Inference in FOL – Unification, Forward &Backward chaining

Universal Instantiation

- Every instantiation of a universally quantified sentence is entailed by it:

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

  - for any variable v and ground term g
    - ground term…a term with out variables

- Example:
  - $\forall$x King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x) yields
    - King(John) $\wedge$ Greedy(John) $\Rightarrow$ Evil(John)
    - King(Richard) $\wedge$ Greedy(Richard) $\Rightarrow$ Evil(Richard)
    - King(Father(John)) $\wedge$ Greedy(Father(John) $\Rightarrow$ Evil(Father(John))
    - …

Existential Instantiation

- For any sentence α, variable *v*, and constant k that does not appear in the KB:

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

- Example:
  - $\exists$x Crown(x) $\wedge$ OnHead(x, John) yields:
    - Crown(C_1) $\wedge$ OnHead(C_1, John)
    - provided C_1 is a new constant (Skolem)

Existential Instantiation

- UI can be applied several times to ***add*** new sentences
  - The KB is logically equivalent to the old
- EI can be applied once to ***replace*** the existential sentence
  - The new KB is not equivalent to the old but is satisfiable iff the old KB was satisfiable

Unification

- ***Unification***: The process of finding all legal substitutions that make logical expressions look identical

- This is a recursive algorithm

Unification

- We can get the inference immediately if we can find a substitution θ such that King(x) and Greedy(x) match King(John) and Greedy(y)
- θ = {x/John, y/John} works
- Unify(α,β) = θ if α θ = β θ

Unification

p	q	θ
$Knows(John, x)$	$Knows(John, Jane)$	$\{x/Jane\}$
$Knows(John, x)$	$Knows(y, OJ)$	$\{x/OJ, y/John\}$
$Knows(John, x)$	$Knows(y, Mother(y))$	$\{y/John, x/Mother(John)\}$
$Knows(John, x)$	$Knows(x, OJ)$	$fail$

Generalized Modus Ponens

- This is a general inference rule for FOL that does not require instantiation
- Given:

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

  p1′, p2′ … pn′ (p1 ∧ … pn) ⇒ q

  Subst(θ, pi′) = subst(θ, pi) for all p
- Conclude:
  - Subst(θ, q)

GMP in "CS terms"

- Given a rule containing variables
- If there is a consistent set of bindings for all of the variables of the left side of the rule (before the arrow)
- Then you can derive the result of substituting all of the same variable bindings into the right side of the rule

Base Cases for Unification

- If two expressions are identical, the result is (NIL) (succeed with empty unifier set)
- If two expressions are different constants, the result is NIL (fail)
- If one expression is a variable *and is not contained in the other,* the result is ((x other-exp))

Recursive Case

- If both expressions are lists,
  - Combine the results of unifying the CAR with unifying the CDR

- In Lisp…

```
(cons (unify (car list1) (car list2))
      (unify (cdr list1) (cdr list2))
```

Algorithm

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
    inputs:  x, a variable, constant, list, or compound
             y, a variable, constant, list, or compound
             θ, the substitution built up so far
    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(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y],θ))
    else if  LIST?(x) and LIST?(y) then
        return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y],θ))
    else return failure
```

Algorithm

function UNIFY-VAR(*var*, *x*, θ) **returns** a substitution

inputs: *var*, a variable

x, any expression

θ, the substitution built up so far.

if {*var*/*val*} $\in$ θ **then return** UNIFY(*val*, *x*, θ)

else if {*x*/*val*} $\in$ θ **then return** UNIFY(*var*, *val*, θ)

else if OCCUR-CHECK? (*var*, *x*) **then return** failure

else return {*var*/*x*} to θ

A few more details…

- Don’t reuse variable names
  - Before actually unifying, give each rule a separate set of variables
  - The lisp function *gentemp* creates uniquely numbered variables
- Keep track of bindings
  - If a variable is already bound to something, it must retain the same value throughout the computation
  - This requires substituting each successful binding in the remainder of the expression

FC: Example Knowledge Base

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

- Prove that Col. West is a criminal.

FC: Example Knowledge Base

- …it is a crime for an American to sell weapons to hostile nations

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

- Nono…has some missiles

  $\exists x\ Owns(Nono, x) \wedge Missiles(x)$

  $Owns(Nono, M_1)$ *and* $Missle(M_1)$

- …all of its missiles were sold to it by Col. West

  $\forall x\ Missle(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$

- Missiles are weapons

  $Missle(x) \Rightarrow Weapon(x)$

FC: Example Knowledge Base

- An enemy of America counts as "hostile"

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

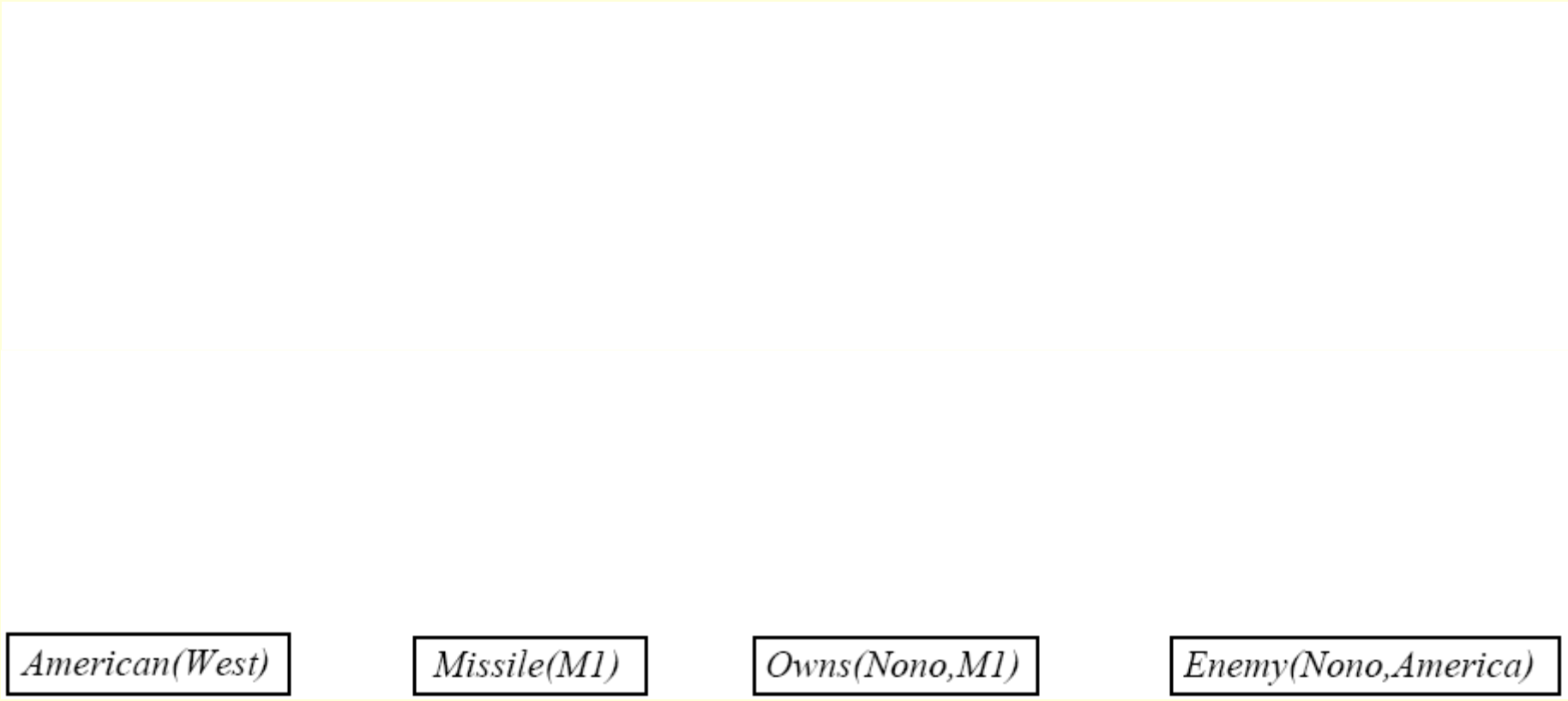
- Col. West who is an American

  American(Col. West)

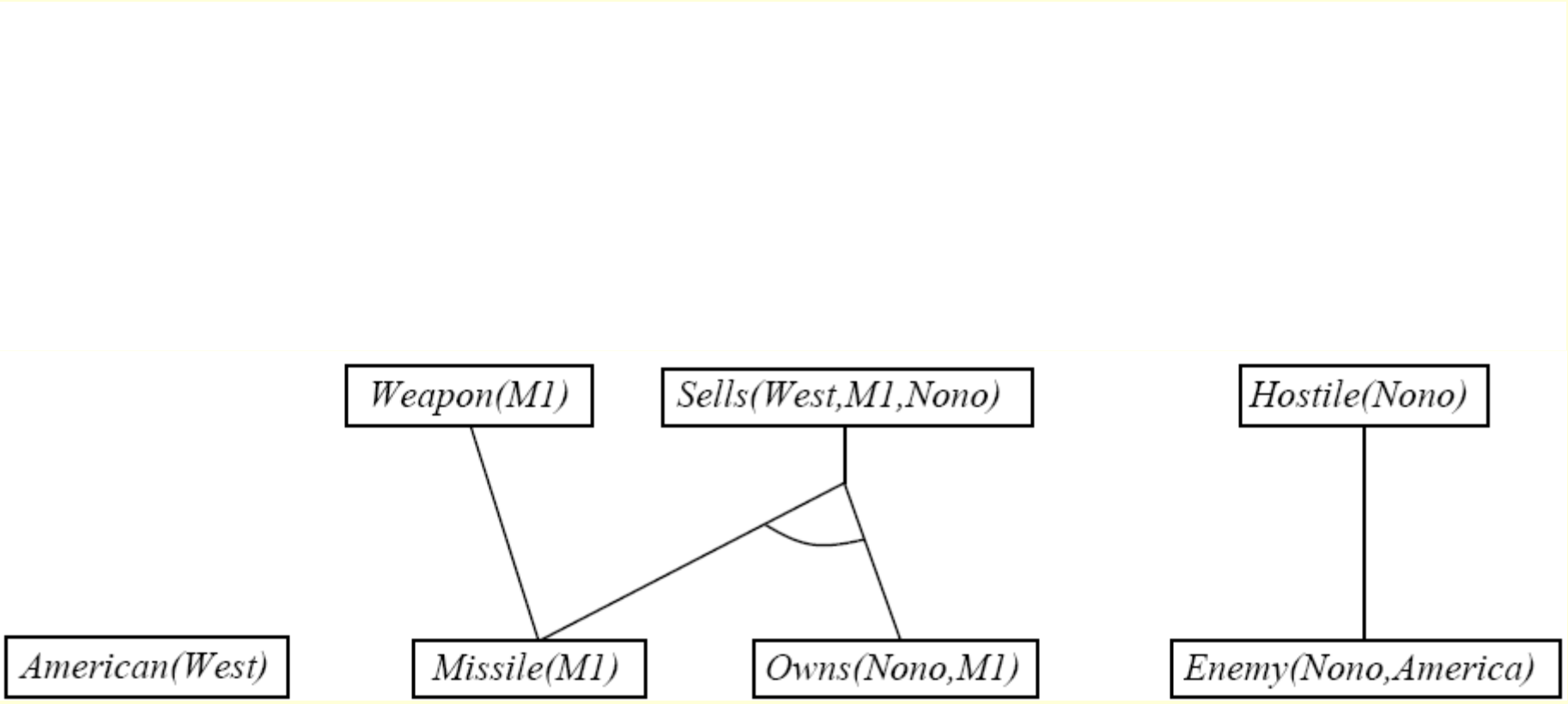
- The country Nono, an enemy of America

  Enemy(Nono, America)

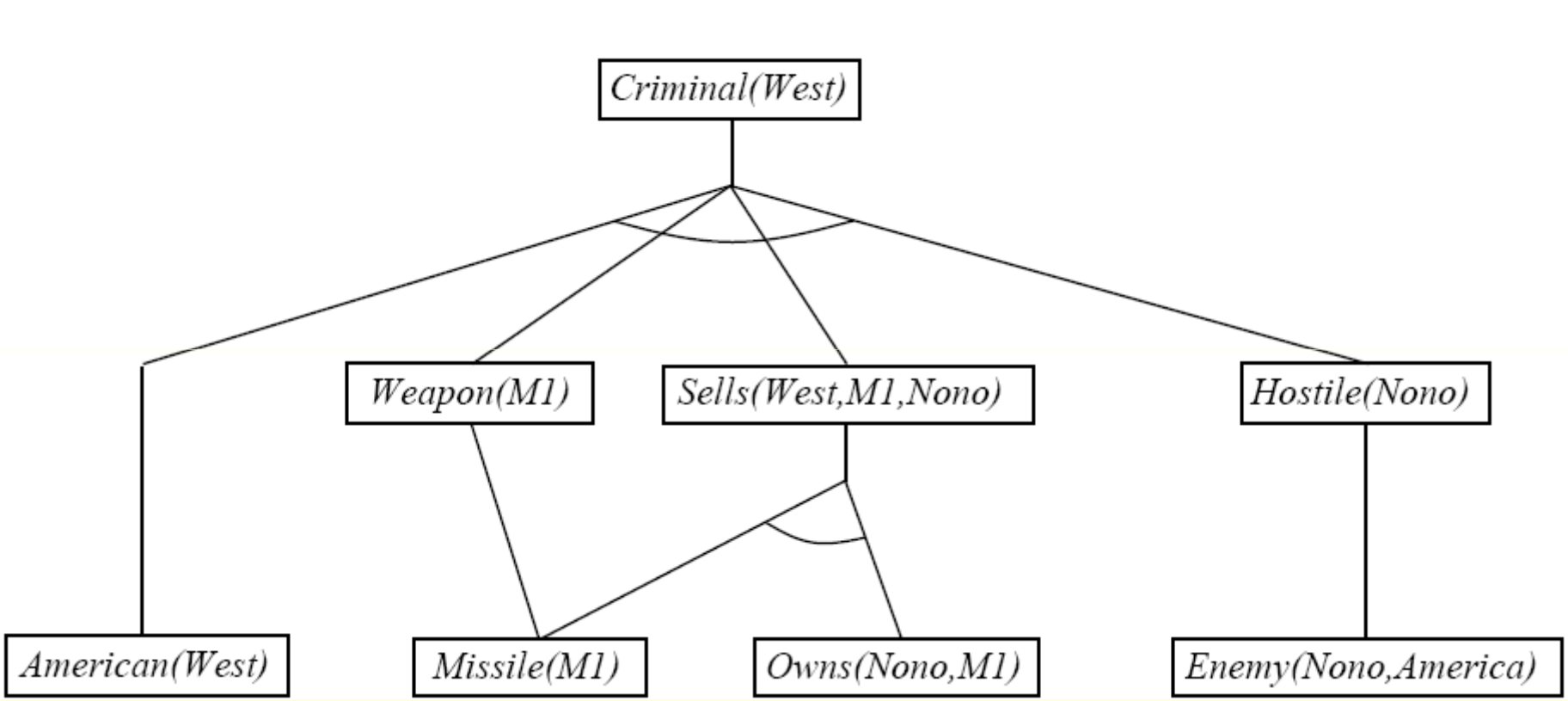
FC: Example Knowledge Base



FC: Example Knowledge Base



FC: Example Knowledge Base



Efficient Forward Chaining

- Order conjuncts appropriately
  - E.g. most constrained variable
- Don’t generate redundant facts; each new fact should depend on at least one newly generated fact.
  - Production systems
  - RETE matching
  - CLIPS

Forward Chaining Algorithm

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

repeat until *new* is empty
  new $\leftarrow \{\}$
  for each sentence r **in** KB **do**
    $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-APART(r)
    for each θ such that $(p_1 \wedge \ldots \wedge p_n)\theta = (p'_1 \wedge \ldots \wedge p'_n)\theta$
        for some $p'_1, \ldots, p'_n$ in KB
      $q' \leftarrow$ SUBST(θ, q)
      if q' is not a renaming of a sentence already in KB or *new* **then do**
        add q' to *new*
        $\phi \leftarrow$ UNIFY(q', α)
        if ϕ is not *fail* **then return** ϕ
  add *new* to KB
return *false*

RETE Network

- Based only on Left Sides (conditions) of rules
- Each condition (test) appears once in the network
- Tests with “AND” are connected with “JOIN”
  - Join means all tests work with same bindings

Example Rules

If (goal put-on <x> <y>) AND (clear <x>) AND (clear <y>) THEN
add (on <x> <y>) delete (clear <x>)

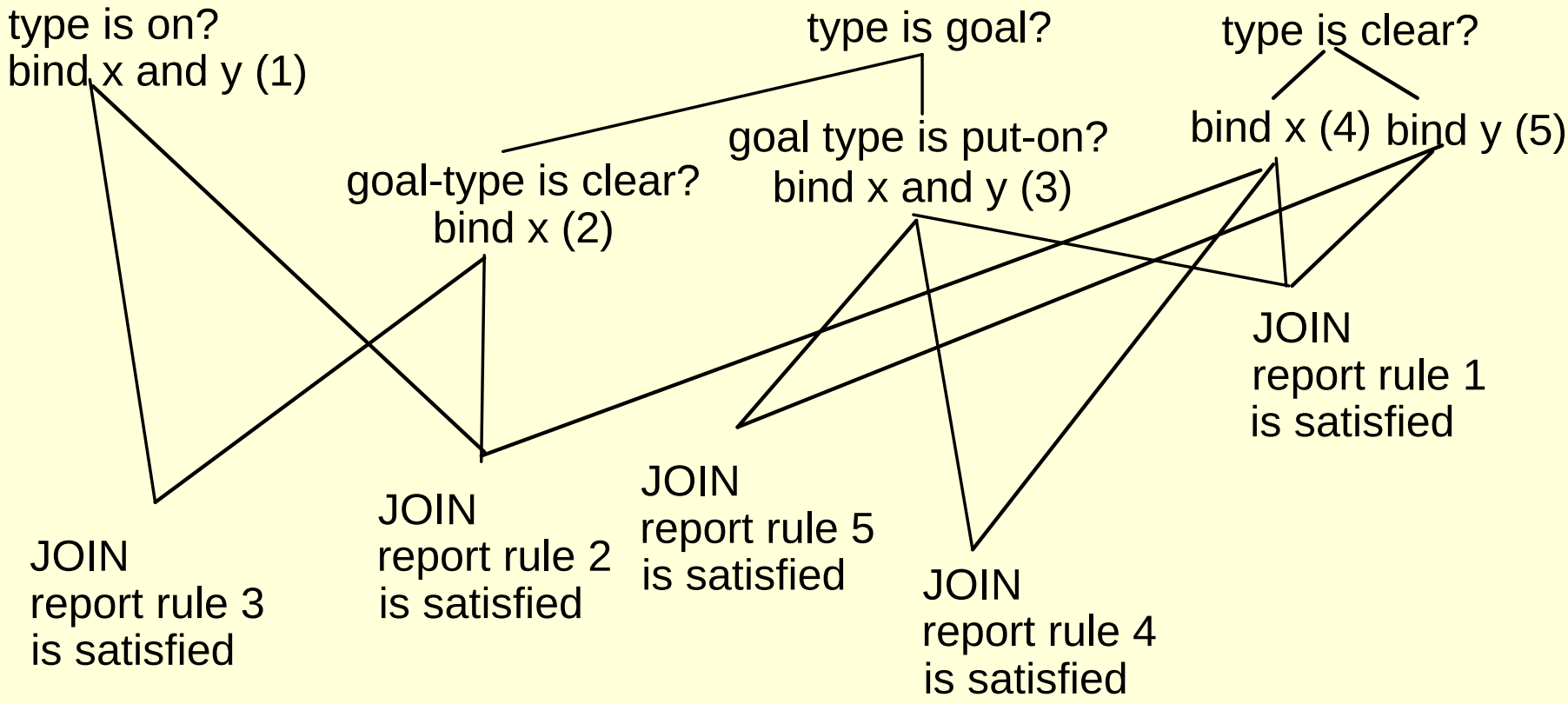
If (goal clear <x>) AND (on <y> <x>) AND (clear <y>) THEN
add (clear <x>) add (on <y> table) delete (on <y> <x>)

If (goal clear <x>) AND (on <y> <x>) THEN
add (goal clear <y>)

If (goal put-on <x> <y>) AND (clear <x>) THEND
add (goal clear <y>)

If (goal put-on <x> <y>) AND (clear <y>) THEN
add (goal clear <x>)

RETE Network



Using the RETE Network

- Each time a fact comes in…
  - Update bindings for the relevant node (s)
  - Update join(s) below those bindings
  - Note new rules satisfied
- Each processing cycle
  - Choose a satisfied rule

Example (Facts)

1. (goal put-on yellow-block red-block)
2. (on blue-block yellow-block)
3. (on yellow-block table)
4. (on red-block table)
5. (clear red-block)
6. (clear blue-block)

Why RETE is Efficient

- Rules are “pre-compiled”
- Facts are dealt with as they come in
  - Only rules connected to a matching node are considered
  - Once a test fails, no nodes below are considered
  - Similar rules share structure
- In a typical system, when rules “fire”, new facts are created / deleted incrementally
  - This incrementally adds / deletes rules (with bindings) to the conflict set

CLIPS

- CLIPS is another forward-chaining production system

- Important commands
  - (assert *fact*) (deffacts *fact1 fact2* …)
  - (defrule *rule-name rule*)
  - (reset) - eliminates all facts except "initial-fact"
  - (load *file*) (load-facts *file*)
  - (run)
  - (watch all)
  - (exit)

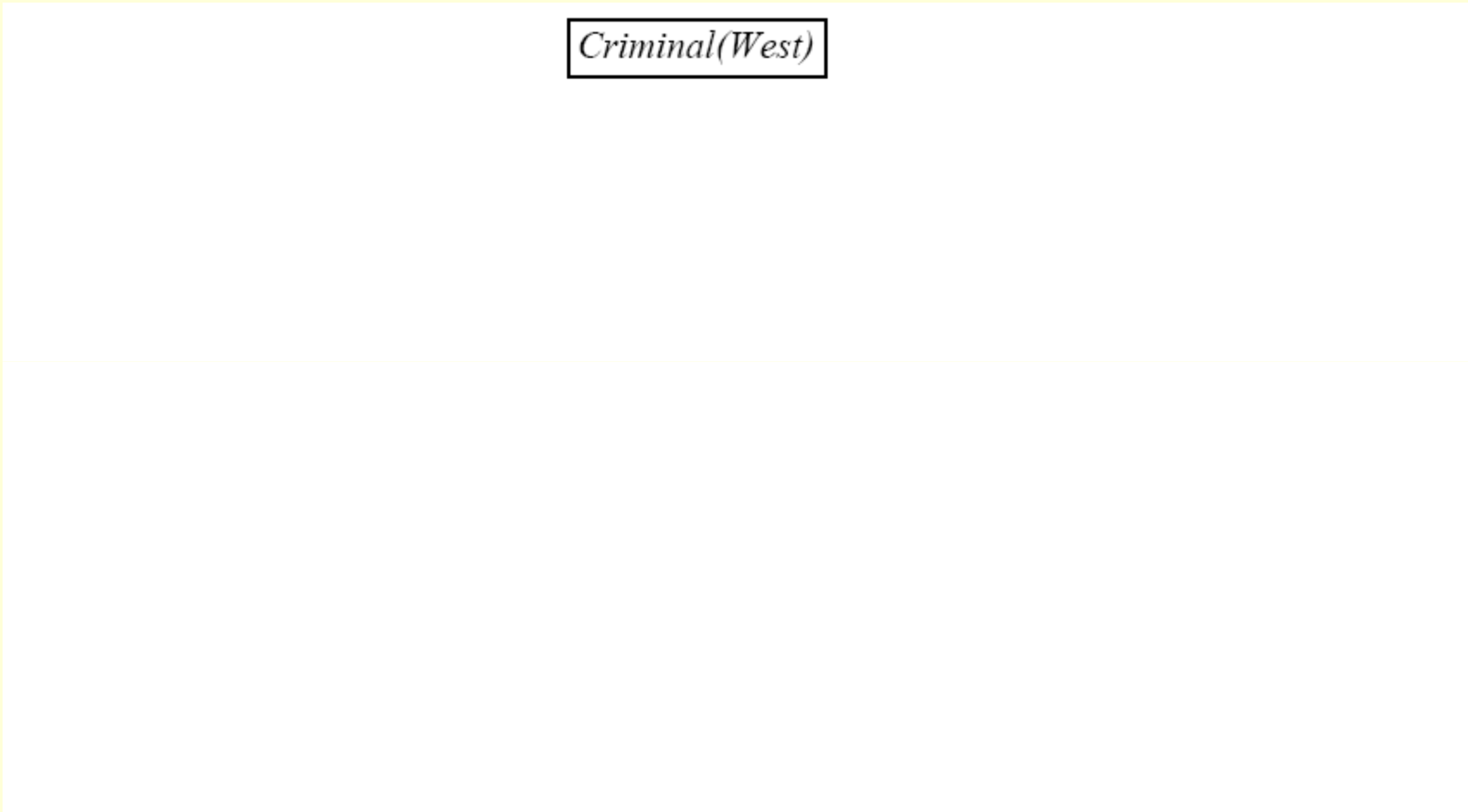
CLIPS Rule Example

```
(defrule putting-on
     ?g <- (goal put-on ?x ?y)
     (clear ?x)
     ?bottomclear <- (clear ?y)
==>
     (assert (on ?x ?y))
     (retract ?g)
     (retract ?bottomclear)
)
```

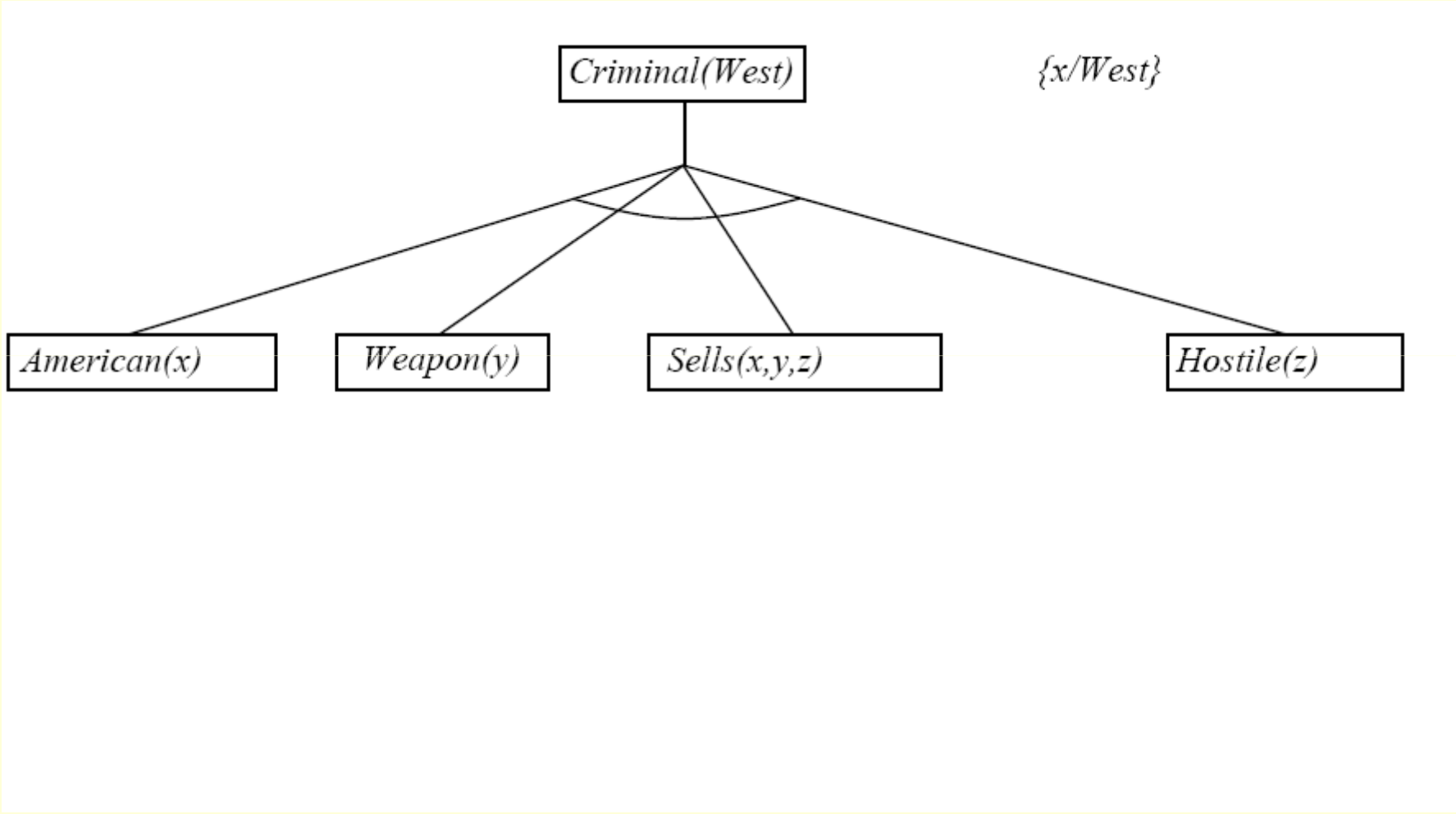
Backward Chaining

- Consider the item to be proven a goal
- Find a rule whose head is the goal (and bindings)
- Apply bindings to the body, and prove these (subgoals) in turn
- If you prove all the subgoals, increasing the binding set as you go, you will prove the item.
- Logic Programming (cprolog, on CS)

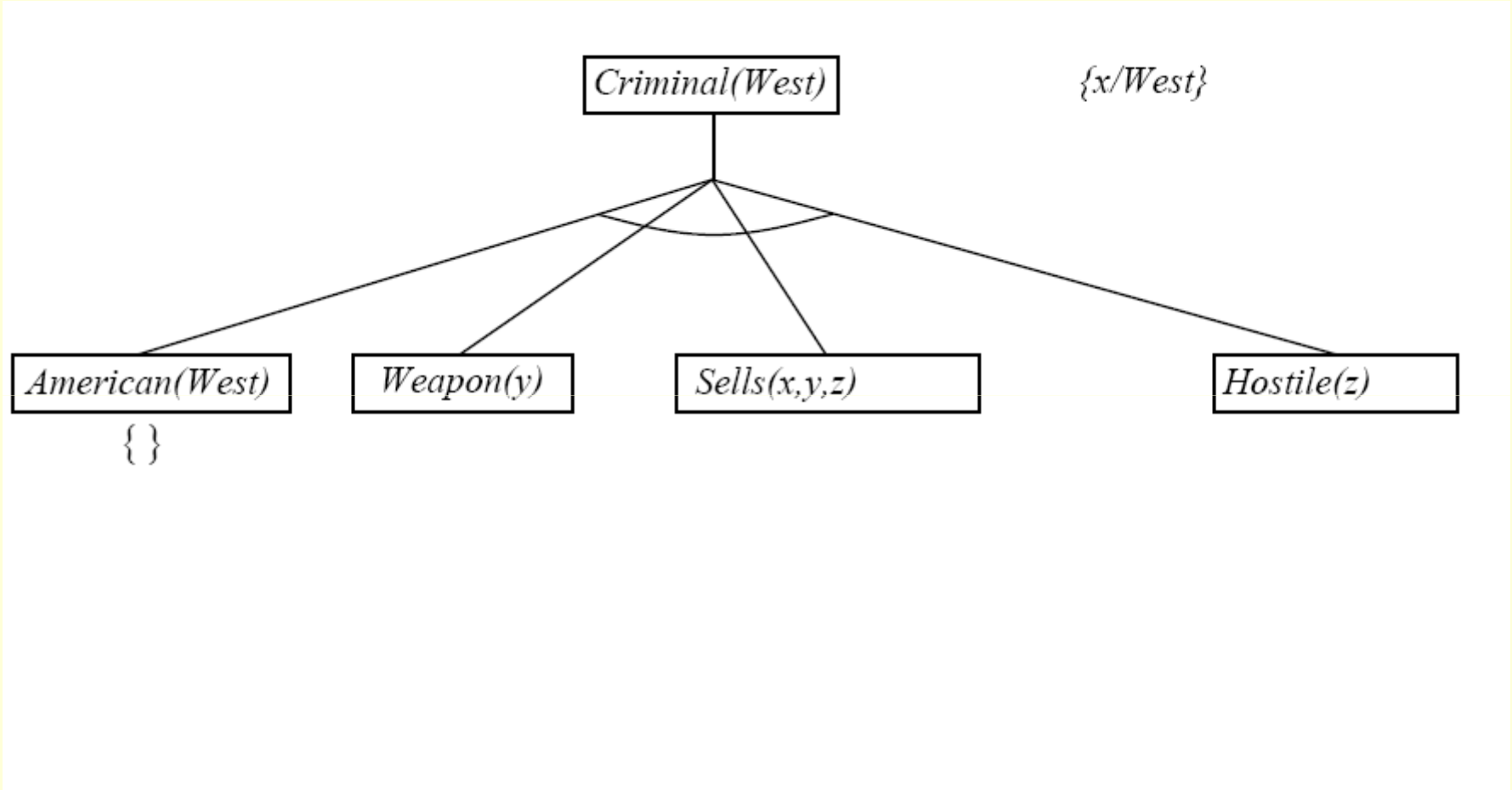
Backward Chaining Example



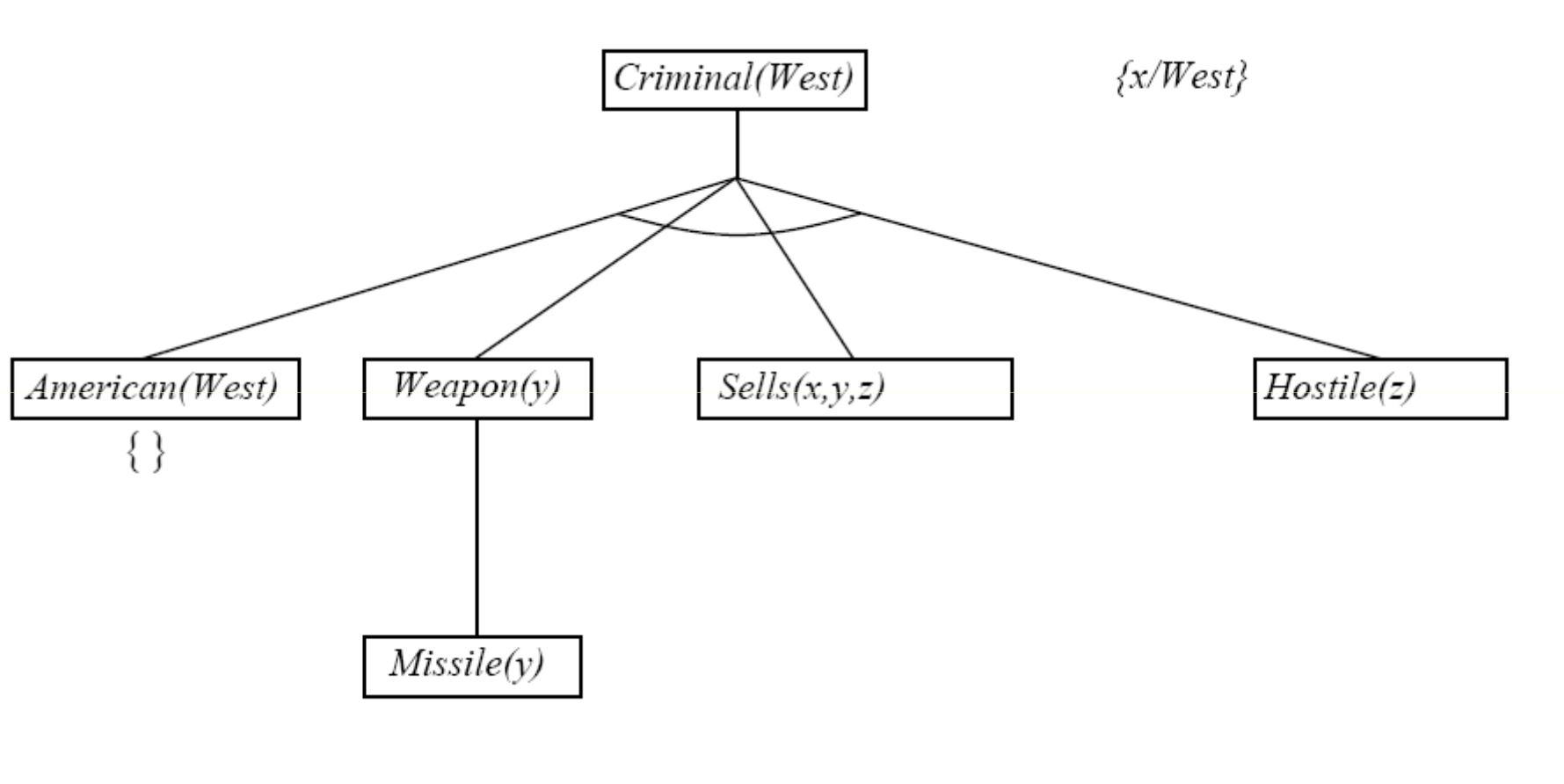
Backward Chaining Example



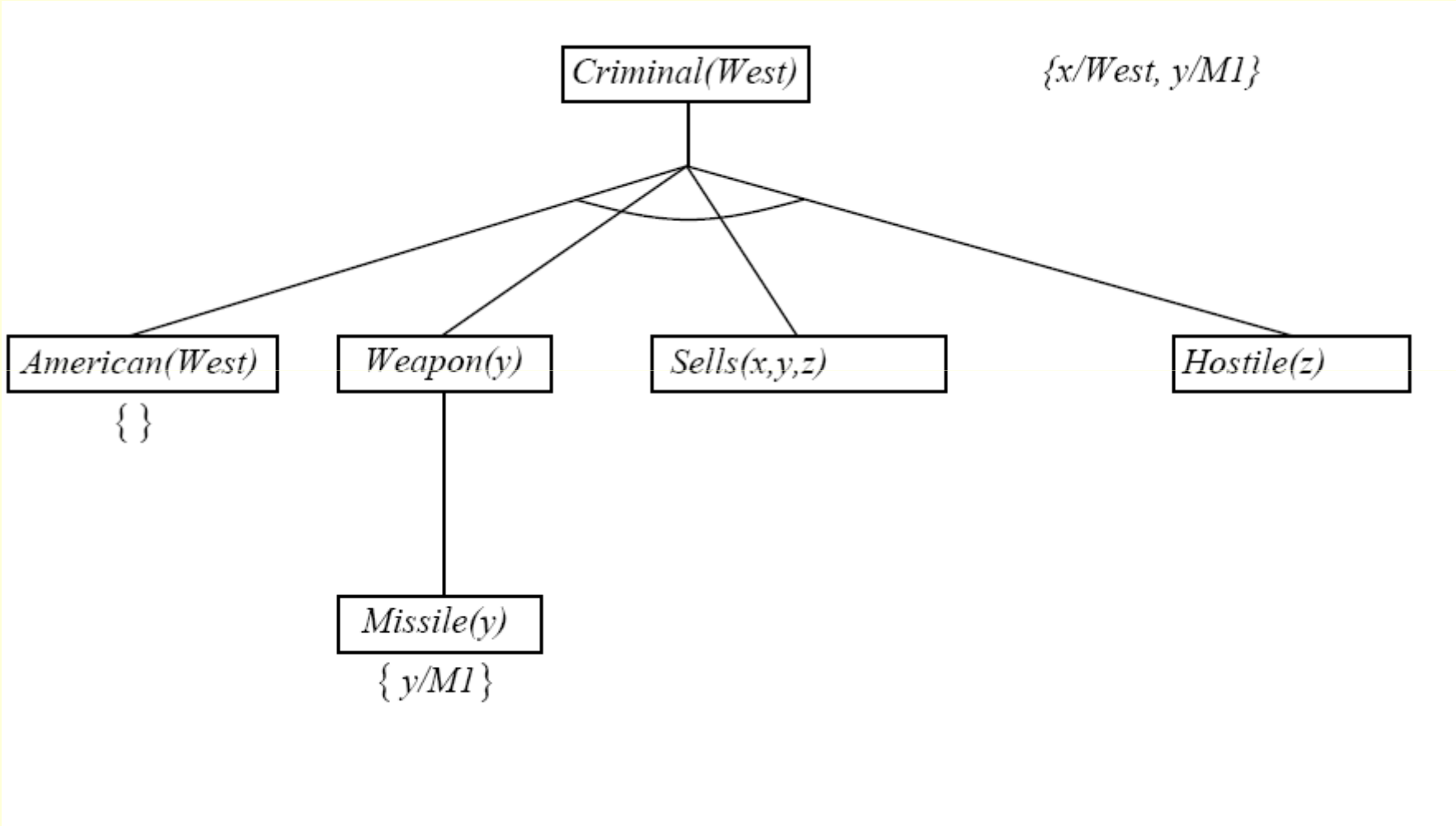
Backward Chaining Example



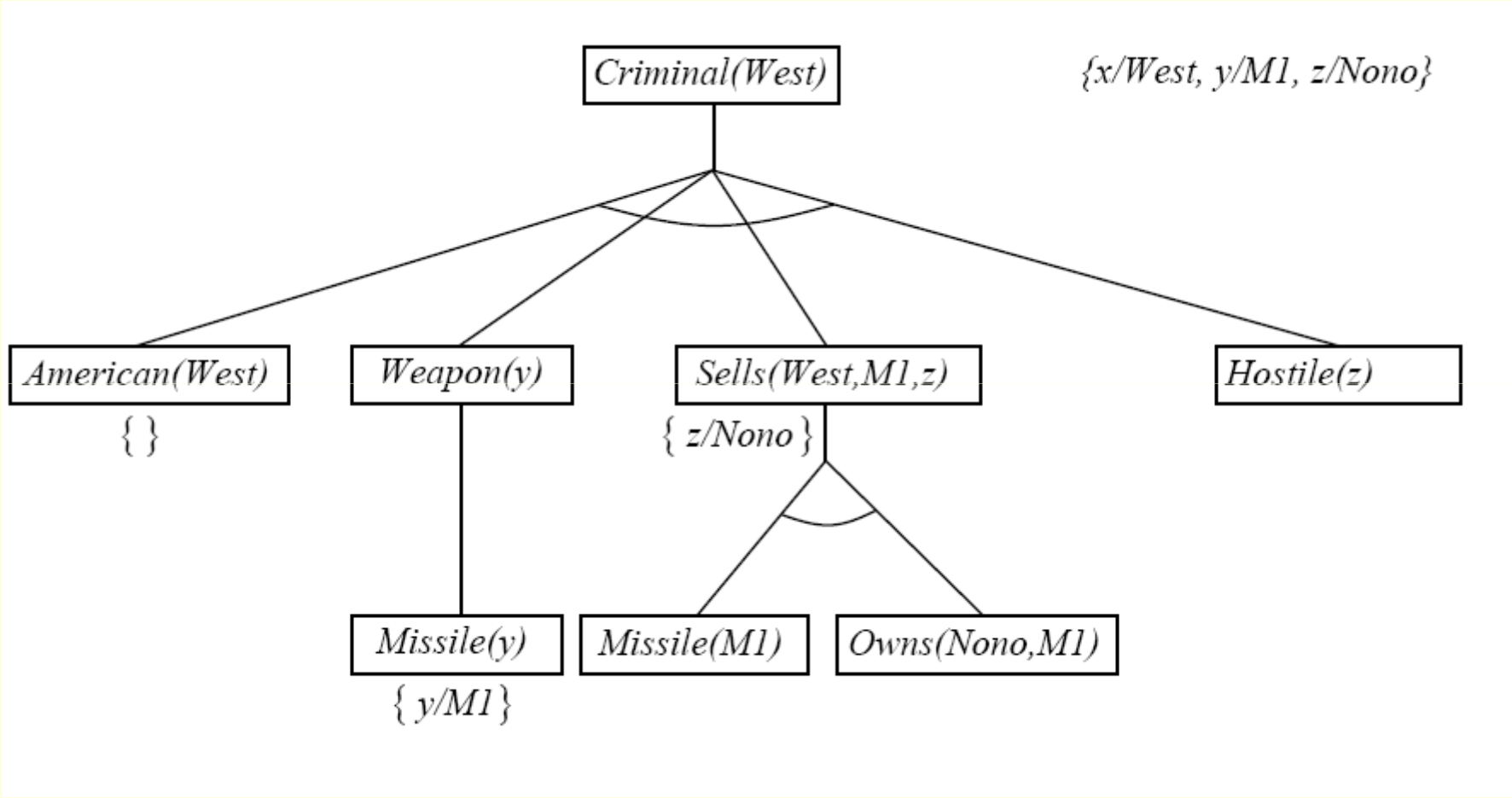
Backward Chaining Example



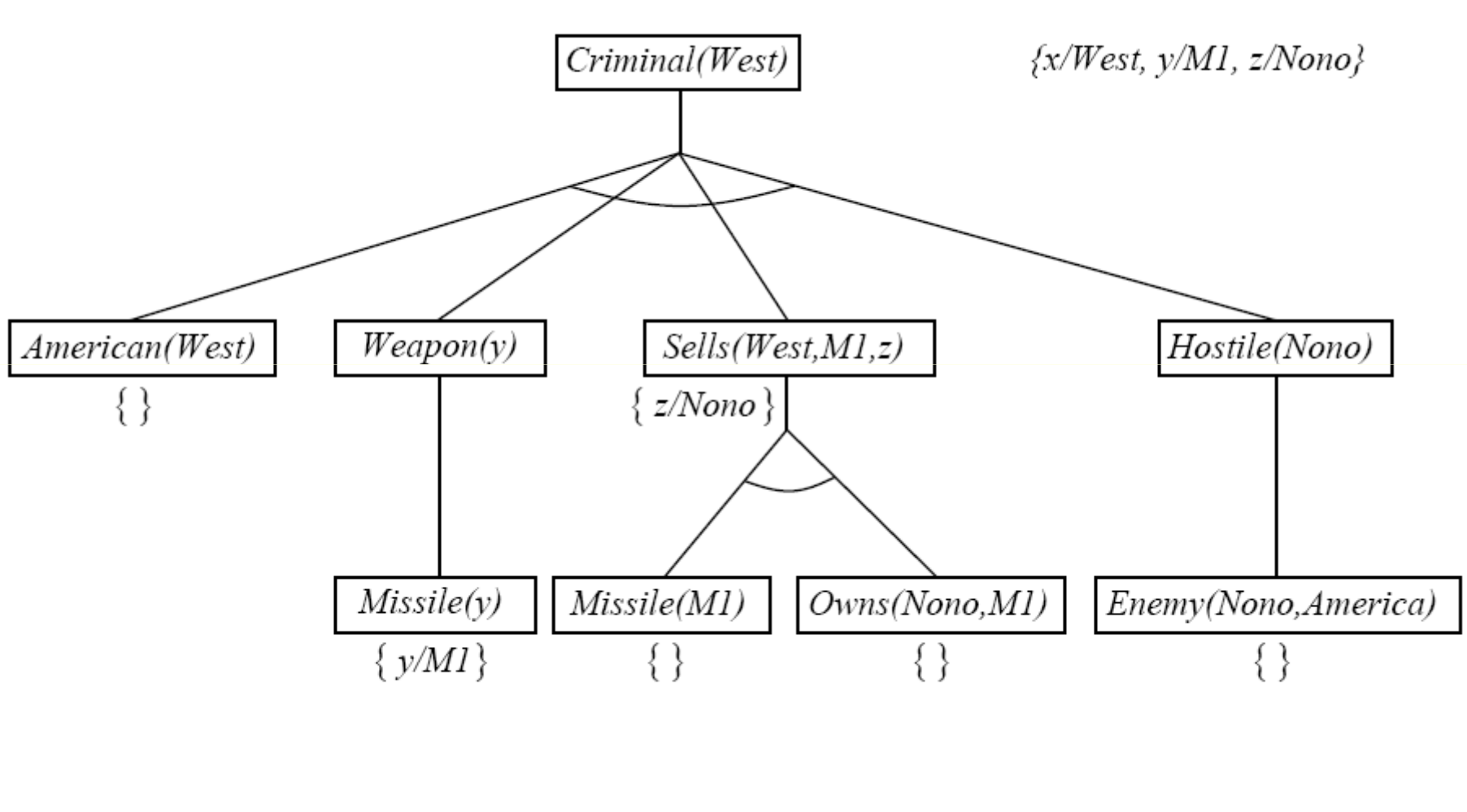
Backward Chaining Example



Backward Chaining Example



Backward Chaining Example



Backward Chaining Algorithm

function FOL-BC-ASK(KB, goals, θ) **returns** a set of substitutions

inputs: KB, a knowledge base

$goals$, a list of conjuncts forming a query

θ, the current substitution, initially the empty substitution $\{\}$

local variables: ans, a set of substitutions, initially empty

if $goals$ is empty **then return** $\{\theta\}$

$q' \leftarrow \text{SUBST}(\theta, \text{FIRST}(goals))$

for each r **in** KB where $\text{STANDARDIZE-APART}(r) = (p_1 \wedge \ldots \wedge p_n \Rightarrow q)$

and $\theta' \leftarrow \text{UNIFY}(q, q')$ succeeds

$ans \leftarrow \text{FOL-BC-ASK}(KB, [p_1, \ldots, p_n | \text{REST}(goals)], \text{COMPOSE}(\theta', \theta)) \cup ans$

return ans

Properties of Backward Chaining

- Depth-first recursive proof search: space is linear in size of proof
- Incomplete due to infinite loops
  - Fix by checking current goal with every subgoal on the stack
- Inefficient due to repeated subgoals (both success and failure)
  - Fix using caching of previous results (extra space)
- Widely used without improvements for logic programming

Logic Programming

- Logic Programming
  - Identify problem
  - Assemble information
  - Tea Break
  - Encode information in KB
  - Encode problem instance as facts
  - Ask queries
  - Find false facts
- Ordinary Programming
  - Identify problem
  - Assemble information
  - Figure out solution
  - Program Solution
  - Encode problem instance as data
  - Apply program to data
  - Debug procedural errors

Logic Programming

- Basis: backward chaining with Horn clauses + lots of bells and whistles
  - Widely used in Europe and Japan
    - Basis of 5th Generation Languages and Projects
- Compilation techniques -> 60 million LIPS
- Programming = set of clauses

  head :- literal1, …, literaln

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

Logic Programming

- Rule Example
  puton(X,Y) :- cleartop(X), cleartop(Y), takeoff(X,Y).

- Capital letters are variables

- Three parts to the rule
  - Head (thing to prove)
  - Neck :-
  - Body (subgoals, separated by ,)

- Rules end with .

Logic Programming

- Efficient unification by open coding
- Efficient retrieval of matching clauses by direct linking
- Depth-first, left-to-right, backward chaining
- Built-in predicate for arithmetic e.g. X is Y*Z+2
- Closed-world assumption ("negation as failure")
  - e.g. given alive(X) :- not dead(X).
  - alive(Joe) succeeds if dead(joe) fails

Logic Programming

- These notes are for gprolog (available on the departmental servers)
- To read a file, consult('file').
- To enter data directly, consult(user). Type control-D when done.
- Every statement must end in a period. If you forget, put it on the next line.
- To prove a fact, enter the fact directly at the command line. gprolog will respond Yes, No, or give you a binding set. If you want another answer, type ; otherwise return.
- Trace(predicate) or trace(all) will allow you to watch the backward chaining process.

Logic Programming

- Depth-first search from start state X
  - dfs(X) :- goal(X).
  - dfs(X) :- successor(X,S), dfs(S).

- No need to loop over S: successor succeeds for each

Logic Programming

- Example: Appending two lists to produce a third
  - append([], Y, Y).
  - append([X|L], Y, [X|Z]) :- append(L, Y, Z).

  - query: append(A, B, [1,2]).
  - answers:
    - A=[] B=[1,2]
    - A=[1] B=[2]
    - A=[1,2] B=[]

Inference Methods

- Unification (prerequisite)
- Forward Chaining
  - Production Systems
  - RETE Method (OPS)
- Backward Chaining
  - Logic Programming (Prolog)
- Resolution
  - Transform to CNF
  - Generalization of Prop. Logic resolution

Resolution

- Convert everything to CNF
- Resolve, with unification
- If resolution is successful, proof succeeds
- If there was a variable in the item to prove, return variable's value from unification bindings

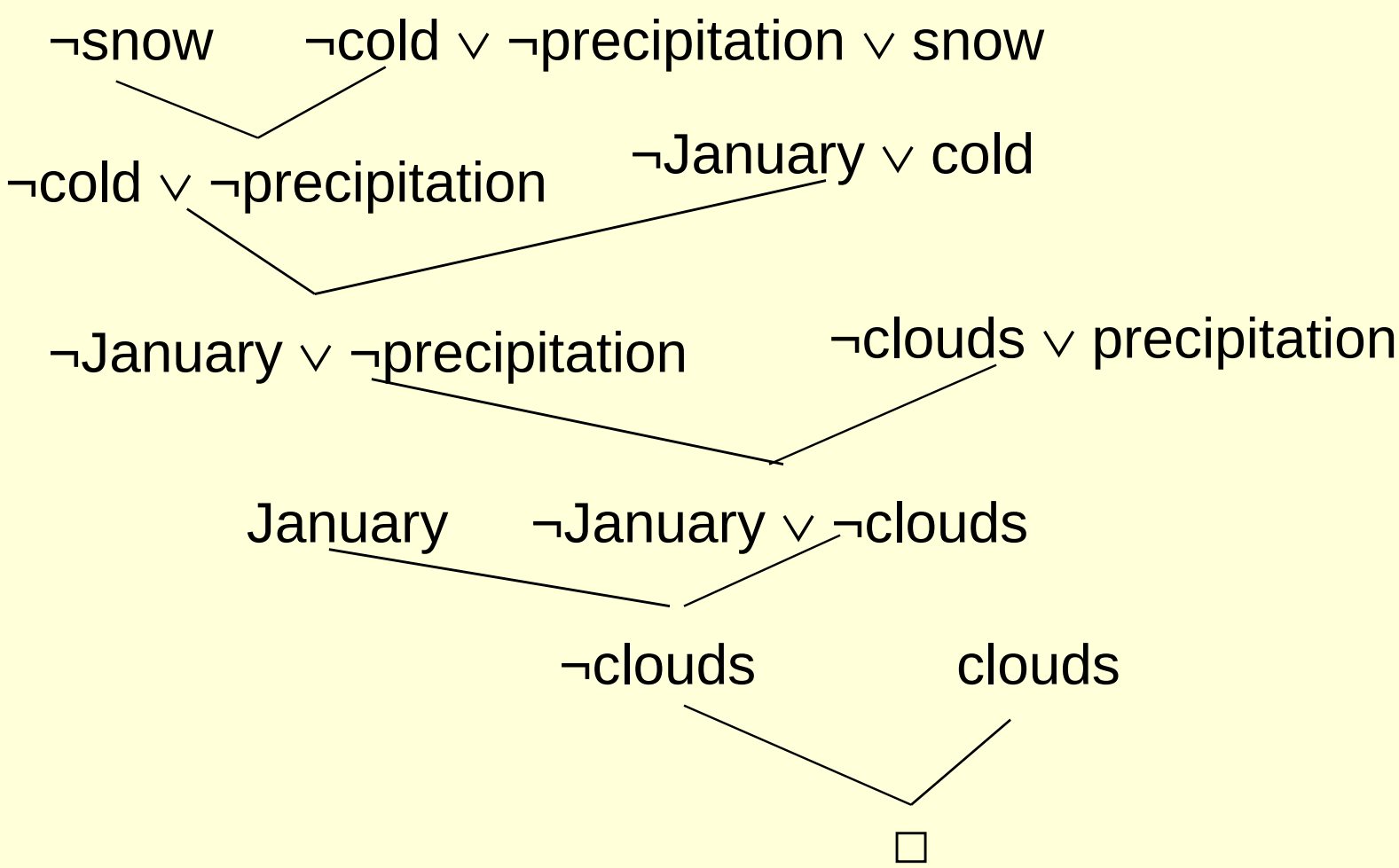
Resolution (Review)

- Resolution allows a complete inference mechanism (search-based) using only one rule of inference
- Resolution rule:
  - Given: $P_1 \vee P_2 \vee P_3 \ldots \vee P_n$, and $\neg P_1 \vee Q_1 \ldots \vee Q_m$
  - Conclude: $P_2 \vee P_3 \ldots \vee P_n \vee Q_1 \ldots \vee Q_m$
    Complementary literals P_1 and $\neg P_1$ "cancel out"
- To prove a proposition F by resolution,
  - Start with $\neg F$
  - Resolve with a rule from the knowledge base (that contains F)
  - Repeat until all propositions have been eliminated
  - If this can be done, a contradiction has been derived and the original proposition F must be true.

Propositional Resolution Example

- Rules
  - Cold and precipitation -> snow
    $\neg$cold $\vee$ $\neg$precipitation $\vee$ snow
  - January -> cold
    $\neg$January $\vee$ cold
  - Clouds -> precipitation
    $\neg$clouds $\vee$ precipitation
- Facts
  - January, clouds
- Prove
  - snow

Propositional Resolution Example



Resolution Theorem Proving (FOL)

- Convert everything to CNF
- Resolve, with unification
  - Save bindings as you go!
- If resolution is successful, proof succeeds
- If there was a variable in the item to prove, return variable's value from unification bindings

Converting to CNF

1. Replace implication ($A \Rightarrow B$) by $\neg A \vee B$
2. Move $\neg$ "inwards"
   - $\neg \forall x\ P(x)$ is equivalent to $\exists x\ \neg P(x)$ & vice versa
3. Standardize variables
   - $\forall x\ P(x) \vee \forall x\ Q(x)$ becomes $\forall x\ P(x) \vee \forall y\ Q(y)$
4. Skolemize
   - $\exists x\ P(x)$ becomes $P(A)$
5. Drop universal quantifiers
   - Since all quantifiers are now $\forall$, we don't need them
6. Distributive Law

Convert to FOPL, then CNF

1. John likes all kinds of food
2. Apples are food.
3. Chicken is food.
4. Anything that anyone eats and isn't killed by is food.
5. Bill eats peanuts and is still alive.
6. Sue eats everything Bill eats.

Prove Using Resolution

1. John likes peanuts.
2. Sue eats peanuts.
3. Sue eats apples.
4. What does Sue eat?
   - Translate to Sue eats X
   - Result is a valid binding for X in the proof

Another Example

- Steve only likes easy courses
- Science courses are hard
- All the courses in the basket weaving department are easy
- BK301 is a basket weaving course

- What course would Steve like?

Another Resolution Example