Artificial Intelligence

7、知识、推理与规划

对应课本7-12章

本节内容

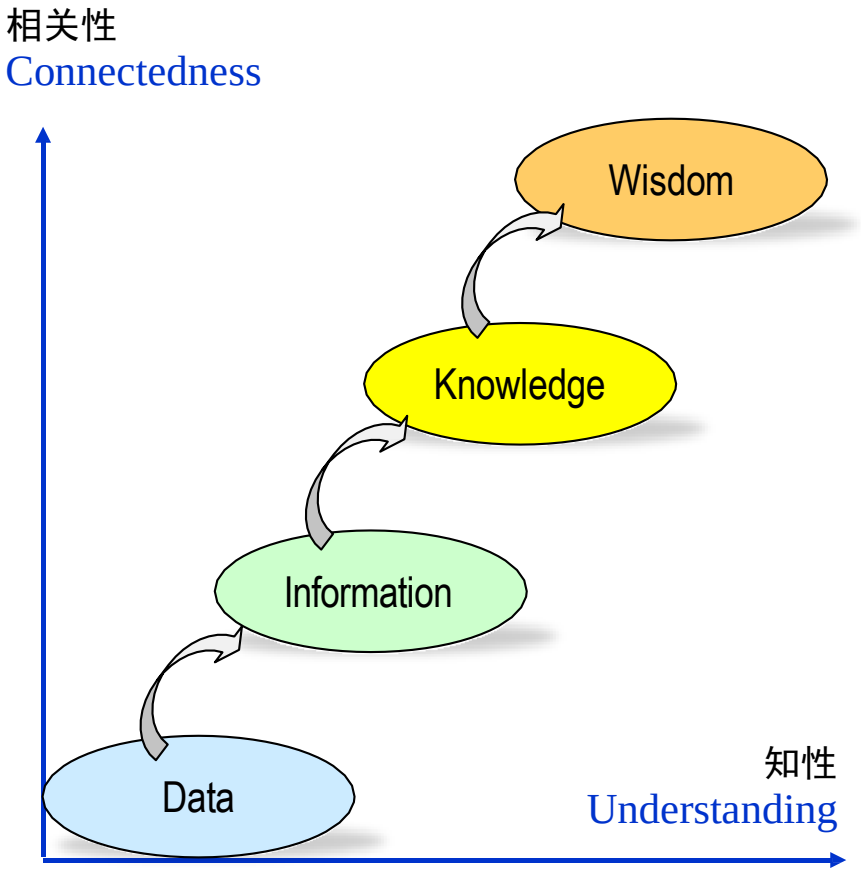
- 知识工程
- 基于知识的Agent
- Wumpus世界
- 逻辑证明
- 一阶逻辑
- 本体工程

数据、信息、知识与智慧

- 数据Data：
  - 世界的计量和表征
  - 表现为事实、信号、或者符号
- 信息Information：
  - 对数据赋予含义而生成
  - 结构与功能的，主观与客观的
- 知识Knowledge：
  - 对信息加工而确立
  - 表现为加工的、过程的或者命题的
- 智慧Wisdom：
  - 做出决定和判断的经验
  - 表现为“知因”、“知然”、或“因何”

例：银行

- 数据
  - 数字100或者5（无上下文）
- 信息
  - 本金：100美元；利率：5%
- 知识
  - 年底拿回105美元
- 智慧
  - 投资?



显性与隐性知识

- 显性知识
  - 可以表示为形式语言，包括语法陈述、数学表达式、等等。
  - 可以快捷地转化成其它形式。
  - 可以容易地用计算机语言、决策树和规则等表示。
- 隐性知识
  - 个人的经验和无形的因素、如观点、等等。
  - 难以用形式化语言来表示。
  - 神经网络提供了表示隐形知识的方法。

知识的类型

类型	特点
静态知识	不太可能改变
动态知识	记录在数据库中
表层知识	通过经验积累
深层知识	理论 / 证明 / 问题细节
过程性知识	描述如何解决问题
陈述性知识	描述已知的问题是什么
元知识	描述知识的知识
启发式知识	引导推理过程的经验法则

知识库和知识库系统

- “知识库 (KB)”这个术语是用于区分更广泛使用的术语“数据库 (DB)”。
- 知识库被用于存储复杂的结构和非结构化知识。它由一套语句组成，每个语句都是由一种被称为知识表示语言来表示的，从而表示关于世界的某些断言。
- 一个知识库系统 (KBS) 由知识库和推理引擎组成，其中，知识库表示关于世界的事实，推理引擎则可以基于这些事实进行推理。

知识工程和基于知识的工程

- 知识工程Knowledge Engineering(KE):
  - 指的是构建、维护和使用知识库系统中所关联的所有技术、科学和社会的方方面面。
- 基于知识的工程Knowledge-based Engineering(KBE):
  - 将基于知识的系统技术用于制造设计和生产领域。

- KB或KBE本质上是在知识模型基础之上的工程，它采用知识表示来表征设计过程的产品。
- KB或KBE最初的应用是专家系统。

知识表示概述

- 什么是知识表示
  - 关注于设计计算机表示来采集关于世界的知识，可用于解决复杂的问题。
  - 与过程性代码相比，使复杂的软件容易定义和维护，可用于专家系统。
- 为什么采用知识表示
  - 传统的过程性代码并非是解决复杂问题的最好形式。

知识表示的核心问题

- 原语
  - 用于表示知识的基础框架，例如：语义网络、一阶逻辑、等等。
- 元表示
  - 知识表示语言用该语言本身表示，例如：在Frame环境中，所有的frames都是某个frame的实例。
- 不完备性
  - 将确定性因子与规则和结论相关联，例如：苏格拉底是人，具有50%的置信度。

知识表示的核心问题

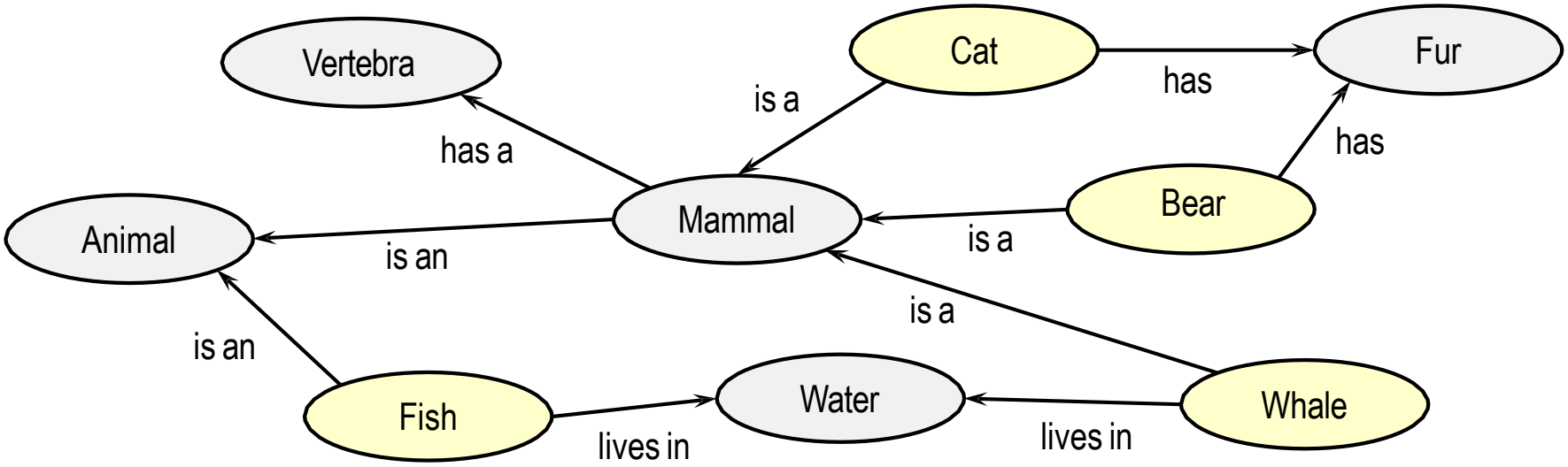
- 共性与事实
  - 共性：关于世界的一般性描述。例如：所有的人都会死。
  - 事实：共性中的具体事例。例如：苏格拉底是人，因此他会死。
- 表现的充分性
  - 它们想要的表示是如何表现的。
- 推理的有效性
  - 指的是该系统运行时的有效性。

典型的知识表示方法

- 贝叶斯网络
- 一阶逻辑
- 基于Frame的系统
- 本体
- 产生式系统
- 脚本
- 语义网络

什么是语义网络

- 它是一种表示概念间语义关系的网络。
- 它是一种由节点和弧组成的有向或无向图，其中，节点：表示概念，弧：概念间的语义关系。



用Lisp语言表示的语义网络

- 采用关联表。

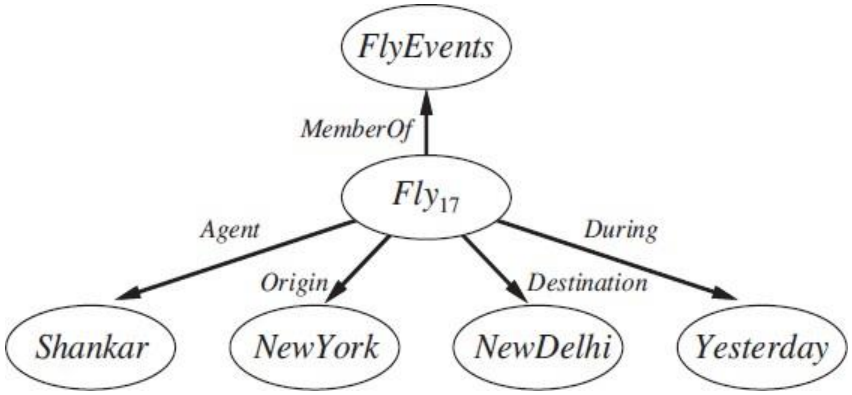
```
(defun a-knowledge-base ()
     ((canary     (is-a bird)
        (color yellow)
        (size small))
      (penguin (is-a bird)
                 (
      movement
      swim))
      (bird

      (is-a
      vertebrate)
         (has-part wings)
                  (reproduction egg-laying))))
```

- 使用“assoc”函数与关键字“canary”来提取关于“canary”类型的所有信息。

语义网络的基本概念

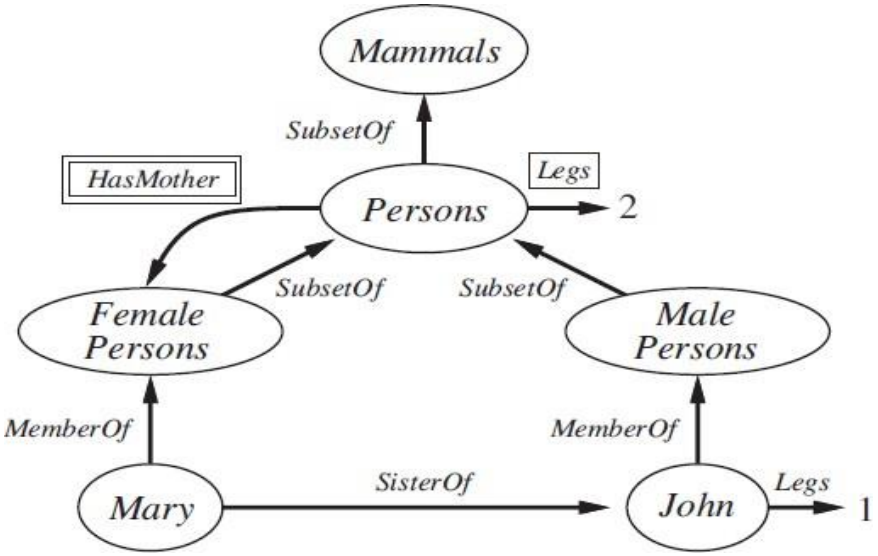
- 语义网络是基于认知的，被组织成为一个分类层次结构。
- 语义网络被采用的情形是当某种知识可以很好地化解为一组彼此相关的概念时。
- 但是，它难以驾驭大型领域，并且不能很好地表现性能或者元知识。
- 某些特性也不易表达，例如：
- 否定、析取、或者一般的非分类知识。

实例



一个逻辑断言的语义网络

Fly(*Shankar, NewYork, NewDelhi , Yesterday*)



一个具有类别和对象的语义网络

$Mary \in FemalePersons$, $John \in MalePersons$,

$SisterOf(Mary, John)$

$\forall x\ x \in Persons \Rightarrow [\forall y\ HasMother(x, y) \Rightarrow y \in FemalePersons]$

过程性与陈述性方法

- 过程性方法
  - C/C++/C#/Java,
  - Lisp,
  - Python.
- 陈述性方法
  - 命题逻辑
  - 一阶逻辑
  - 时序逻辑

基于知识的Agent

- 推理引擎：领域无关算法
- 知识库：领域相关的算法
- 知识库在实现上，等价于一组规则化描述的语句
  - 我们需要告诉它所有它需要知道的
  - 之后它就能进行自我问答
- 在这种定义下，Agent可以看做是知识层面的实体
  - 也就是说，他们只是知道问题的答案，但是不管如何实施
- 如果将Agent定义在实现层面上的话，那知识库中的知识和算法就应该指导它如何实现操作

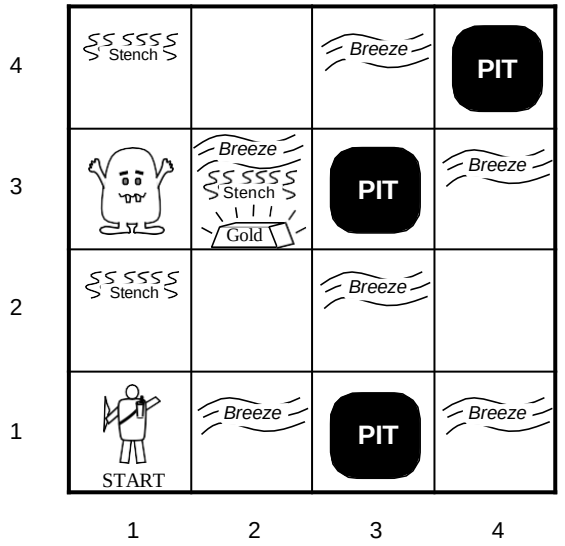
简单的基于知识的Agent

- Agent需要做到：
  - 表示状态、动作
  - 合并新的感知
  - 更新世界的内部表示
  - 演绎世界的隐藏属性
  - 演绎适当的动作

```
function KB-Agent(percept) returns an action
    static: KB, a knowledge base
            t, a counter, initially 0, indicating time
    Tell(KB, Make-Percept-Sentence(percept, t))
    action ← Ask(KB, Make-Action-Query(t))
    Tell(KB, Make-Action-Sentence(action, t))
    t ← t + 1
    return action
```

Wumpus世界

- 性能指标:
  - +1000:黄金
  - -1000:死亡
  - -1:每一步
  - -10:用箭
  - -20:爬出陷阱
- 执行器:
  - 向左、向右、前进
  - 射击: 发射一支箭
  - 抓住: 拾起黄金
  - 攀爬: 攀越陷阱
- 感受器:
  - Stench 臭气
  - Bump 碰撞
  - Breeze 微风
  - Glitter 闪光
  - Scream 尖叫

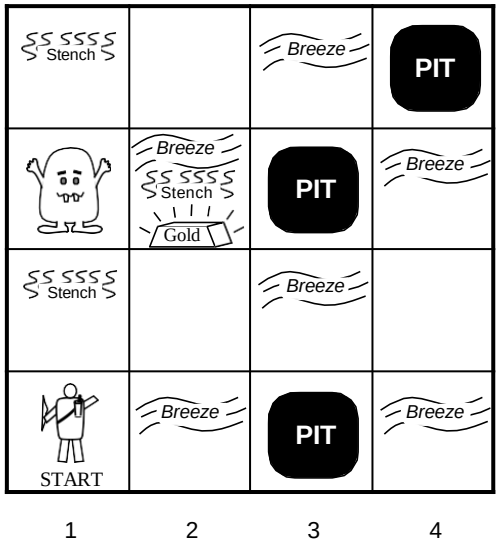


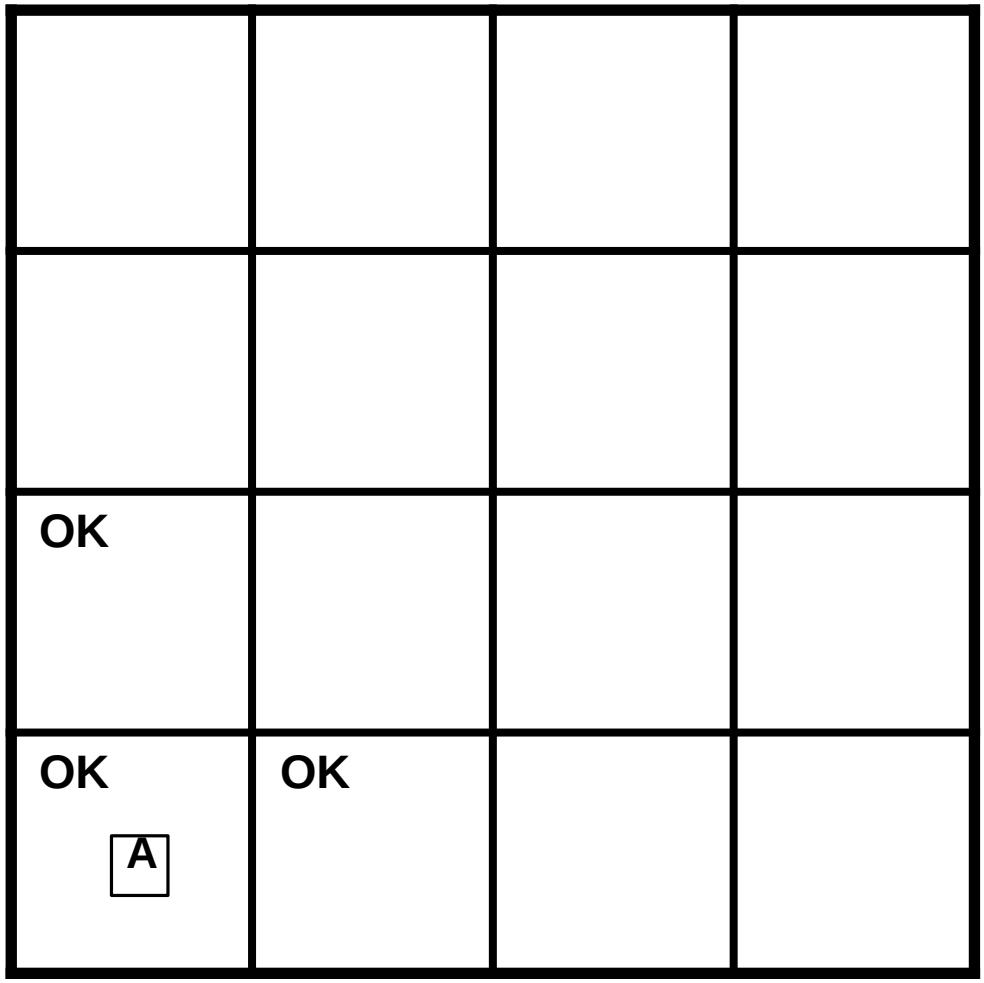
- 环境:
  - Agent智能体
  - Wumpus
  - Gold黄金
  - Pit陷阱

Wumpus世界

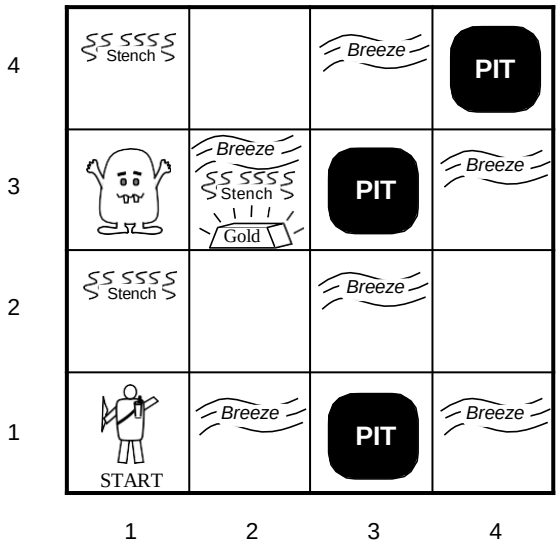
- 可观察性或不可观察：局部可观察
- 单或多Agent：单
- 确定性或不确定：环境是确定的
- 片段或延续：片段的，因为操作级别上没有顺序
- 静态或动态：静态的
- 离散或连续：离散的

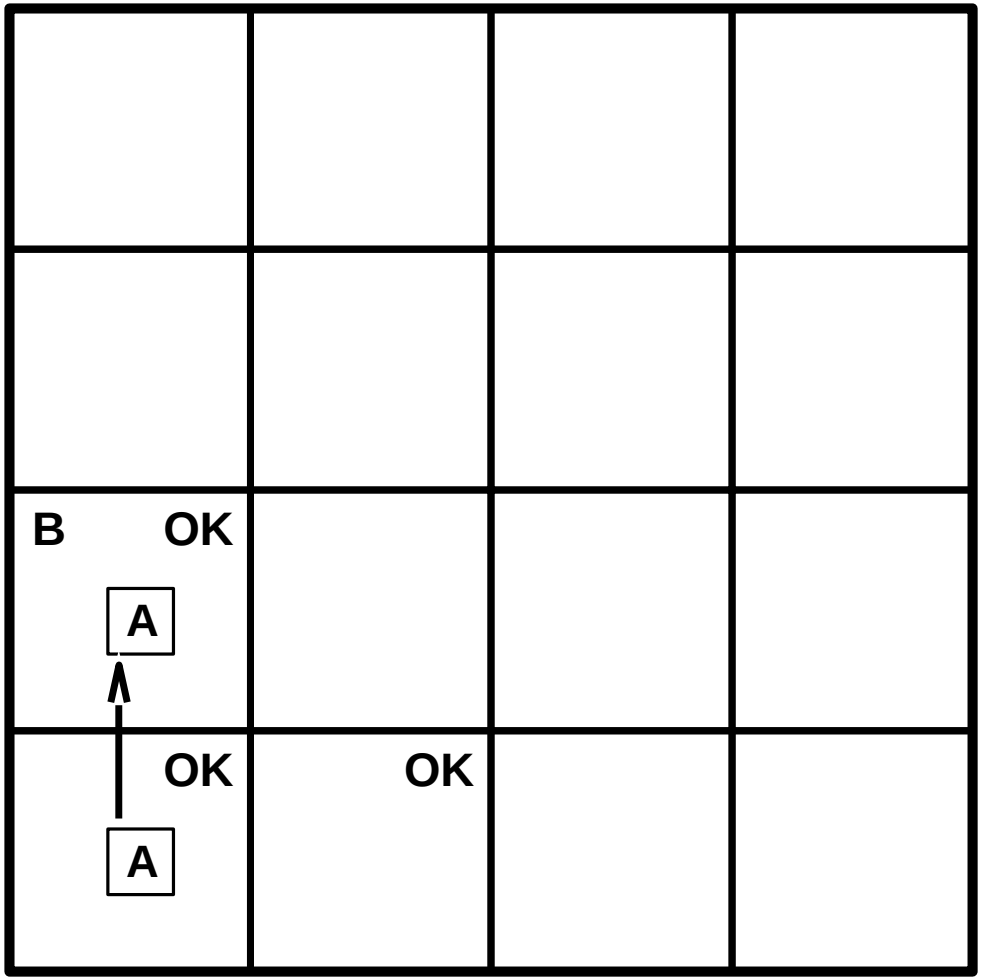
Wumpus世界探索


4
3
2
1
Stench
Breeze
PIT
Breeze
Stench
Gold
Breeze
Stench
Breeze
Breeze
PIT
Breeze
START
1
2
3
4

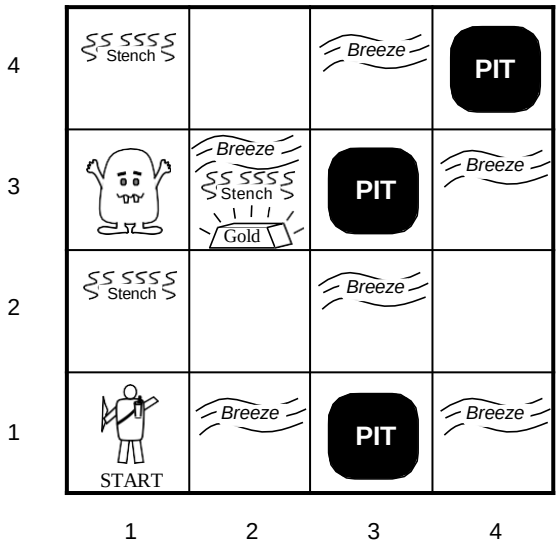

OK
OK
OK
A

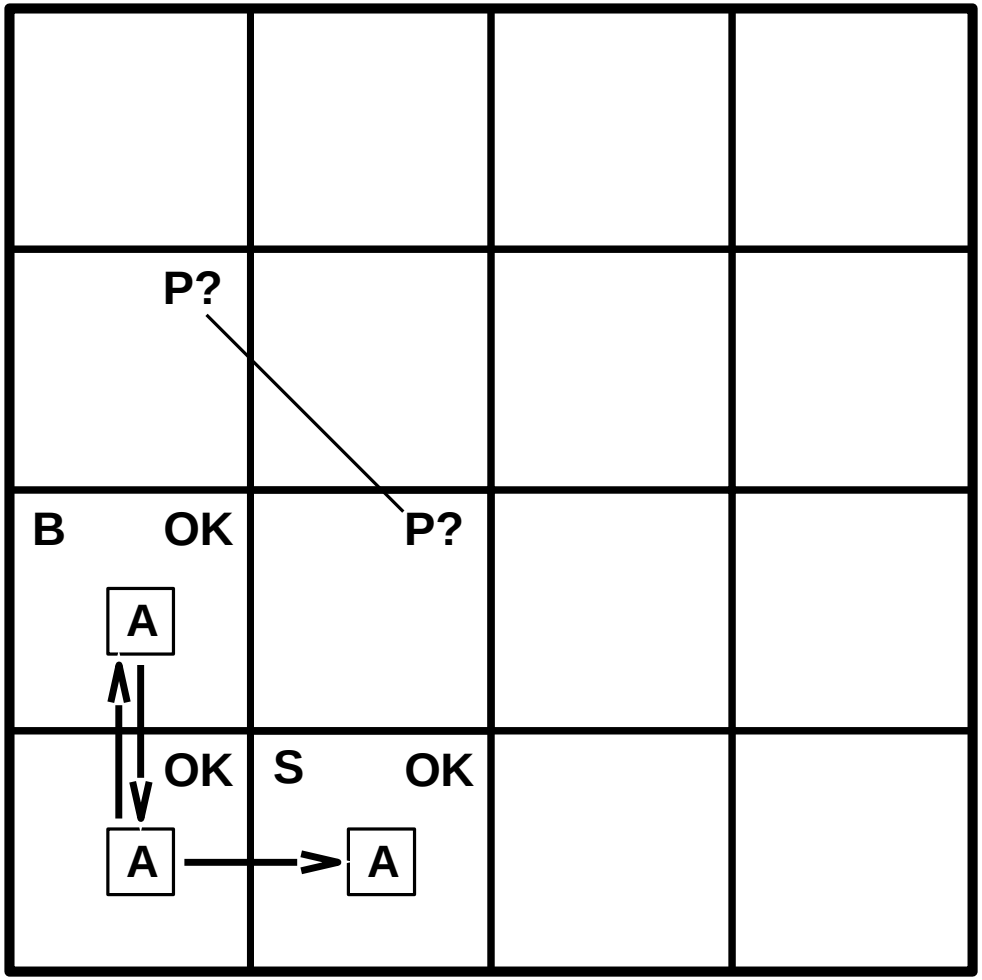
Wumpus世界探索


4
3
2
1
Stench
Breeze
PIT
Stench
Breeze
Gold
PIT
Breeze
Stench
Breeze
Breeze
PIT
Breeze
START
1
2
3
4

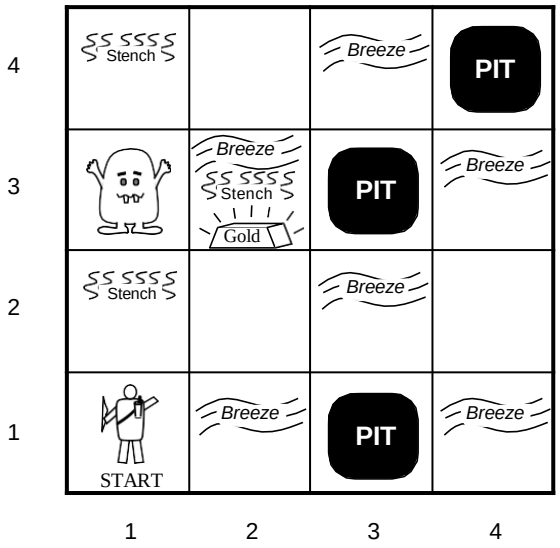

B
OK
A
OK
OK
A

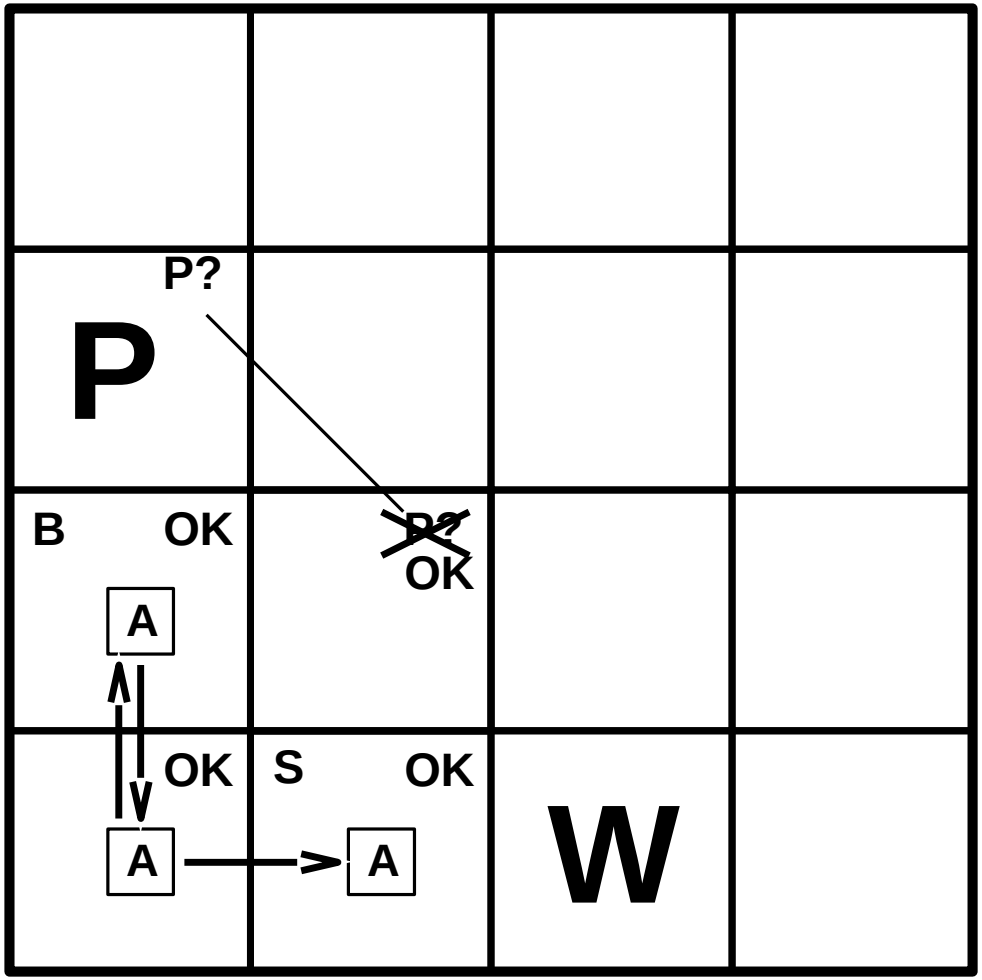
Wumpus世界探索


4
3
2
1
Stench
Breeze
PIT
Stench
Breeze
Gold
PIT
Breeze
Stench
Breeze
START
Breeze
PIT
Breeze
1
2
3
4

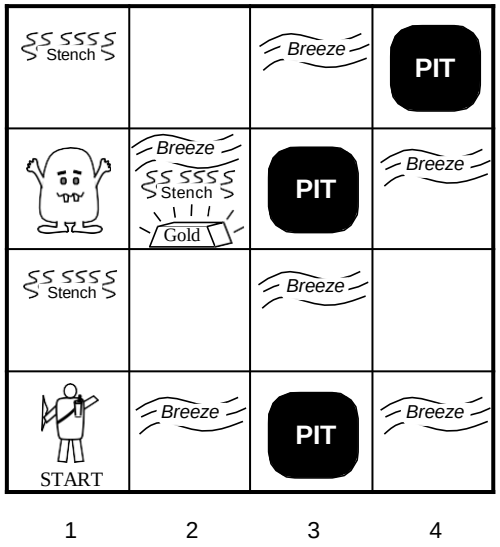

P?
B
OK
P?
A
OK
S
OK
A
A

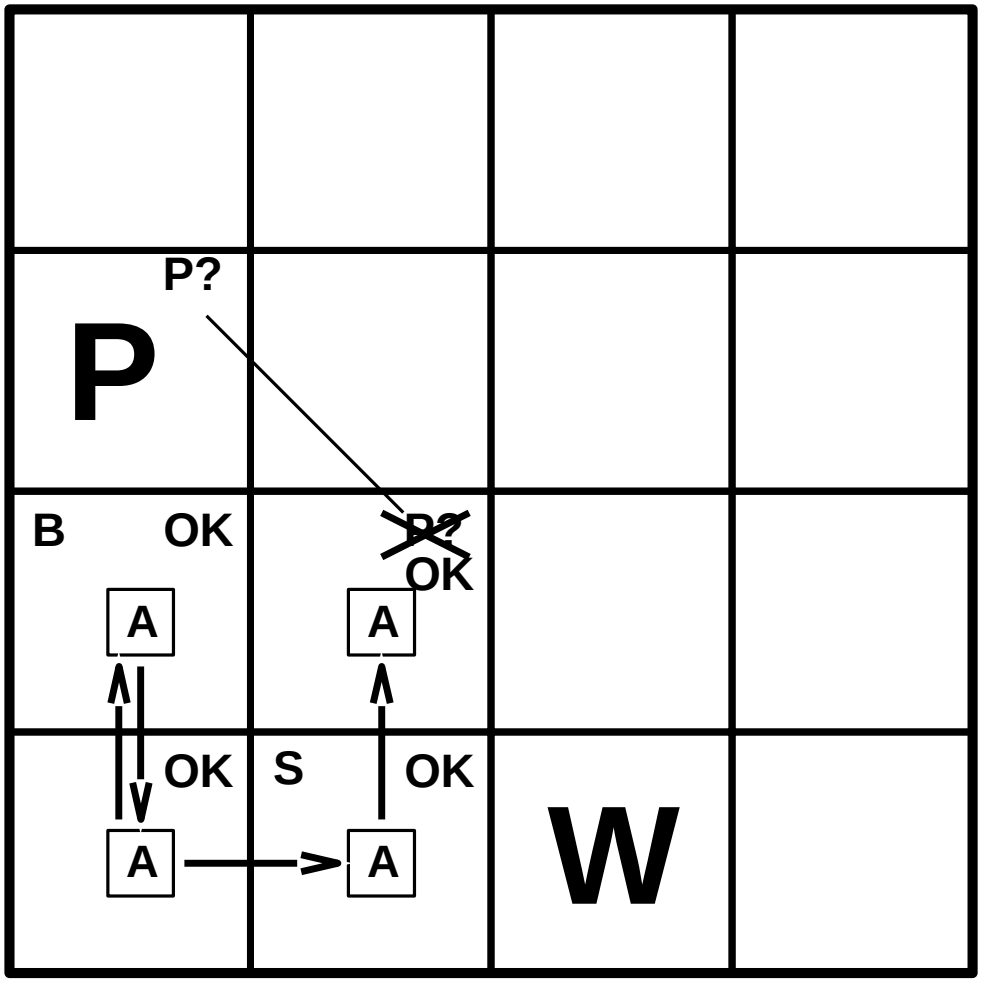
Wumpus世界探索


4
3
2
1
Stench
Breeze
PIT
Stench
Breeze
Gold
PIT
Breeze
Stench
Breeze
START
Breeze
PIT
Breeze
1
2
3
4

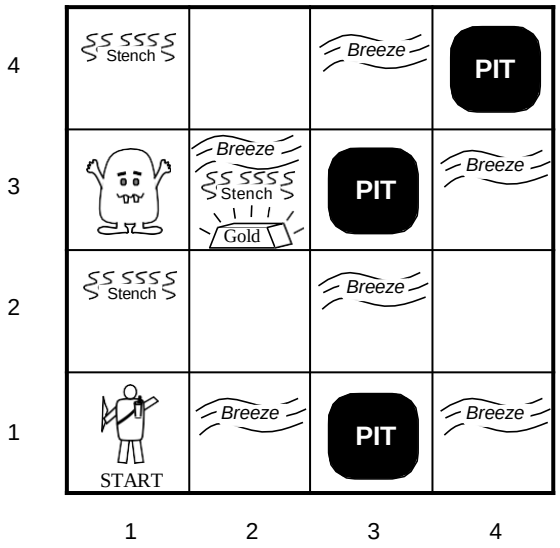

P?
P
B
OK
P?
OK
A
OK
S
OK
A
A
W

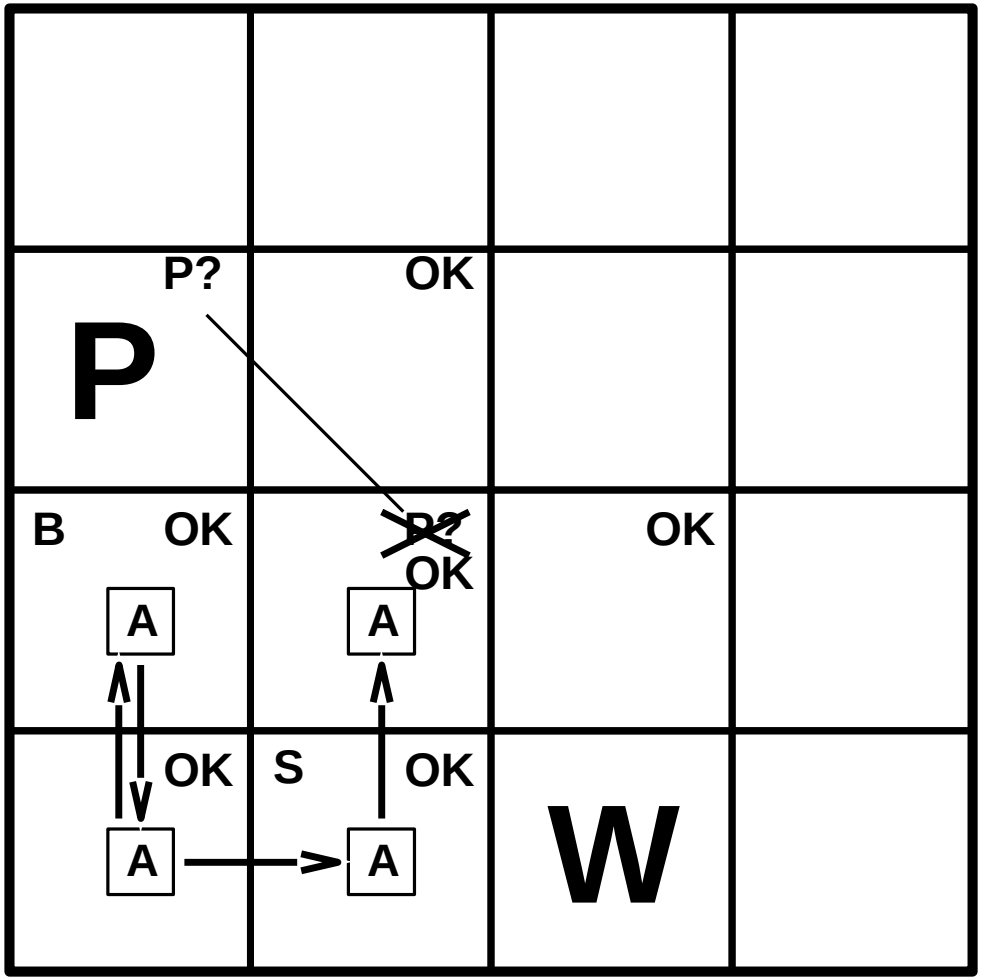
Wumpus世界探索


4
3
2
1
Stench
Breeze
PIT
Stench
Breeze
Gold
PIT
Breeze
Stench
Breeze
START
Breeze
PIT
Breeze
1
2
3
4

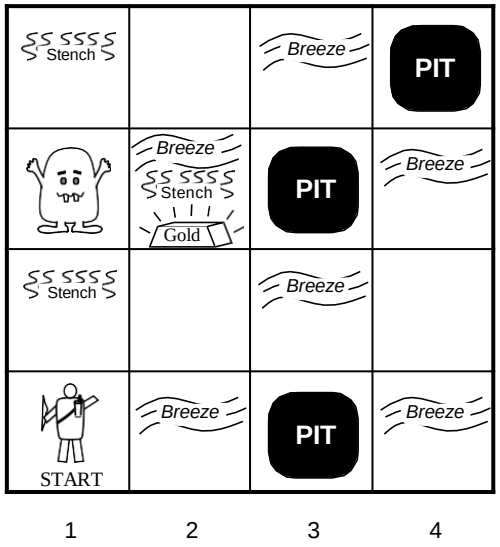

P?
P
B
OK
P?
OK
A
A
OK
S
OK
A
A
W

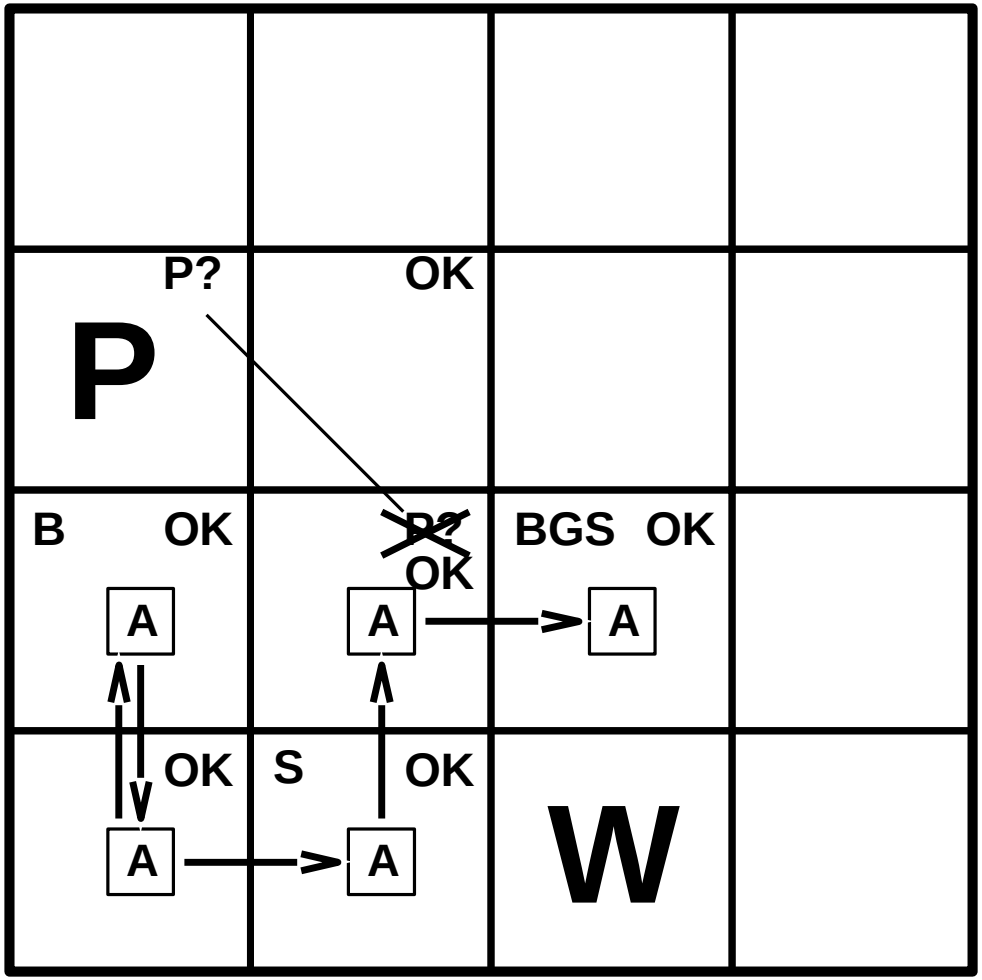
Wumpus世界探索


4
3
2
1
Stench
Breeze
PIT
Breeze
Stench
Gold
Breeze
Stench
Breeze
Breeze
PIT
Breeze
START
1
2
3
4


P?
OK
P
B
OK
P?
OK
OK
A
A
OK
S
OK
A
A
W

Wumpus世界探索


4
3
2
1
Stench
Breeze
PIT
Gold
START
1
2
3
4


P?
OK
P
B
OK
P?
OK
BGS
OK
A
A
A
OK
S
OK
A
A
W

逻辑学

- 逻辑学是表示信息的正式语言
  - 语法定义了语言中的句子
  - 语义定义了句子的“意义”
  - 也就是说，定义了在一个“世界”中的真理
- 例如，算术语言
  - $x+2 \geq y$ 是一个句子
  - $x+y >$ 不是一个句子
- x+2只有在x+2不小于y的时候成立
- x+2≥y在 x=7, y=1 的世界中为真
- x+2≥y在 x=0, y=6 的世界中为假

知识库蕴含一个语句

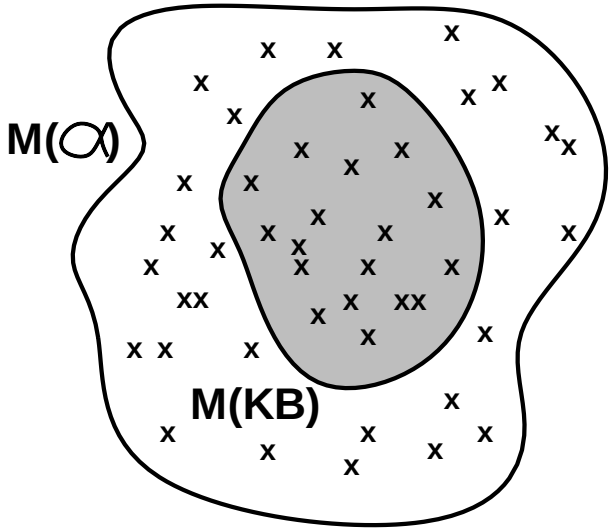
- 定义一个知识库蕴含某个语句表述如下：

$$KB \models \alpha$$

- 知识库 KB 蕴含一个语句 α 当且仅当 α 在所有知识库 KB 为真的“世界”中为真
- 例：
  - x+y=4蕴含4=x+y
  - 如果知识库KB中有两个事实：“河南建业获胜”、“广州恒大获胜”，那么这个知识库KB就蕴含一个语句：“河南建业和广州恒大二者其一获胜”

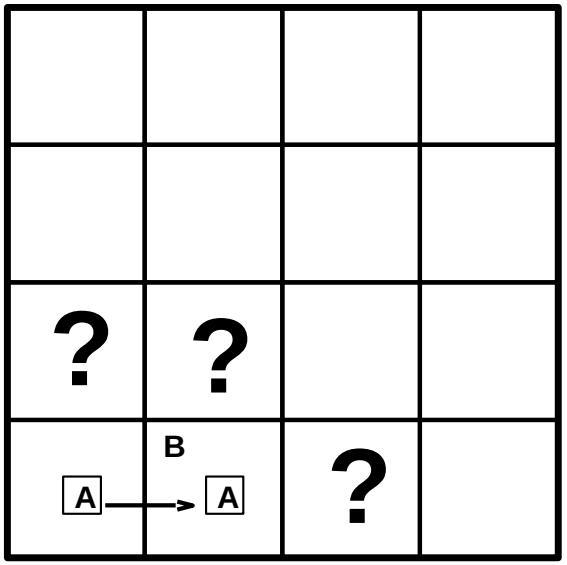
逻辑模型

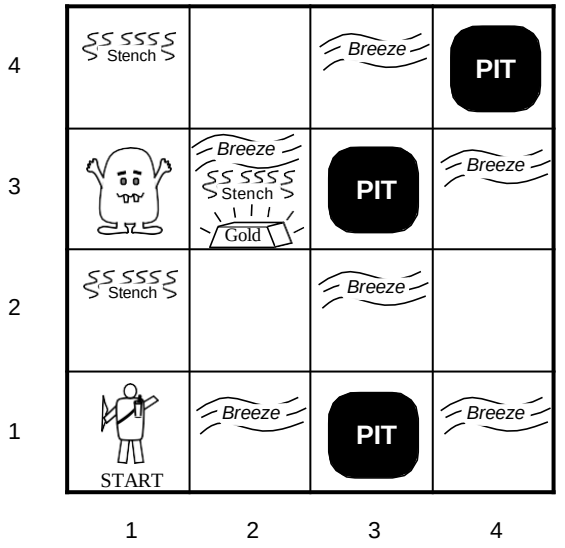
- 逻辑学通过建模（Model）对这样的“世界”进行描述
- 如果在模型m中语句α是真，那么称 m 是语句 α 的一个模型
- 称M(α)是α所有模型的集合
- 那么 $KB \models \alpha$ *当且仅当* $M(KB) \subseteq M(\alpha)$
- 比如：
  - $KB=$ “河南建业获胜”和“广州恒大获胜”
  - $\alpha=$ “河南建业获胜”



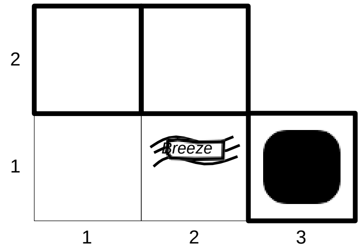
wumpus世界中的蕴含关系

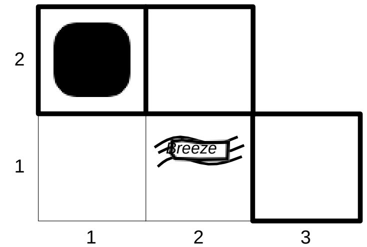
- 如果[1,1]中什么都没有，则右移，发现[2,1]中有微风
- 此时Agent有3中移动的方式（图中问号位置）
- 每一种移动方式Agent都有选和不选两种情况，那么所有选项组成的模型共有 $2^3=8$ 种

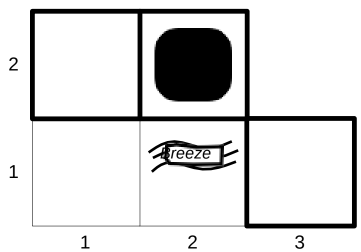


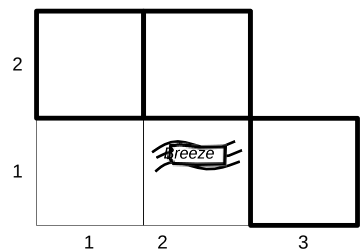


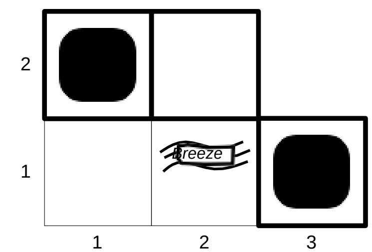
Wumpus世界的8个模型

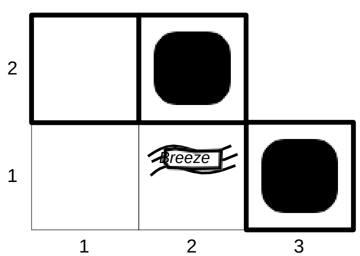


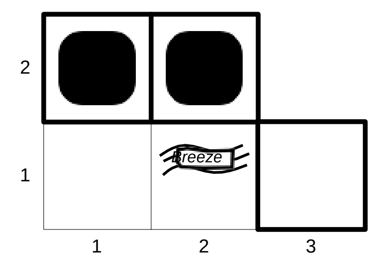


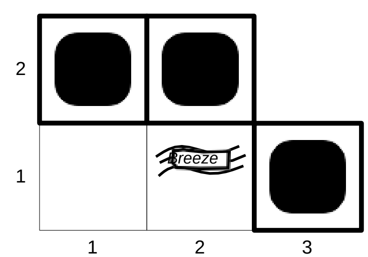




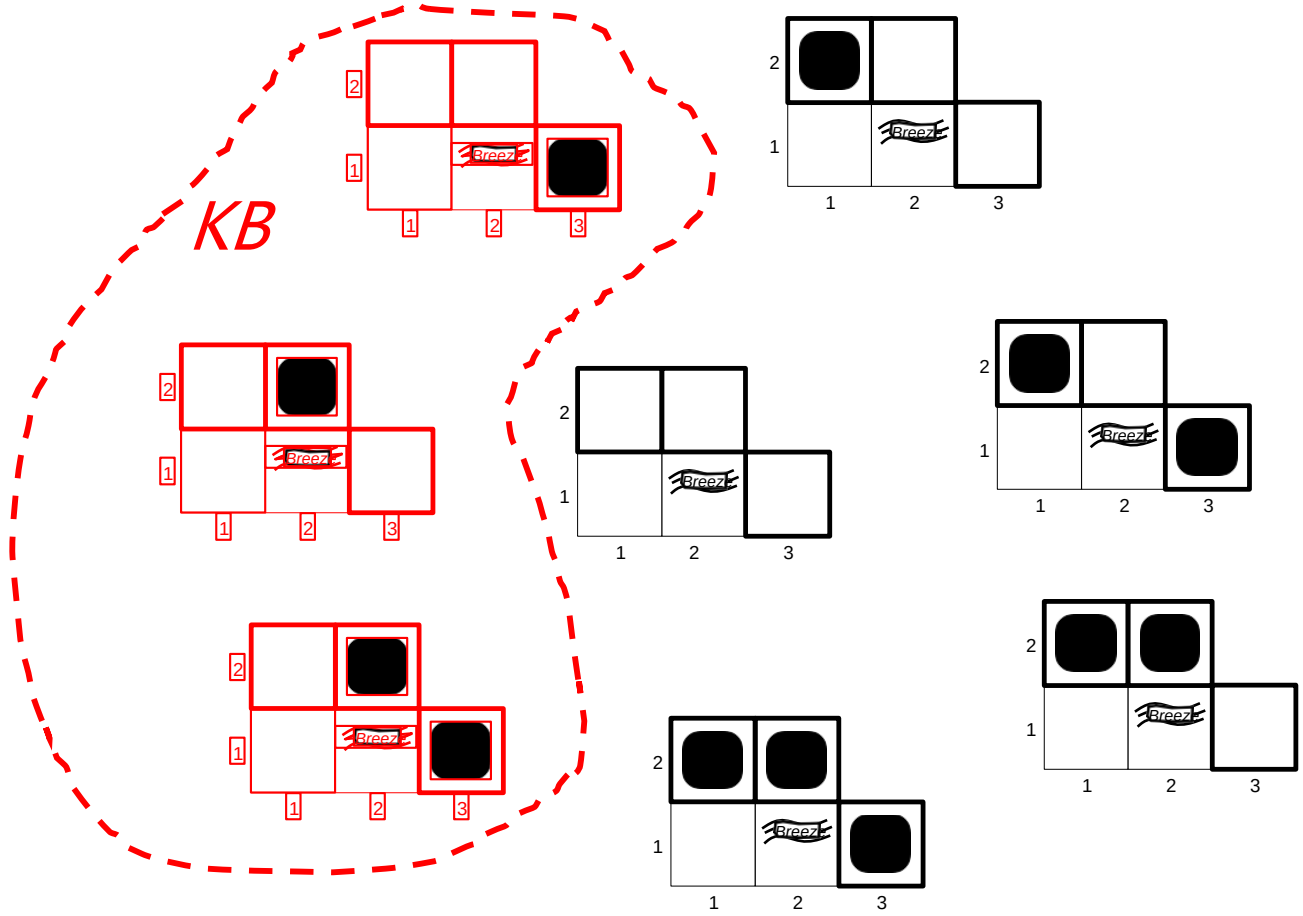






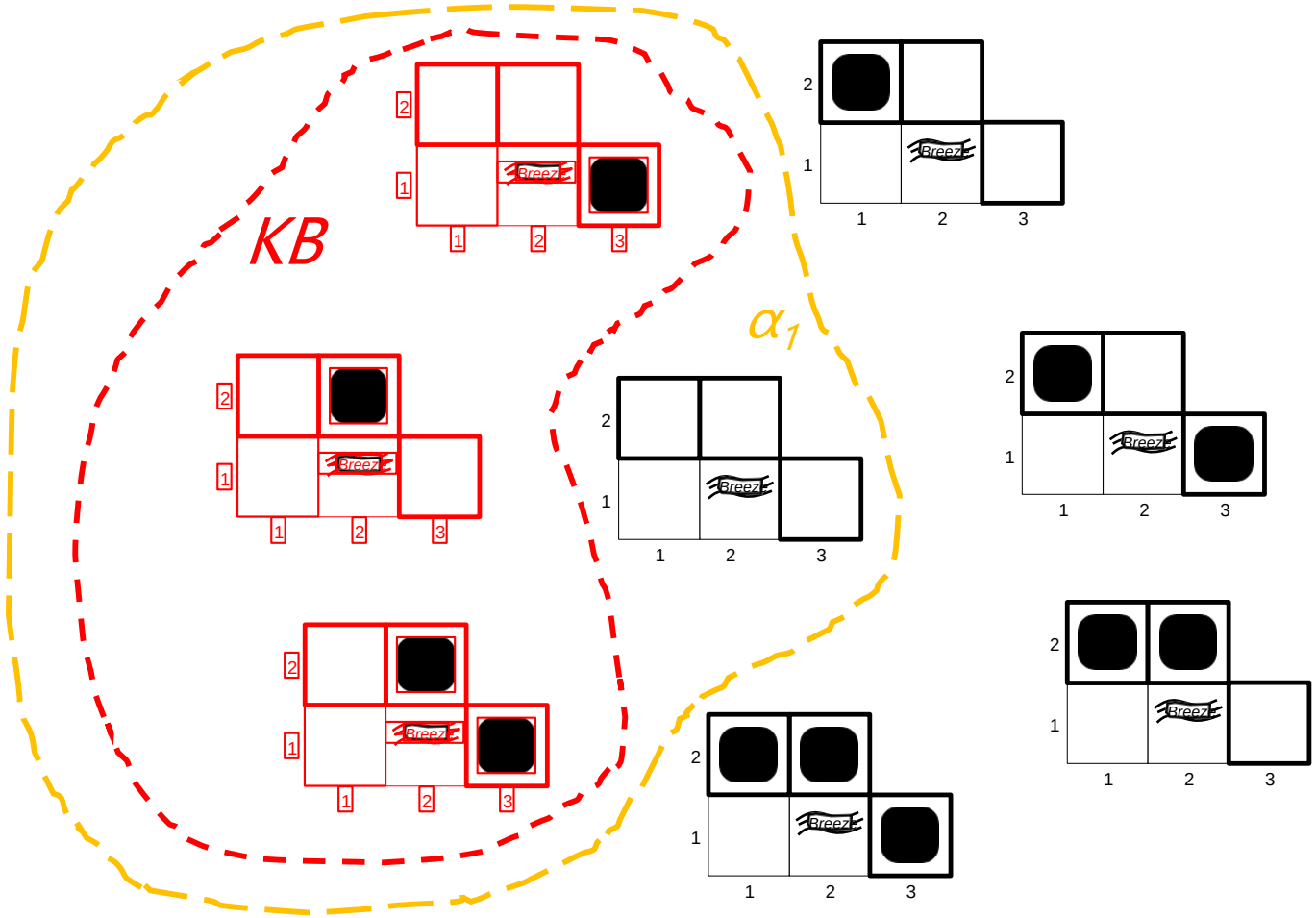


Wumpus世界的8个模型



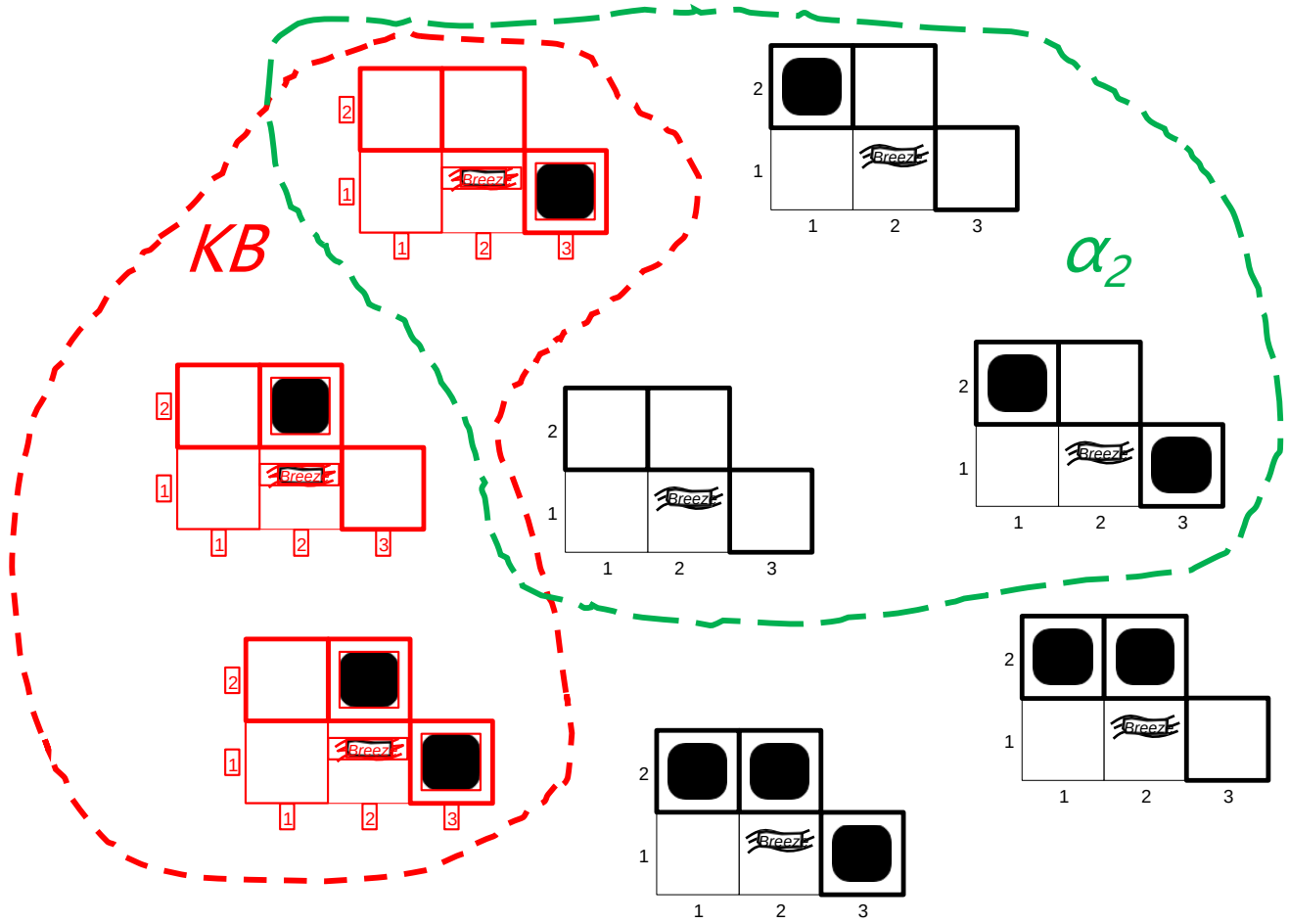
- *知识库KB=wumpus世界的规则+观察到的情况*

Wumpus世界的8个模型



- *知识库KB = wumpus世界的规则 + 观察到的情况*
- α_1 = “[1,2] 安全”, $KB \models \alpha_1$, 这一结论可以通过模型检查推出

Wumpus世界的8个模型



- *知识库KB = wumpus世界的规则 + 观察到的情况*
- α_2 = “[2,2] is safe”, 但不能推出 $KB \models \alpha_2$

推理Inference

- 断定符：$\vdash$（推出），满足符：$\models$（永真）
- $KB \vdash_i \alpha$ =语句 α 可以通过在 KB 上执行 i 过程得到
- 知识库就像一个大海，α是一根针。
- 蕴含过程就好比大海中有一根针，推理就像大海捞针
- 推理 i 是健全(soundness)的意味着：
  - 对任意一个$KB \vdash_i \alpha$, 可以得到 $KB \models \alpha$
- 推理 i 完整(Completeness)意味着：
  - 对任意一个$KB \models \alpha$,可以得到 $KB \vdash_i \alpha$
- 我们的目标是定义一个逻辑(一阶逻辑)，它具有足够的表达能力，可以表达几乎所有感兴趣的内容，并且有一个完善的推理过程。
- 也就是说，这样的过程可以回答任何问题，而这些问题的答案来自于知识库中已知的内容。

逻辑证明

- 证明方法大致分为两种:
- 1、应用推理规则
  - 从旧句中合法生成新句
  - 证明=推理规则应用的序列
  - 可以在标准搜索算法中使用推理规则作为操作符
  - 通常需要将句子翻译成标准形式
- 2、模型检查
  - 真值表枚举(总是以n为指数)
  - 改进的回溯
  - 在模型空间中的启发式搜索
    - 最小化冲突比如爬山算法

前向和反向链接

- Horn子句
  - 命题符号
  - 结合符号
  - 如$C \wedge (B \Rightarrow A) \wedge (C \wedge D \Rightarrow B)$
- Horn范式
  - KB=Horn子句的连接

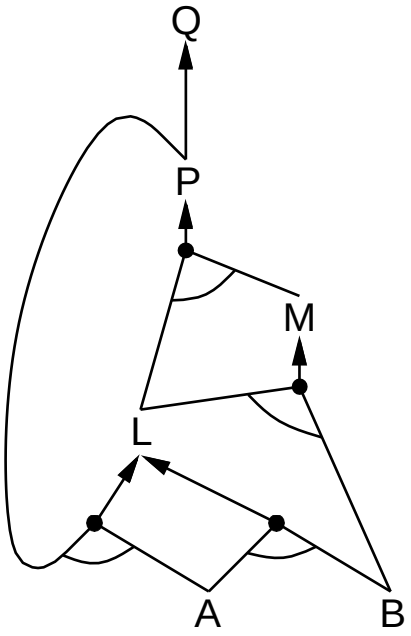
$$\frac{\alpha_1, \ldots, \alpha_n, \alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow \beta}{\beta}$$

- Horn范式能够使用前向和反向链接求解
- 其算法思路简洁，能够在线性时间内计算完成

前向链接

- 思路：触发知识库中满足其前提的任何规则，将其结论添加到知识库中，直到找到查询为止

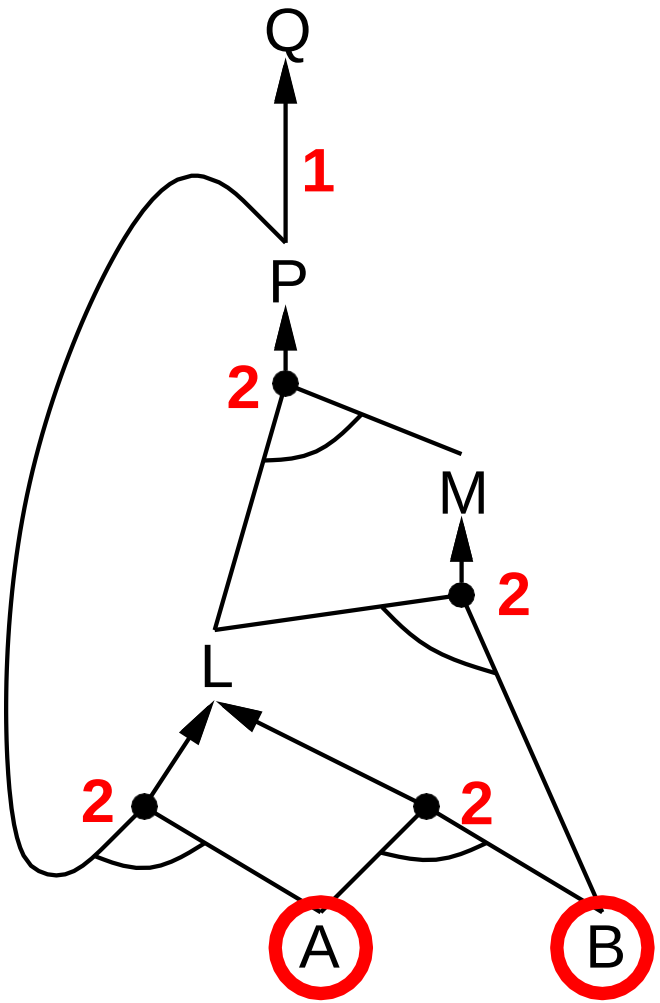
$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$A$$
$$B$$



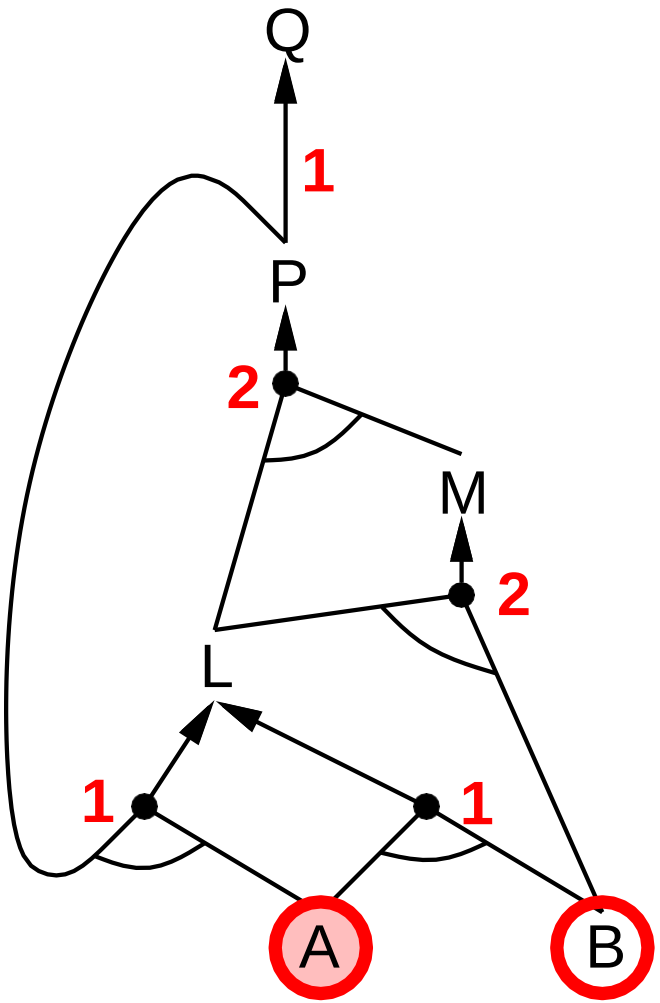
前向链接算法

```
function PL-FC-Entails?(KB, q) returns true or false
    inputs: KB, the knowledge base, a set of propositional Horn clauses
            q, the query, a proposition symbol
    local variables: count, a table, indexed by clause, initially the number of premises
                     inferred, a table, indexed by symbol, each entry initially false
                     agenda, a list of symbols, initially the symbols known in KB

    while agenda is not empty do   p ←
    Pop(agenda)  unless inferred[p] do
            inferred[p] ← true
            for each Horn clause c in whose premise p appears do
                decrement count[c]
                if count[c] = 0 then  do
                    if Head[c] = q then return true
                    Push(Head[c], agenda)
    return false
```

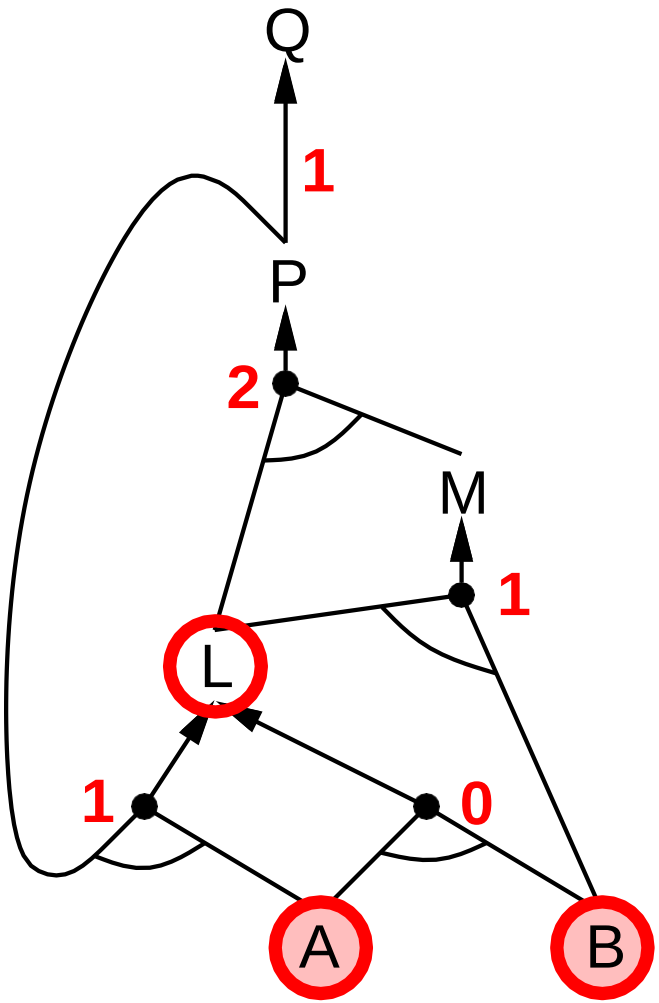
前向链接过程



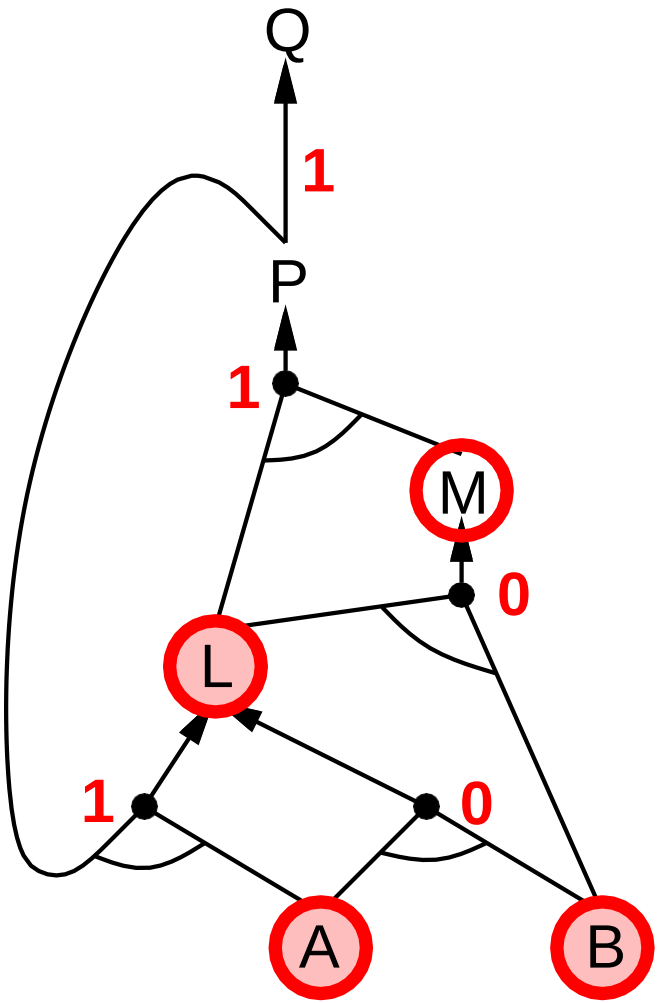
前向链接过程



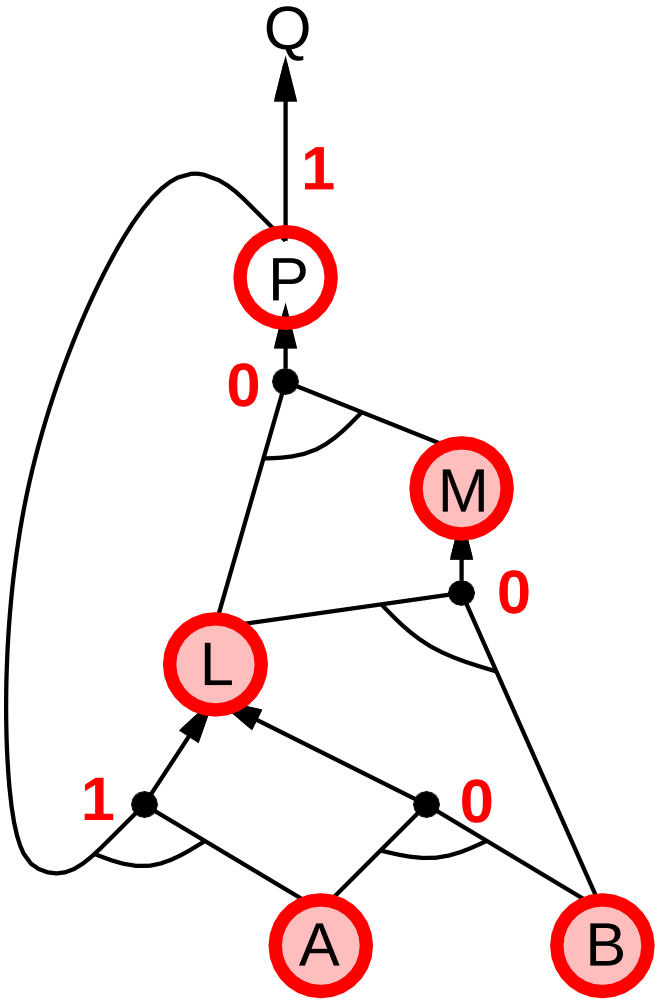
前向链接过程



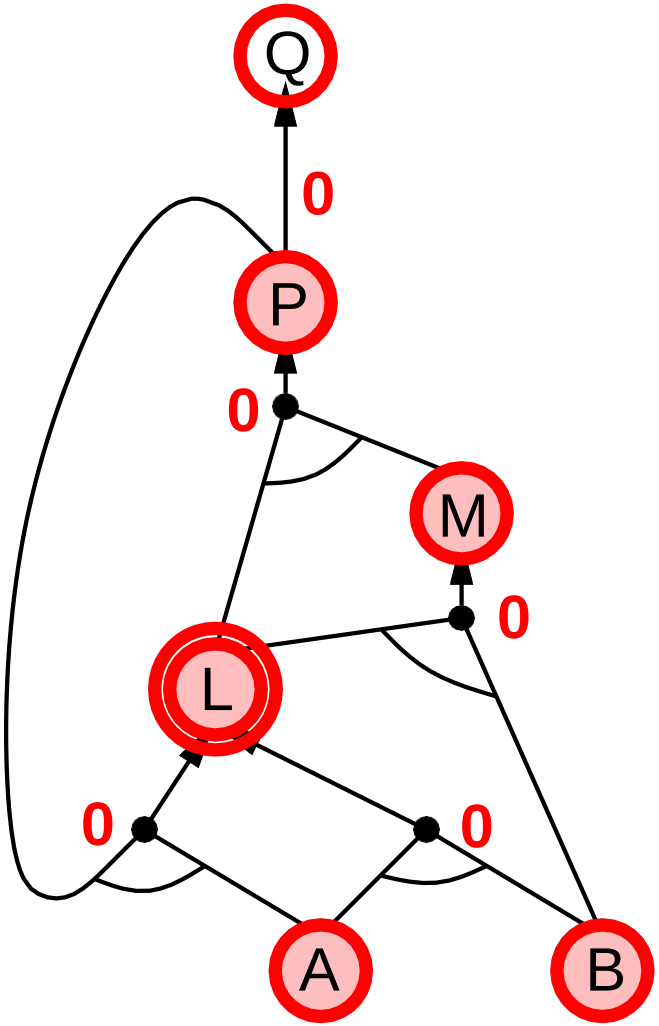
前向链接过程



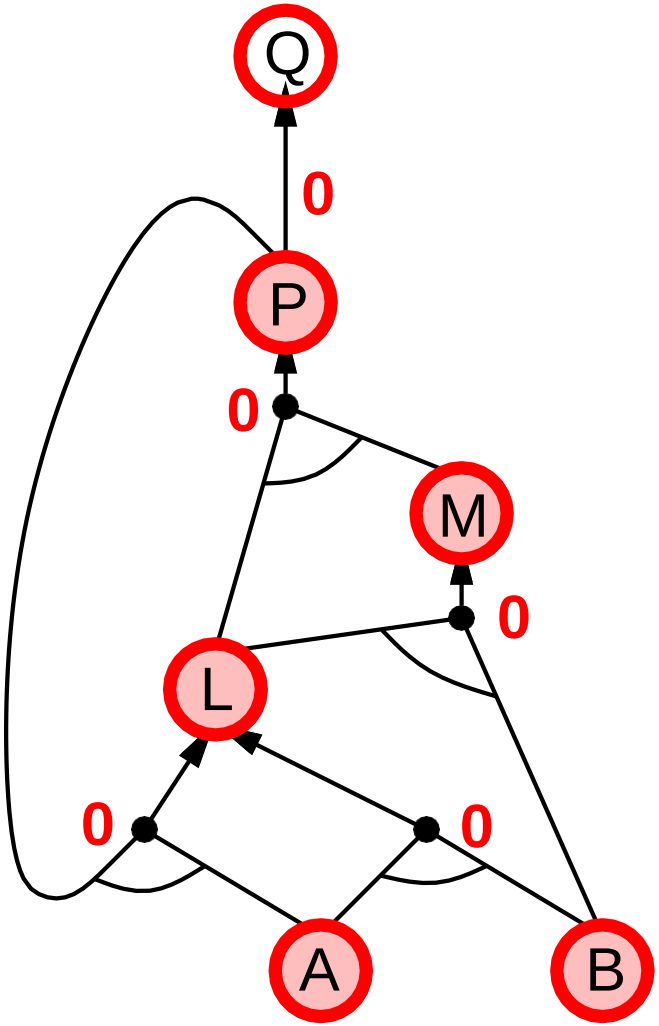
前向链接过程



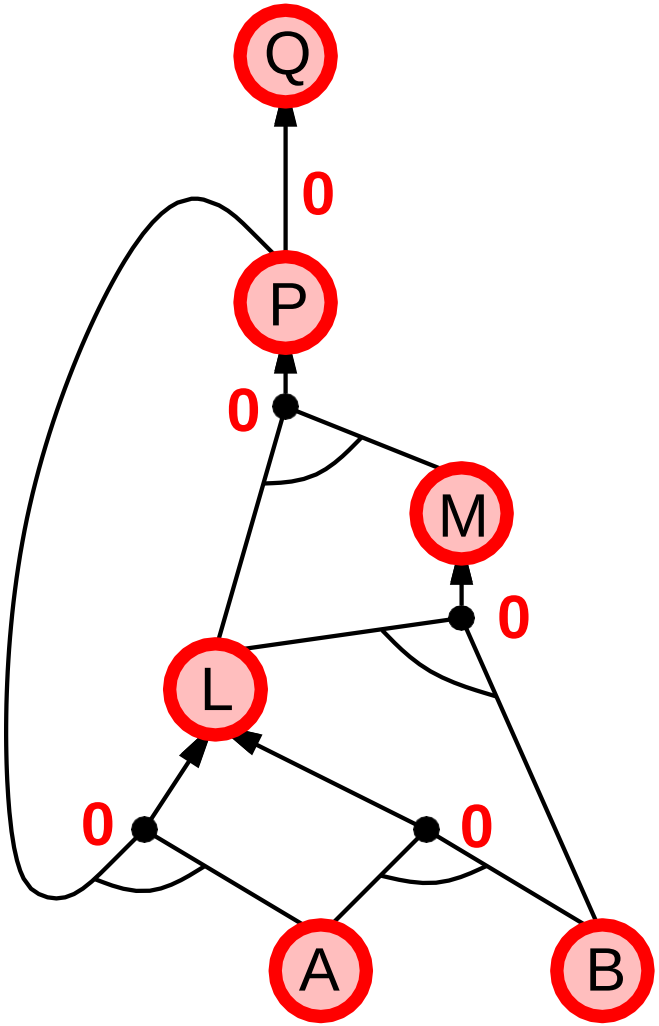
前向链接过程



前向链接过程



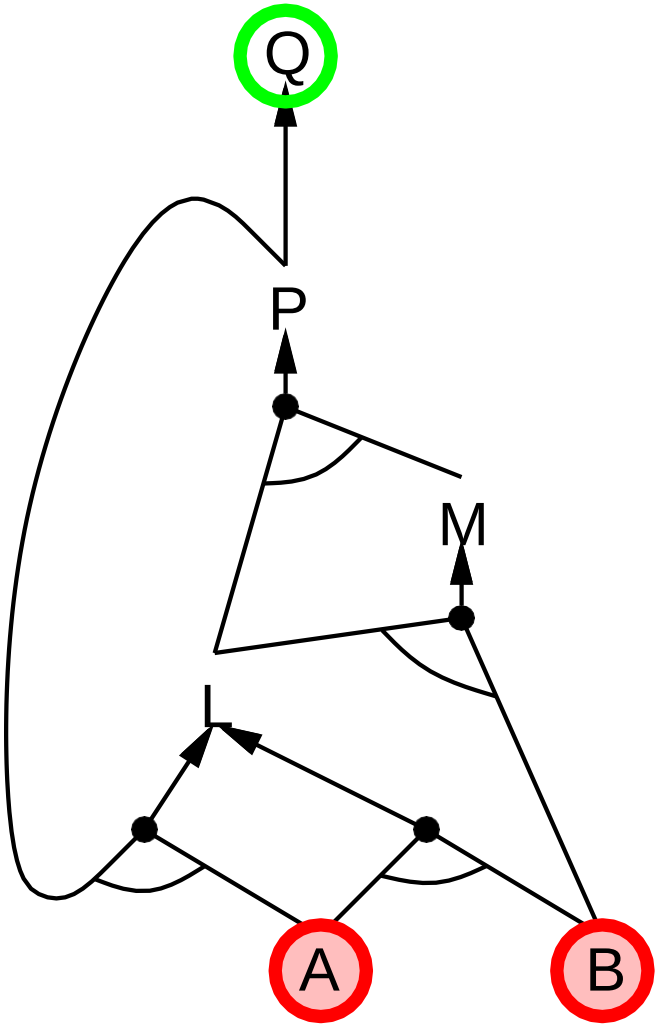
前向链接过程



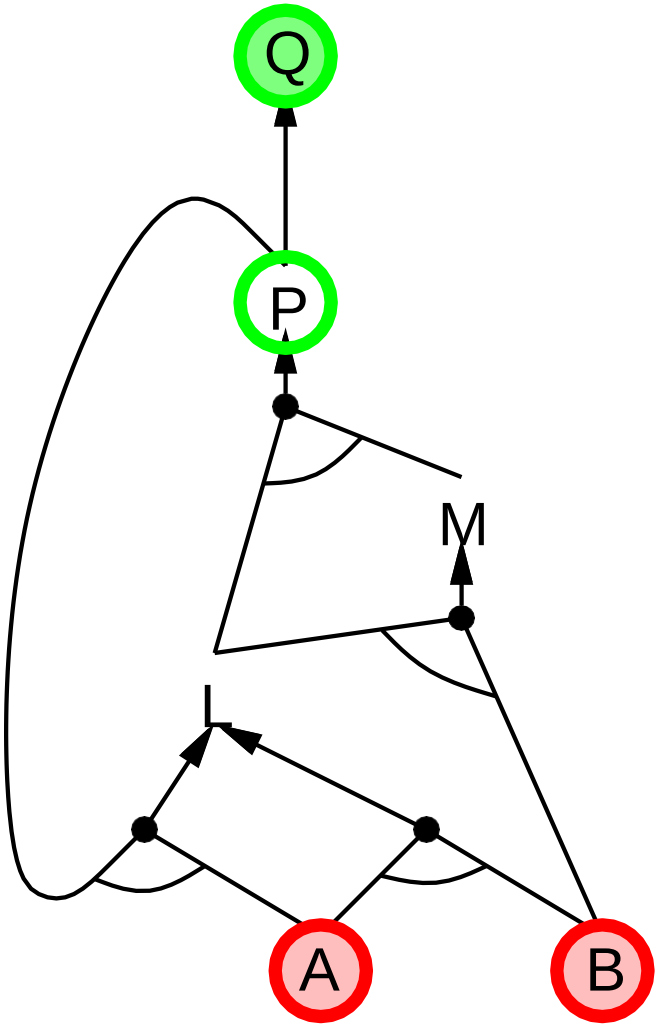
反向链接过程

- 思路：从查询q逆向推理：
  - 用P证明q，
  - 检查P是否已知，或者
  - 用其它证明一些规则结论P的所有前提
- 避免循环：检查新的子目标是否已经在目标栈上
- 避免重复工作：检查是否有新的子目标
  - 已经被证明是真的，还是
  - 已经失败了

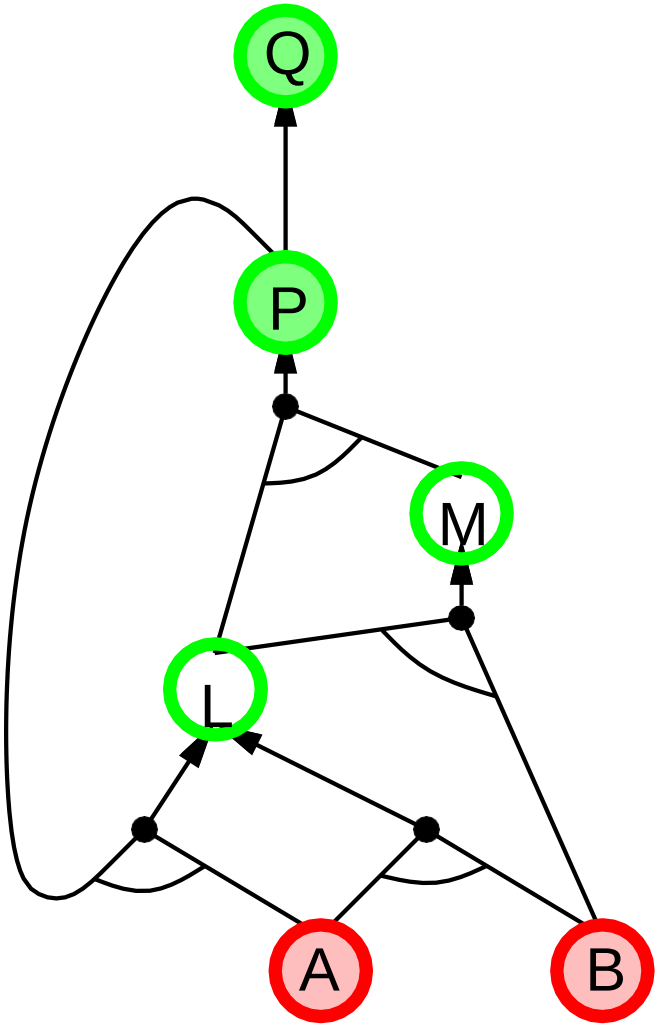
反向链接过程



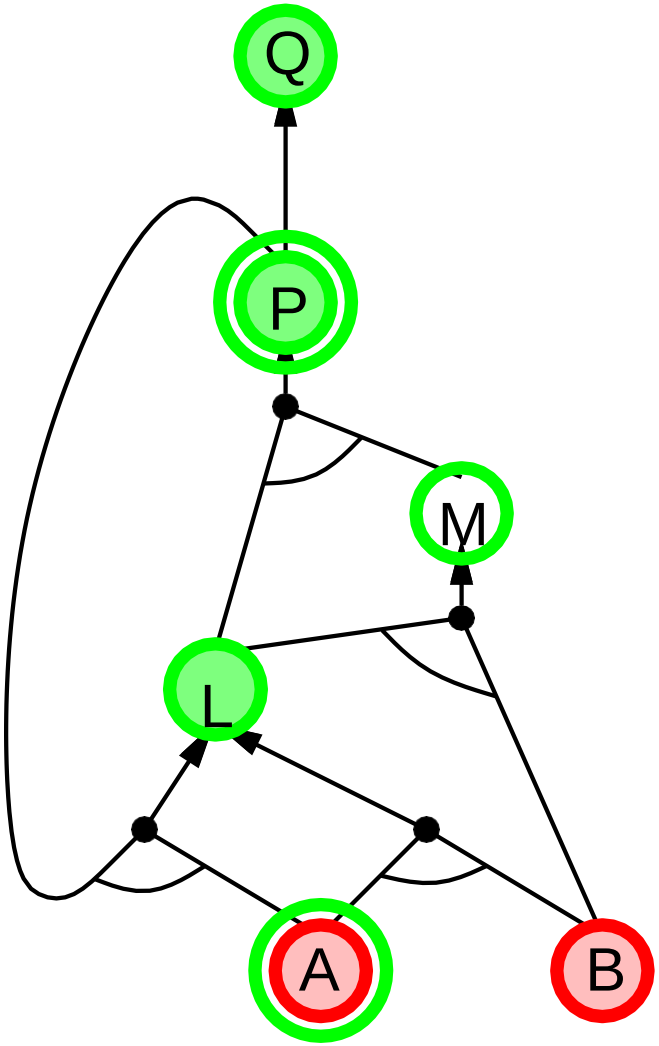
反向链接过程



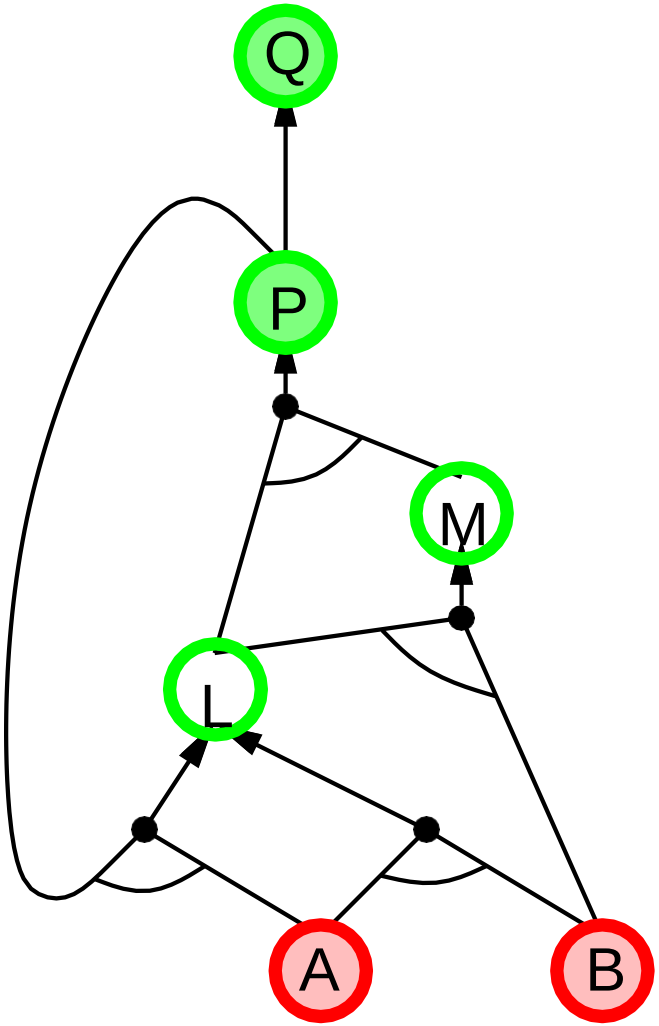
反向链接过程



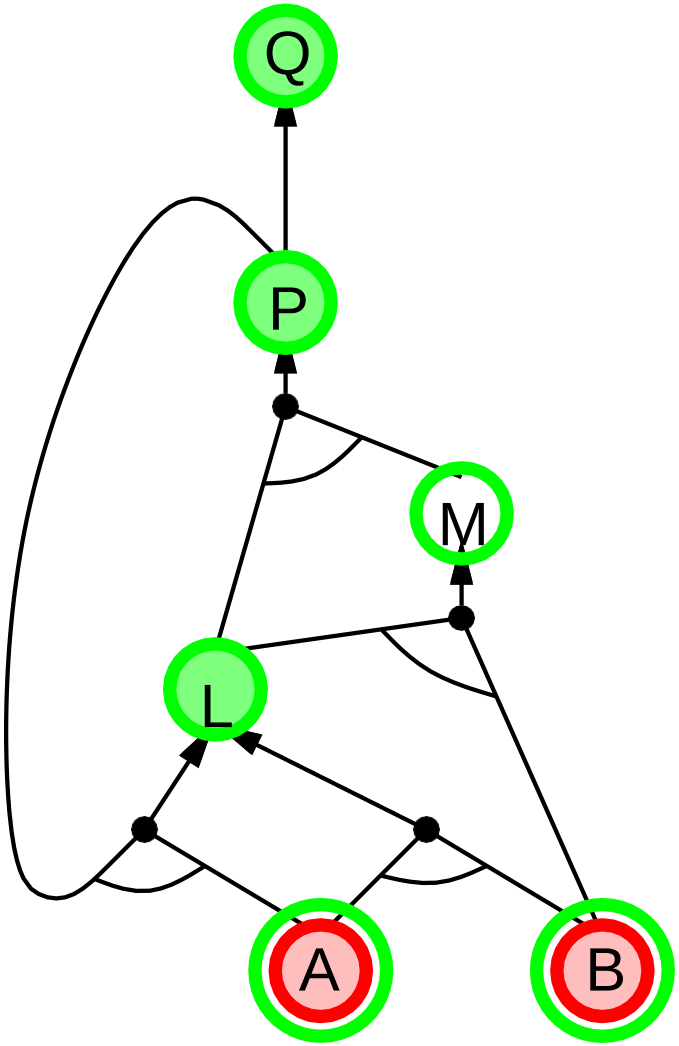
反向链接过程



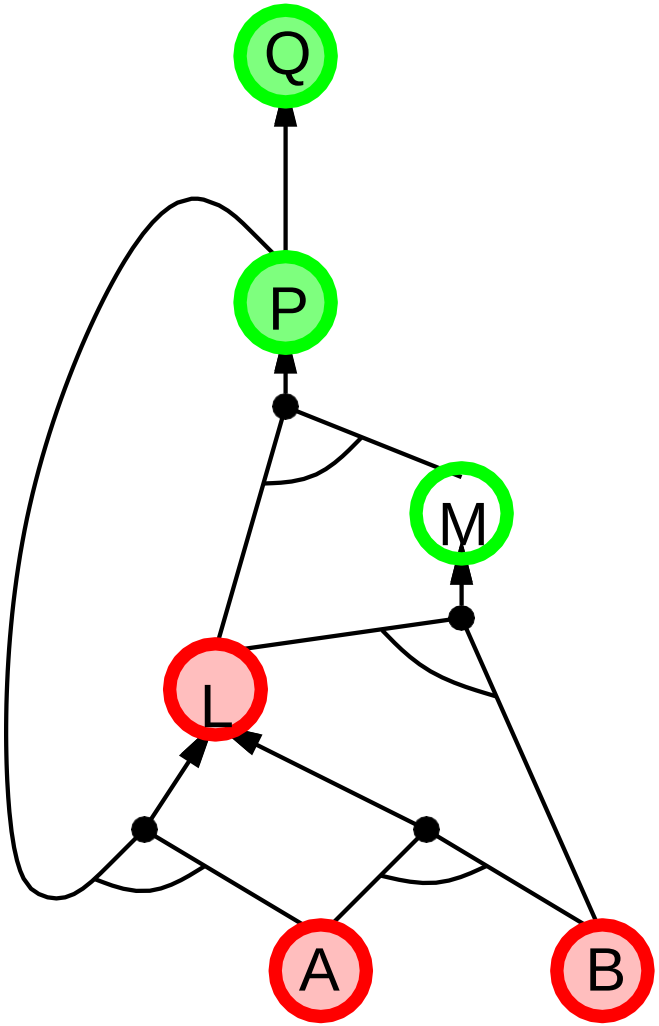
反向链接过程



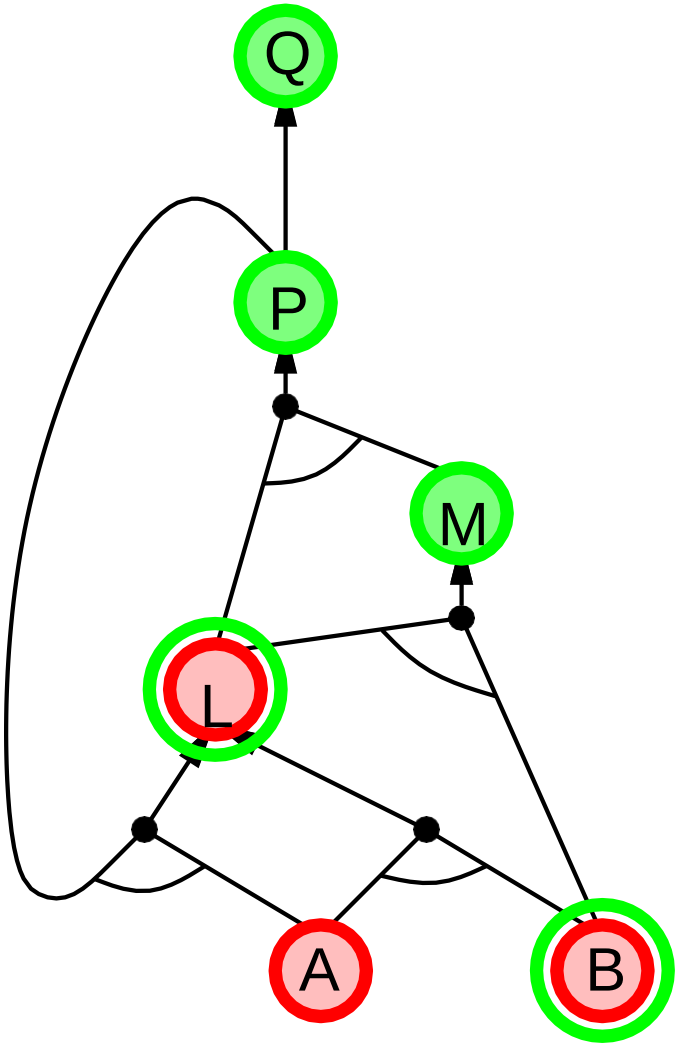
反向链接过程



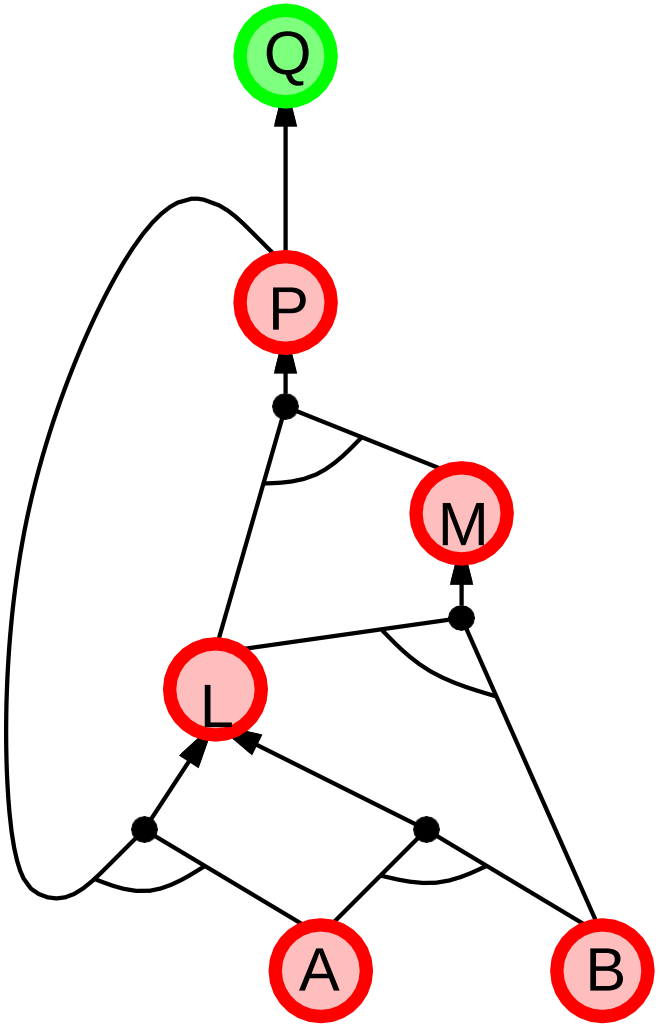
反向链接过程



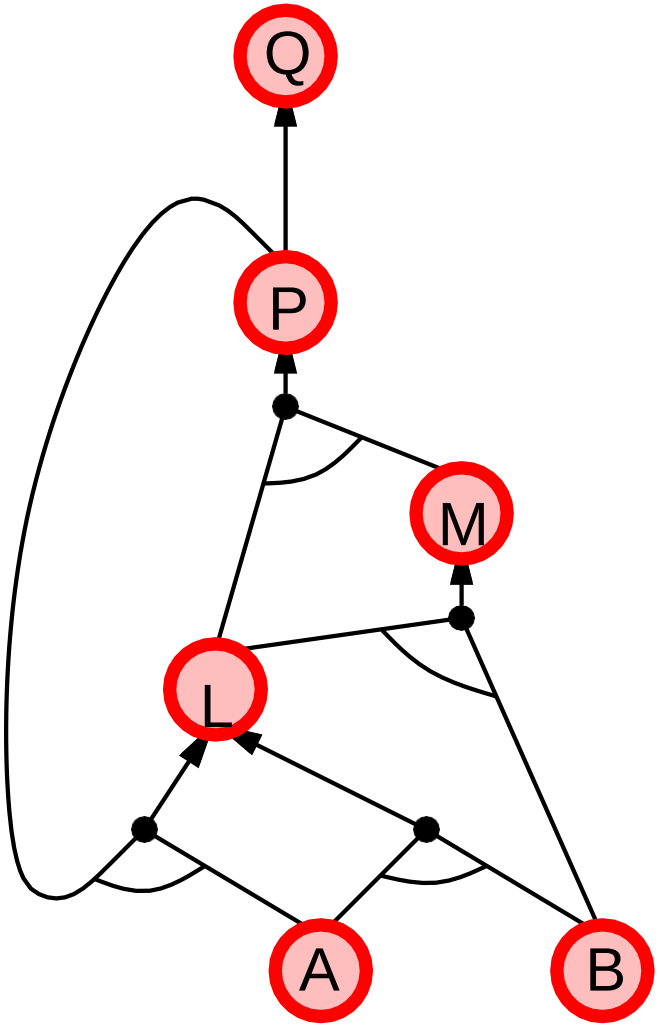
反向链接过程



反向链接过程



反向链接过程



前向和反向链接对比

- FC是数据驱动的，自动的，无意识的处理。
  - 目标识别
  - 日常决策
- 可能做了很多与目标无关的工作

- BC以目标为导向，适合解决问题，
  - 我的钥匙在哪儿?
  - 我怎样才能拿到毕业证?
- BC的复杂度在KB大小上远小于线性复杂度

五种不同的逻辑

形式语言	本体论约定	认识论约定
命题逻辑	事实	真/假/未知
一阶逻辑	事实、对象、关系	真/假/未知
时序逻辑	事实、对象、关系、时间	真/假/未知
概率论	事实	可信度
模糊逻辑	事实具有真实度	已知区间值

逻辑符号

类别	符号	含义
连接词	¬	非
	∧	与
	∨	或
	⇒	蕴含
	⇔	当且仅当
	⊨	导出
	⊭	
限量词	∀	所有
	∃	存在
等量词	=	等于

命题逻辑与一阶逻辑

- 命题逻辑：
  - 亦被称为命题演算
  - 使用逻辑连接词，用于处理简单的陈述性命题。
- 一阶逻辑:
  - 亦被称为一阶谓词演算，
  - 此外，还使用限量词、等量词、以及谓词（通常与集合相关联）。

用BNF表述的命题逻辑语法

Sentence		*AtomicSentence* | *ComplexSentence*
AtomicSentence		*True* | *False* | *P* | *Q* | *R* | ...
ComplexSentence		< *Sentence* > | [*Sentence*]
	|	$\neg$ *Sentence*
	|	*Sentence* $\wedge$ *Sentence*
	|	*Sentence* $\vee$ *Sentence*
	|	*Sentence* $\Rightarrow$ *Sentence*
	|	*Sentence* $\Leftrightarrow$ *Sentence*
OPERATOR PRECEDENCE	:	$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

用BNF表述的一阶逻辑的语法

Sentence	→	*AtomicSentence* | *ComplexSentence*
AtomicSentence	→	*Predicate* | *Predicate*(*Term*, ...) | *Term* = *Term*
ComplexSentence	→	< *Sentence* > | [*Sentence*] | ¬ *Sentence* | *Sentence* ∧ *Sentence*
	|	*Sentence* ∨ *Sentence* | *Sentence* ⇒ *Sentence* | *Sentence* ⇔ *Sentence*
	|	*Quantifier Variable*, ... *Sentence*
Term	→	*Function*(*Term*, …) | *Constant* | *Variable*
Quantifier	→	∀ | ∃
Constant	→	*A* | X_1 | *John* | ...
Variable	→	*a* | *x* | *s* | ...
Predicate	→	*True* | *False* | *After* | *Loves* | *Raining* | ...
Function	→	*Mother* | *LeftLeg* | ...
OPERATOR PRECEDENCE	:	¬, =, ∧, ∨, ⇒, ⇔

一阶逻辑的形式规则

- 该形式规则定义
  - 项
  - 公式
- 该形式规则可以用于书写项和公式的形式文法。
- 形式规则通常是上下文无关的，即
- 每个产生式左侧有一个单一的符号。

一阶逻辑的形式规则：项

- 规则1：变量
  - 任何变量都是一个项
- 规则2：常数
  - 任何常数也都是一个项
- 规则3：函数
  - 任何n个参数的表达式f(t1,...,tn)都是一个项，其中每个参数ti是一个项，并且f是一个价n的函数符号。 尤其是，表示个体常量的符号是0元函数符号，因此也是一个项。

一阶逻辑的形式规则：公式

- 谓词符号：若P是一个n元谓词符号并且$t_1,...t_n$是项，则 $P(t_1,...t_n)$是一个公式。
- 等量：若等量符号被认为是逻辑的一部分，并且 t_1和 t_2是项，则 $t_1=t_2$ 是一个公式。
- 否定：若φ是一个公式，则¬φ是一个公式。
- 二元连接：若φ和ψ是公式，则(φ=>ψ)是一个公式。类似的规则可用于其他二元逻辑连接。
- 限量：若φ是一个公式并且x是一个变量，则 $\forall x\varphi$ 和 $\exists x\varphi$ 是公式。

用一阶逻辑描述魔兽世界

- 感知语句
  - $Percept([Stench, Breeze, Glitter, None, None], 5)$
  - $\forall t, s, g, m, c \quad Percept([s, Breeze, g, m, c], t) \Rightarrow Breeze(t)$
  - $\forall t, s, b, m, c \quad Percept([s, b, Glitter, m, c], t) \Rightarrow Glitter(t)$

- 动作语句
  - $Turn(Right)$, $Turn(Left)$, $Forward$, $Shoot$, $Grab$, $Climb$.

- 查询语句
  - $\text{ASKVARS}(\exists a, BestAction(a, 5))$

Prolog语言

- Prolog语言起源于一阶逻辑。
- Prolog是一种通用的逻辑编程语言，已经被用于定理证明、专家系统、自然语言处理，等等。
- 不同于其它编程语言， Prolog是陈述性的：程序逻辑由关系来表达，表示为事实与规则。

```
likes(bill, car).
animal(X) :- cat(X).
bird(X) :- animal(X), has(X, feather).
```

本体工程Ontology

- 本体论是关于生物、生成、存在或现实的本质、以及生物及其关系的基本类别的哲学研究。
- 一个本体是一种对若干实体的类型、特性和相互关系的形式化命名和定义，它真实的、或根本地存在于一个特定范围的论域。
- 一个本体提供了一个领域的公共词汇，并且用不同层次的形式定义一些术语的含义和它们之间的关系。

什么是本体

- 某些领域创建本体来组织信息，然后再将这些本体用于问题求解，包括：
  - 人工智能
  - 语义Web
  - 系统工程
  - 软件工程
  - 生物医学信息学
  - 图书馆学
  - 信息架构

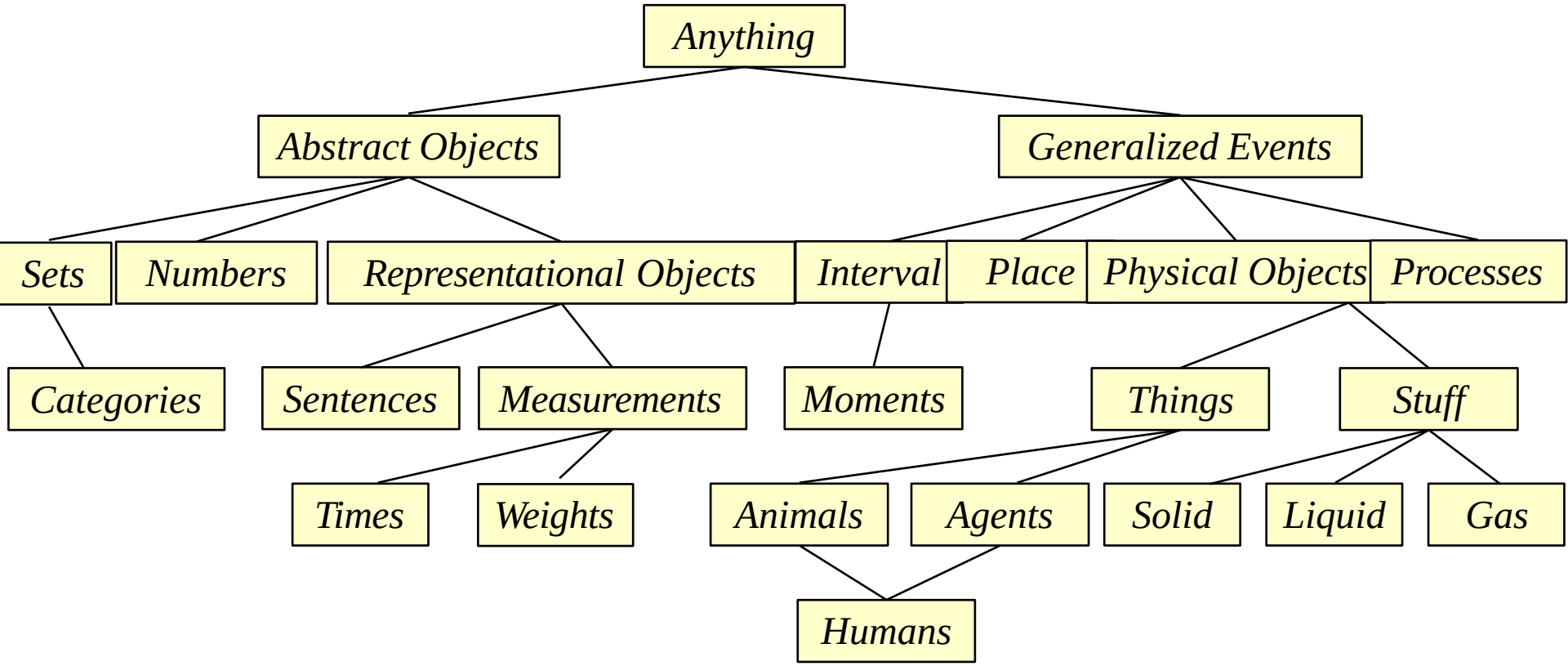
本体的应用

- 知识管理
- 自然语言处理
- 电子商务
- 智能信息集成
- 生物信息学
- 教育
- 语义Web

本体的类型

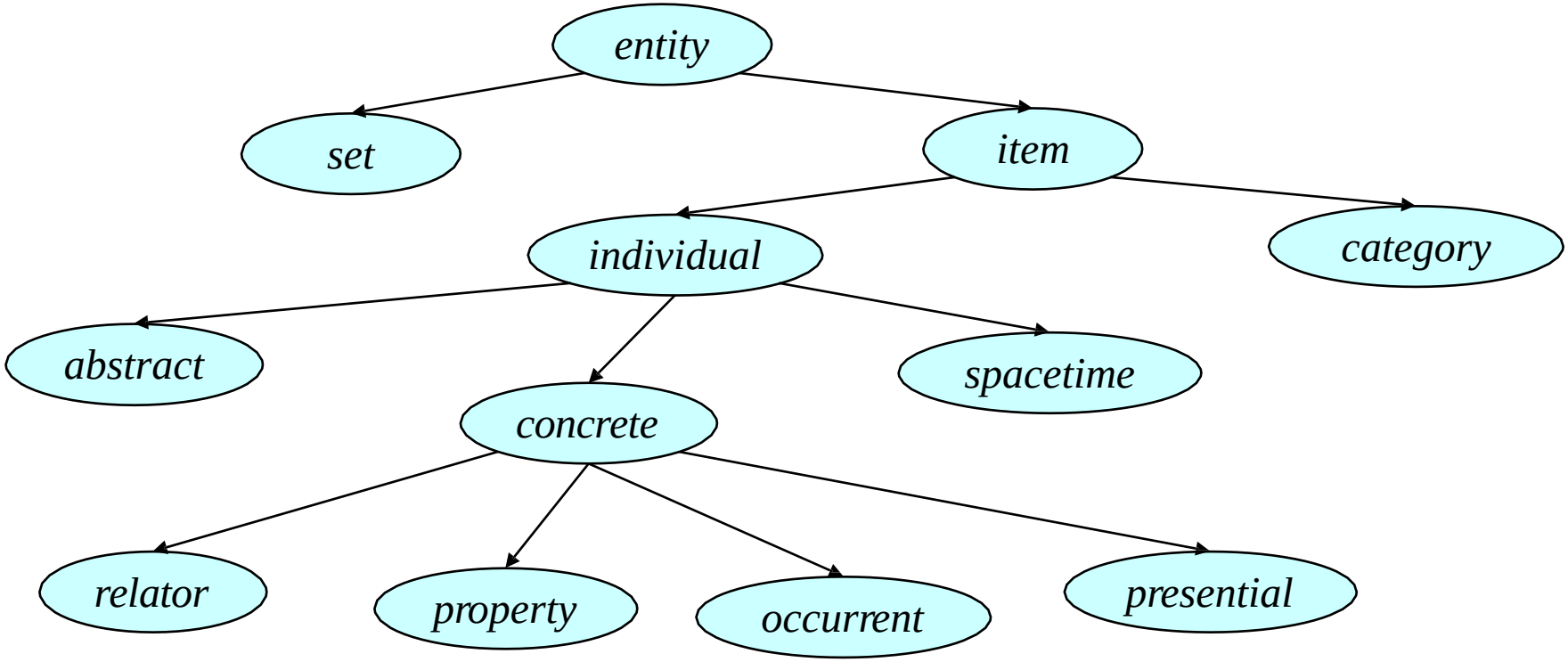
类型	概念
上层本体	一种公共对象的模型，通常可应用于广泛的领域本体。
领域本体	与一个特定的主题或兴趣领域有关，如：信息技术或科学的某个分支。
混合本体	一个上层本体与一个领域本体的结合。

世界的上层本体



- 底层概念是上层概念的一个特化

一个领域本体



- 一个本体用来表示一个领域中概念的集合，以及这些概念之间的关系。

本体的成分

成分	实例
个体	实例或对象
类	集、集合、概念、编程中的类、对象的类型、或者事物的种类
属性	对象（以及类）所能够具有的方面、属性、特性、特征、或者参数
关系	类和个体可以相互关联的方式
功能项	从某些关系所形成的复杂结构，可以用来替代某个陈述中的独立项。

本体的成分

成分	实例
限定	正式规定的必须为真的描述，为了使某些断言被接受为输入
规则	采用if-then（前因-后果）语句形式的陈述，描述从一个特定形式的断言得到的逻辑推理
公理	逻辑形式的断言（包括规则），共同构成本体在它的应用领域描述的全部理论
事件	属性或关系的改变

什么是本体工程

- 一个研究构建本体的方法和方法学的领域：
  - 一组某个领域概念的形式化表示，以及
  - 这些概念之间的关系。
- 要表示的一般概念，即：
  - 事件、
  - 时间、
  - 物理对象、以及
  - 置信度。
- 本体工程研究：
  - 本体开发过程，
  - 本体生命周期，
  - 构建本体的方法和方法学，
  - 支持本体的工具套件和语言的新领域。

两类本体语言

- 传统语法的本体语言
  - Common Logic
  - DOGMA (Developing Ontology-Grounded Methods and Applications)
  - F-Logic (Frame Logic)
  - KIF (Knowledge Interchange Format)
  - KM programming language
  - LOOM (ontology)
  - OCML (Operational Conceptual Modelling Language)
  - OKBC (Open Knowledge Base Connectivity)
- 标记式本体语言：使用标记方案对知识进行编码
  - XML
  - DAML+OIL
  - OIL (Ontology Inference Layer)
  - OWL (Web Ontology Language)
  - RDF (Resource Description Framework)
  - RDFS (RDF Schema)
  - SHOE

典型的本体语言

- Common logic
  - 已成为ISO 24707标准，是一套本体语言的规范，这些语言彼此之间可以被精确地转换。
- OWL (Web Ontology Language)
  - 是一种用于本体陈述的语言，目的是在Web上使用。
  - 所有的元素（类、特性和个体）都被定义为RDF (资源描述框架) 资源，并且通过URIs（统一 资源标识）加以识别。

RDF

```
<rdf:RDF
    xmlns ="http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#"
    xmlns:vin ="http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#"
    xml:base ="http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine#"
    xmlns:food="http://www.w3.org/TR/2004/REC-owl-guide-20040210/food#"
    xmlns:owl ="http://www.w3.org/2002/07/owl#"
    xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd ="http://www.w3.org/2001/XMLSchema#">
```

OWL

```
<owl:Ontology rdf:about="">
  <rdfs:comment>An example OWL ontology</rdfs:comment>
  <owl:priorVersion>
    <owl:Ontology rdf:about="http://www.w3.org/TR/2003/CR-owl-guide-20030818/wine"/>
  </owl:priorVersion>
  <owl:imports rdf:resource="http://www.w3.org/TR/2003/PR-owl-guide-20031209/food"/>
  <rdfs:comment>Derived from the DAML Wine ontology at
    http://ontolingua.stanford.edu/doc/chimaera/ontologies/wines.daml
    Substantially changed, in particular the Region based relations.
  </rdfs:comment>
  <rdfs:label>Wine Ontology</rdfs:label>
</owl:Ontology>
```

owl类

<rdf:Description rdf:about=“Professor” >
    <rdf:type rdf:resource=“owl:Class”/>
</rdf:Description>

- 这等价于

    <owl:Class rdf:about=“Professor”/>

- 这个声明的类实例化可用

    <Professor rdf:about=“张三”/>

Owl角色

- Owl属性又叫为角色，有两种不同类型的角色：抽象角色和具体角色。
  - 抽象角色用于连接个体与个体，具体角色用于连接个体与数据值。
- 角色的声明方式和类的声明方式类似：
- <owl:ObjectProperty rdf:about=“has Affiliation”/>抽象角色，表示人隶属的机构
- <owl:DatatypeProperty rdf:about=“firstName”/>具体角色，赋名

角色的定义域和值域

```
<owl:ObjectProperty rdf:about= “has Affiliation” >
    <rdfs:domain rdf:resource= “Person” />
    <rdfs:range rdf:resource= “Organisation” />
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:about= “firstName” >
    <rdfs:domain rdf:resource= “Person” />
    <rdfs:range rdf:resource= “xsd:string” />
</owl:DatatypeProperty>
```

类之间的关系

- 1.rdfs:subClassOf声明类的传递性
- 2.owl:disjointWith声明类不想交
- 3.owl:equivalentClass声明类等价
- 例:

```
<owl:Class rdf:about= “Professor” >
    <rdfs:subClassOf rdf:resource= “FacultyMember” />
</owl:Class>
```

个体之间的关系

- 用sameas声明两个个体之间是一样的:

```
<professor rdf:about=“rudiStuder”/>
<rdf:Description rdf:about=“rudiStuder”>
    <owl:sameAs rdf:resource=“professorStuder”/>
</rdf:Description>
```

封闭式类

```
<owl:Class rdf:about= “SecretariesOfStuder” >
    <owl:oneOf rdf:parseType= “Collection” >
        <Person rdf:about= “giselaSchillinger” />
        <Person rdf:about= “annneEberhardt” />
    </owl:oneOf>
</owl:Class>
```

- one of表明giselaSchillinger和anneEberhardt是类SecretariesOfStuder中仅有的个体。

与Owl:intersectionOf

- 下例声明了类SecretariesOfStuder恰好由Secretaries和MemberOfStudersGroup的对象构成

```
<owl:Class rdf:about= “SecrtariesOfStuder” >
    <owl:intersectionOf rdf:parseType= “Collection” >
        <owl:Class rdf:about= “Secretaries” />
        <owl:Class rdf:about= “MembersOfStudersGroup” />
    </owl:intersectionOf>
</owl:Class>
```

或Owl:UnionOf

- 下例表示教授可能是两个属性中的一个，或者两个属性同时存在

```
<owl:Class rdf:about= “Professor” >
    <rdfs:subClassOf>
        <owl:unionOf rdf:parseType= “Collection” >
            <owl:Class rdf:about= “ActivelyTeaching” />
            <owl:Class rdf:about= “Retired” />
        </owl:unionOf>
    </rdf:subClassOf>
</owl:Class>
```

非owl:complement

- 下例表明没有教员能是出版物

```
<owl:Class rdf:about=“FacultyMember” >
    <rdfs:subClassOf>
        <owl:Class rdf:ID=“non-publication” >//不写的话是匿名类
            <owl:complementOf rdf:resource=“Publication” />
        </owl:Class>
    </rdfs:subClassOf>
</owl:Class>
```

角色限制

- 下例声明一场考试中所有主考官都必须是教授：

```
<owl:Class rdf:about= “Exam” >
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource= “hasExaminer” />
            <owl:allValuesFrom rdf:resource= “Professor” />
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

- 如果把allValuesFrom变为someValuesFrom则意思是至少一个主考官是Professor

角色限制

- 也可以定义数量的上下限

```
<owl:Class rdf:about= “Exam” >
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource= “hasTopic” />
            <owl:maxCardinality
            rdf:datatype= “xsd:nonNegativeInteger” />
            2
            </owl:maxCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

- 如果用minCardinality代表的是下限，如果直接用Cardinality就意味着正好涵盖三个主题

角色限制

```
<owl:Class rdf:about= “ExamStuder” >
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource= “hasExaminer” />
            <owl:hasValue rdf:resource= “rudiStuder” />
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

- 如果用hasValue的话，意味着EaxmStuder中必须有rudiStuder这个人的存在

角色关系

<owl:ObjectProperty rdf:about=“hasExaminer” >
    <rdfs:subPropertyOf rdf:resource=“hasParticipant” />
</owl:ObjectProperty>

- 角色关系是相关的，如果用equivalentProperty意味着两个关系是相同的。
- 如果换做inverseOf的话，就意味着两个关系是相反的。

角色特性

- 可以考虑为ObjectProperty定义特性，包括传递性、对称性、函数性和反函数性。

```
<owl:ObjectProperty rdf:about= “hasColleague” >
    <rdf:type rdf:resource= “owl;TransitiveProperty” />
    <rdf:type rdf:resource= “owl;SymmetricProperty” />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about= “hasProjectiLeader” >
    <rdf:type rdf:resource= “owl;FunctionalProperty” />
</owl:ObjectProperty>
<owl:ObjectProperty
rdf:about= “isProhectLeaderFor” >
    <rdf:type
    rdf:resource= “owl;InverseFunctionalProperty” />
</owl:ObjectProperty>
```

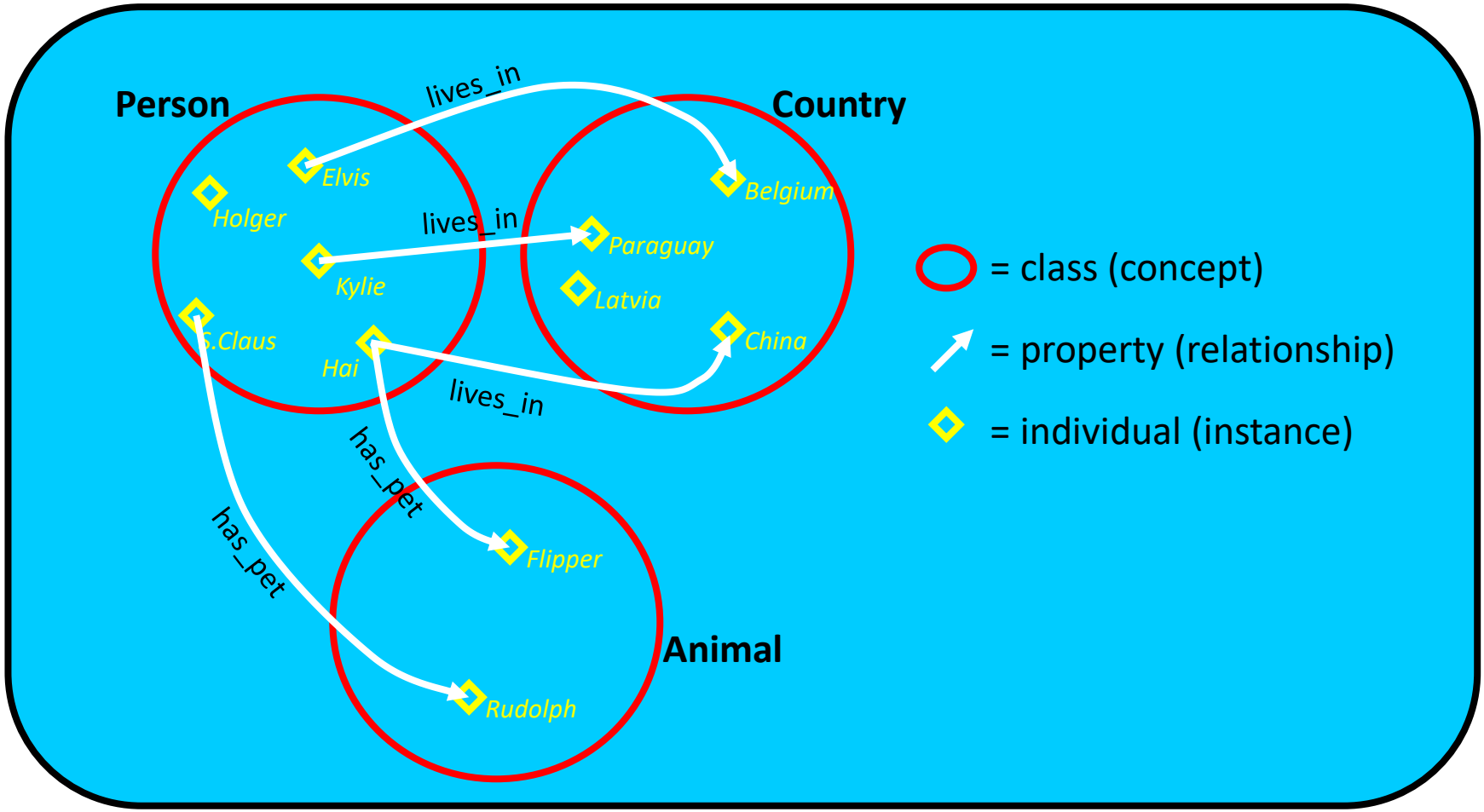
角色特性的使用

```
<Person rdf:about= “Peter” >
    <hasColleague rdf:resource= “Phillip” >
    <hasColleague rdf:resource= “Steven” >
    <isProjectLeaderFor rdf:resource= “neOn” />
</Person>
<Project rdf:about= “x-Media” >
    <hasProjectLeader rdf:resource= “Phillip” />
    <hasProjectLeader rdf:resource= “张三” />
</Project>
```

owl处理框架

- AllegroGraph RDF Store (triple store programming environment reasoner development environment rdfs reasoner). Directly usable from Java LISP Python Prolog C Ruby Perl
- Apache Jena (triple store programming environment reasoner rule reasoner owl reasoner rdfs reasoner parser). Directly usable from Java
- FRED (rdf generator tagging knowledge graph extractor).
- Mobi (programming environment development environment). Directly usable from Java Javascript
- OpenLink Virtuoso (triple store reasoner rdf generator sparql endpoint owl reasoner rdfs reasoner rdb2rdf). Directly usable from C C++ Python PHP Java Javascript ActionScript Tcl Perl Ruby Obj-C
- Oracle Spatial and Graph 19c (triple store reasoner owl reasoner). Directly usable from Java
- GraphDB (triple store reasoner sparql endpoint rdfs reasoner owl reasoner). Directly usable from Java C
- RDFox (triple store reasoner owl reasoner rdfs reasoner rule reasoner). Directly usable from C++ Java
- Altova's SemanticWorks (editor development environment).

本体的一个例子

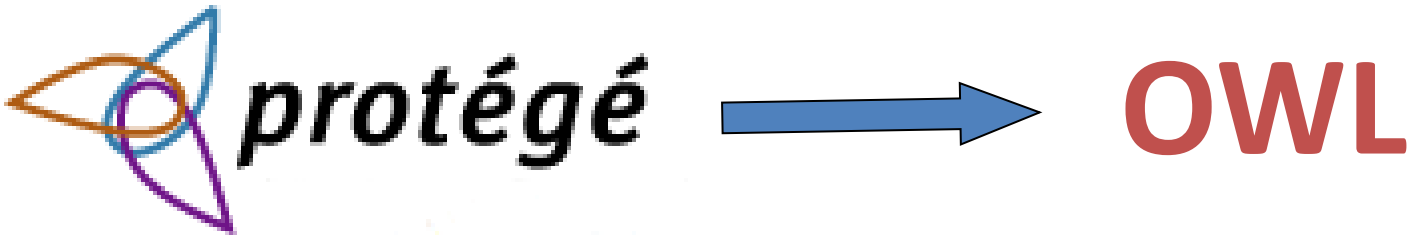

Person
Country
Animal
lives_in
lives_in
lives_in
has_pet
has_pet
Elvis
Holger
Kylie
S.Claus
Hai
Belgium
Paraguay
Latvia
China
Flipper
Rudolph
= class (concept)
= property (relationship)
= individual (instance)

用protégé构建本体

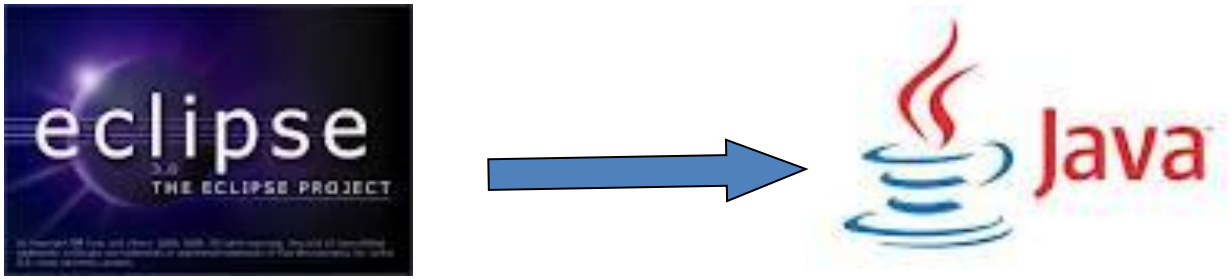
- 由斯坦福生物医学研究中心（BMIR）开发
- 基于java、开源、可扩展
- 下载地址：https://protege.stanford.edu/

protégé是什么

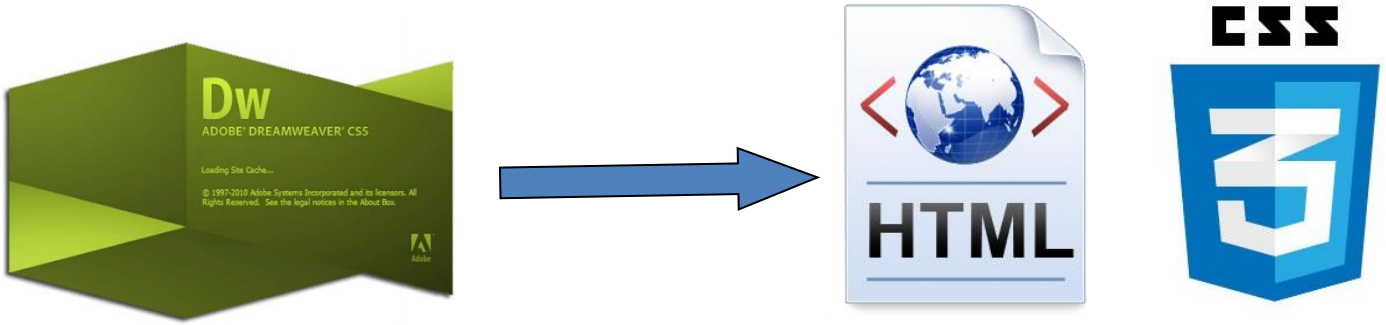
- 本体编辑工具



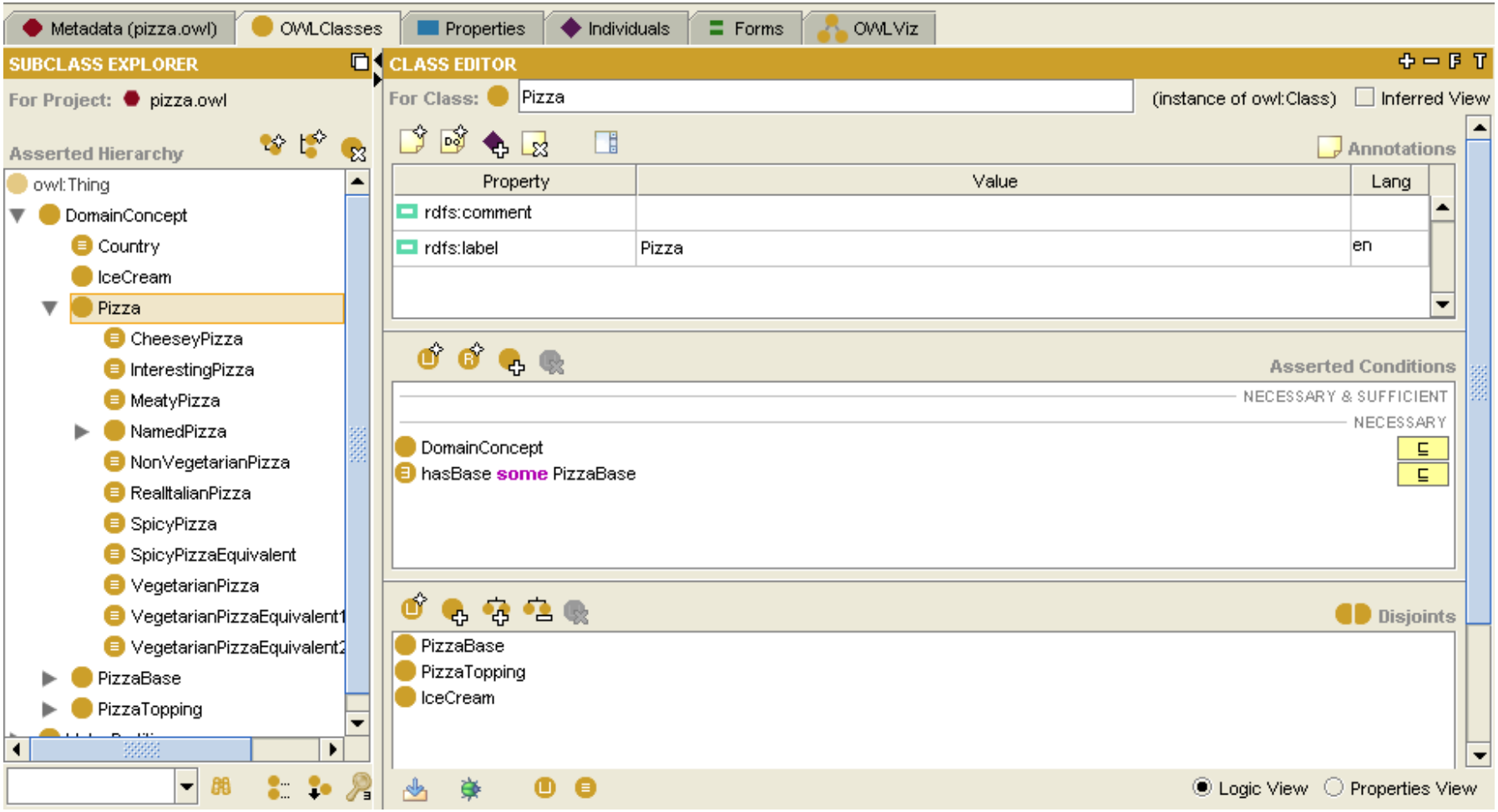
- eclipse 之于 java



- Dreamweaver 之于 HTML和CSS



protégé


Metadata (pizza.owl)
OWLClasses
Properties
Individuals
Forms
OWLViz
SUBCLASS EXPLORER
For Project: pizza.owl
Asserted Hierarchy
owl:Thing
DomainConcept
Country
IceCream
Pizza
CheeseyPizza
InterestingPizza
MeatyPizza
NamedPizza
NonVegetarianPizza
RealItalianPizza
SpicyPizza
SpicyPizzaEquivalent
VegetarianPizza
VegetarianPizzaEquivalent1
VegetarianPizzaEquivalent2
PizzaBase
PizzaTopping
CLASS EDITOR
For Class: Pizza
(instance of owl:Class)
Inferred View
Annotations
Property
Value
Lang
rdfs:comment
rdfs:label
Pizza
en
Asserted Conditions
NECESSARY & SUFFICIENT
NECESSARY
DomainConcept
hasBase some PizzaBase
Disjoints
PizzaBase
PizzaTopping
IceCream
Logic View
Properties View