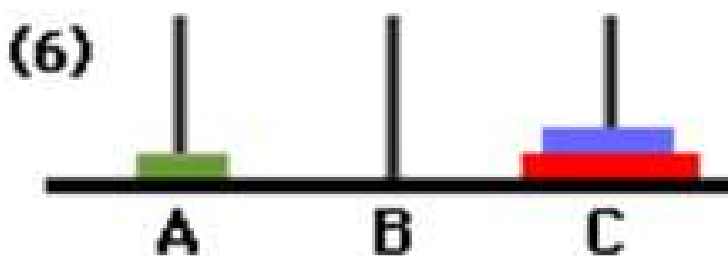
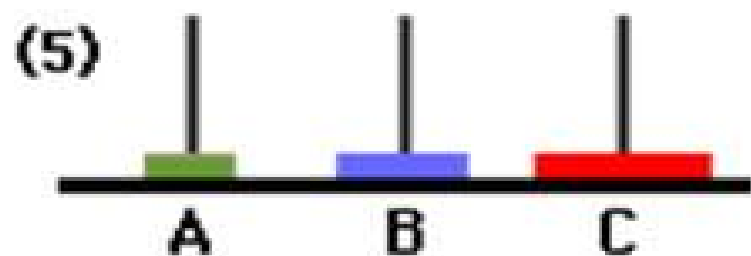
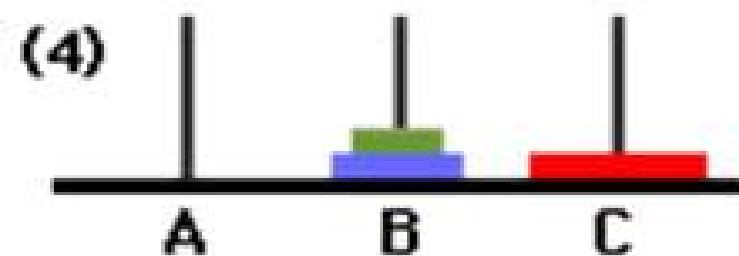
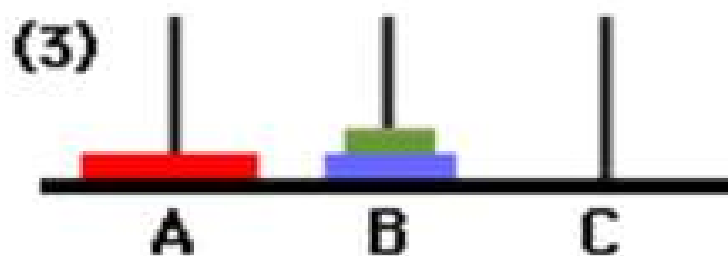
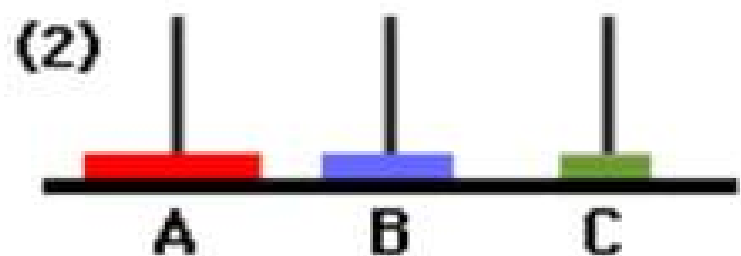
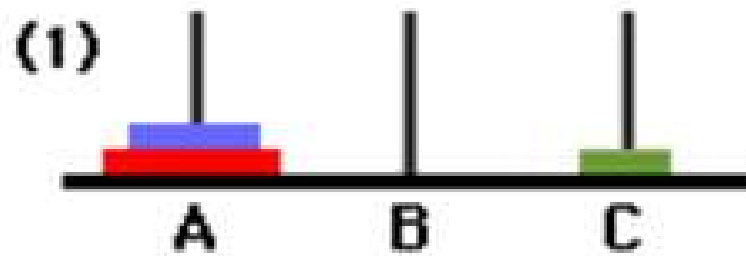
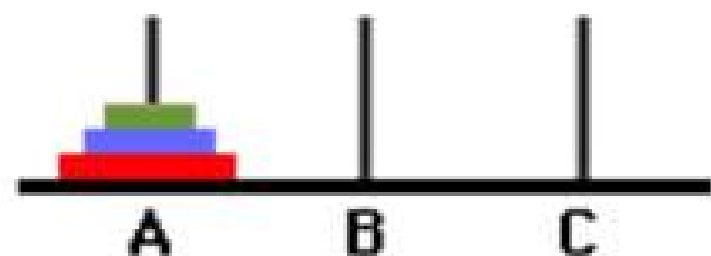
# 汉内塔 使用递归

开天辟地的神勃拉玛在一个庙里留下了三根金刚石的棒，第一根上面套着64个圆的金片，最大的一个在底下，其余一个比一个小，依次叠上去，庙里的众僧不倦地把它们一个个地从这根棒搬到另一根棒上，规定可利用中间的一根棒作为帮助，但每次只能搬一个，而且大的不能放在小的上面。看来，众僧们耗尽毕生精力也不可能完成金片的移动。



# 汉内塔 使用递归

3 DISKS



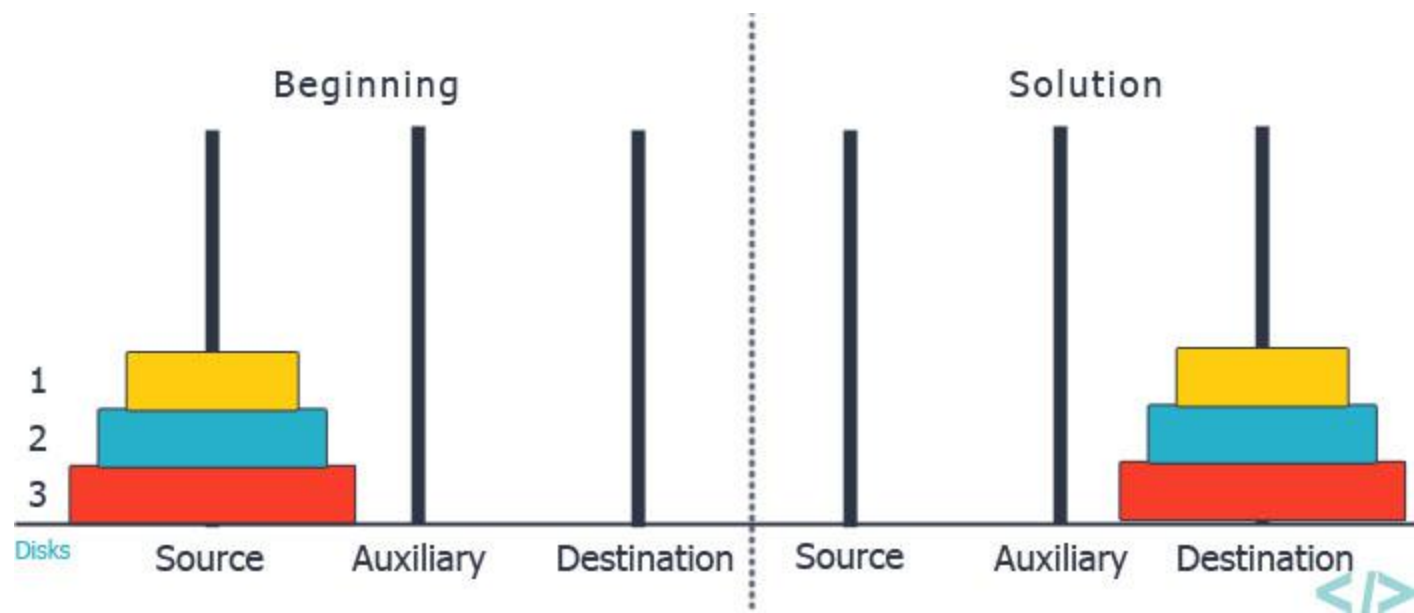
# 汉内塔 使用递归

```
move(1,X,Y,_):-
```

```
    write('Move top disk from '),  
    write(X), write(' to '), write(Y),  
    nl.
```

```
move(N,X,Y,Z):-
```

```
    N>1,  
    M is N-1,  
    move(M,X,Z,Y),  
    move(1,X,Y,_),  
    move(M,Z,Y,X).
```



# Prolog列表

Prolog用 [...] 构建列表，有限列表可以直接写，如 [1,2,3,4]。[X|Y] 表示一个列表的头元素是 X，尾部是 Y。如果某个列表可以和[X|Y] 合一，则该列表的头元素必然可以和X合一，且其尾部可以和Y合一。比如：

```
member(X,[X|R]).
```

```
member(X,[Y|R]) :- member(X,R).
```

表示

X is a member of a list whose first element is X.

X is a member of a list whose tail is R if X is a member of R.

# Prolog列表

`member(X,[X|R]).`

`member(X,[Y|R]) :- member(X,R).`

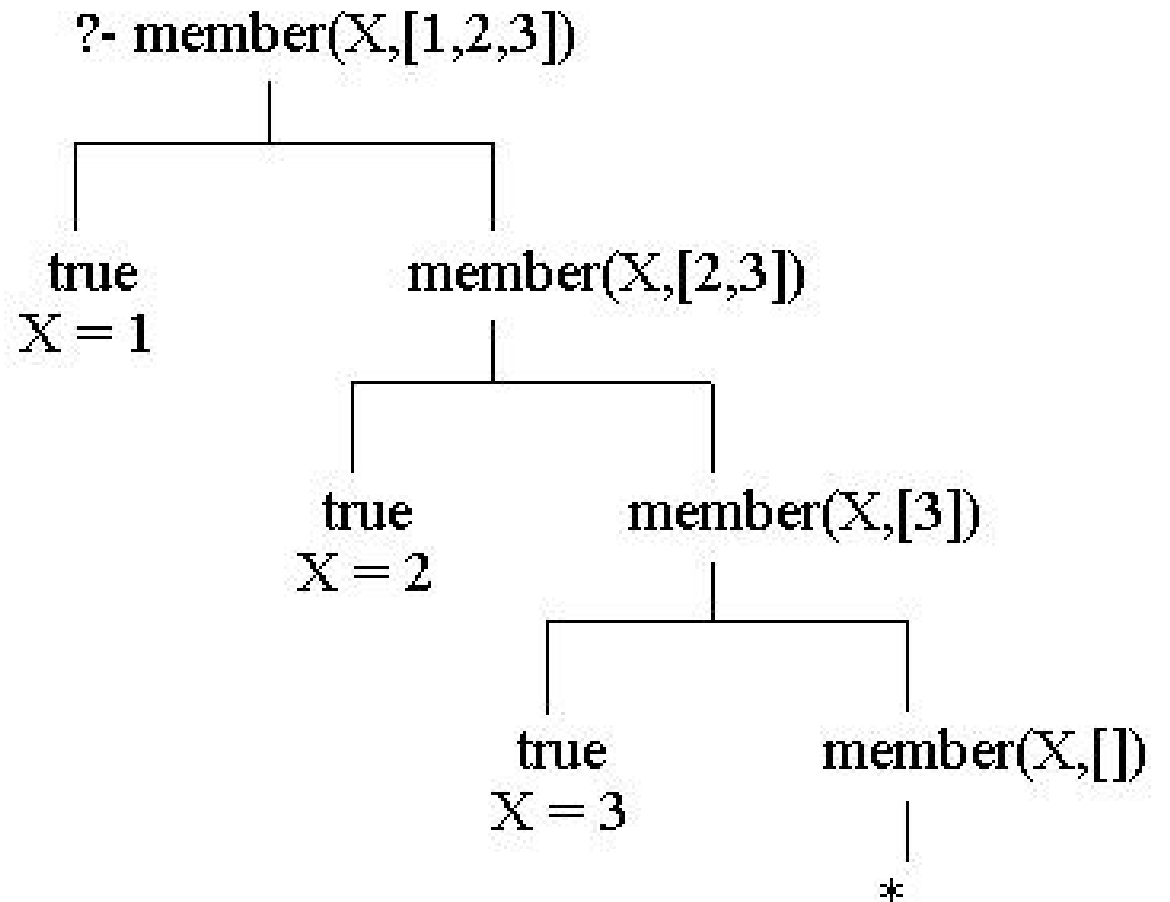
`?- member(X,[1,2,3]).`

`X = 1 ;`

`X = 2 ;`

`X = 3 ;`

`No`





# Prolog列表

?- member([3,Y], [[1,a],[2,m],[3,z],[4,v],[3,p]]).

Y = z ;

Y = p ;

?- member(X,[23,45,67,12,222,19,9,6]), Y is X\*X, Y < 100.

X = 9   Y = 81 ;

X = 6   Y = 36 ;

# Prolog列表

```
member(X,[X|R]).
```

```
member(X,[Y|R]) :- member(X,R).
```

其实可以定义为：

```
member(X,[X|_]).
```

```
member(X,[_|R]) :- member(X,R).
```

# Prolog列表

翻转一个列表：

```
reverse([X|Y],Z,W) :- reverse(Y,[X|Z],W).  
reverse([],X,X).
```

# Prolog列表

翻转一个列表：

```
reverse([X|Y],Z,W) :- reverse(Y,[X|Z],W).  
reverse([],X,X).
```

```
?- reverse([1,2,3],[],A)  
  |  
  |  
reverse([2,3],[1],A)  
  |  
  |  
reverse([3],[2,1],A)  
  |  
  |  
reverse([], [3,2,1], A)  
  |  
  |  
true  
A = [3,2,1]
```

# Prolog列表

和 'member' 类似, 我们可以定义 'takeout'.

```
takeout(X,[X|R],R).
```

```
takeout(X,[F|R],[F|S]) :- takeout(X,R,S).
```

表示

When  $X$  is taken out of  $[X|R]$ ,  $R$  results.

When  $X$  is taken out of the tail of  $[X|R]$ ,  $[X|S]$  results, where  $S$  is the result of taking  $X$  out of  $R$ .

# Prolog列表

?- takeout(X,[1,2,3],L).

X=1 L=[2,3] ;

X=2 L=[1,3] ;

X=3 L=[1,2] ;

takeout(3,[1,2,3],[1,2])

|

takeout(3,[2,3],[2])

|

takeout(3,[3],[])

|

true

# Prolog列表

基于'takeout', 我们可以定义'perm'.

```
perm([X|Y],Z) :- perm(Y,W), takeout(X,Z,W).
```

```
perm([],[]).
```

表示：

Z is a permutation of [X|Y] provided W is a permutation of Y and then X is put into W to produce Z .

[] is the (only) permutation of [].

# Prolog列表

?- perm([1,2,3],P).

P = [1,2,3] ;

P = [2,1,3] ;

P = [2,3,1] ;

P = [1,3,2] ;

P = [3,1,2] ;

P = [3,2,1] ;

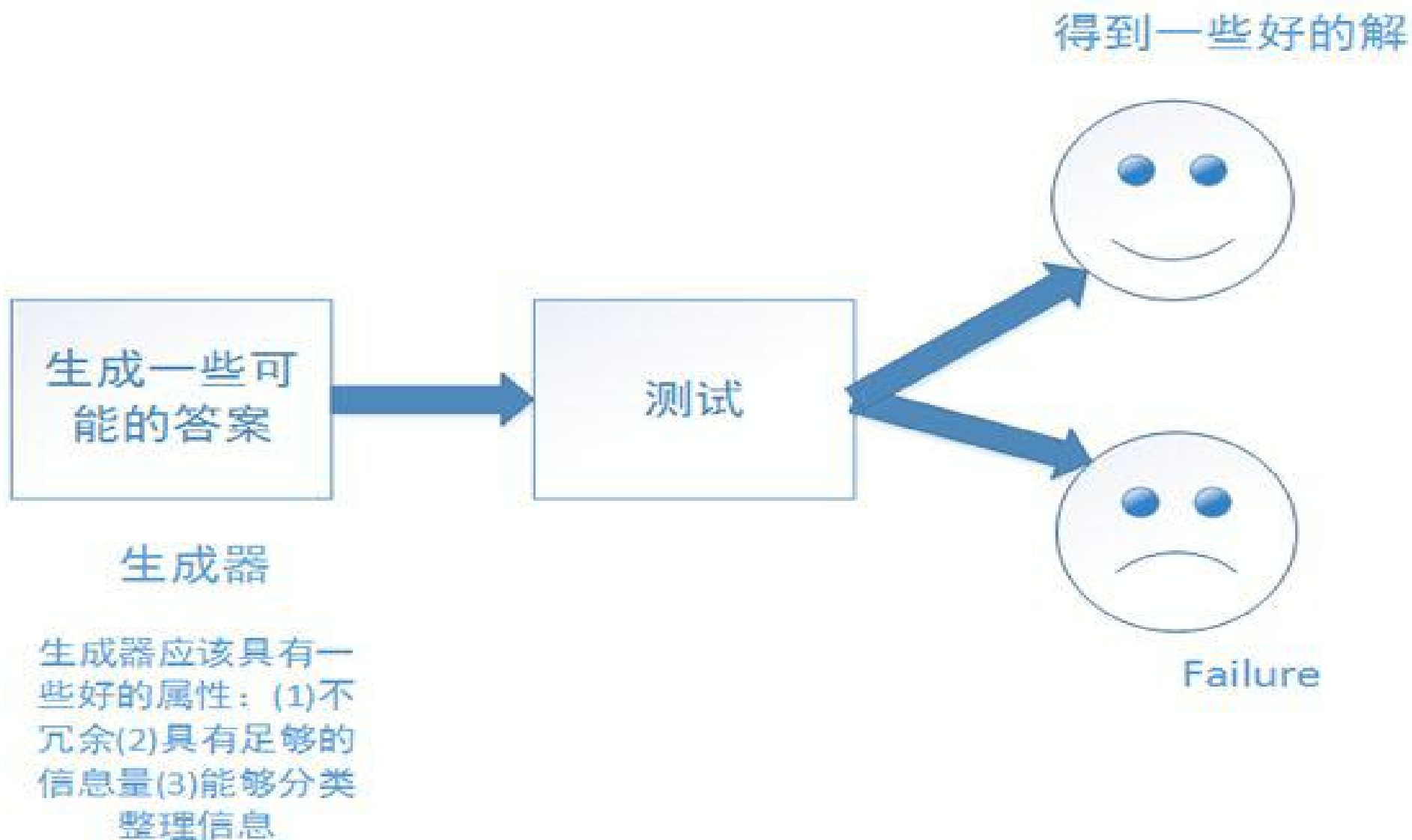


# 农夫过河问题



农夫带着一匹狼、一只羊、一些白菜来到河边，想要乘坐一只小船过河去，但是小船很小，每次只能搭载两样物品，请问农夫如何过河？

# 生成测试法



# 问题状态表示

- 四元组表示  $(w, w, w, w)$  分别表示农夫、狼、羊、白菜的位置

取值：**w**est表示左岸；**e**ast 表示右岸

起始状态  $(w, w, w, w)$ ；目标状态  $(e, e, e, e)$

- 如果农夫带着狼过河  $(w, w, w, w) \rightarrow (e, e, w, w)$

羊会吃掉白菜，否决！

# 问题迁移表示

- 每次移动，农夫可以最多携带一种物品，迁移的物品可以有四种：wolf, goat, cabbage, nothing
- (nothing 表示农夫自己过河不带物品)

# 安全检查

- 如何判断一个状态是安全的？

当狼或者羊跟人在同一边；

并且

当羊或者白菜跟人在同一边

# 搜索过程

- 定义边界条件
- 保证安全
- 递归搜索

# 参考代码

change(e,w).

change(w,e).

move([X,X,Goat,Cabbage],wolf,[Y,Y,Goat,Cabbage]) :- change(X,Y).

move([X,Wolf,X,Cabbage],goat,[Y,Wolf,Y,Cabbage]) :- change(X,Y).

move([X,Wolf,Goat,X],cabbage,[Y,Wolf,Goat,Y]) :- change(X,Y).

move([X,Wolf,Goat,C],nothing,[Y,Wolf,Goat,C]) :- change(X,Y).





# 参考代码

```
solution([e,e,e,e],[]).
```

```
solution(Config,[Move|Rest]) :-
```

```
    move(Config,Move,NextConfig),
```

```
    safe(NextConfig),
```

```
    solution(NextConfig,Rest).
```

```
%length(X,7), solution([w,w,w,w],X).
```