Knowledge Representation – Logic-based Knowledge & Ontology

PRAKARN UNACHAK

Software

- Prolog
  - Most widely-used logic programming language
    - Based on first-order logic
    - Use backward chaining for inference
  - We'll used SWI Prolog: http://www.swi-prolog.org/
- Protégé https://protege.stanford.edu/
  - Ontology editor
  - Requires Java Runtime Environment https://www.java.com/en/download/manual.jsp
  - With OWLViz for graphical interaction.
    - https://protegewiki.stanford.edu/wiki/OWLViz
    - Need Graphviz: https://www.graphviz.org/download/
  - Example ontology: Pizza
    - https://protege.stanford.edu/ontologies/pizza/pizza.owl

Outline

- Overview of Knowledge Representation
- Propositional Logic
  - Syntax and Semantics
  - Inference
- First-order Logic
  - Syntax, Semantics, and Assumptions
  - Creating First-order Logic Knowledge Base
  - Inference
  - Application Examples – Logic Programming (Prolog) & Rule-based Expert System
- Ontology

Knowledge Representation

Knowledge Base

- Knowledge base (KB) consists of sentences
- A sentence show specific knowledge
- A piece of knowledge can be data, information derived from data or directives derived from data or information
- In a KB, the sentences are represented by knowledge representation language

Knowledge Representation Language

- The language should provide enough information for showing whether a sentence is true or false

Components:

- Object – nouns in the problem domain such as person, places, etc.
- Relation – true/false for existence of relation between objects
- Function – Relation identifier, but provide object instead of true/false value
- Interpretation of Symbols – meaning for each symbol used

Knowledge Representation

- Knowledge representation is the act of keeping knowledge and facts in certain format for future use (in KB)

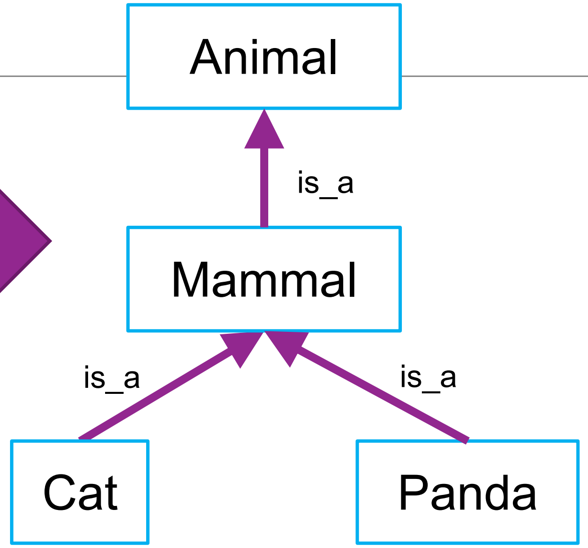
- A good knowledge representation should:
  - Reduce/remove ambiguity
  - Allow (different levels of) abstraction
  - Allow inference to create new knowledge, to aid in decision
  - Allow acquisition of new knowledge
  - Allow update of KB

One set of facts can have many representations

Facts

- Cat is a mammal
- Panda is a mammal
- All mammals are animals

Representation #1



Representation #2

CatIsMammal

PandaIsMammal

MammalIsAnimal

Representation #3

$Mammal(Cat)$

$Mammal(Panda)$

$\forall x\ Mammal(x) \Rightarrow Animal(x)$

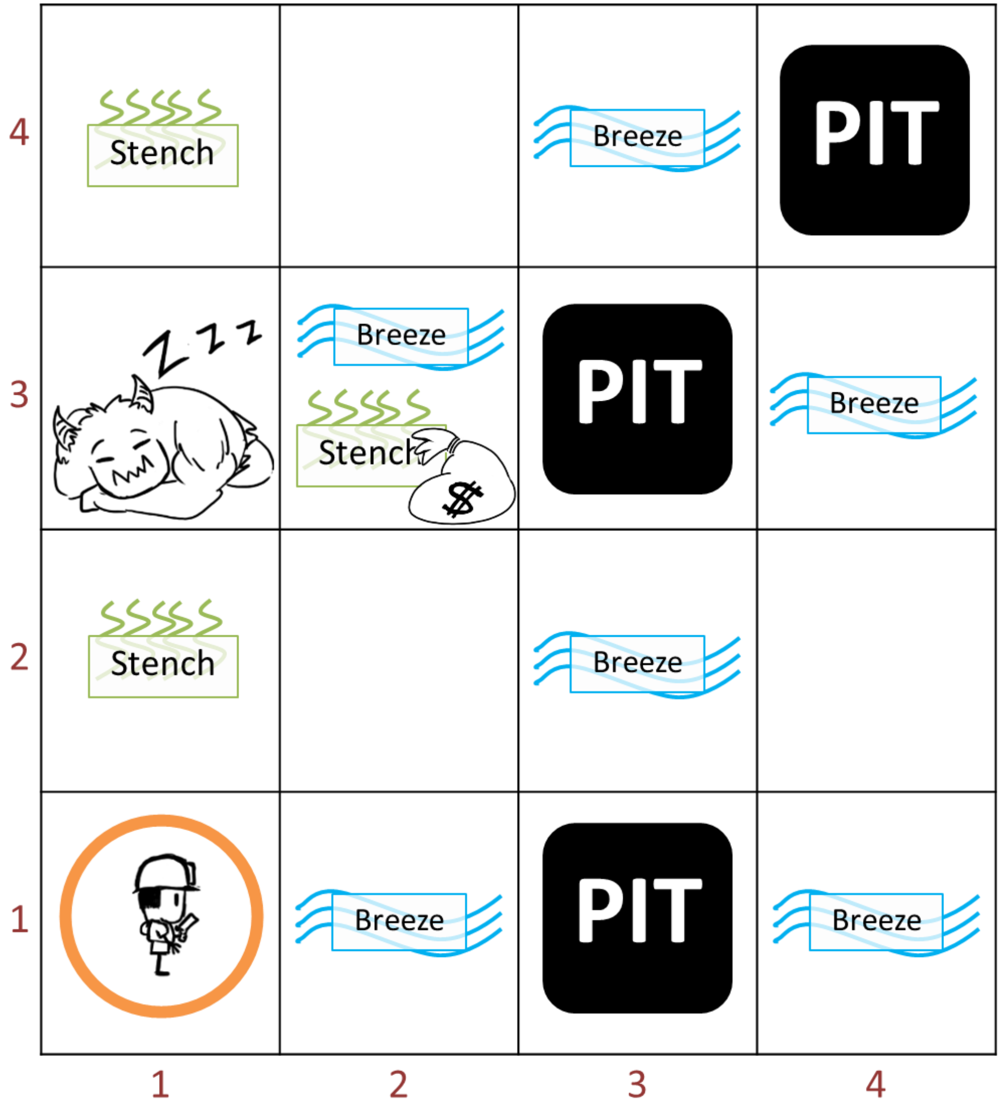
Four Properties of Suitable Knowledge Representation

Is this knowledge representation suitable for our problem?

- **Representational Adequacy**
  - Can it represent knowledge for this problem domain?
- **Inferential Adequacy**
  - Can new knowledge be inferred and stored properly?
- **Inferential Efficiency**
  - Can new knowledge be inferred to the direction of our interest?
- **Acquisitional Efficiency**
  - How easy it is to add new knowledge?

Example Problem – Wumpus World

- Helping explorer to obtain treasure and leave the dark cave safely
- The cave is 4 x 4 grid of rooms, with wall on all sides
- The cave is completely dark, the explorer need to rely on other senses
- Goal: guide explorer to the treasure, pick it up, and come back to the entrance (and leave) without being eaten by Wumpus and falling into the pit



Wumpus World (cont.) – Dangers

- The cave will have a **Wumpus** at random location (not entrance nor treasure)
  - Wumpus is sleep and will not move
  - But if the explorer enter the room with Wumpus, they will be eaten
- The cave will also have one or more **pit** at random locations (not entrance nor treasure nor Wumpus)
  - If the explorer enter the pit room, they will fall down and die

Wumpus World (cont.) – The Explorers

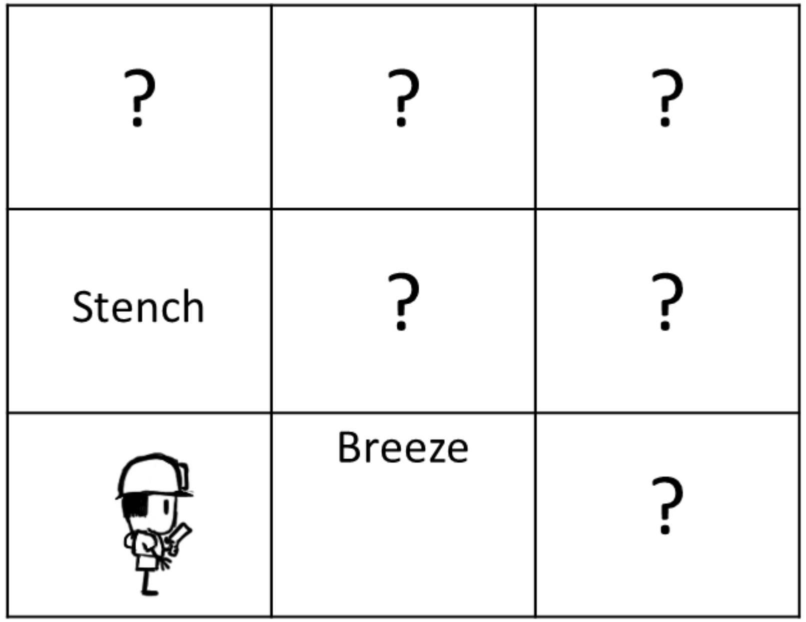
- The explorer will start at [1, 1], lower-left corner
- Only senses available — [Stench, Breeze, Glitter, Bump]
  - The explorer will smell stench if the Wumpus is in one of the adjacent (non-diagonal) rooms
  - The explorer will feel breeze if there is at least one pit in adjacent (non-diagonal) rooms
  - The explorer will see glitter if they are in the same room as the treasure
  - The explorer will feel a bump if they move into a wall

Wumpus World (cont.) – The Explorers (cont.)

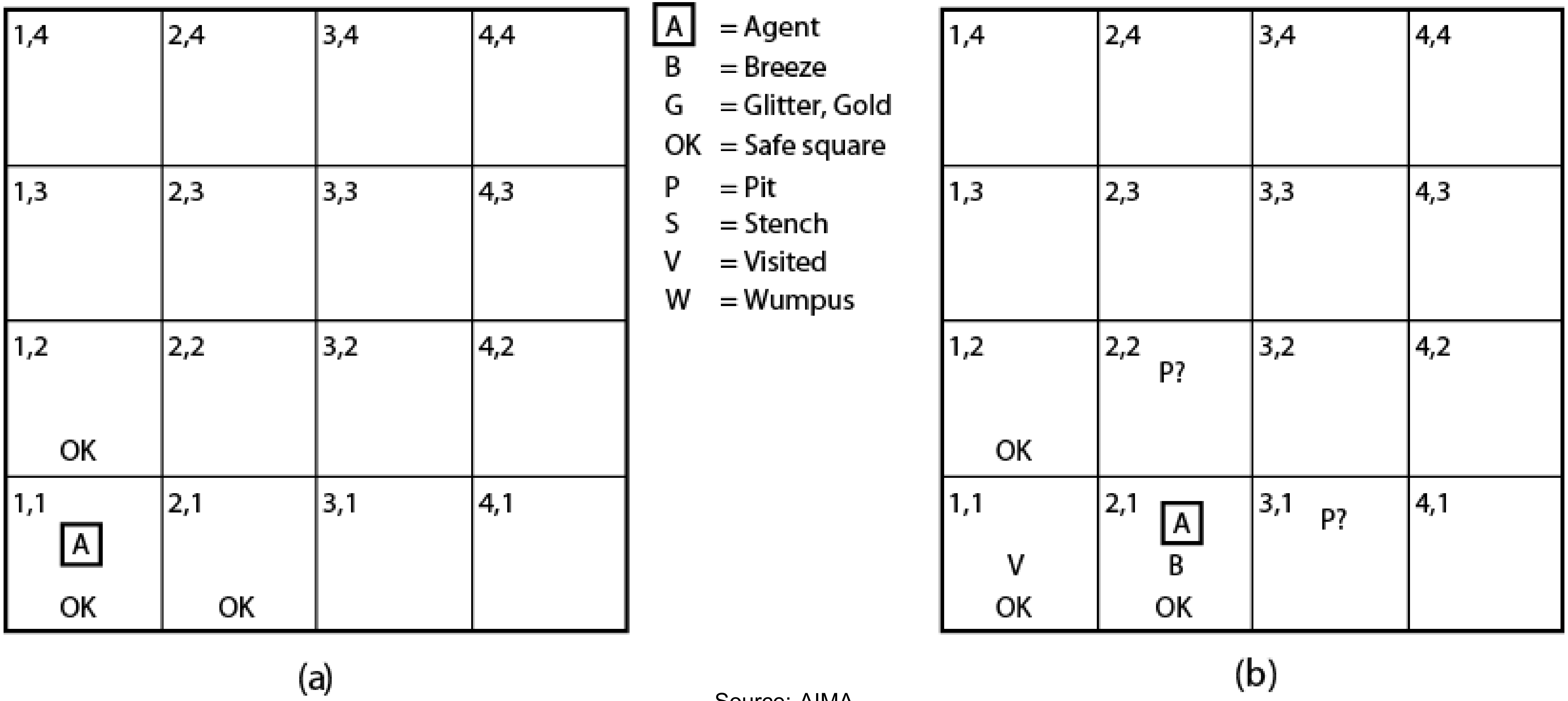
- Possible Actions
  - Move forward 1 room on the direction the explorer is facing, or bump into a wall
  - Turn left 90°
  - Turn right 90°
  - Grab the treasure, if it is at the same room as the explorer
  - Climb out of the cave, only available at [1, 1]

Wumpus World (cont.)

- Performance Measure (Scoring)
  - If the explorer leaves the cave with the treasure → 1,000 points
  - If they get eaten by the Wumpus or fall down a pit → -1,000 points
  - -1 points per action
- To play Wumpus world, we need to interpret senses to infer which room is safe (or not)

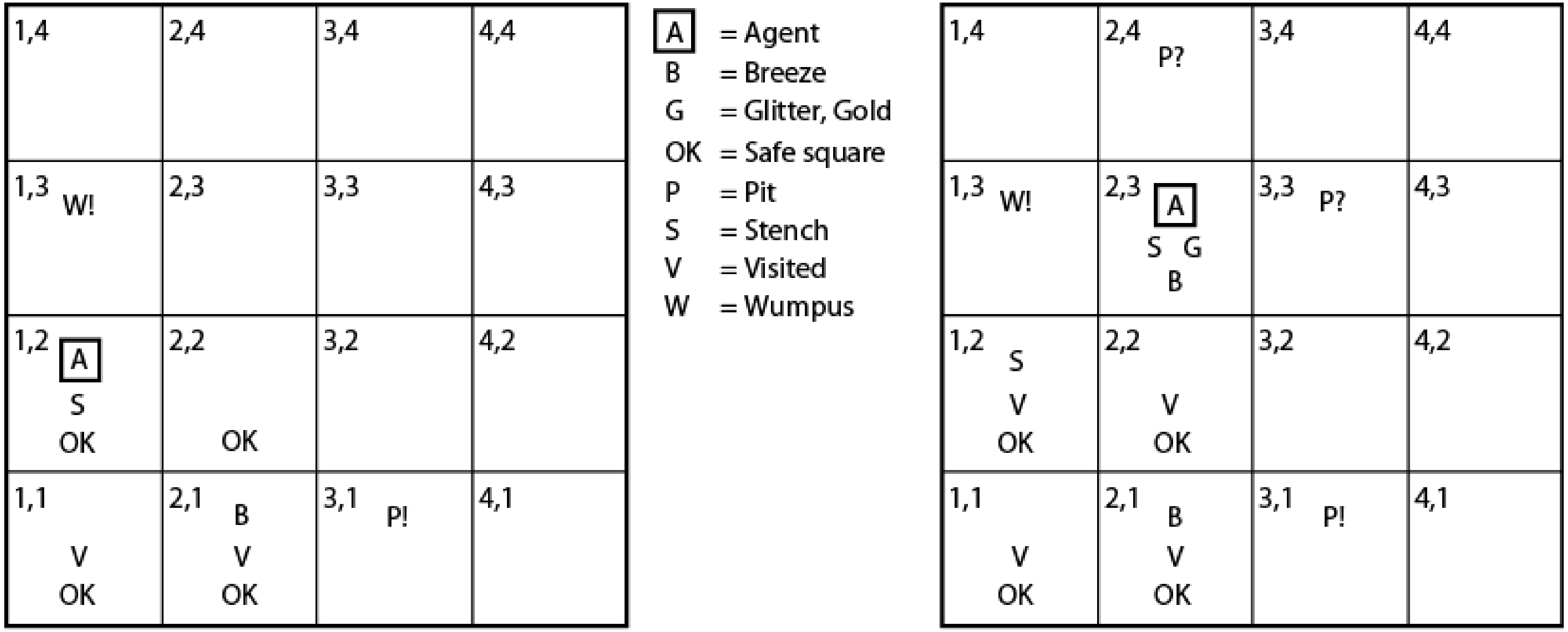


Wumpus World (cont.)



Source: AIMA

Wumpus World (cont.)



(a) (b)

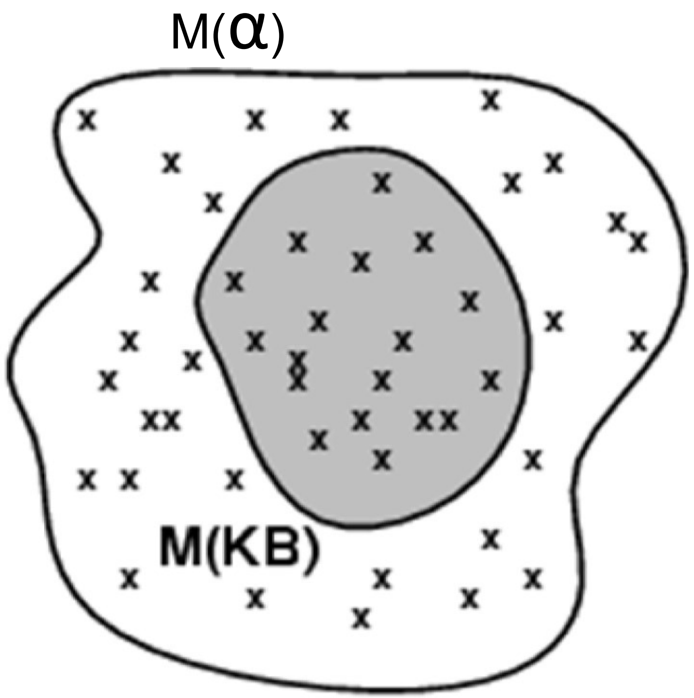
Logic

Logic in General

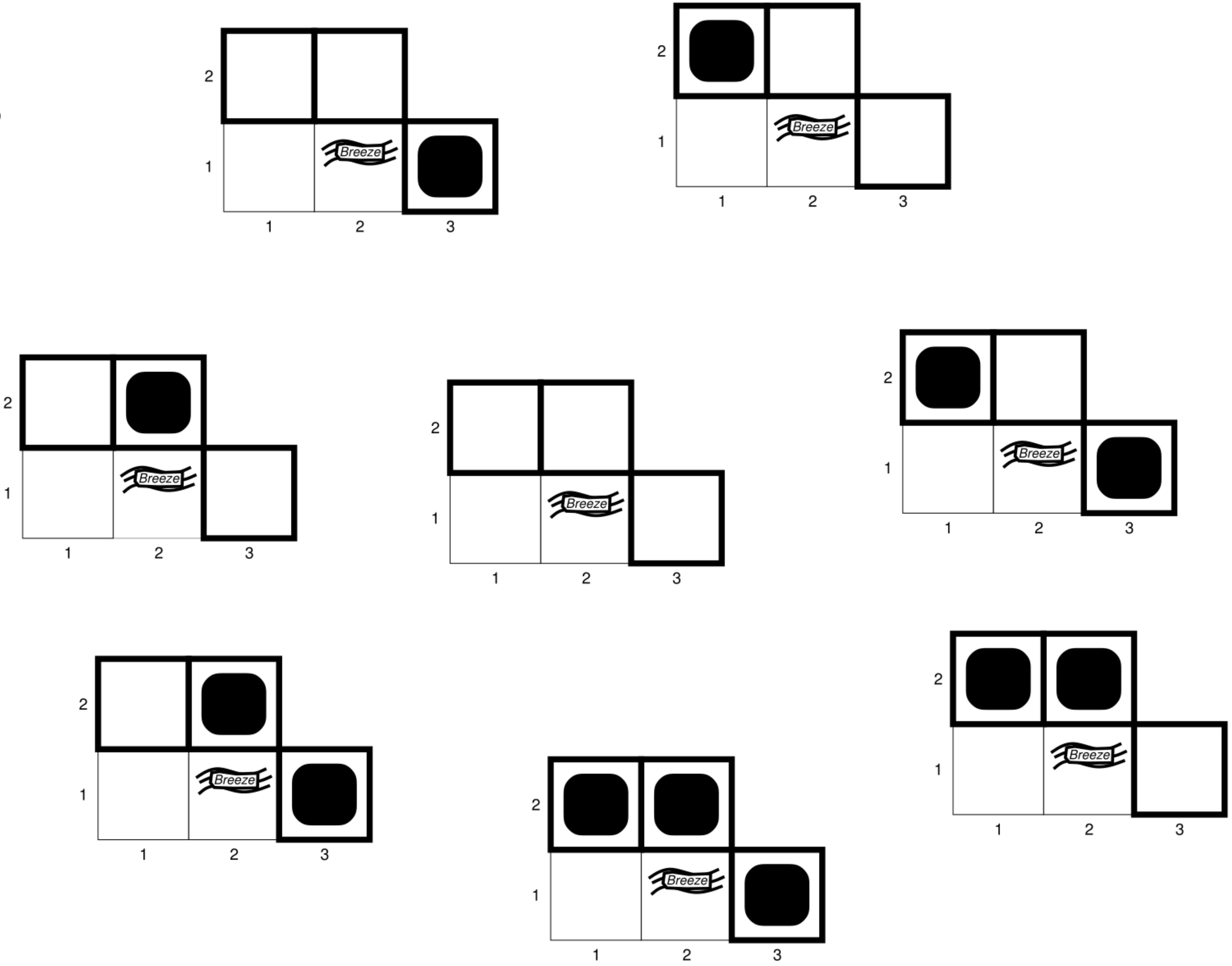
- Logics is a formal language to represent knowledge so that new knowledge can be inferred from them
- Syntax specify the format of such formal language
- Semantics define meaning of sentences in that language
  - For example, truth value of a sentence
- For example, in mathematics:
  - $x + 2 \geq y$ is a sentence, but $x2 + y >$ isn't
  - $x + 2 \geq y$ is true iff $x + 2$ is no less than y
  - $x + 2 \geq y$ is true in a world (model) that $x = 7$,and $y = 1$
  - $x + 2 \geq y$ is false in a world that $x = 0$, and $y = 6$

Models & Entailment

- Models: "Formally Structured Worlds with Respect to Which Truth Can Be Evaluated"
  - Model *M* is a model of A if A is true on *M*
- Entailment (⊨)
  - Let M(*X*) be set of all models of sentence *X*
  - "*A* ⊨ *B* iff M(*a*) ⊆ M(*b*)"
  - Or A is true in a model that, B will also be true in that model
  - <u>Example of</u> KB ⊨ A
    - Kb = "I won a lottery and I went to Bangkok"
    - A = "I went to Bangkok"

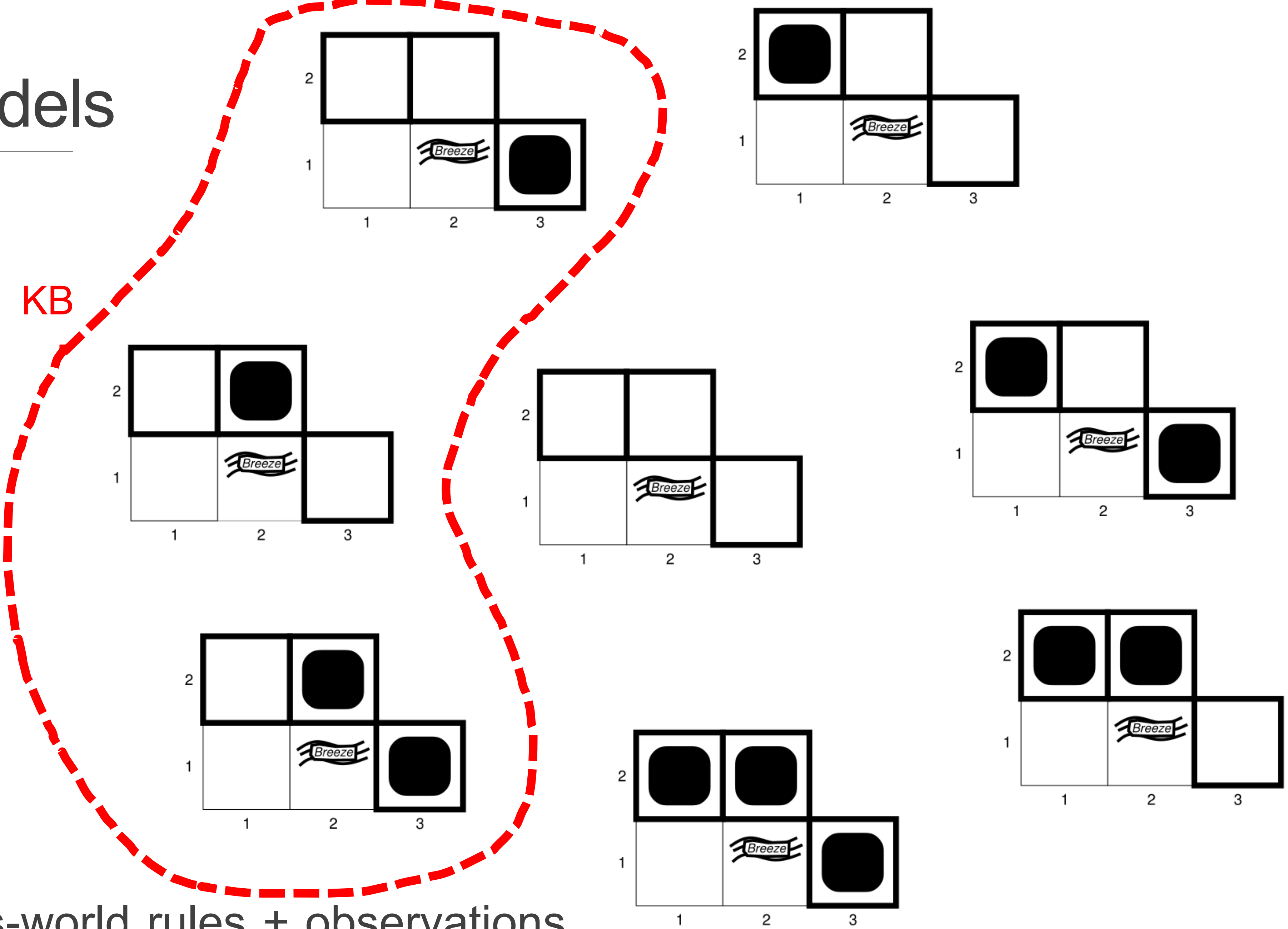


Wumpus Models



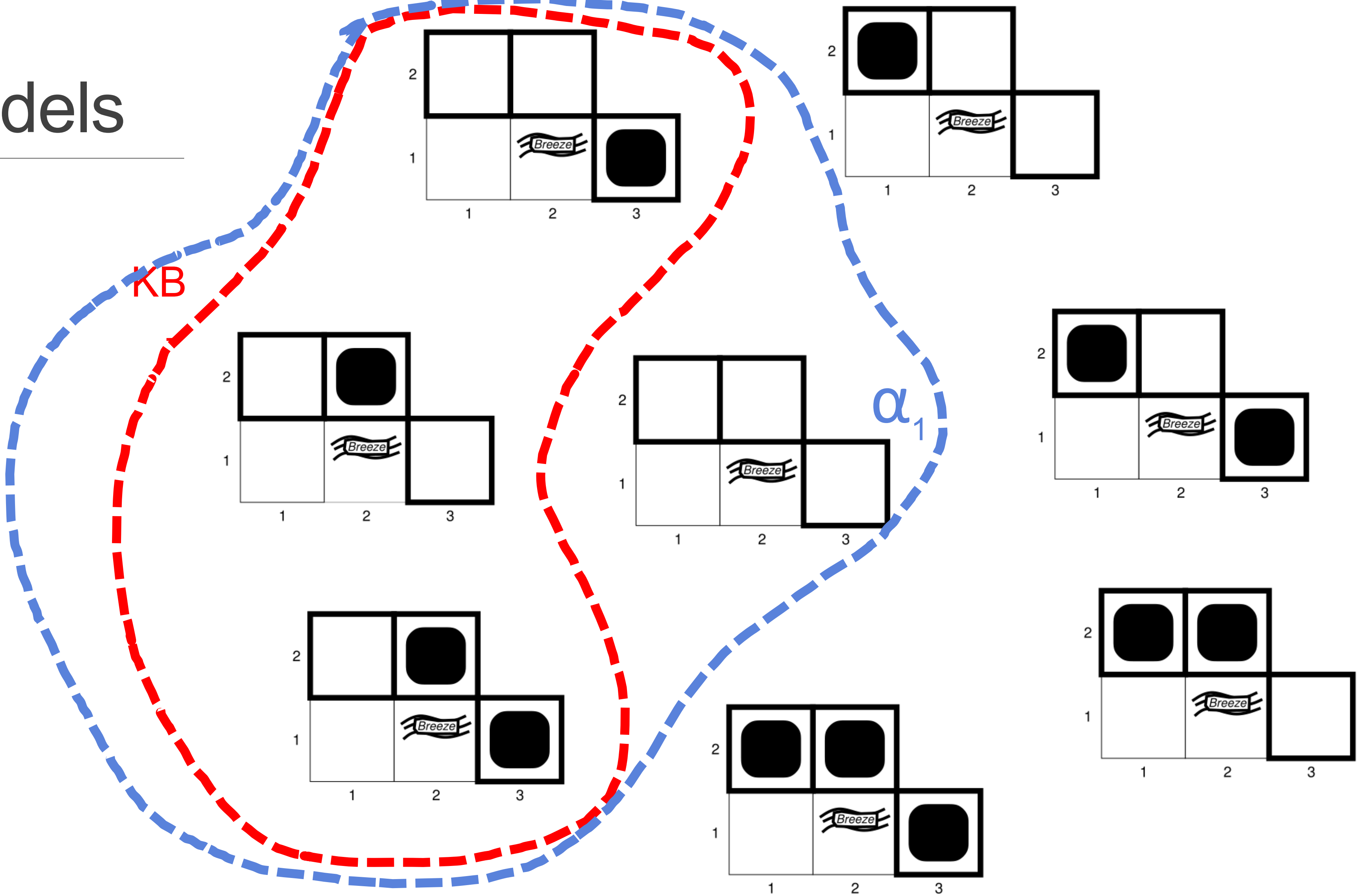
- “Start at [1, 1]”
- “Nothing at [1, 1]”
- “Move to [2, 1]”
- “[2, 1] is breezy”

Wumpus Models



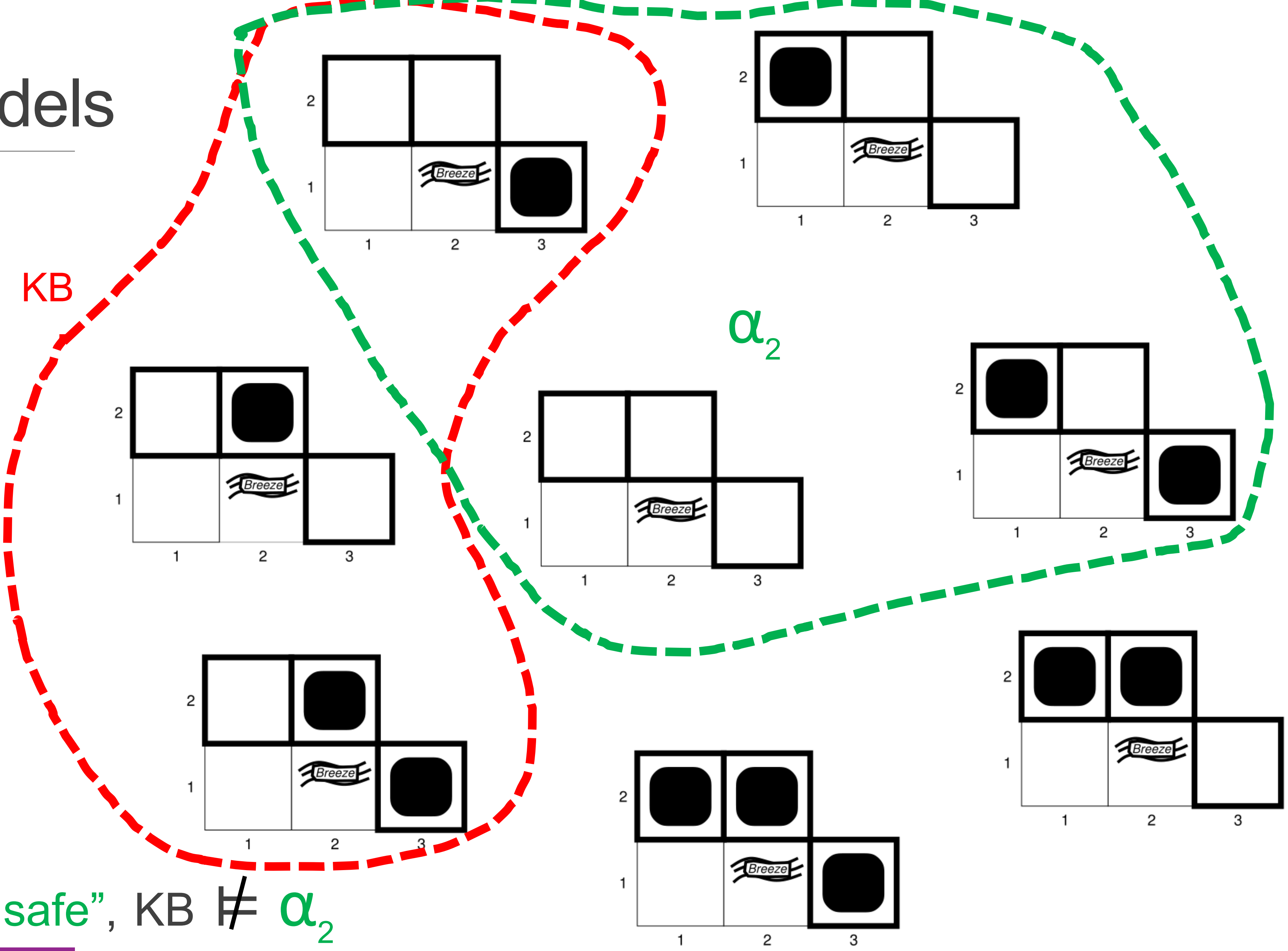
- KB = Wumpus-world rules + observations

Wumpus Models



- α_1 = "[1,2] is safe", KB $\models$ α_1 which can be proven by model-checking

Wumpus Models



- α_2 = "[2,2] is safe", KB $\not\models$ α_2

Inference

- $KB \vdash_i \alpha$ – the sentence α can be inferred from KB with the method *i*
  - For example, from last pages: $KB \vdash_{\textit{model-checking}} \alpha_1$
- Soundness: *i* will be a sound method if:

  If $KB \vdash_i \alpha$ then $KB \vDash \alpha$ always
- Completeness: *i* will be a complete method if:

  If $KB \vDash \alpha$ then $KB \vdash_i \alpha$ always

Propositional Logic

Propositional Logic: Syntax

- Propositional logic is the simple logic syntax
- Use proposition symbols (literals) P_1, P_2, $B_{1,3}$, *North* for sentences
- Use logical connectives to create complex sentence:
  - $\neg$ (not) Negation
  - $\wedge$ (and) Conjunction
  - $\vee$ (or) Disjunction
  - $\Rightarrow$ (implies) Implication
    - *Antecedent/Premise/Body* $\Rightarrow$ *Conclusion/Consequent/Head*
  - $\Leftrightarrow$ (if and only if, iff) Biconditional

BNF (Backus-Naur Form) Grammar

$$
\begin{array}{lcl}
\text{Sentence} & \longrightarrow & \text{AtomicSentence} \mid \text{ComplexSentence} \\
\text{AtomicSentence} & \longrightarrow & \text{True} \mid \text{False} \mid P \mid Q \mid R \mid \ldots \\
\text{ComplexSentence} & \longrightarrow & (\ \text{Sentence}\ ) \mid [\ \text{Sentence}\ ] \\
& \mid & \neg\ \text{Sentence} \\
& \mid & \text{Sentence} \land \text{Sentence} \\
& \mid & \text{Sentence} \lor \text{Sentence} \\
& \mid & \text{Sentence} \Rightarrow \text{Sentence} \\
& \mid & \text{Sentence} \Leftrightarrow \text{Sentence}
\end{array}
$$

OPERATOR PRECEDENCE : $\neg, \land, \lor, \Rightarrow, \Leftrightarrow$

Propositional Logic: Semantics

- Each model must assign truth value (true/false) to each propositional symbol
  - Example: m1 = $\{P_{1,2}$ = false, $P_{2,2}$ = false, $P_{3,1}$ = true$\}$
  - If there are 3 symbols $\rightarrow 2^3 = 8$ possible models
- For complex sentence:
  - $\neg P$ is true **iff** P is false
  - $P \wedge Q$ is true **iff** P is true **AND** Q is true
  - $P \vee Q$ is true **iff** at least either P is true **OR** Q is true
  - $P \Rightarrow Q$ is true **EXCEPT** when P is true **AND** Q is false
  - $P \Leftrightarrow Q$ is true **iff** P $\Rightarrow$ Q **AND** Q $\Rightarrow$ P

Truth Tables for Connectives

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Proof Method

To check that KB $\models \alpha$

2 main groups:

1. Model checking – enumerates all possible models to check if α is true in all model in which KB is true
   - Truth table enumeration – very costly for complex problem
2. Proof by inference rules
   - Infer new sentence that does not contradict KB (sound) from existing sentences
   - Proving by applying inference rules in sequence until is α found, or new sentences cannot be created

Enumeration Inference with Truth Tables

- Create truth table for all literals involved
- Consider only row where KB is true
  - If the query *q* is also true/false, it can be proven true/false by KB
- Example: let $P_{i,j}$ be true if there is a pit at [*i*, *j*], and $B_{i,j}$ be true if there is breeze at [*i*, *j*]
- Initial Sentences:

  R_1: $\neg P_{1,1}$

  - "a pit make nearby square breezy"

  R_2: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

  R_3: $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$
- After travel to [2, 1] and observe a breeze

  R_4: $\neg B_{1,1}$

  R_5: $B_{2,1}$

?	?	?
?	?	?
	Breeze **A**	?

Enumeration Inference with Truth Tables

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	<u>*true*</u>
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	<u>*true*</u>
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	<u>*true*</u>
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

- We got $P_{1,2}$ = *false* in all row that *KB* is true

Proof by Inference Rules

- Searching with KB as state
- Initial State:
  - Initial KB
- Actions:
  - Using an inference rules with matching conditions
- Result:
  - Adding the result of the inference rule into KB
- Goal:
  - KB with query sentence

Logical Equivalence

- Logical equivalence: when two sentences are true to the same set of models: $\alpha \equiv \beta$ iff $\alpha \models \beta$ and $\beta \models \alpha$

$$
\begin{array}{rcll}
(\alpha \wedge \beta) & \equiv & (\beta \wedge \alpha) & \text{commutativity of } \wedge \\
(\alpha \vee \beta) & \equiv & (\beta \vee \alpha) & \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) & \equiv & (\alpha \wedge (\beta \wedge \gamma)) & \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) & \equiv & (\alpha \vee (\beta \vee \gamma)) & \text{associativity of } \vee \\
\neg(\neg\alpha) & \equiv & \alpha & \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) & \equiv & (\neg\beta \Rightarrow \neg\alpha) & \text{contraposition} \\
(\alpha \Rightarrow \beta) & \equiv & (\neg\alpha \vee \beta) & \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) & \equiv & ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) & \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) & \equiv & (\neg\alpha \vee \neg\beta) & \text{de Morgan} \\
\neg(\alpha \vee \beta) & \equiv & (\neg\alpha \wedge \neg\beta) & \text{de Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) & \equiv & ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) & \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) & \equiv & ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) & \text{distributivity of } \vee \text{ over } \wedge
\end{array}
$$

Validity and Satisfiability

- A sentence is valid when it is true in all possible model
  - Examples: $True$, $A \lor \neg A$, $A \Rightarrow A$, $(A \land (A \Rightarrow B)) \Rightarrow B$
- Validity helps with inference by deduction theorem:
  - $KB \models \alpha$ iff $(KB \Rightarrow \alpha)$ is valid
- A sentence is satisfiable if it can be true in some models
  - Examples: $A \lor B$, C
- A sentence is unsatisfiable if it **cannot** be true in any model
  - Examples: $A \land \neg A$
- Satisfiability relates to inference as followed:
  - $KB \models \alpha$ iff $(KB \land \neg \alpha)$ is unsatisfiable

Example of Inference Rules

- Modus Ponens

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

- And-Elimination

$$\frac{\alpha \land \beta}{\alpha}$$

- Biconditional Elimination

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)} \text{ AND } \frac{(\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}$$

Forward and Backward Chaining

- Prove by inference rules
- KB in Horn From
  - KB = Conjunction ($\wedge$) of Horn clauses
  - Horn clauses include
    - Fact: single proposition symbol (literal)
    - Implication: Premise (Body) $\Rightarrow$ Conclusion (Head)
    - No Negation
- Modus ponens results of horn clauses will be also be horn clauses

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

- Can be used with forward chaining and backward chaining
  - Linear-time

Forward Chaining

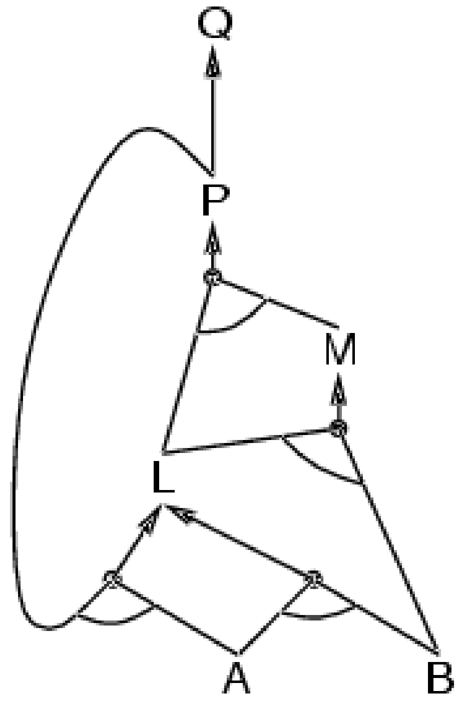
```
function PL-FC-ENTAILS?(KB, q) returns true or false
    inputs:    KB, the knowledge base, a set of propositional definite clauses
                   q, the query, a proposition symbol
    count ⟵ a table, where count[c] is the number of symbols in c's premise
    inferred ⟵ a table, where inferred[s] is initially false for all symbols
    agenda ⟵ a queue of symbols, initially symbols known to be true in KB
    while agenda is not empty do
        p ⟵ POP(agenda)
        if p = q then return true
        if inferred[p] = false then
            inferred[p] ⟵ true
            for each clause c in KB where p is in c.PREMISE do
                decrement count[c]
                if count[c] = 0 then add c.CONCLUSION to agenda
    return false
```

Forward Chaining

- Idea: Use rules in KB with all true literals in premise
  - Adding conclusion into KB until:

1. Found clause we want to prove (query proven true) OR
2. No more clauses can be added (can't prove query)

$$
\begin{aligned}
P &\Rightarrow Q \\
L \wedge M &\Rightarrow P \\
B \wedge L &\Rightarrow M \\
A \wedge P &\Rightarrow L \\
A \wedge B &\Rightarrow L \\
A & \\
B &
\end{aligned}
$$



- Sound and complete

Forward chaining example

Proving Q

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

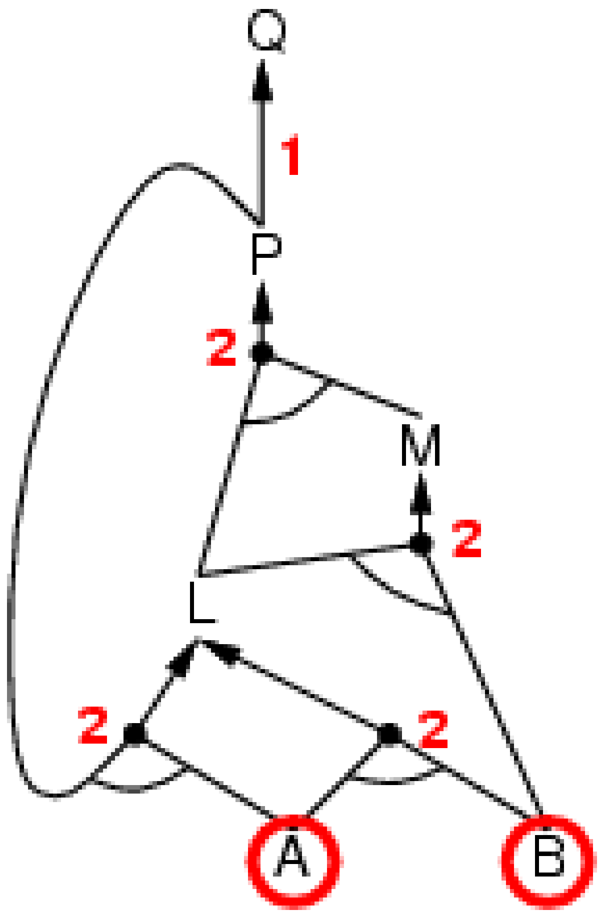
$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

$\boxed{A}$

$\boxed{B}$



Agenda = {*A*, *B*}

Forward chaining example

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

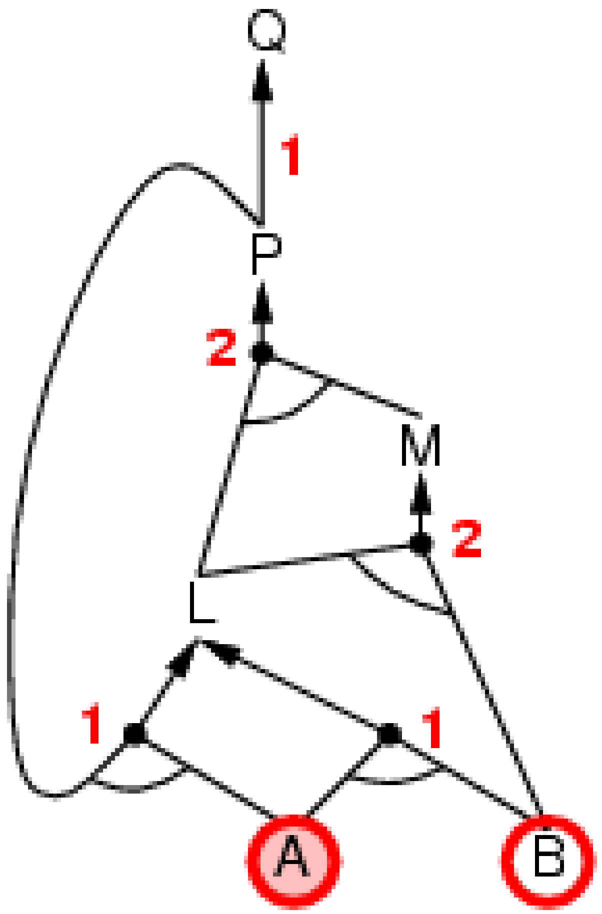
$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

$\boxed{A}$

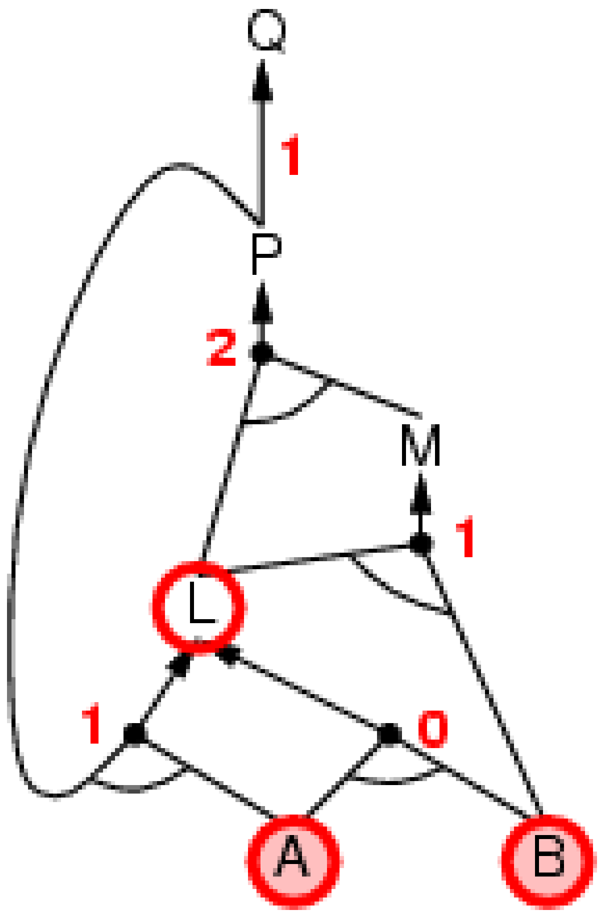
$\boxed{B}$



Agenda = {~~A~~, B}

Forward chaining example

$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$\boxed{A \wedge B \Rightarrow L}$$
$$\boxed{A}$$
$$\boxed{B}$$



Agenda = {~~A~~, ~~B~~, L}

Forward chaining example

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

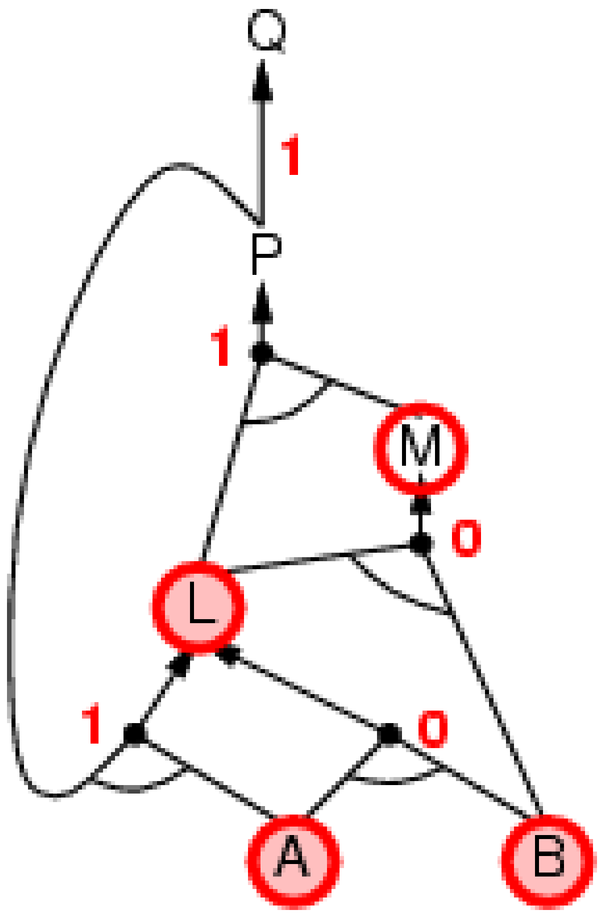
$\boxed{B \wedge L \Rightarrow M}$

$A \wedge P \Rightarrow L$

$\boxed{A \wedge B \Rightarrow L}$

$\boxed{A}$

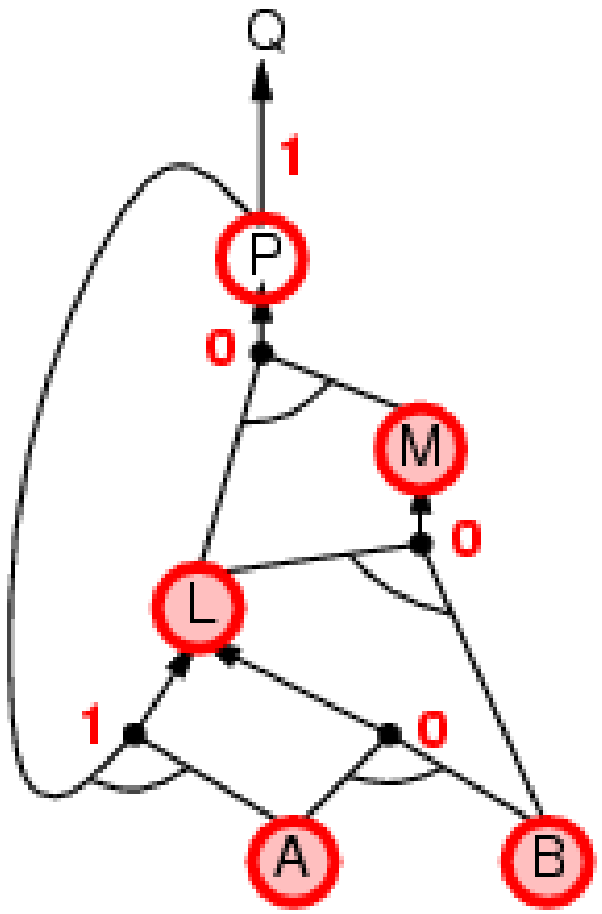
$\boxed{B}$



Agenda = {~~A~~, ~~B~~, ~~L~~, M}

Forward chaining example

$$P \Rightarrow Q$$
$$\boxed{L \wedge M \Rightarrow P}$$
$$\boxed{B \wedge L \Rightarrow M}$$
$$A \wedge P \Rightarrow L$$
$$\boxed{A \wedge B \Rightarrow L}$$
$$\boxed{A}$$
$$\boxed{B}$$



Agenda = {~~A~~, ~~B~~, ~~L~~, ~~M~~, P}

Forward chaining example

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

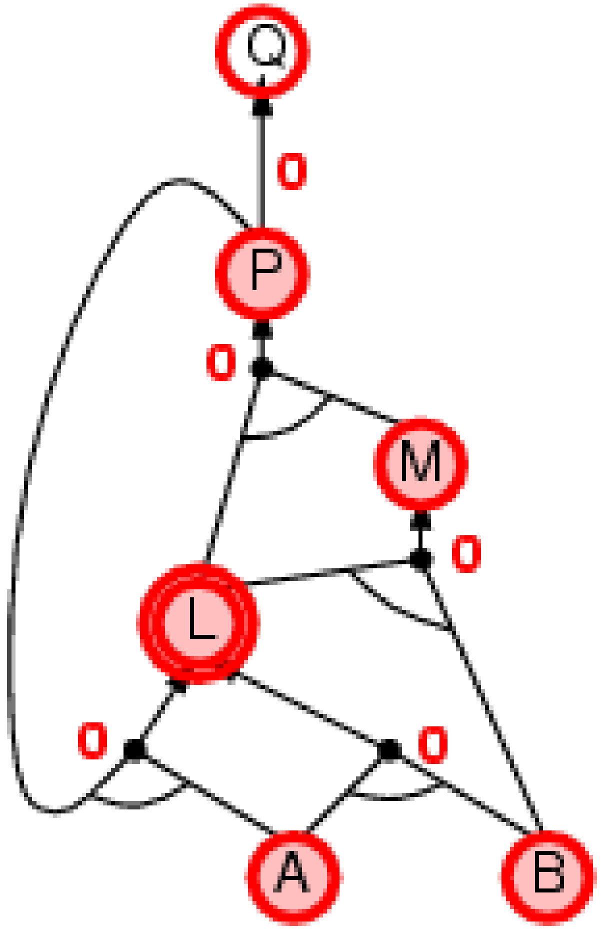
$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A

B

Agenda = {~~A~~, ~~B~~, ~~L~~, ~~M~~, ~~P~~}



Forward chaining example

$P \Rightarrow Q$

$\boxed{L \wedge M \Rightarrow P}$

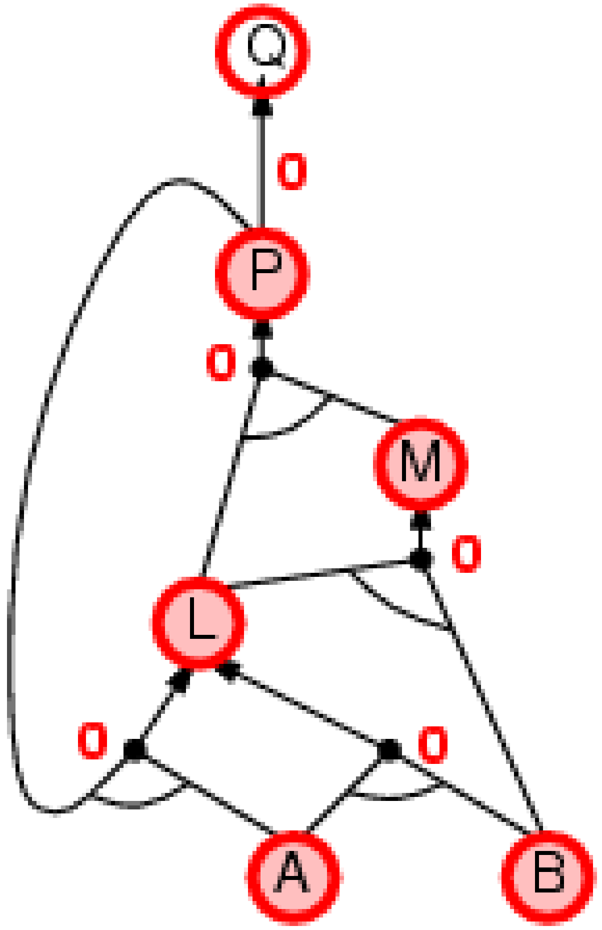
$\boxed{B \wedge L \Rightarrow M}$

$\boxed{A \wedge P \Rightarrow L}$

$\boxed{A \wedge B \Rightarrow L}$

$\boxed{A}$

$\boxed{B}$



Agenda = {~~A~~, ~~B~~, ~~L~~, ~~M~~, ~~P~~}

Forward chaining example

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

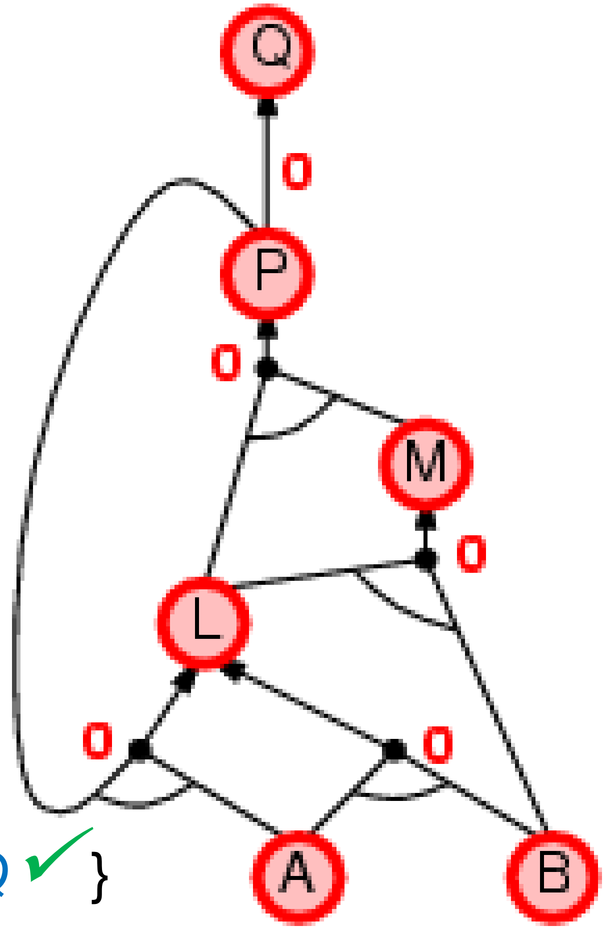
$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A

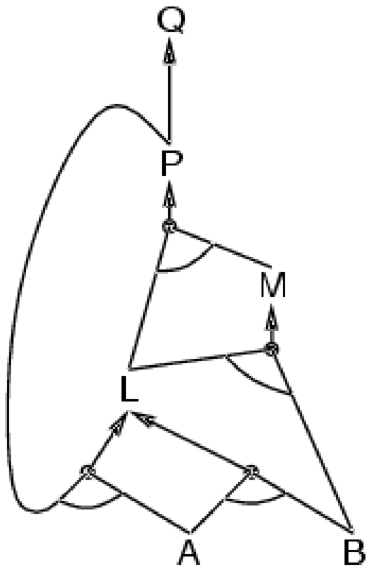
B



Agenda = {~~A~~, ~~B~~, ~~L~~, ~~M~~, ~~P~~, Q ✓}

Backward Chaining

$$
\begin{aligned}
&P \Rightarrow Q\\
&L \wedge M \Rightarrow P\\
&B \wedge L \Rightarrow M\\
&A \wedge P \Rightarrow L\\
&A \wedge B \Rightarrow L\\
&A\\
&B
\end{aligned}
$$



- Idea: Start from the target query *q*
  - Prove *q* by:

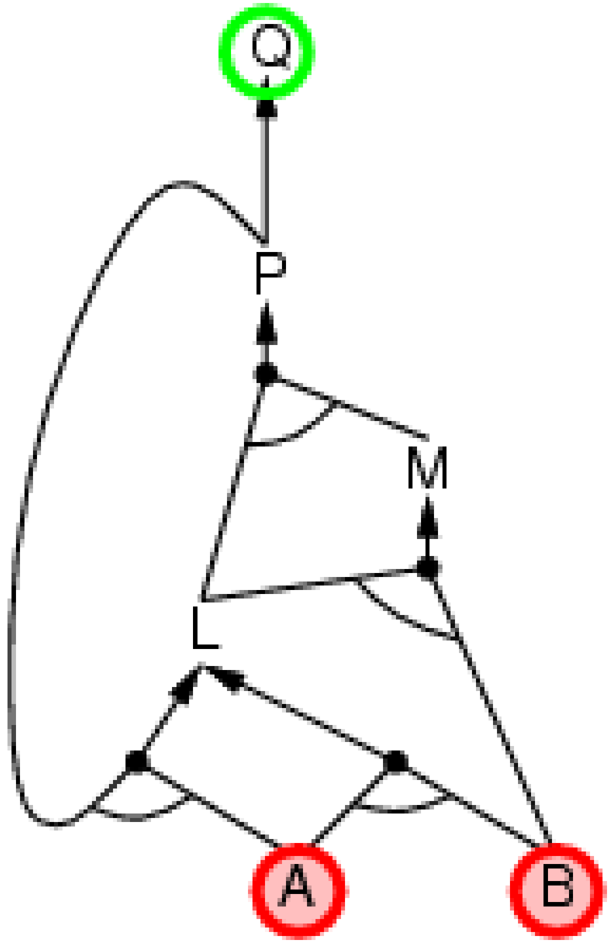
1. Check whether q is already proven true
2. If not, check if there is an implication with *q* as conclusion. If so, add the premise of that implication as subgoals to be proven

  - To avoid looping: check first if a subgoal already exists
  - To avoid repition: check first if a subgoal has already been proven true/false

Backward chaining example

Proving Q

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

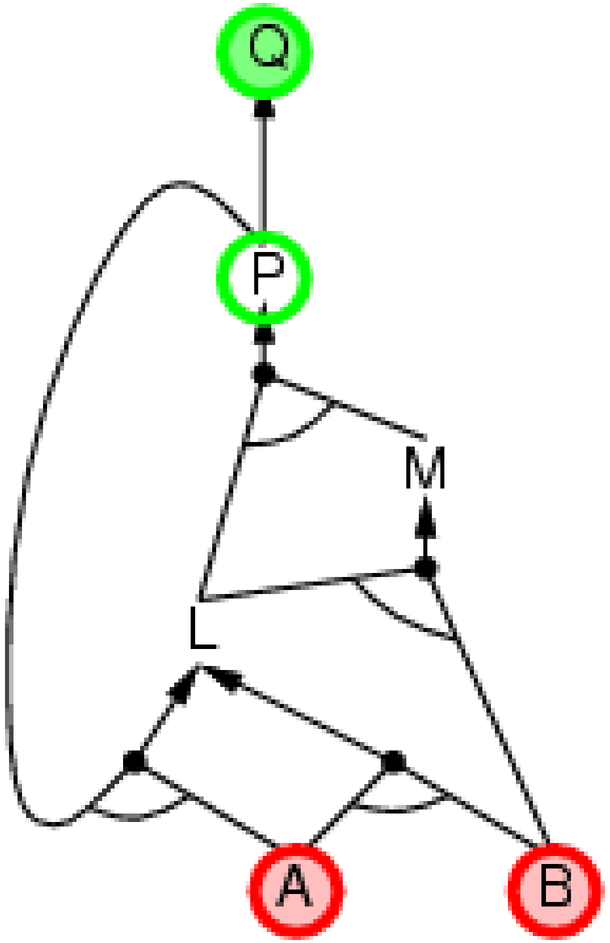


Query = Q

Subgoals = { Q }

Backward chaining example

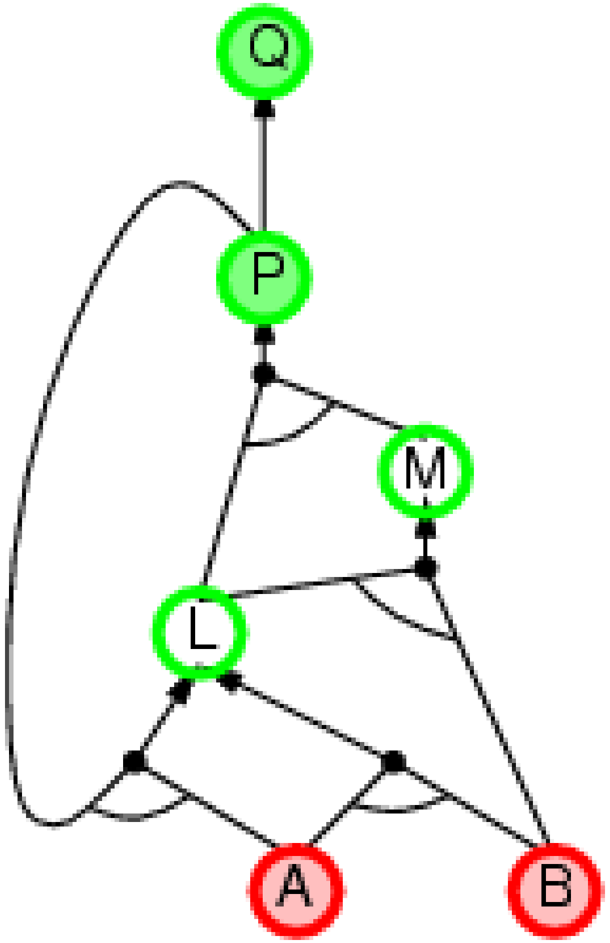
$$\boxed{P \Rightarrow Q}$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$A$$
$$B$$



Subgoals = { Q, P }

Backward chaining example

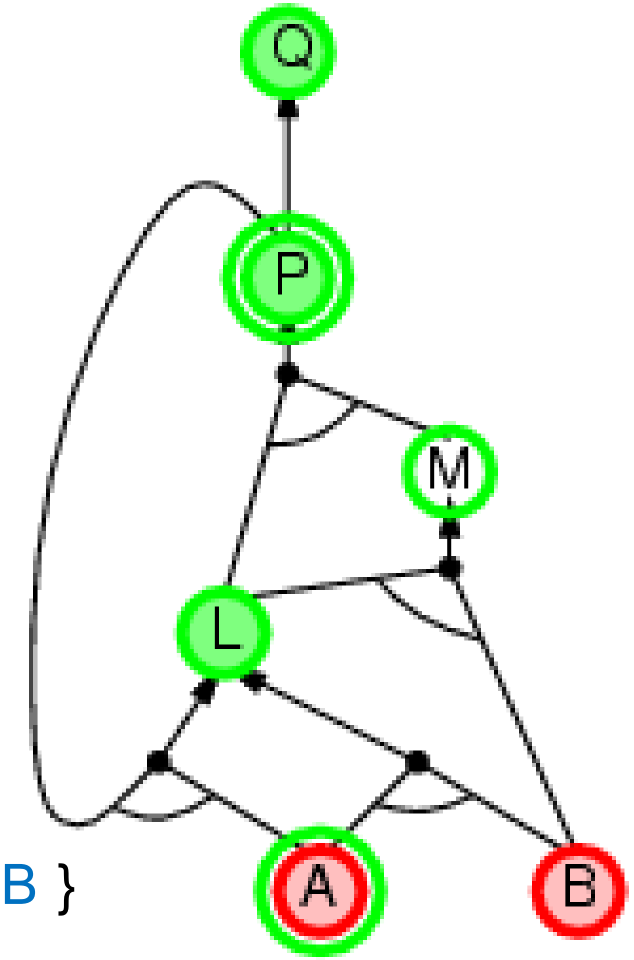
$\boxed{P \Rightarrow Q}$
$\boxed{L \wedge M \Rightarrow P}$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
A
B



Subgoals = { Q, P, L, M }

Backward chaining example

$\boxed{P \Rightarrow Q}$
$\boxed{L \wedge M \Rightarrow P}$
$B \wedge L \Rightarrow M$
$\boxed{A \wedge P \Rightarrow L}$
$\boxed{A \wedge B \Rightarrow L}$
A
B



Subgoals = { Q, P, L, M, A, B }

Backward chaining example

$\boxed{P \Rightarrow Q}$

$\boxed{L \wedge M \Rightarrow P}$

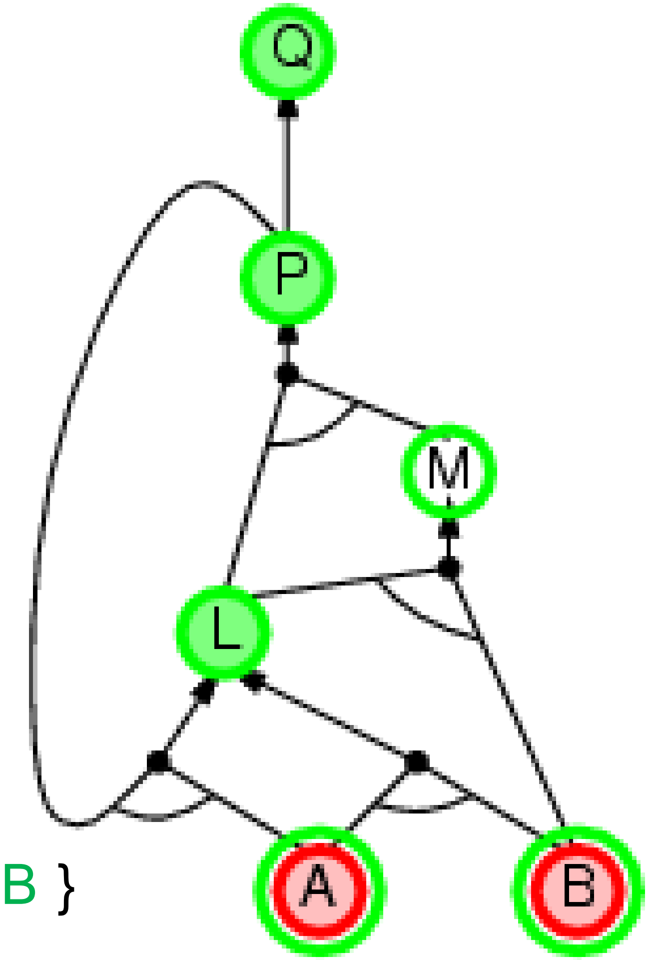
$B \wedge L \Rightarrow M$

$\boxed{A \wedge P \Rightarrow L}$

$\boxed{A \wedge B \Rightarrow L}$

A

B



Subgoals = { Q, P, L, M, A, B }

Backward chaining example

$\boxed{P \Rightarrow Q}$

$\boxed{L \wedge M \Rightarrow P}$

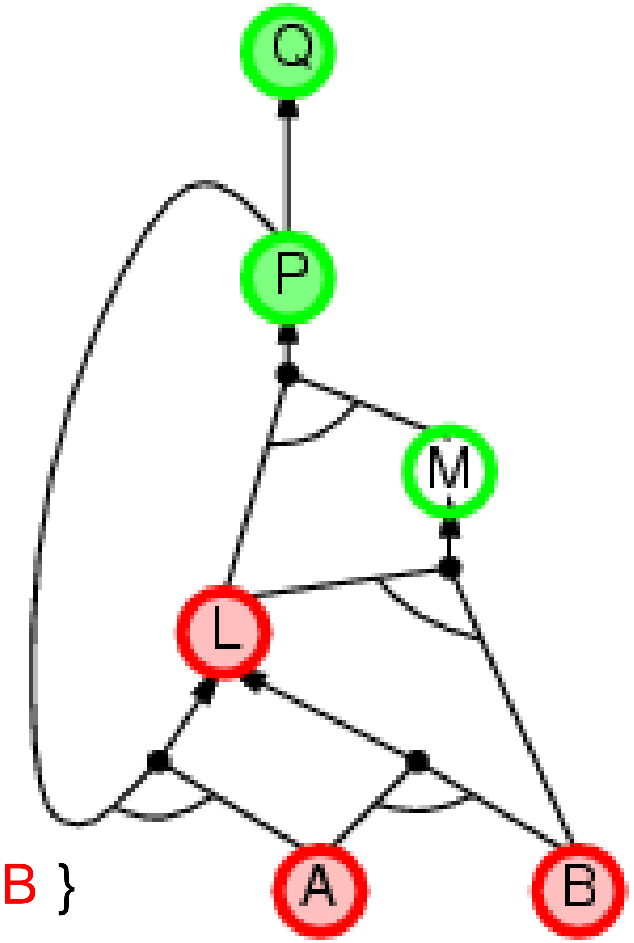
$B \wedge L \Rightarrow M$

$\boxed{A \wedge P \Rightarrow L}$

$\boxed{A \wedge B \Rightarrow L}$

A

B



Subgoals = { Q, P, L, M, A, B }

Backward chaining example

$\boxed{P \Rightarrow Q}$

$\boxed{L \wedge M \Rightarrow P}$

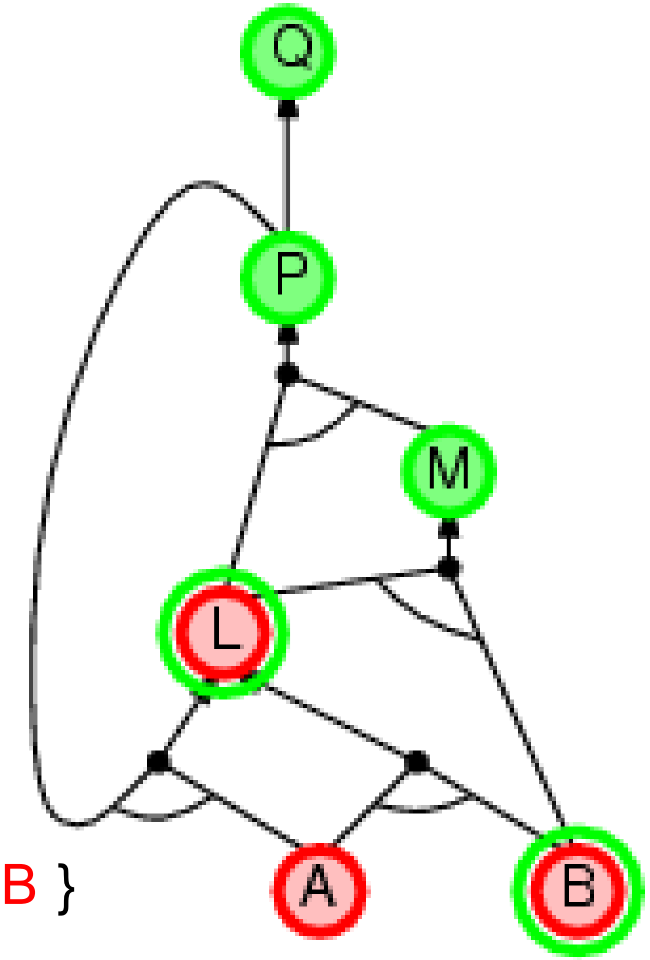
$\boxed{B \wedge L \Rightarrow M}$

$\boxed{A \wedge P \Rightarrow L}$

$\boxed{A \wedge B \Rightarrow L}$

A

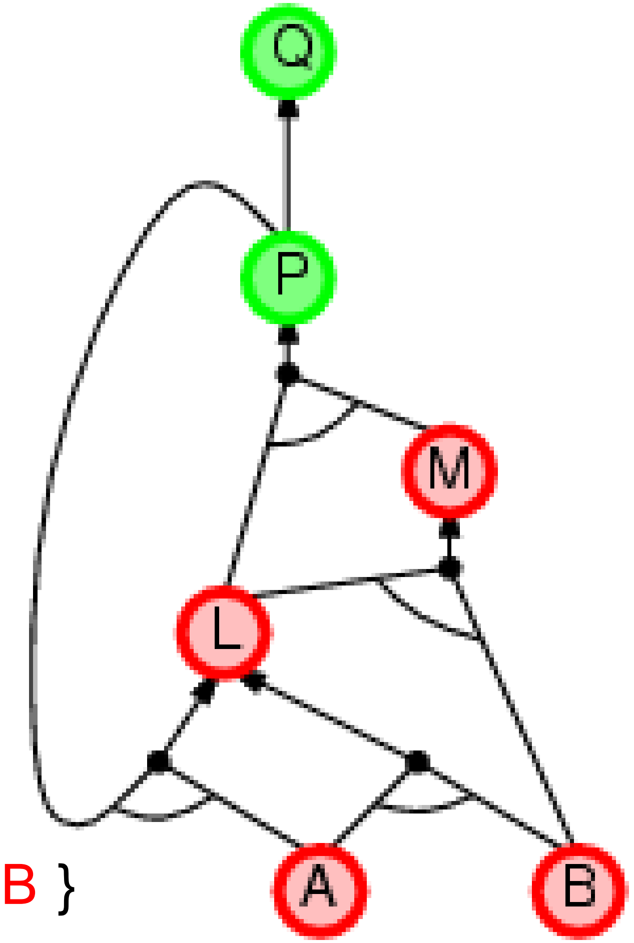
B



Subgoals = { Q, P, L, M, A, B }

Backward chaining example

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



Subgoals = { Q, P, L, M, A, B }

Backward chaining example

$\boxed{P \Rightarrow Q}$

$\boxed{L \wedge M \Rightarrow P}$

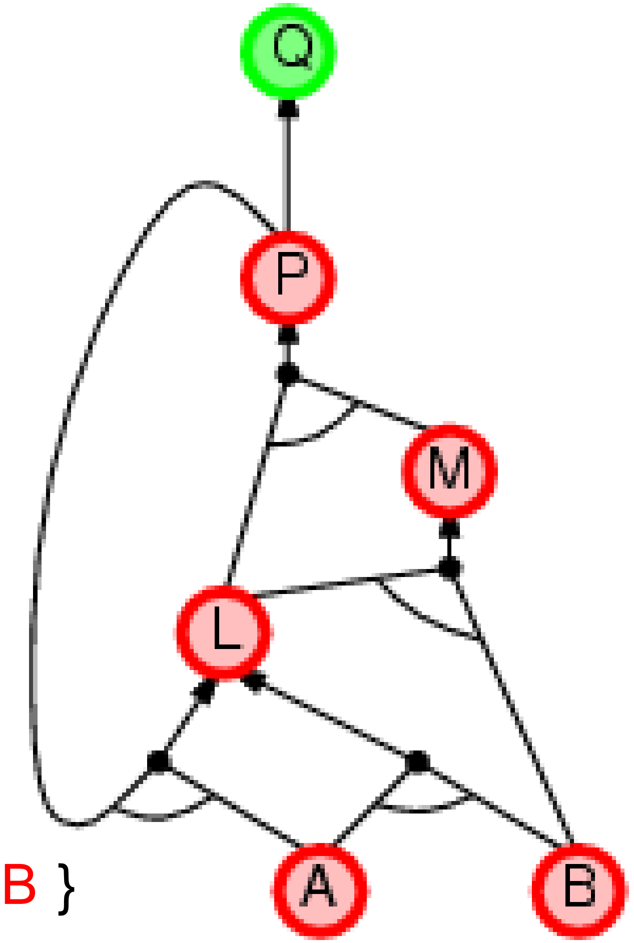
$\boxed{B \wedge L \Rightarrow M}$

$\boxed{A \wedge P \Rightarrow L}$

$\boxed{A \wedge B \Rightarrow L}$

A

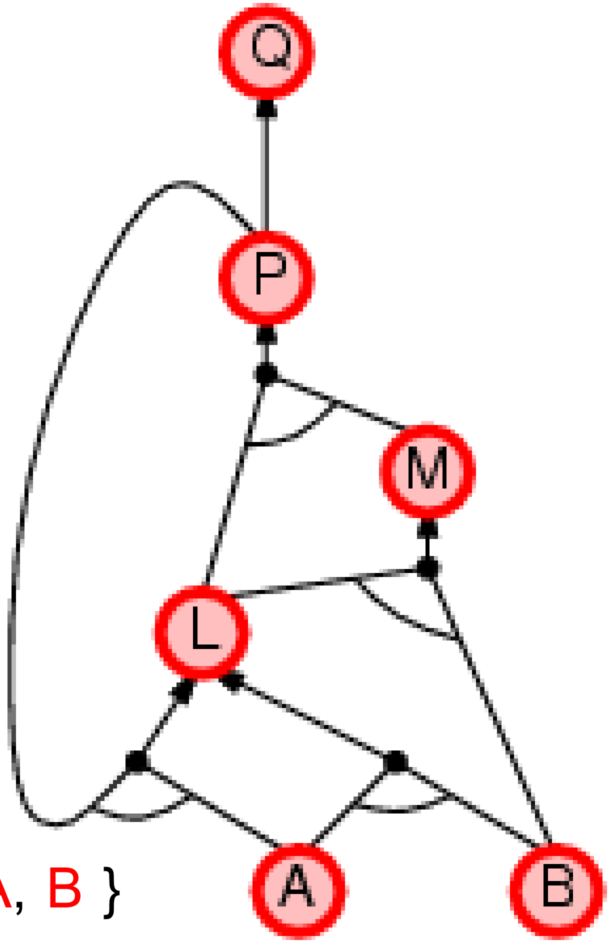
B



Subgoals = { Q, P, L, M, A, B }

Backward chaining example

$\boxed{P \Rightarrow Q}$
$\boxed{L \wedge M \Rightarrow P}$
$\boxed{B \wedge L \Rightarrow M}$
$\boxed{A \wedge P \Rightarrow L}$
$\boxed{A \wedge B \Rightarrow L}$
A
B



Subgoals = { Q✓, P, L, M, A, B }

Forward vs. Backward Chaining

- Forward chaining is data-driven reasoning
  - Start with known data/facts
  - Good for deriving new conclusions for incoming facts
  - Can create unnecessary works, need to keep track of which conclusions are useful/desired
- Backward chaining is goal-driven reasoning
  - The process only touches relevant facts – much less cost
  - Suitable of specific questions
    - “Should I retire?”
    - “Where are my keys?”

Resolution

- Logic Representation: Use conjunctive normal form (CNF)
  - Conjunction ($\wedge$) of disjunctions ($\vee$)
  - Example: $(A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$
- Use resolution reference rule

$$\frac{l_1 \vee \ldots \vee l_k, \quad m_1 \vee \ldots \vee m_n}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n}$$

  - When $l_i = \neg m_j$ these two literals in effect eliminate each other
- Example

$$\frac{P_{1,3} \vee P_{2,2} \quad , \neg P_{2,2}}{P_{1,3}}$$

- Sound and complete

Converting Sentences to CNF

Example: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

1. Eliminate $\Leftrightarrow$, by replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$

   $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$

2. Eliminate $\Rightarrow$, by replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.

   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$

3. Distribute $\neg$ inside the parentheses with de Morgan's rules and Double-negation:

   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$

4. Use distributivity law to distribute $\wedge$ inside the parentheses, and $\vee$ outside the parentheses :

   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

Resolution Algorithm

- To prove α
- Use proof by contradiction: prove KB $\models \alpha$ by showing that we cannot satisfy (KB $\land \neg\alpha$)

1. Add $\neg\alpha$ into KB to get (KB $\land \neg\alpha$)
2. Convert (KB $\land \neg\alpha$) into CNF
3. Use resolution inference rule and add the results into KB until:
    1. Resolve into empty clause $\rightarrow$ KB $\models \alpha$
    2. No more clauses to be added $\rightarrow$ KB does not entail α

Resolution Algorithm

function PL-RESOLUTION(*KB*, α) **returns** *true* or *false*

  inputs: *KB*, the knowledge base, sentences in propositional logic

  α, the query, a sentence in propositional logic

  clauses ⟵ the set of clauses in the CNF representation of *KB* $\wedge \neg\alpha$

  new ⟵ {}

  loop do

    for each pair of clauses C_i, C_j **in** *clauses* **do**

      resolvents ⟵ PL-RESOLVE(C_i, C_j)

      if *resolvents* contains the empty clause **then return** *true*

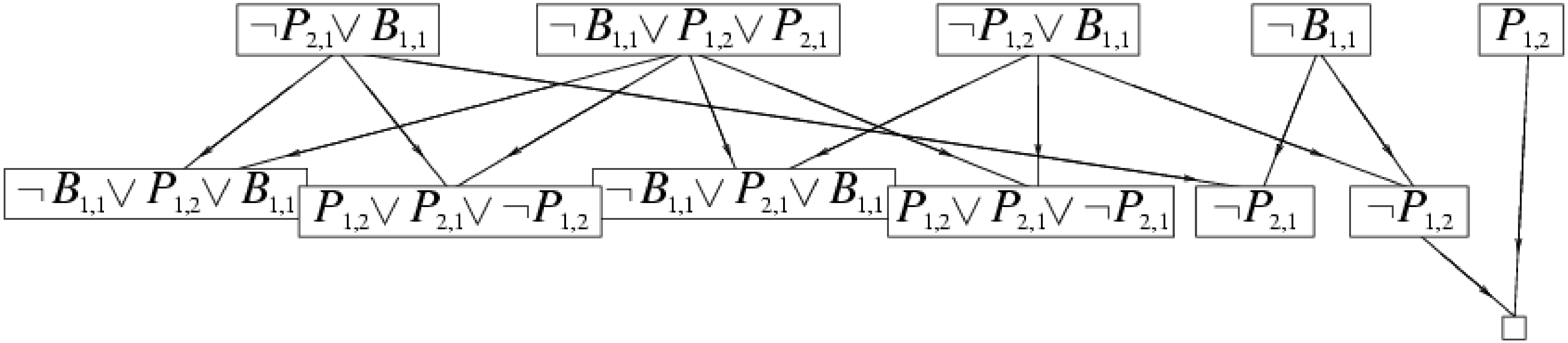
      new ⟵ *new* ∪ *resolvents*

    if *new* ⊆ *clauses* **then return** *false* #no new clause

    clauses ⟵ *clauses* ∪ *new*

Example of Resolution

- $KB = (B_{1,1} \Leftrightarrow (P_{1,2} \lor P_{2,1})) \land \neg B_{1,1}$ $\alpha = \neg P_{1,2}$



Limits of Propositional Logic

- No variable, so you will need a rule for each object
- Example (Wumpus' world): you will need a rule for the existence of pit in a square
  - $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
  - $B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{1,3} \vee P_{2,2})$
  - …
- Need a more expressive language

First-order Logic

First-order Logic

- First-order logic (FOL) consists of:
  - Objects: nouns and noun phrases in the problem domain
    - Examples: people, houses, numbers, colors, baseball games, wars, …
  - Relations: verbs, verb phrases, adjectives and adverbs that refer to relation among objects.
    - Examples: red, round, prime, brother of, bigger than, part of, comes between, …
  - Functions: relations that represents an object
    - Examples: father of, best friend, one more than, +, …
  - Which explain domain elements in a problem domain
- Each symbol in FOL needed to be given interpretation: what it means

Syntax of First-order Logic

Components

- Constants: *Tom, Bob, 2*
- Variables (lowercase): x, y
- Predicates: *HasCold(Steven)*
- Functions: +, *Father*(*Steven*)
- Connectives: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$
- Equality: $=$
- Quantifiers: $\forall$, $\exists$

Sentence	$\rightarrow$	*AtomicSentence* | *Sentense*
AtomicSentence	$\rightarrow$	*Predicate* | *Predicate*(Term,…) | *Term* = *Term*
ComplexSentence	$\rightarrow$	(*Sentence*)
	|	$\neg$ *Sentence*
	|	*Sentence* $\wedge$ *Sentence*
	|	*Sentence* $\vee$ *Sentence*
	|	*Sentence* $\Rightarrow$ *Sentence*
	|	*Sentence* $\Leftrightarrow$ *Sentence*
	|	*Quantifier Variable,… Sentence*

Term	$\rightarrow$	*Function*(*Term*,…)
	|	*Constant*
	|	*Variable*
Quantifier	$\rightarrow$	$\forall$ | $\exists$
Constant	$\rightarrow$	*A* | X_1 | *John* | …
Variable	$\rightarrow$	*a* | *x* | *s* | …
Predicate	$\rightarrow$	*True* | *False* | *After* | *Loves* | *HasCold*
Function	$\rightarrow$	*Mother* | *LeftLeg* | …
Operator Precedence	:	$\neg$, =, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$

Function vs. Predicate

- Function symbol represents a function will return an object

  Advisor(John) = Tom

- Predicate symbol represents relation, and will return a truth value (true/false/unknown)

  IsAdvisor(ProfRoberts, John) = false

- Both will have arity: number of arguments they can take

Terms & Quantifiers

- Terms include everything that represent objects
  - Constants, variables, functions
  - A term with no variable is called a ground term
- Quantifiers are symbols for expressing properties for the entire collections of objects
  - We have universal quantification (for all… , $\forall$) and existential quantification (for some… , $\exists$)

Quantifiers (cont.)

- Quantifiers can be nested, but order of nesting is important
  - Considering *FBFriend*(*x*, *y*) to mean "*x* is a Facebook friend with *y*", the compared the following sentences

$$\forall x\ \exists y\ FBFriend(x, y)$$

$$\exists y\ \forall x\ FBFriend(x, y)$$

- Quantifiers and negations

$$\neg\forall x P(x) \equiv \exists x \neg P(x)$$
$$\neg\exists x P(x) \equiv \forall x \neg P(x)$$

Assumptions:

- Unique-names Assumption: every constant symbol refer to a distinct object Object
- Close World Assumption: atomic sentences not known to be (or, cannot proven to be) true are in fact false
- Domain Closure: each model contains no more domain elements than those named by the constant symbols

Using First-order Logic

- FOL KB can be updated using assertion, adding new sentence into KB and retraction, removing sentence from KB
- KB can be asked with queries or goals
  - And the answer can be *true*/*false* of substitution list, list of variable substitution to make the goal true

Example: Wumpus World

- Precept predicate can be used to describe what is perceive at the time

$$Percept([Stench, Breeze, Glitter, None, None])$$

- And can be used to check current status at time t

$$\forall t, s, g, m, c\ Percept([s, Breeze, g, m, c], t) \Rightarrow Breeze(t)$$

$$\forall t, s, b, m, c\ Percept([s, b, Glitter, m, c], t) \Rightarrow Glitter(t)$$

- Also used for reflex behavior

$$\forall t\ Glitter(t) \Rightarrow Grab(t)$$

Example: Wumpus World (cont.)

- Can also be used to explain more complex relationship, for example, whether the location [*x*, *y*] is adjacent to location [*a*, *b*]

$$\forall x, y, a, b\ Adjacent([x, y], [a, b]) \Leftrightarrow \big(x = a \land (y = b - 1 \lor y = b + 1)\big) \lor \big(y = b \land (x = a - 1 \lor x = a + 1)\big)$$

- And explain the relationship between location (s) and time (t). $At(s, t)$. Note that variable of the same name on different sentences can represent different types of objects altogether

$$\forall s, t\ At(s, t) \land Breeze(t) \Longrightarrow Breezy(s)$$

Example: Wumpus World (cont.)

- Then, we use those to explain the relationship between pit and breeze. Variable *r* and *s* represent locations

$$\forall s\ Breezy(s) \Leftrightarrow \exists r Adjacent(r, s) \land Pit(r)$$

- Need fewer rules than propositional logic

Knowledge Engineering in FOL

- Or, how to create a knowledge base in FOL form

1. Identify the questions
   - All questions that can asked in the problem domain
   - KB should have enough knowledge to cover those questions
2. Assemble the relevant knowledge – knowledge acquisition
   - Not yet represent formally (by FOL, in this case)
3. Decide on a vocabulary of predicates, functions, and constants
   - Need to identify all symbols in the problem domain
   - Create ontology: determines what kinds of things exist, but does not determine their specific properties and interrelationships

Knowledge Engineering in FOL (cont.)

4. Encode general knowledge about the domain
   - Meaning of symbols
   - General relationship
   - May need to include common knowledge not specified in the problem statement
5. Encode a description of the problem instance
   - Writing simple sentences about instances of concepts that are already part of the ontology
6. Pose queries to the inference procedure and get answers
7. Debug and evaluate the knowledge base

Example – Criminal Problem

"The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

is Colonel West a criminal?"

- Part 1 – 2 are done.

Example – Criminal Problem (cont.)

3. Deciding on a vocabulary (and their meaning, which is part 4.)

- Constants: West, America, Nono
- Predicates:
  - *American*(x) – x is an American
  - *Criminal*(x) – x is a criminal
  - *Enemy*(x, y) – x is an enemy of y (to America)
  - *Hostile*(x) – x is a hostile nation
  - *Missile*(x) – x is a missile
  - *Owns*(x, y) – x owns y
  - *Weapon*(x) – x is a weapon
  - *Sells*(x, y, z) – x sells z y

Example – Criminal Problem (cont.)

4. Encode general knowledge about the domain

- (Common knowledge) “A missile is a weapon”

$$\forall x\ Missile(x) \Rightarrow Weapon(x)$$

- (Common knowledge) “An enemy of America is a hostile nation”

$$\forall x,y\ Enemy(x, America) \Rightarrow Hostile(x)$$

- “The law says that it is a crime for an American to sell weapons to hostile nations”

$$\forall x,y,z\ American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$$

Example – Criminal Problem (cont.)

5. Encode a description of the problem instance

- “The country Nono is an enemy of America”

$$Enemy(Nono, America)$$

- “The country Nono has some missiles” (use conjunction for $\exists$)

$$\exists y\ Missile(y) \land Owns(Nono, y)$$

- “Colonel West is American”

$$American(West)$$

- “All of Nono’s missiles were sold to it by Colonel West”

$$\forall x\ Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$$

Example – Criminal Problem (cont.)

We now have a KB

- $American(West)$
- $Enemy(Nono, America)$
- $\forall x\ Missile(x) \Rightarrow Weapon(x)$
- $\forall x,y\ Enemy(x, America) \Rightarrow Hostile(x)$
- $\forall x,y,z\ American(x) \land Weapon(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x)$
- $\forall x\ Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$
- $\exists y\ Missile(y) \land Owns(Nono, y)$

Inference with FOL

- Create new sentences from existing KB ($KB \models \alpha_1$)
  - Either just true/false answer, or substitution list
- To use inference technique mentioned earlier, FOL sentences need to be converted to propositional logic first
  - Need to substitute all variables, or dealing with quantifiers

Dealing with Quantifiers

- Substitute variables to get rid of quantifiers
- SUBST($\{v/g\}$, α), substitutes ground term g into variable v in sentence α
- UNIFY
  - UNIFY(p, q) = θ when SUBST(θ, p) = SUBST(θ, q)
  - θ is a unifier between p and q
    - θ is a substitution list that will make sentences p and q turn into the same sentence

Unification Algorithm

```
function UNIFY(x , y, θ) returns a substitution to make x and y identical
    inputs:     x , a variable, constant, list, or compound expression (F(x))
                y, a variable, constant, list, or compound expression
                θ, the substitution built up so far (optional, defaults to empty)
    if θ = failure then return failure
    else if x = y then return θ
    else if VARIABLE?(x) then return UNIFY-VAR(x , y, θ)
    else if VARIABLE?(y) then return UNIFY-VAR(y, x , θ)
    else if COMPOUND?(x) and COMPOUND?(y) then
        return UNIFY(x .ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
    else if LIST?(x) and LIST?(y) then
        return UNIFY(x .REST, y.REST, UNIFY(x .FIRST, y.FIRST,θ))
    else return failure

function UNIFY-VAR(var , x ,θ) returns a substitution
    if {var/val} ∈ θ then return UNIFY(val, x ,θ)
    else if {x/val} ∈ θ then return UNIFY(var, val, θ)
    else if OCCUR-CHECK?(var, x ) then return failure
    else return add {var/x} to θ
```

In case of compound F(s, t)

F(s, t).OP = F

F.ARGS = {s, t}

OCCUR-CHECK will check whether a term exists in another term, such as $S(x)$ in $S(S(x))$

SUBST and UNIFY Examples

- SUBST(*{x/Mary}, Knows(John,x)) = Know(John, Mary)*

- *UNIFY(Knows(John, x), Knows(John, Jane)) = {x/Jane}*
- *UNIFY(Knows(John, x), Knows(y, Bill)) = {x/Bill, y/John}*
- *UNIFY(Knows(John, x), Knows(y, Mother (y))) = {y/John, x/Mother (John)}*
- *UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail*

Dealing with Quantifiers (cont.)

- Universal Instantiation (UI)

$$\frac{\forall v\ \alpha}{\mathrm{SUBST}(\{v/g\}, \alpha)}$$

- Where g is a ground term
- Creating new sentence
- Can be done multiple time on α with different ground term

Dealing with Quantifiers (cont.)

Example of UI

- If in KB, we have

  $\forall x\ CMUStudent(x) \Rightarrow GoodLooking(x)$

  and the Constants: *Ken*, *Toon*, *Yo*

- UI will be able to create the following:

  $CMUStudent(Ken) \Rightarrow GoodLooking(Ken)$

  $CMUStudent(Toon) \Rightarrow GoodLooking(Toon)$

  $CMUStudent(Yo) \Rightarrow GoodLooking(Yo)$

Dealing with Quantifiers (cont.)

- Existential Instantiation (EI)

$$\frac{\exists v\ \alpha}{\mathrm{SUBST}(\{v/k\}, \alpha)}$$

- *k* is a Skolem Constant, a constant that **<u>does not appear elsewhere in the KB</u>**
- Example:
  - KB: $\exists x\ Car(x) \land OwnBy(x, John)$
  - EI can yield $Car(C1) \land OwnBy(C1, John)$
    - If C1 does not appear in KB before

Dealing with Quantifiers (cont.)

- UI can be used multiple time to the same sentence, adding new sentences
  - The new KB is equivalent to the old KB
- EI can only be used once per one sentence, replacing the sentence
  - ∃ can then be discarded
  - The new KB is **NOT** equivalent to the old KB
  - But the new KB is only satisfiable **iff** the old KB is satisfiable

Inference Example

- KB
  1. StudyHard(John)
  2. StudyHard(Tom)
  3. Sick(Tom)
  4. $\neg$Sick(John)
  5. $\forall$x StudyHard(x) $\wedge\neg$Sick(x) $\Rightarrow$ GetA(x)
- Query

  GetA(*John*)

- Inference

1) SUBST({x/John},5) $\rightarrow$

6. StudyHard(John) $\wedge\neg$Sick(John) $\Rightarrow$ GetA(John)

2) From 1, 2, and 6 use modus ponens:

   GetA(John)

Answer: True

Dealing with Quantifiers (cont.)

- Generalized Modus Ponens
  - For atomic sentences p_i, p_i' and q
  - Where there is a substitution θ such that SUBST(θ, p_i') = SUBST(θ, p_i), for all i

$$\frac{p_1', p_2', \ldots, p_n', \quad (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\mathrm{SUBST}(\theta, q)}$$

- This can be use for forward and backward chaining in FOL

Preparing FOL KB for Generalized Modus Ponens

- Need to convert sentence into Horn form
  - Convert sentences to implication $\Rightarrow$
  - Use EI to get rid of $\exists$
  - $\forall$ can be omitted
- Example: Criminal Problem
  - For sentence $\exists y \; Missile(y) \land Owns(Nono, y)$
  - Use EI and replace y with Skolem constant $M1$
  - Result: $Missile(M1) \land Owns(Nono, M1)$ – 2 sentences

Converted Criminal Problem KB

- $American(West)$
- $Enemy(Nono, America)$
- $Missile(x) \Rightarrow Weapon(x)$
- $Enemy(x, America) \Rightarrow Hostile(x)$
- $American(x) \land Weapon(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x)$
- $Missile(x) \land Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$
- $Missile(M1)$
- $Owns(Nono, M1)$

Inference in FOL – Forward Chaining

- Data-driven
  - Start with existing sentences in KB
  - Use generalized modus ponens to create new sentences, until
    1. Goal is created (succeed)
    2. No sentences can be created (fail)
  - If succeed, return substitution list Θ that prove the goal

function FOL-FC-ASK(KB, α) **returns** a substitution or *false*

inputs: KB, the knowledge base, a set of first-order definite clauses

α, the query, an atomic sentence

local variables: *new*, the new sentences inferred on each iteration

Change the name of variables, so unrelated variables do not share a name

repeat until *new* is empty

new ⟵ {}

for each *rule* **in** *KB* **do**

$(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \longleftarrow$ STANDARDIZE-VARIABLES(*rule*)

for each θ such that SUBST(θ, $p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta$, $p'_1 \wedge \ldots \wedge p'_n$) for some $p'_1, \ldots, p'_n$ in KB

$q' \longleftarrow$ SUBST(θ, q)

if q' does not unify with some sentence already in *KB* or *new* **then**

add q' to *new*

$\phi \longleftarrow$ UNIFY(q', α)

if ϕ is not *fail* **then return** ϕ

add *new* to *KB*

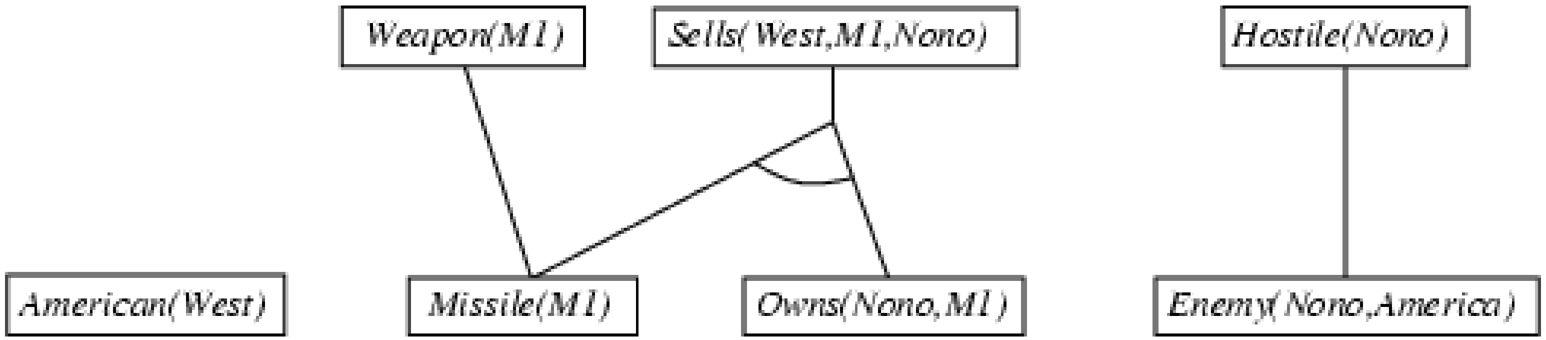
return *false*

Russell, Peter Norvig Stuart (2009-12-01). Artificial Intelligence: A Modern Approach, 3/e (Page 332). Pearson HE, Inc.. Kindle Edition.

Example of Forward Chaining

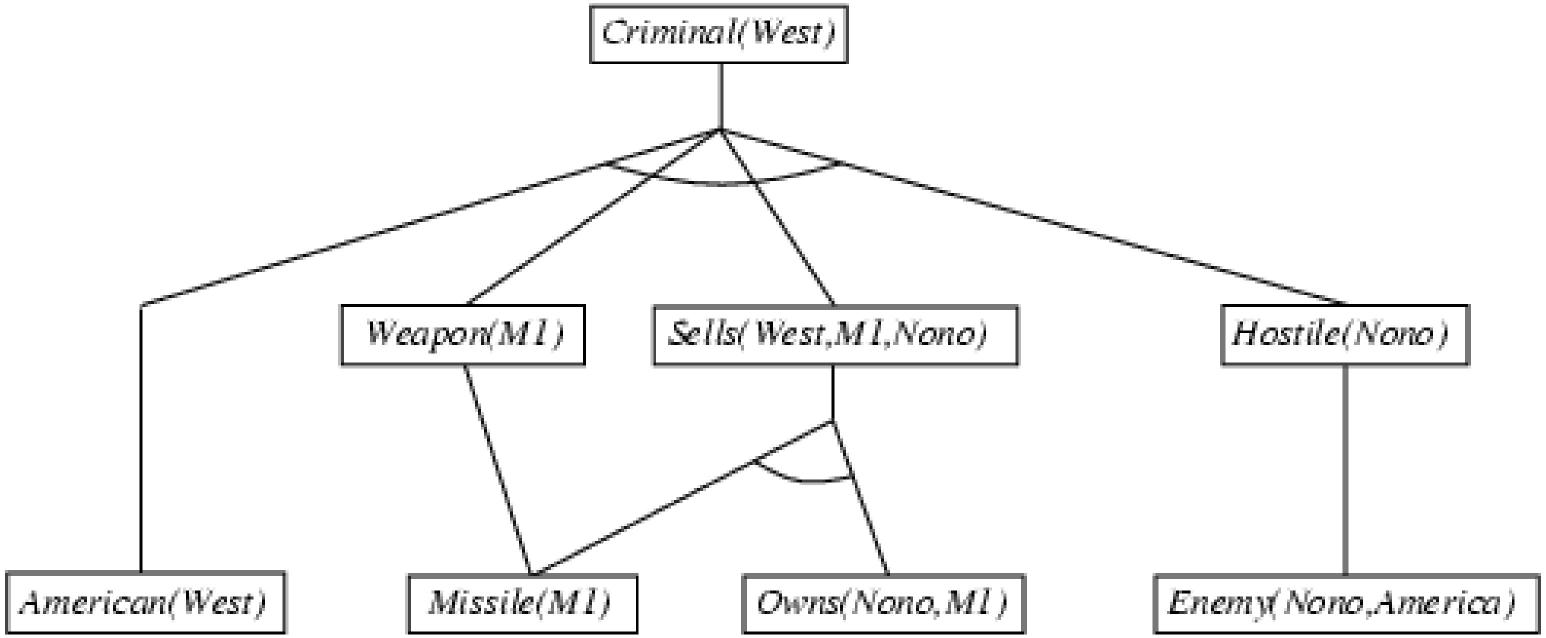
Source: AIMA

Example of Forward Chaining (cont.)



Source: AIMA

Example of Forward Chaining (cont.)



Source: AIMA

Inference in FOL – Backward Chaining

- Goal-driven
  - Start from goal and find implication with the target as conclusion

$$P \Longrightarrow Goal$$

  - The recursively trying to prove that the premise P is true
  - If succeed, return substitution list Θ that prove the goal

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
    return FOL-BC-OR(KB, query, { })

generator FOL-BC-OR(KB, goal, θ) yields a substitution
    for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
        (lhs, rhs) ⟵ STANDARDIZE-VARIABLES((lhs, rhs))
        for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
            yield θ'

generator FOL-BC-AND(KB, goals, θ) yields a substitution
    if θ = failure then return
    else if LENGTH(goals) = 0 then yield θ
    else do
        first, rest ⟵ FIRST(goals), REST(goals)
        for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
            for each θ'' in FOL-BC-AND(KB, rest, θ' ) do
                yield θ''
```

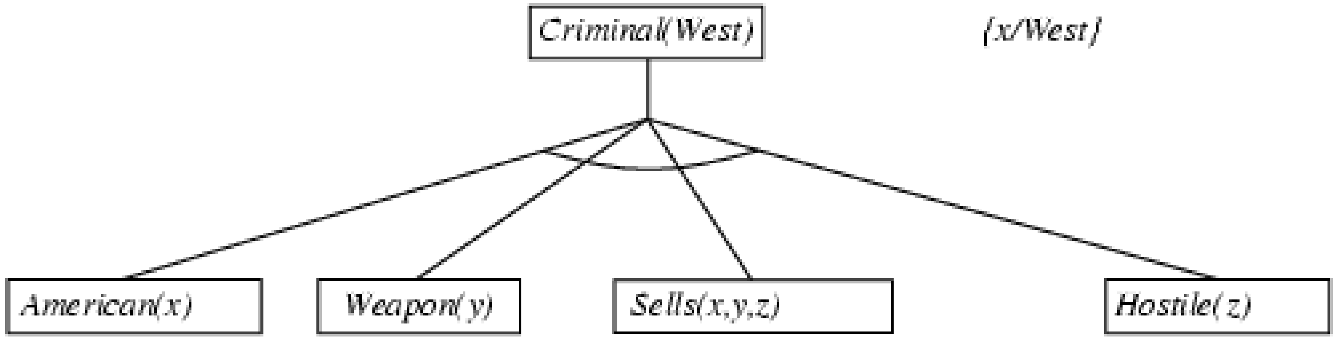
Source: AIMA

Backward chaining example

$\boxed{Criminal(West)}$

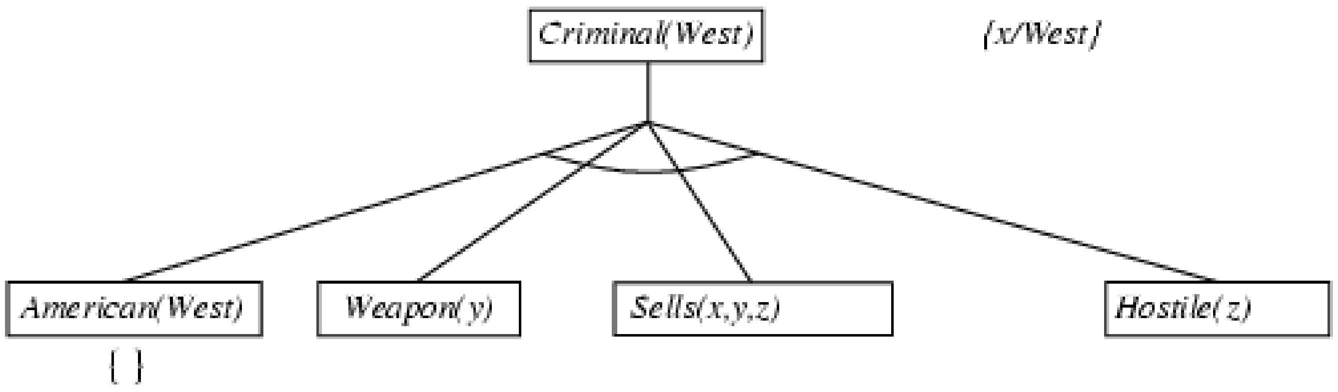
Source: AIMA

Backward chaining example



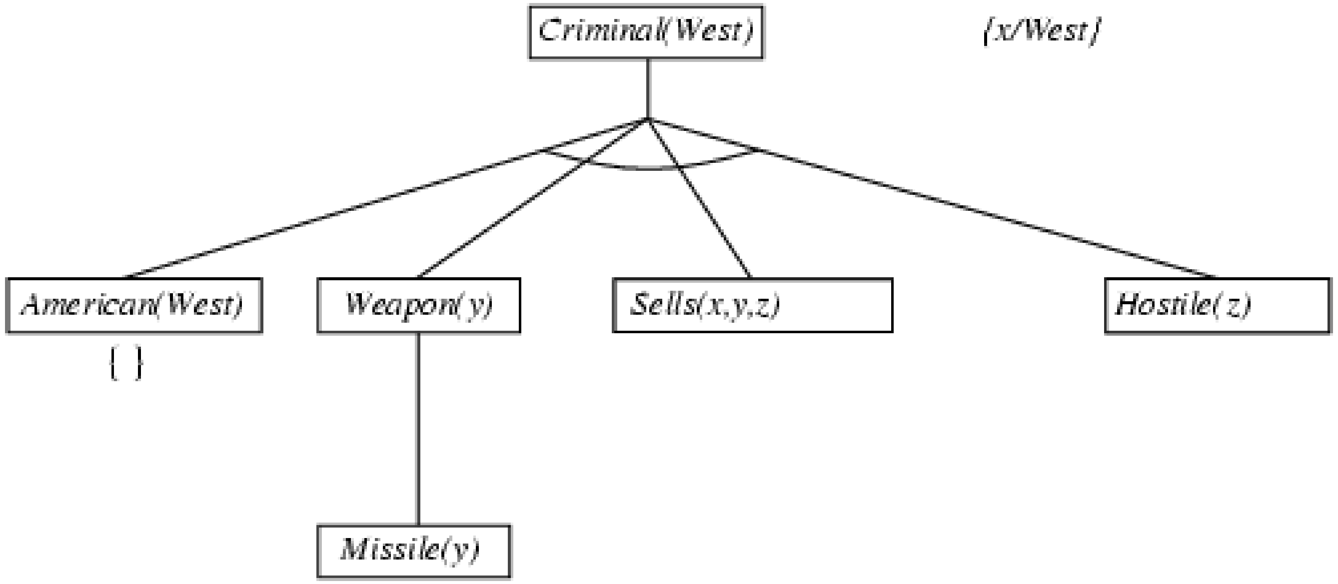
Source: AIMA

Backward chaining example



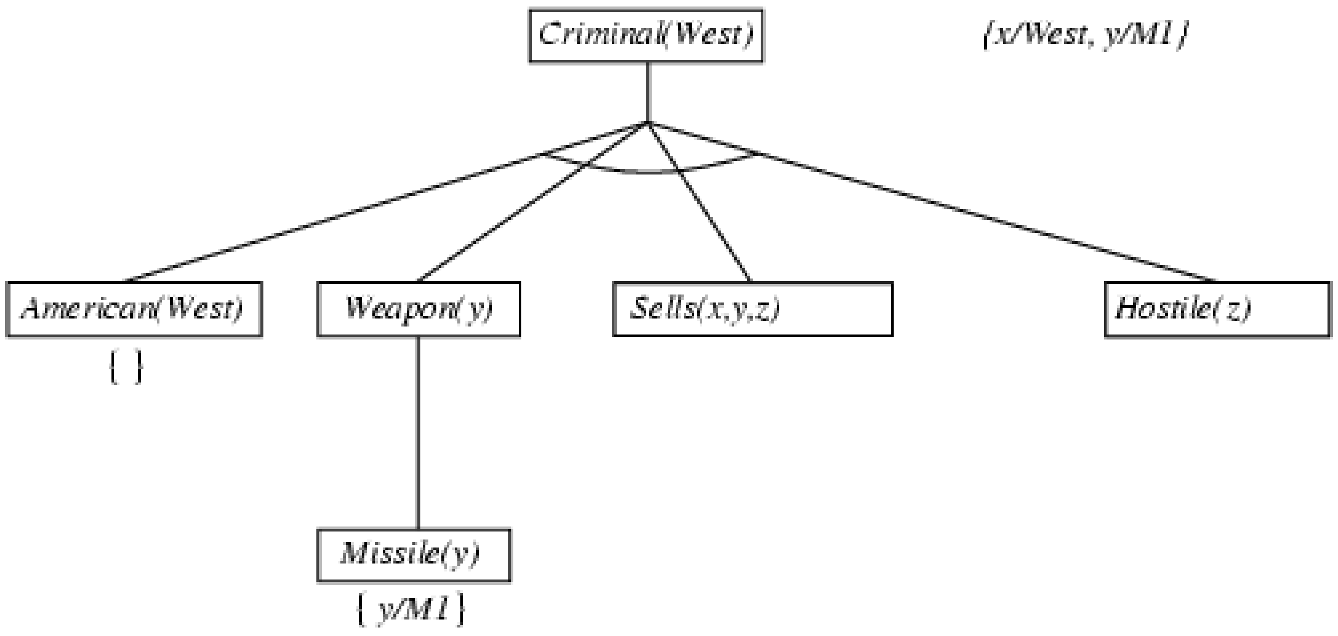
Source: AIMA

Backward chaining example



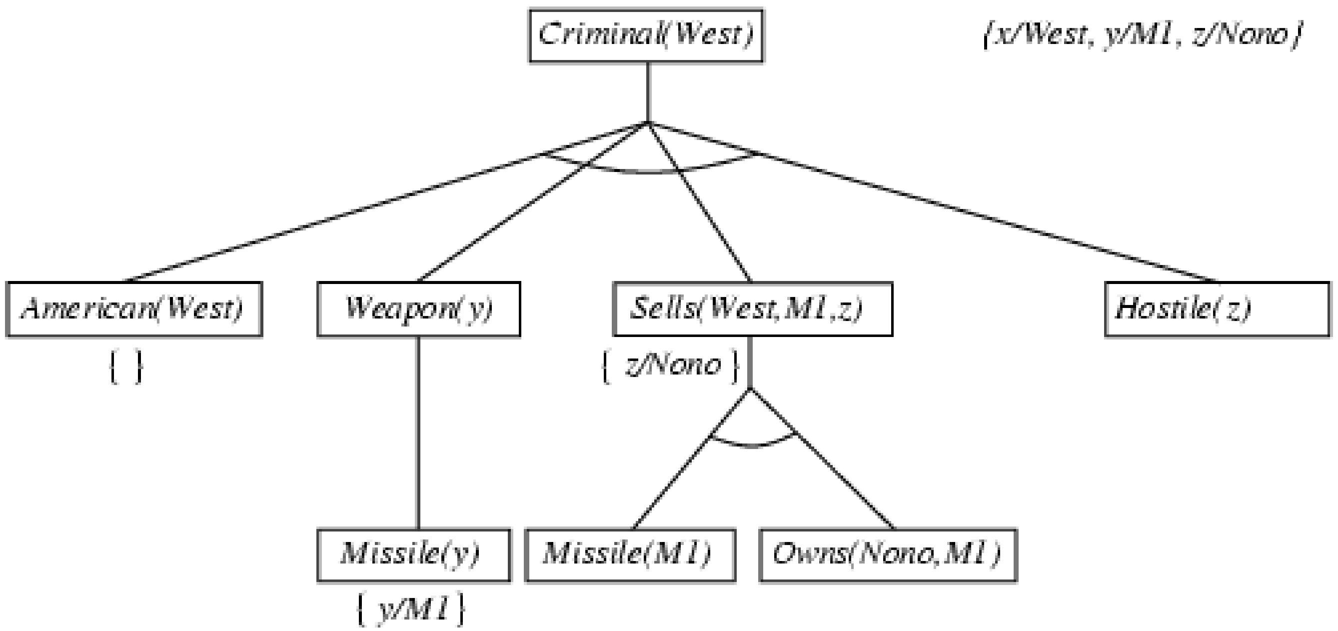
Source: AIMA

Backward chaining example



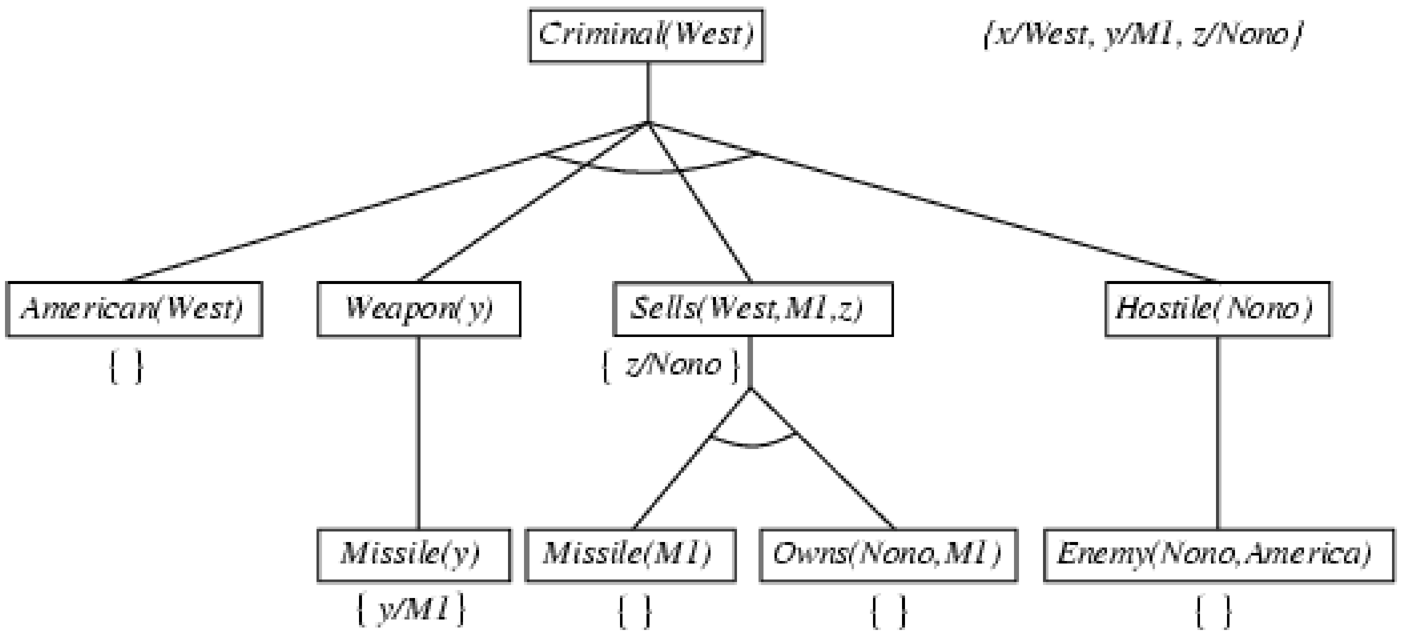
Source: AIMA

Backward chaining example



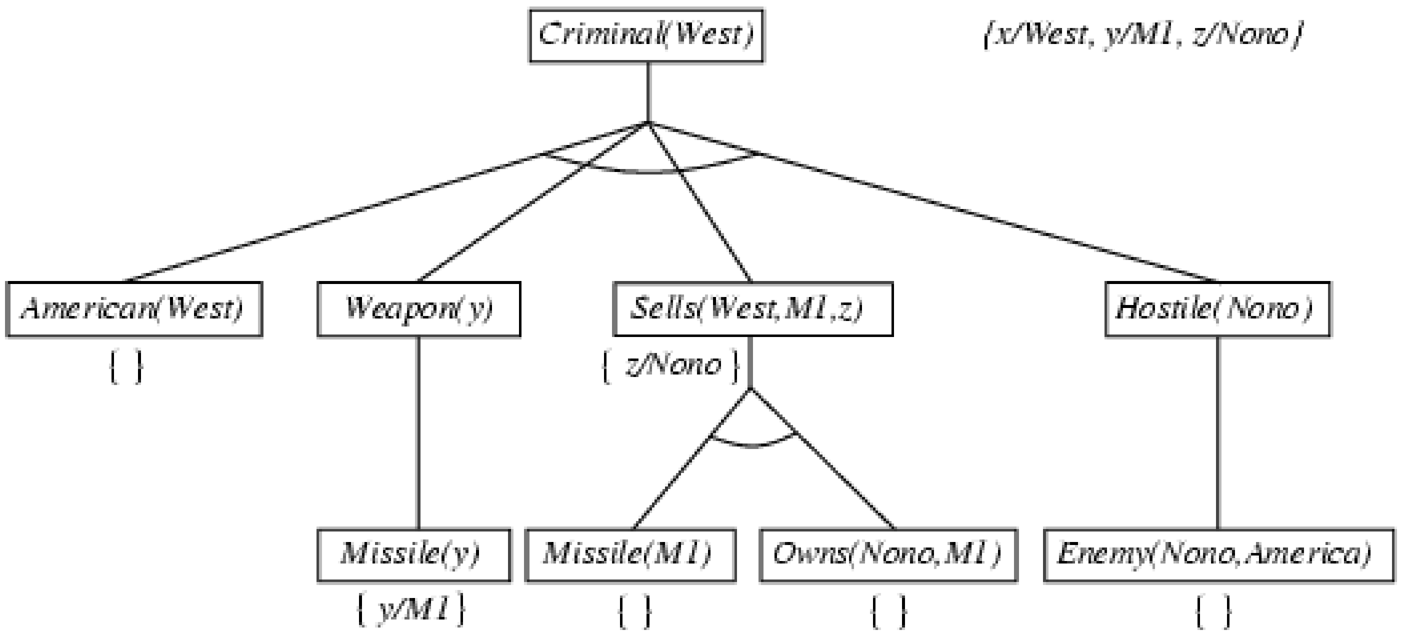
Source: AIMA

Backward chaining example



Source: AIMA

Backward chaining example



Source: AIMA

Resolution: Brief Summary

- Full first-order version:

$$\frac{l_1 \vee \ldots \vee l_k \quad m_1 \vee \ldots \vee m_n}{SUBST(\theta, l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n)}$$

  When Unify(l_i, $\neg m_j$) = θ.

  - The sentences are standardized ⟶ no shared variables
- Example:

$$\frac{\neg Rich(x) \vee Unhappy(x) \quad Rich(Ken)}{Unhappy(Ken)}$$

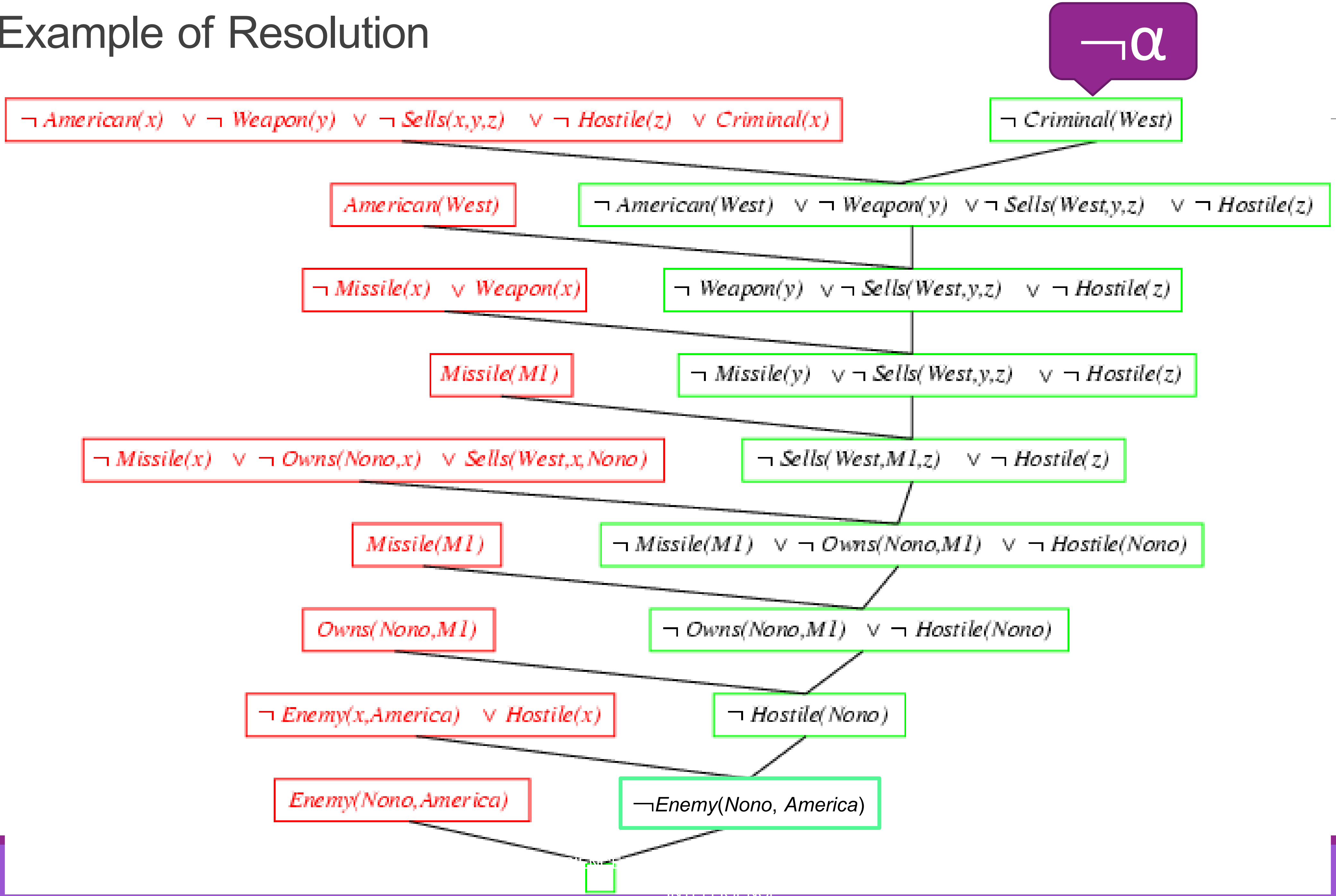
  with θ = {x/Ken}

- Usage: perform Resolution on $CNF(KB \wedge \neg\alpha)$ step-by-step – only work with conjunctive normal form (CNF)
  - If a contradiction (empty result) occur ⇒ KB ⊨ α ;
- Resolution is complete for FOL

Criminal Problem KB in CNF

- $American(West)$
- $Enemy(Nono, America)$
- $\neg\ Missile(x) \lor Weapon(x)$
- $\neg\ Enemy(x, America) \lor Hostile(x)$
- $\neg\ American(x) \lor \neg\ Weapon(y) \lor \neg\ Sells(x, y, z) \lor \neg\ Hostile(z) \lor Criminal(x)$
- $\neg\ Missile(x) \lor \neg\ Owns(Nono, x) \lor Sells(West, x, Nono)$
- $Missile(M1)$
- $Owns(Nono, M1)$

Example of Resolution



FOL Application: Logic Programming with Prolog

- Logic Programming is declarative programming instead of procedural programming
  - Describe the problem, and have inference engine answer the question
  - No need to write steps of solutions
- Need to:
  1. Describe the problem by Providing general rules and facts for KB
  2. Use the KB by asking question (query)
  3. Debug as needed

Prolog

- Is the most widely-used logic programming language
- Sentences are in first-order logic form
- Inference is done by backward chaining
- KB consists of:
  - Constant (Atom), Predicate – name starts with lowercase letter
  - Variables – name starts with capital letters (and underscore(_))
  - Sentences are either facts or rules, always end with full stop(.)

Prolog (cont.)

- Fact
  - Single predicate, ground sentence
  - Example: love(**sam**, **jill**)
- Rule
  - Implication, but written backward
    - From implication

$$love(X, Y) \wedge love(Y, X) \Rightarrow happy(X)$$

    - … to Prolog rule

```
happy(X) :- love(X, Y), love(Y, X).
```

Prolog Rules

- From: premise $\Rightarrow$ consequent
- To: consequent :- premise or, using Prolog's terms:

  head :- body
- If all terms in body are proven true, the term in head will also be proven true
- For connectives, use:
  - **:-** instead of $\Rightarrow$ (and in reverse)
  - **,** instead of $\wedge$
  - **;** instead of $\vee$

Example Prolog KB: Criminal Problem

```
owns(nono, m1).
american(west).
missile(m1).
enemy(nono, america).

criminal(X) :- american(X), weapon(Y), sells(X, Y, Z), hostile(Z).
sells(west, X, nono) :- missile(X), owns(nono, X).
weapon(X) :- missile(X).
hostile(X) :- enemy(X, america).
```

Skolem Constant

criminal_01.pl

Running Prolog

- After the KB is compiled (Compile $\longrightarrow$ Compile Buffer), a query can be asked
  - Can be question for yes/no (true/false) answers
  - Or the query can have variables, in that case the answer will be substitution list
- Example:

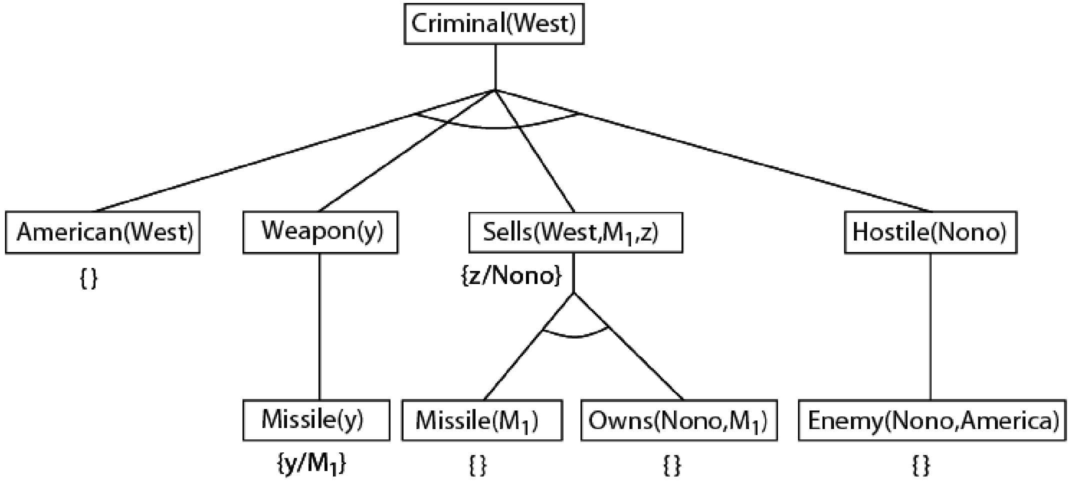
```
7 ?- criminal(west).
true.
8 ?- sells(west, X, Y).
X = m1,
Y = nono.
```

- A query can be continued with comma(,) to find another possible substitution list

Tracing

```
[trace] 6 ?- visible(+all), trace, criminal(west).
   Call: (8) visible(+all) ? creep
   Exit: (8) visible(+all) ? creep
   Call: (8) criminal(west) ? creep
   Unify: (8) criminal(west)
   Call: (9) american(west) ? creep
   Unify: (9) american(west)
   Exit: (9) american(west) ? creep
   Call: (9) weapon(_G4725) ? creep
   Unify: (9) weapon(_G4725)
   Call: (10) missile(_G4725) ? creep
   Unify: (10) missile(m1)
   Exit: (10) missile(m1) ? creep
   Exit: (9) weapon(m1) ? creep
   Call: (9) sells(west, m1, _G4727) ? creep
   Unify: (9) sells(west, m1, nono)
   Call: (10) missile(m1) ? creep
   Unify: (10) missile(m1)
   Exit: (10) missile(m1) ? creep
   Call: (10) owns(nono, m1) ? creep
   Unify: (10) owns(nono, m1)
   Exit: (10) owns(nono, m1) ? creep
   Exit: (9) sells(west, m1, nono) ? creep
   Call: (9) hostile(nono) ? creep
   Unify: (9) hostile(nono)
   Call: (10) enemy(nono, america) ? creep
   Unify: (10) enemy(nono, america)
   Exit: (10) enemy(nono, america) ? creep
   Exit: (9) hostile(nono) ? creep
   Exit: (8) criminal(west) ? creep
true.
```

- Trace command will show the search step in backward chaining
- For example, trace, criminal(west), visible(+all). Numbers in parentheses is the level of the step in the search tree.
- Use notrace to quit tracking



Other command

- Assertion – add a clause (fact or rule) into the KB
  - asserta(clause) assert the argument as first clause
  - assertz(clause) assert the argument as last clause
- retract(clause) – remove the clause from the KB
- listing(predicate)
  - Show facts or rules with predicate as head

```
?- listing(american).
american(west).

true.

?- listing(criminal).
criminal(X) :-
        american(X),
        weapon(Y),
        sells(X, Y, Z),
        hostile(Z).

true.
```

Recursion

- Recursion can be used (ancestor in the example) but be careful of infinite loop!
- Priority is given to upper clause, so put base case in the top.

```
mother_child(trude, sally).
father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).
ancestor(X, Y)     :- parent_child(X, Y).
ancestor(X, Z)     :- ancestor(X, Y), parent_child(Y, Z).
```

family_fixed.pl

FOL Application: Rule-based Expert System

- Expert system is a computer (AI usually) system that provide user with expertise, such as advices or actions
- Usually, the expert system maintain a knowledge base (KB) and use inference engine to infer the most response to the situation
  - A situation is determined by user's update of facts and query
- First-order logic can be used as KB for an expert system
  - Such system is called rule-based expert system or production system

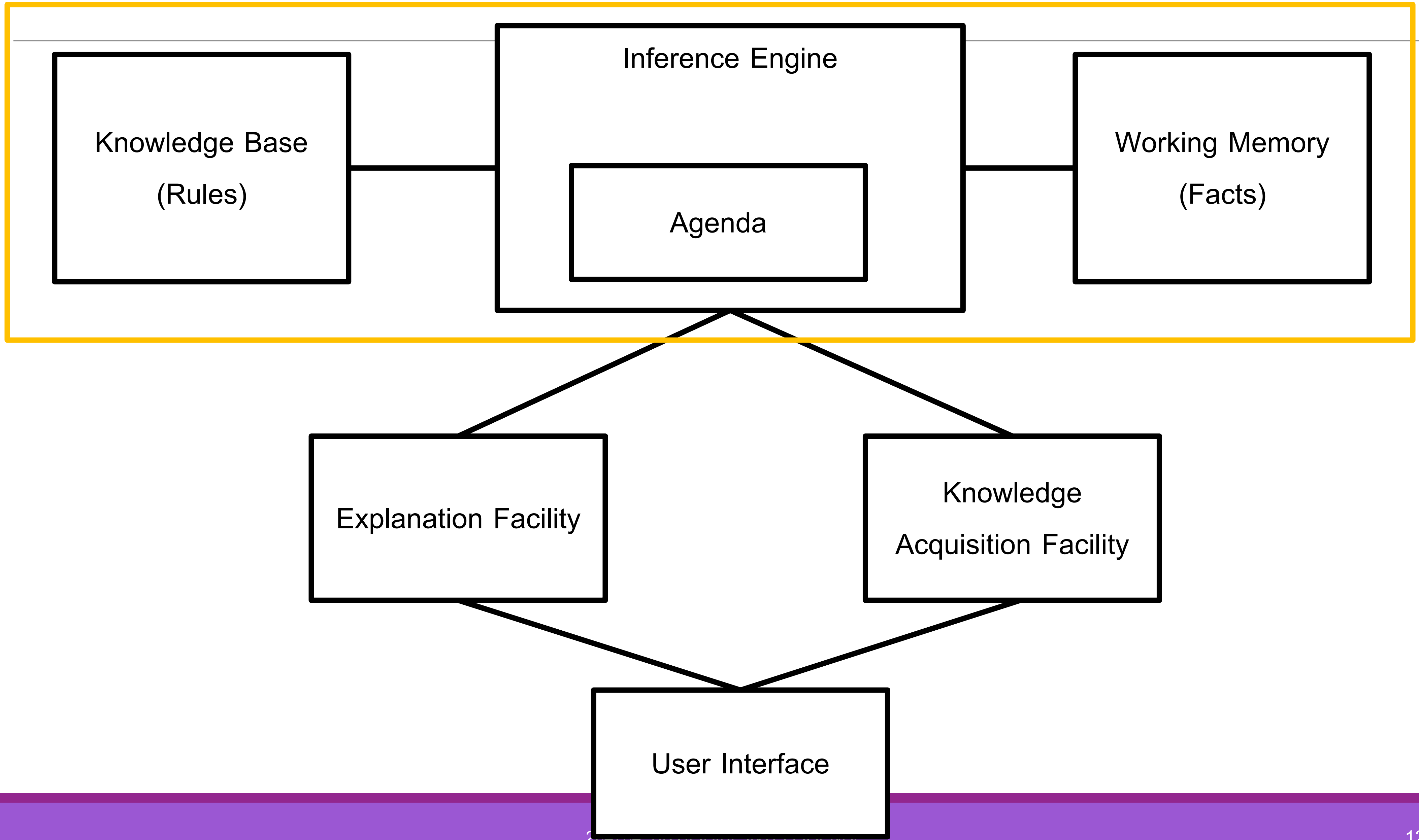
Rule-based Expert Systems

- KB (or production memory) consists of rules in the form of post production system (FOL)

antecedent (LHS, condition, premise) $\rightarrow$ consequence (RHS, actions, conclusion)

- The system also has user interface, explanation facility, working memory, inference engine, agenda, and (optional) knowledge acquisition facility
  - **Agenda** holds activated rule/activation – rules that are ready to be used
  - **Inference engine** perform inference. Forward chaining is used in this case
  - **Working memory** hold facts that occur during the operation

Expert System



Example of KB in Production System

1. car(x) won't start $\longrightarrow$ check (x)'s battery
2. car(x) won't start $\longrightarrow$ check (x)'s gas
3. check (x)'s battery $\wedge$ (x)'s battery bad $\longrightarrow$ replace (x)'s battery
4. check (x)'s gas $\wedge$ (x) has no gas $\longrightarrow$ fill (x)'s gas tank
5. check (x)'s battery $\wedge$ (x)'s battery good $\longrightarrow$ check (x)'s headlight
6. check (x)'s headlight $\wedge$ (x)'s headlight on $\longrightarrow$ replace (x)'s starter

Inference Engine

- Forward Chaining: Work in cycle (select-execute, situation-response/action)
  - Rule with substituted LHS becomes true become activation, select one to execute
  - Repeat until and halt condition is met

WHILE not done

Conflict Resolution: If there are more than one activations, pick the one with highest priority

Act: Perform RHS of chosen activation

update working memory with new facts

Remove executed activation from agenda

Match: Add rules that LHS become true as activation in agenda (pattern/predicate matching)

Also remove activations that LHSs are no longer true

Check for Halt:

If halt action or break command occur → break from WHILE loop

END-WHILE

Pattern Matching

Need control strategy to consider which rules to activated

- Markov Algorithm
  - Rules $\longrightarrow$ Facts
  - Sort rules by priority, then check them. one by one
  - If high-priority cannot be activated yet, consider the next one in line
  - Can be inefficient if KB is large
- Rete algorithm
  - Facts $\longrightarrow$ Rules
  - Only consider matches (facts) that change

Markov Example

Knowledge Base

car(x) won't start
⟶ check (x)'s battery
car(x) won't start
⟶ check (x)'s gas
check (x)'s battery ∧ (x)'s battery
bad ⟶ replace (x)'s battery
…

Inference Engine

Agenda

car(my car) won't start ⟶
check (my car)'s battery

car(my car) won't start ⟶
check (my car)'s gas

Working Memory

car(my car) won't start

- Matching: Activate all (substituted) rules that LHS become true and put them in agenda

Example

Knowledge Base

car(x) won't start

⟶ check (x)'s battery

car(x) won't start

⟶ check (x)'s gas

check (x)'s battery ∧ (x)'s battery

bad ⟶ replace (x)'s battery

…

Inference Engine

Agenda

car(my car) won't start ⟶

check (my car)'s battery

car(my car) won't start ⟶

check (my car)'s gas

Working Memory

car(my car) won't start

- Check for Halt: no halting action yet
- Conflict Resolution: Pick activation to execute (need priority system)

Example

Knowledge Base

car(x) won't start

→ check (x)'s battery

car(x) won't start

→ check (x)'s gas

check (x)'s battery ∧ (x)'s battery bad → replace (x)'s battery

…

Inference Engine

Agenda

~~car(my car) won't start → check (my car)'s battery~~

car(my car) won't start → check (my car)'s gas

Working Memory

car(my car) won't start

check (my car)'s battery

(my car)'s battery bad

Update from user

- Act: execute chosen activation
  - Update working memory
  - Delete execute activation from agenda

Example

Knowledge Base

car(x) won't start

⟶ check (x)'s battery

car(x) won't start

⟶ check (x)'s gas

check (x)'s battery ∧ (x)'s battery

bad ⟶ replace (x)'s battery

…

Inference Engine

Agenda

car(my car) won't start ⟶

check (my car)'s gas

check (my car)'s battery ∧

(my car)'s battery bad ⟶

replace (x)'s battery

Working Memory

car(my car) won't start

check (my car)'s battery

(my car)'s battery bad

- Matching: find rules with matched LHS to facts, activate them
- Repeat the cycle until halting action is detected (or other halting condition)

Rete Algorithm

- Markov Algorithm consider all rules, which can be time-consuming
- However, facts in working memory are unlikely to change in great number $\longrightarrow$ only a few rules need to be considered
  - Rules with LHS that match the added facts
- Also, many rules will share patterns (predicates)

Rete Algorithm (cont.)

- Rete (Latin for web) algorithm work by creating graph for LHS of rules, consisted of
- Root Node: where the facts start
- Pattern Network checks if a fact match a pattern
  - Consists if pattern (alpha) nodes for each pattern, with alpha memory to remember substituted patterns already found
- Join Network combine substituted patterns together
  - There is one join (beta) nodes for each rule, with beta memory to check whether a substitution is enough to activate a rule
- Fact flow: Facts ⟶ Root ⟶ Pattern Nodes ⟶ Join Nodes ⟶ Agenda

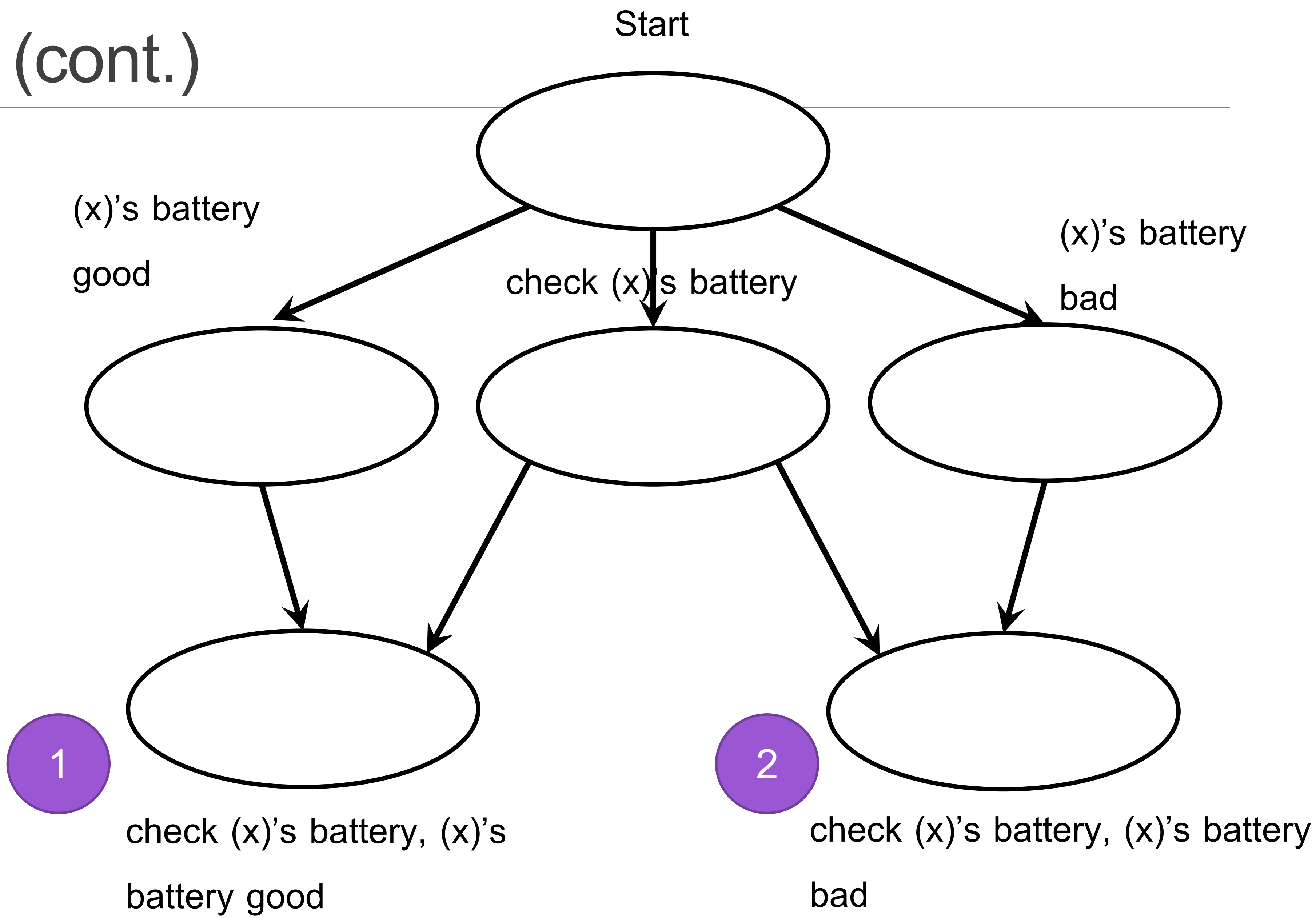
Example of Rete Algorithm

- Using 2 rules
  1. check (x)'s battery $\wedge$ (x)'s battery bad $\longrightarrow$ replace (x)'s battery
  2. check (x)'s battery $\wedge$ (x)'s battery good $\longrightarrow$ check (x)'s headlight
- With 3 Patterns:
  1. check (x)'s battery
  2. (x)'s battery bad
  3. (x)'s battery good

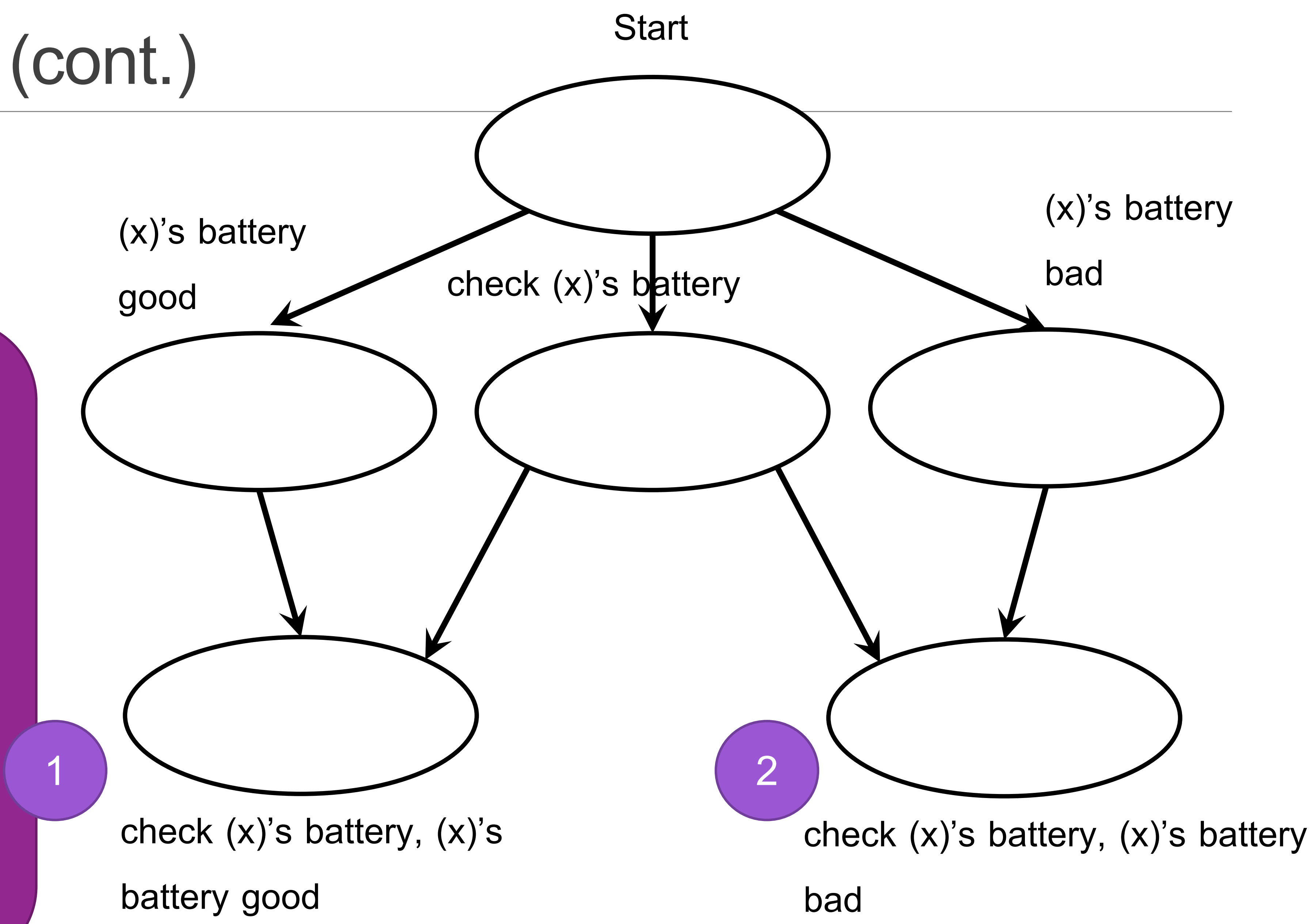
Rete Example (cont.)

Pattern Nodes:

Join Nodes:



Rete Example (cont.)



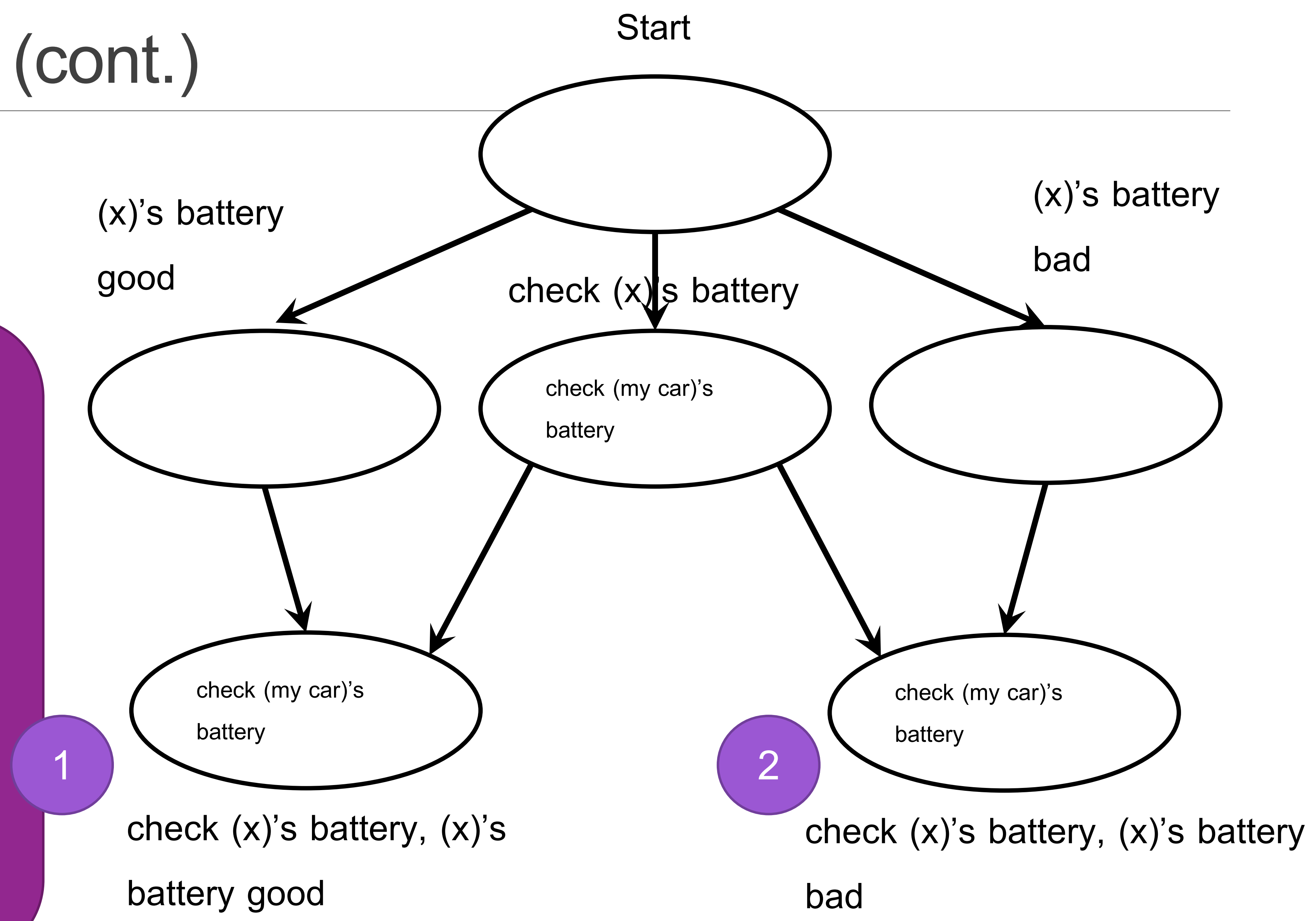
Incoming Facts

- check (my car)'s battery
- (my car)'s battery bad

Rete Example (cont.)

Incoming Facts

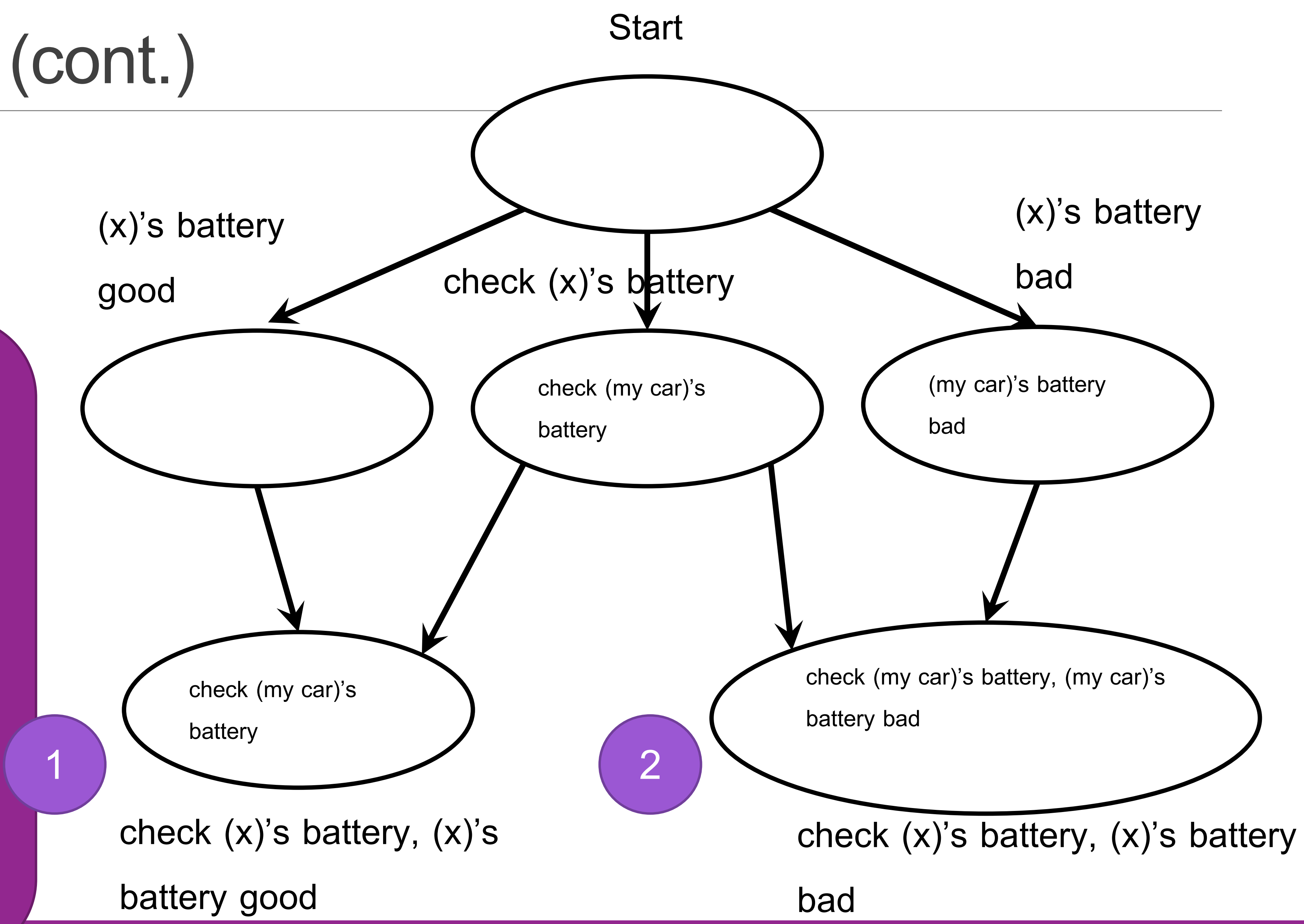
- ✓ check (my car)'s battery
- (my car)'s battery bad



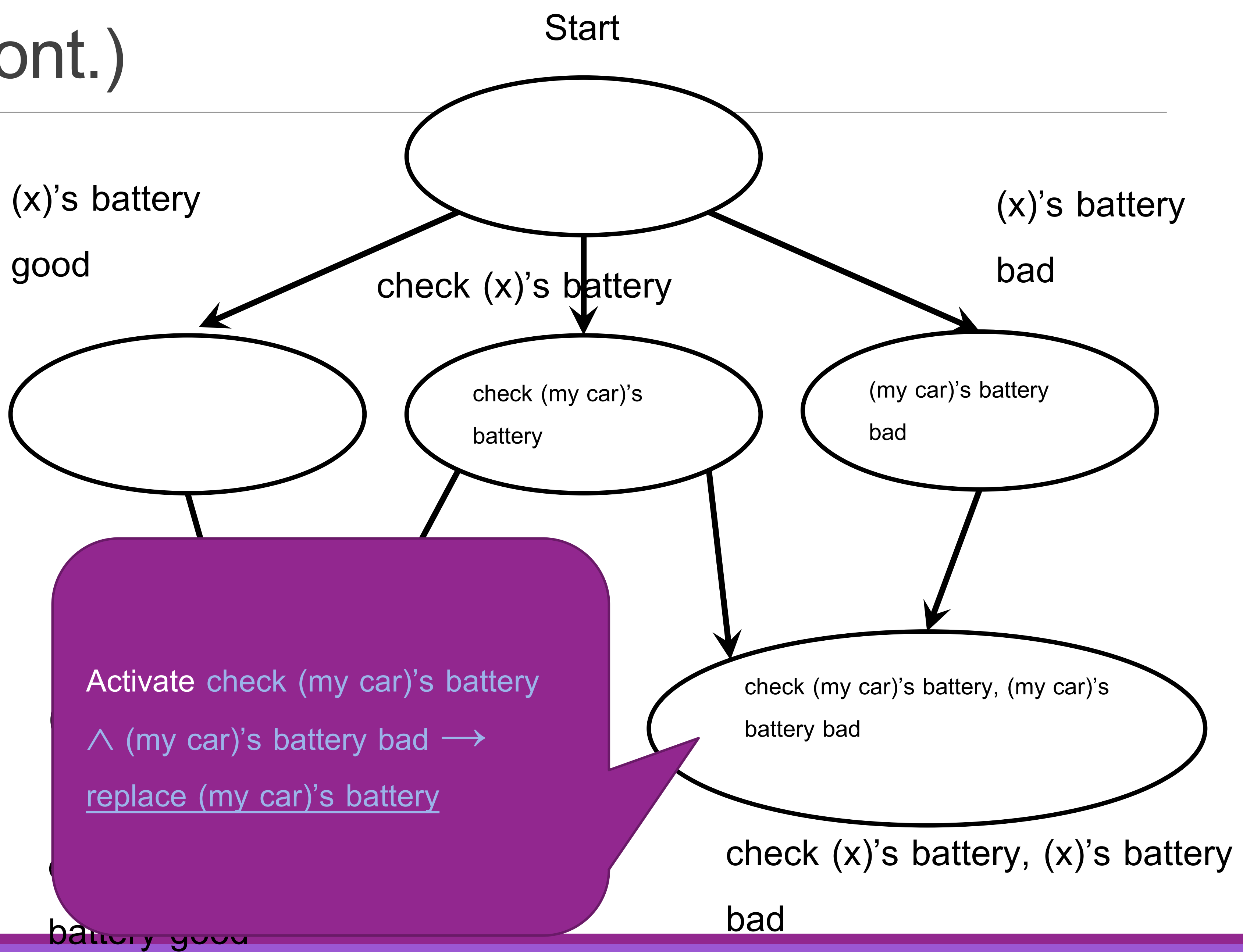
Rete Example (cont.)

Incoming Facts

- ✓ check (my car)'s battery
- ✓ (my car)'s battery bad



Rete Example (cont.)



Incoming Facts

- ✓ check (my car)'s battery
- ✓ (my car)'s battery bad

Activate check (my car)'s battery ∧ (my car)'s battery bad ⟶ replace (my car)'s battery

Ontology

Ontology

- Ontology is an explicit formal specifications of the terms in the domain and relations among them (Gruber 1993)
- Why ontology?
  - To share common understanding of the structure of information among people or software agents
  - To enable reuse of domain knowledge
  - To make domain assumptions explicit
  - To separate domain knowledge from the operational knowledge
  - To analyze domain knowledge
- Ontology can be used as a knowledge base

Class, Instance, and Properties

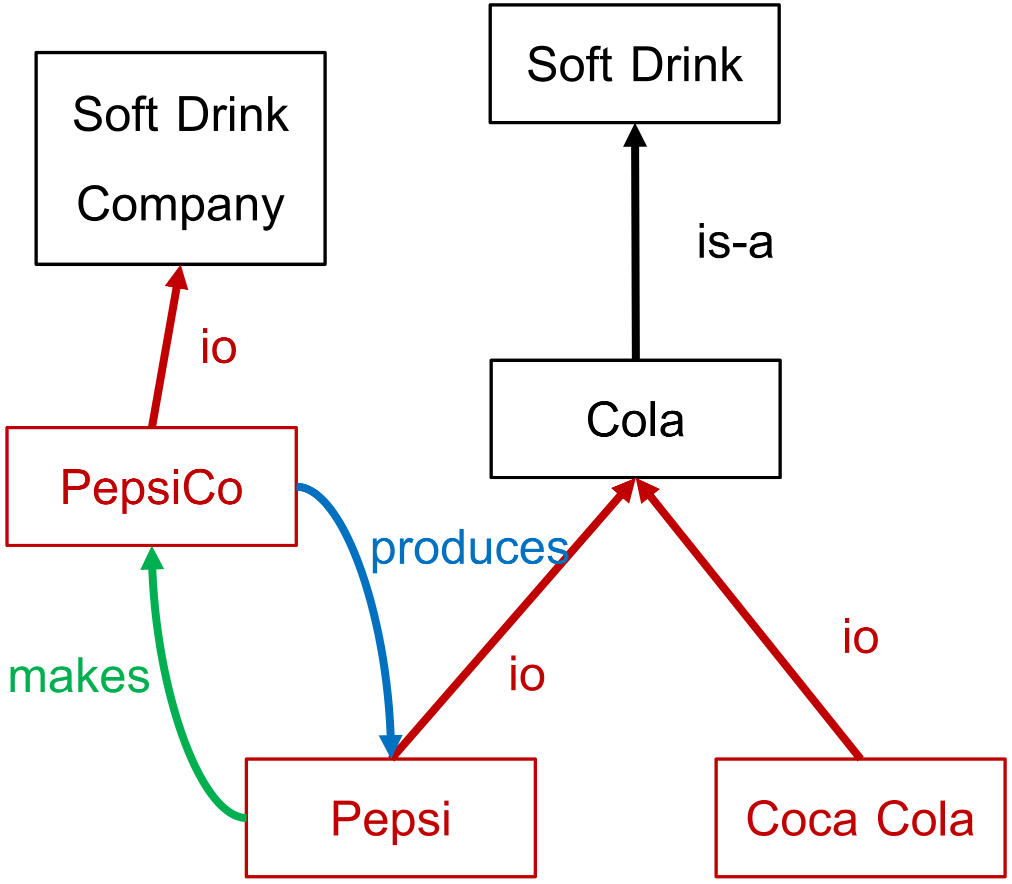
An ontology consists of classes, properties, and individuals.

- Class describe concepts in the domain of interest
  - Classes can have superclass (more general) – subclass (more specific) hierarchy
- Instances, or individuals, represent objects in the domain
  - Instances will belong to at least one classes
- A class can be considered a set of individuals that are members of that class
- Properties, or slots, are binary relation on individuals, describe certain characteristic or relationship of a class or an instance
  - A slot will have domain (class it belong to) and range of allowed (instances of) classes/values it can be
  - Domain $\longrightarrow$ Range

Example of an Ontology

Domain: soft drink (cola in particular)

- Black boxes are classes
- Red boxes are instance
- Direct edge represents slots
  - is_a denote "…is a kind of…" for subclass ⟶ superclass
  - io = "instance of"



Creating an Ontology

Using Knowledge Engineering, from *Ontology Development 101*

1. Determine the domain and scope of the ontology
2. Consider reusing existing ontologies
3. Enumerate important terms in the ontology
4. Define the classes and the class hierarchy
5. Define the properties of classes—slots
6. Define the facets of the slots
7. Create instances

Uses of Ontology

- Semantic Web
  - Encode semantics with the data, making them machine-readable
  - Web Ontology Language (OWL) is widely used for ontology development
- Question-answering
  - Encode knowledge from human-readble source, such as Wikipedia
  - Question-answering system can then parse a question, search relevant information, and then answer the question
  - Example: chapter 6 of YAGO2007.pdf

Reference

- Artificial Intelligence: A Modern Approach
- SWI-Prolog Reference Manual
  - https://www.swi-prolog.org/pldoc/doc_for?object=manual
- Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2), 199-220.
- Ontology Development 101: A Guide to Creating Your First Ontology
  - https://protegewiki.stanford.edu/wiki/Ontology101
- A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools, Edition 1.3
  - http://mowl-power.cs.man.ac.uk/protegeowltutorial/resources/ProtegeOWLTutorialP4_v1_3.pdf
- Suchanek, F. M., Kasneci, G., & Weikum, G. (2008). Yago: A large ontology from wikipedia and wordnet. *Journal of Web Semantics*, 6(3), 203-217.
- Völkel, M., Krötzsch, M., Vrandecic, D., Haller, H., & Studer, R. (2006, May). Semantic wikipedia. In *Proceedings of the 15th international conference on World Wide Web* (pp. 585-594).