Chapter 10
Knowledge Representation

BBM 405 – Artificial Intelligence

Pinar Duygulu

Slides are mostly adapted from AIMA and MIT Open Courseware

Universal instantiation (UI)

- 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*

- E.g., $\forall x\ King(x) \land Greedy(x) \Rightarrow Evil(x)$ yields:

  $King(John) \land Greedy(John) \Rightarrow Evil(John)$

  $King(Richard) \land Greedy(Richard) \Rightarrow Evil(Richard)$

  $King(Father(John)) \land Greedy(Father(John)) \Rightarrow Evil(Father(John))$

  .

  .

  .

Existential instantiation (EI)

- For any sentence α, variable *v*, and constant symbol *k* that does not appear elsewhere in the knowledge base:

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

- E.g., $\exists x\ Crown(x) \wedge OnHead(x, John)$ yields:

$$Crown(C_1) \wedge OnHead(C_1, John)$$

provided C_1 is a new constant symbol, called a Skolem constant

Reduction to propositional inference

Suppose the KB contains just the following:

$\forall x$ King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x)
King(John)
Greedy(John)
Brother(Richard,John)

- Instantiating the universal sentence in all possible ways, we have:

King(John) $\wedge$ Greedy(John) $\Rightarrow$ Evil(John)
King(Richard) $\wedge$ Greedy(Richard) $\Rightarrow$ Evil(Richard)
King(John)
Greedy(John)
Brother(Richard,John)

- The new KB is propositionalized: proposition symbols are

King(John), Greedy(John), Evil(John), King(Richard), etc.

Reduction contd.

- Every FOL KB can be propositionalized so as to preserve entailment
- (A ground sentence is entailed by new KB iff entailed by original KB)
- Idea: propositionalize KB and query, apply resolution, return result
- Problem: with function symbols, there are infinitely many ground terms,
  - e.g., *Father*(*Father*(*Father*(*John*)))

Reduction contd.

Theorem: Herbrand (1930). If a sentence α is entailed by an FOL KB, it is entailed by a finite subset of the propositionalized KB

Idea: For $n = 0$ to ∞ do

create a propositional KB by instantiating with depth-n terms

see if α is entailed by this KB

Problem: works if α is entailed, loops if α is not entailed

Theorem: Turing (1936), Church (1936) Entailment for FOL is semidecidable (algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every nonentailed sentence.)

Problems with propositionalization

- Propositionalization seems to generate lots of irrelevant sentences.

- E.g., from:

  $\forall x$ King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x)

  King(John)

  $\forall y$ Greedy(y)

  Brother(Richard,John)

- it seems obvious that *Evil*(*John*), but propositionalization produces lots of facts such as *Greedy*(*Richard*) that are irrelevant

- With *p* *k*-ary predicates and *n* constants, there are $p \cdot n^k$ instantiations.

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)*

$\theta = \{x/John, y/John\}$ works

- Unify(α,β) = θ if αθ = βθ

p	q	θ
Knows(John,x)	Knows(John,Jane)	
Knows(John,x)	Knows(y,Elizabeth)	
Knows(John,x)	Knows(y,Mother(y))	
Knows(John,x)	Knows(x, Elizabeth)	

- Standardizing apart eliminates overlap of variables, e.g., Knows(z_{17}, Elizabeth)

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 $\alpha\theta = \beta\theta$

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

- Standardizing apart eliminates overlap of variables, e.g., Knows(z_{17}, Elizabeth)

Unification

- To unify *Knows(John,x)* and *Knows(y,z)*,

  θ = {y/John, x/z } or θ = {y/John, x/John, z/John}

- The first unifier is more general than the second.
- There is a single most general unifier (MGU) that is unique up to renaming of variables.

  MGU = { y/John, x/z }

The unification 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
```

The unification 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 $\{var/val\} \in \theta$ **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** failure
  else return add $\{var/x\}$ to θ

Generalized Modus Ponens (GMP)

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

where $p_i'\theta = p_i\,\theta$ for all i

p_1' is *King*(*John*) p_1 is *King*(*x*)

p_2' is *Greedy*(*y*) p_2 is *Greedy*(*x*)

θ is {x/John,y/John} q is *Evil*(*x*)

q θ is *Evil*(*John*)

- GMP used with KB of definite clauses (exactly one positive literal)
- All variables assumed universally quantified

Soundness of GMP

- Need to show that

$$p_1', \ldots, p_n', (p_1 \wedge \ldots \wedge p_n \Rightarrow q) \models q\theta$$

provided that $p_i'\theta = p_i\theta$ for all *I*

- Lemma: For any sentence *p*, we have $p \models p\theta$ by UI

1. $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \models (p_1 \wedge \ldots \wedge p_n \Rightarrow q)\theta = (p_1\theta \wedge \ldots \wedge p_n\theta \Rightarrow q\theta)$
2. p_1', \; …, \;$p_n' \models p_1' \wedge \ldots \wedge p_n' \models p_1'\theta \wedge \ldots \wedge p_n'\theta$
3. From 1 and 2, $q\theta$ follows by ordinary Modus Ponens

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 of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

- Prove that Col. West is a criminal

Example knowledge base contd.

... 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, i.e., $\exists x\ Owns(Nono,x) \wedge Missile(x)$:

$Owns(Nono,M_1)$ and $Missile(M_1)$

… all of its missiles were sold to it by Colonel West

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

Missiles are weapons:

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

An enemy of America counts as "hostile“:

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

West, who is American …

$American(West)$

The country Nono, an enemy of America …

$Enemy(Nono,America)$

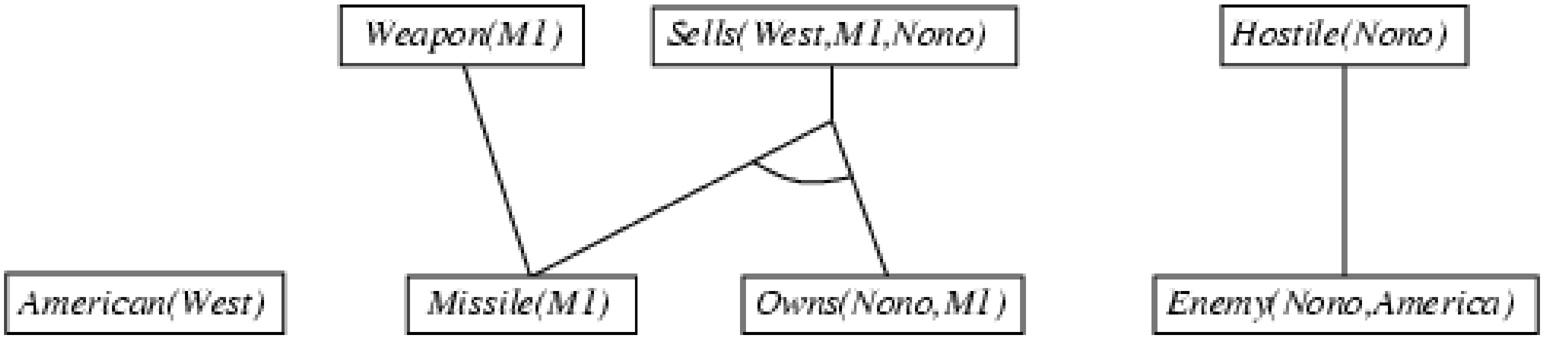
Forward chaining algorithm

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

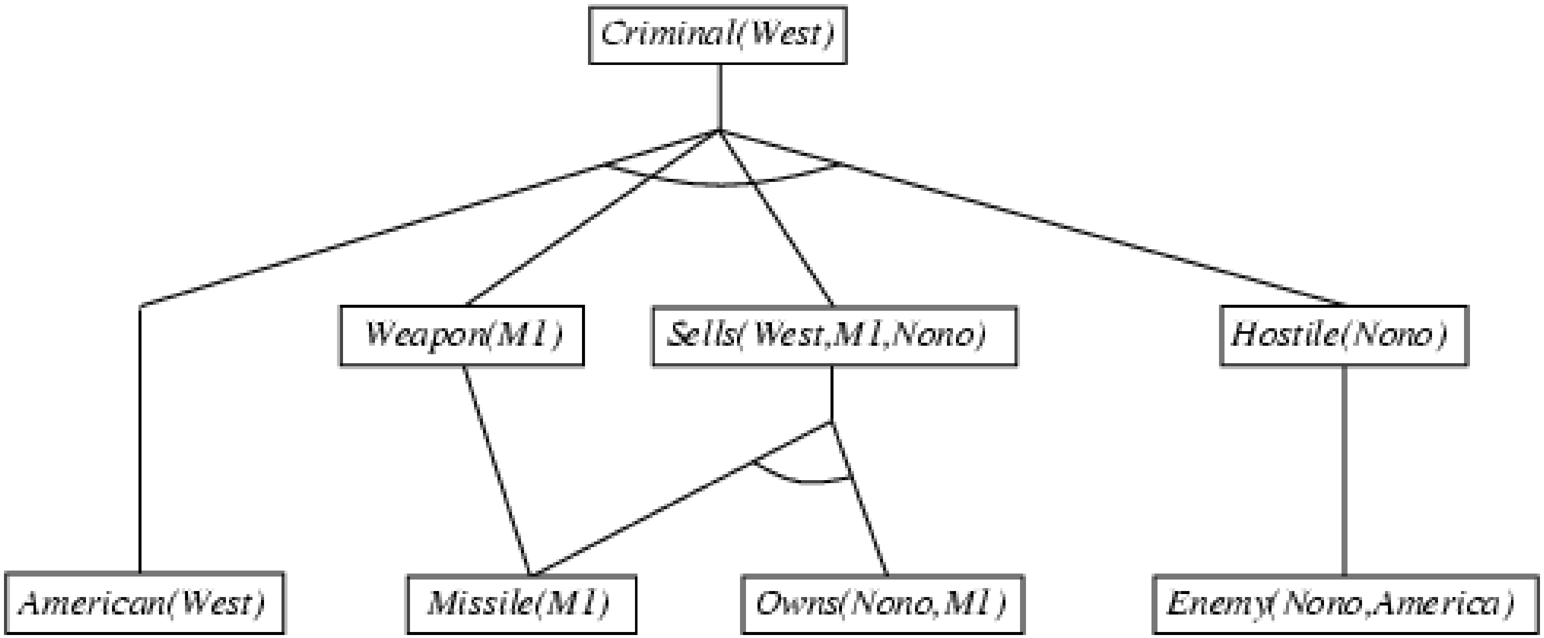
repeat until *new* is empty
  $new \leftarrow \{\ \}$
  for each sentence r **in** KB **do**
    $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-APART(r)
    for each θ such that $(p_1 \wedge \ldots \wedge p_n)\theta = (p'_1 \wedge \ldots \wedge p'_n)\theta$
      for some $p'_1, \ldots, p'_n$ 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*

Forward chaining proof

Forward chaining proof



Forward chaining proof



Properties of forward chaining

- Sound and complete for first-order definite clauses

- Datalog = first-order definite clauses + no functions
- FC terminates for Datalog in finite number of iterations

- May not terminate in general if α is not entailed

- This is unavoidable: entailment with definite clauses is semidecidable

Efficiency of forward chaining

Incremental forward chaining: no need to match a rule on iteration k if a premise wasn't added on iteration k-1

$\Rightarrow$ match each rule whose premise contains a newly added positive literal

Matching itself can be expensive:

Database indexing allows O(1) retrieval of known facts

- e.g., query *Missile(x)* retrieves $Missile(M_1)$

Forward chaining is widely used in deductive databases

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

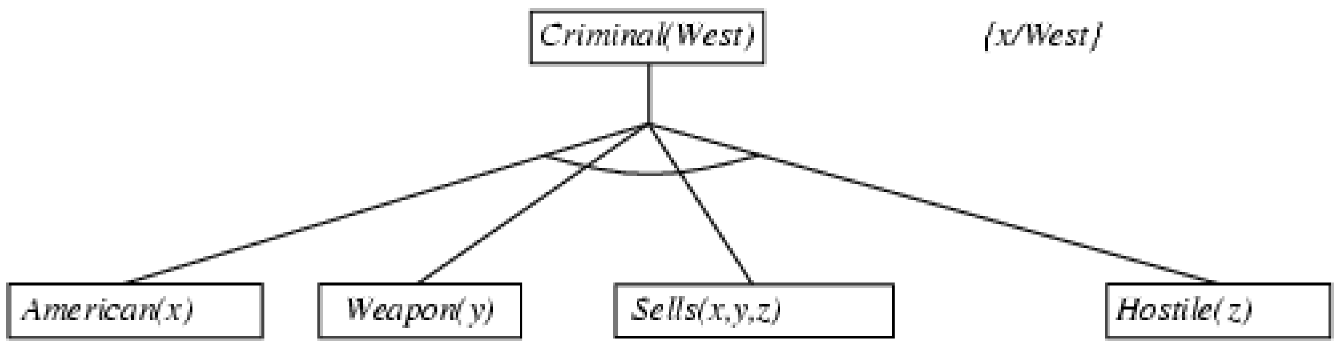
  if $goals$ is empty **then return** $\{\theta\}$
  $q' \leftarrow \textsc{Subst}(\theta, \textsc{First}(goals))$
  for each r **in** KB where $\textsc{Standardize-Apart}(r) = (\, p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
    and $\theta' \leftarrow \textsc{Unify}(q, q')$ succeeds
   $ans \leftarrow \text{FOL-BC-}\textsc{Ask}(KB, [p_1, \ldots, p_n | \textsc{Rest}(goals)], \textsc{Compose}(\theta, \theta')) \cup ans$
  return ans

SUBST(COMPOSE(θ_1, θ_2), p) =
 SUBST(θ_2, SUBST(θ_1, p))

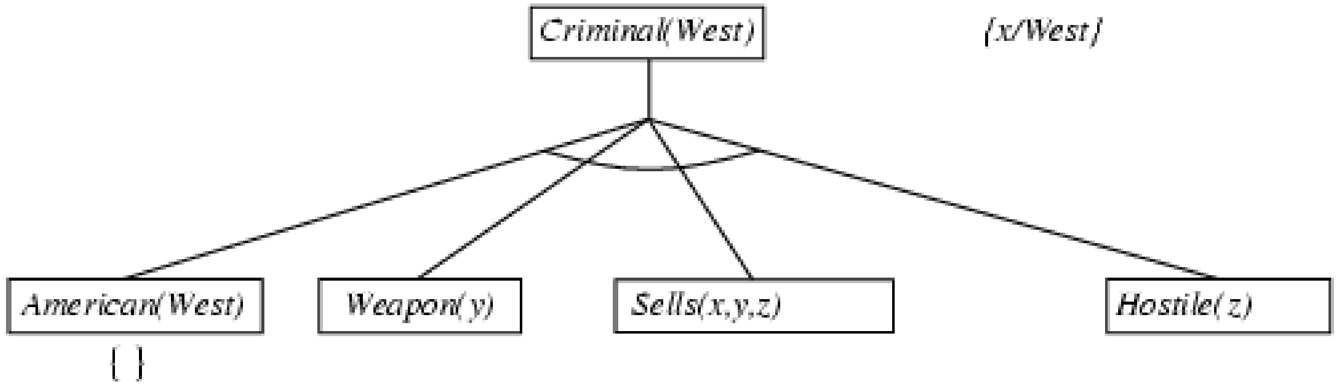
Backward chaining example

$\boxed{Criminal(West)}$

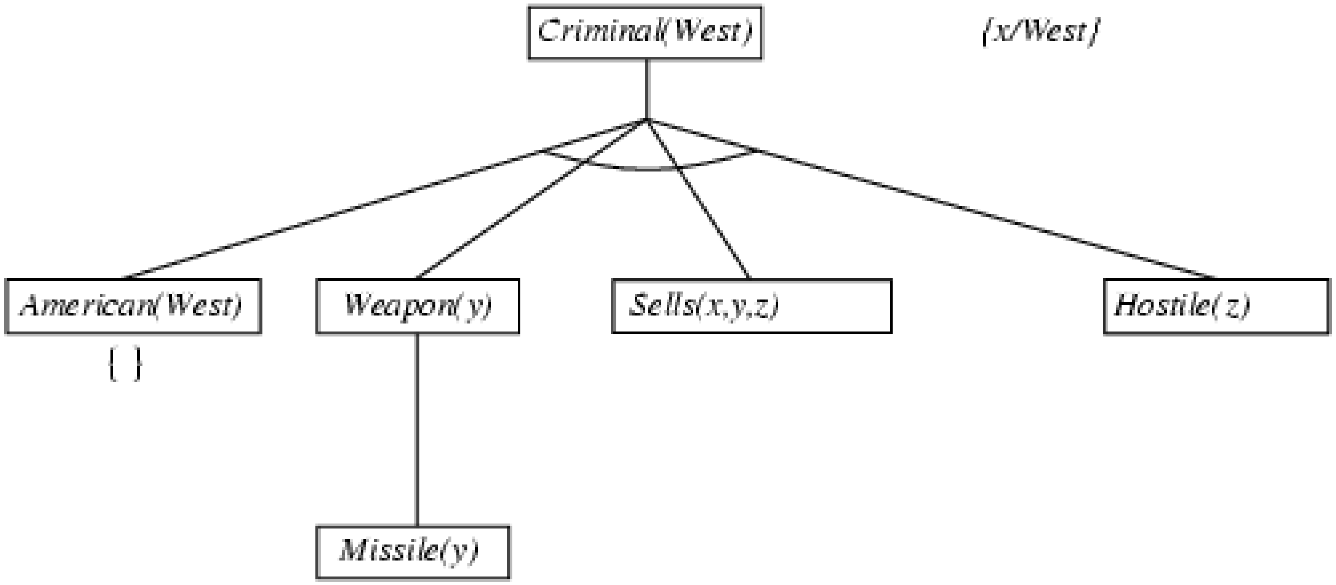
Backward chaining example



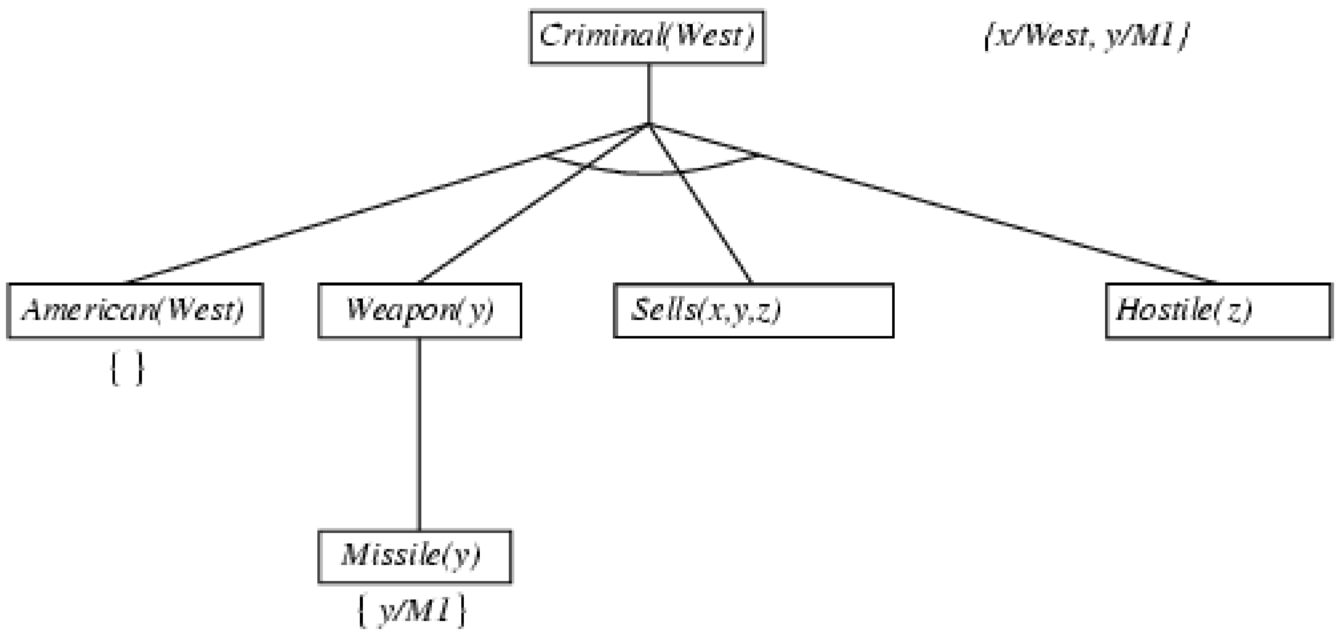
Backward chaining example



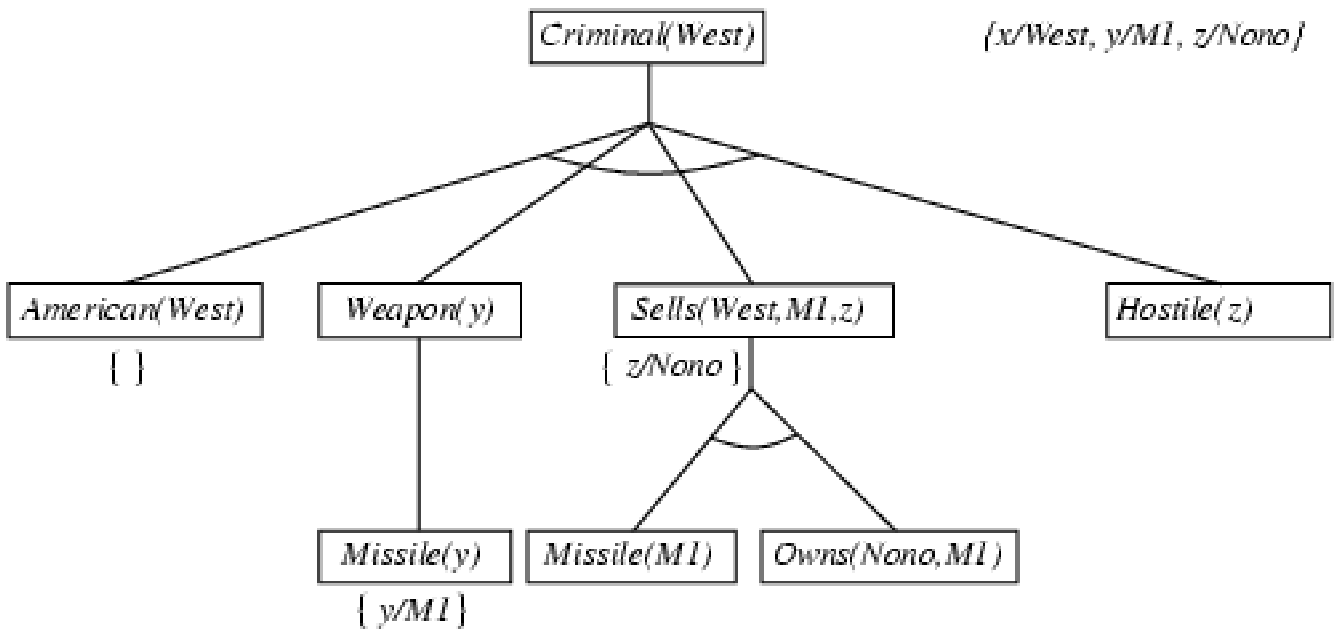
Backward chaining example



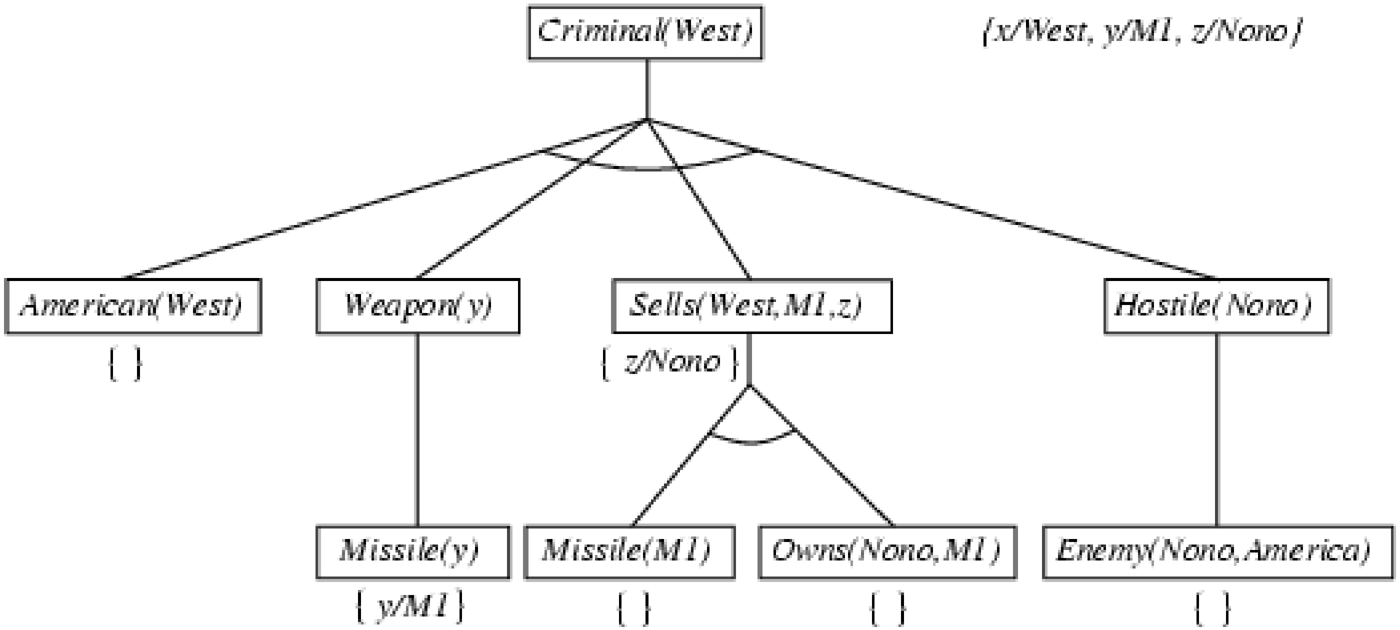
Backward chaining example



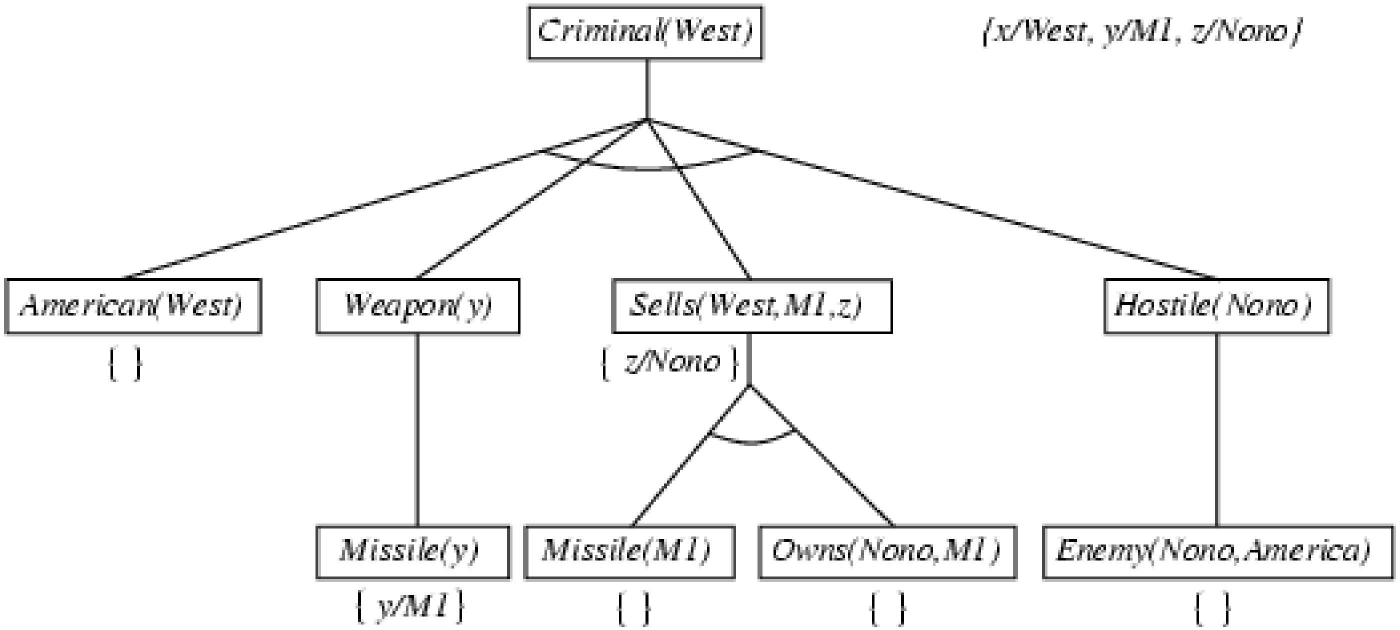
Backward chaining example



Backward chaining example



Backward chaining example



Properties of backward chaining

- Depth-first recursive proof search: space is linear in size of proof
- Incomplete due to infinite loops
  - $\Rightarrow$ fix by checking current goal against every goal on stack
- Inefficient due to repeated subgoals (both success and failure)
  - $\Rightarrow$ fix using caching of previous results (extra space)
- Widely used for logic programming

Logic programming: Prolog

- Algorithm = Logic + Control
- Basis: backward chaining with Horn clauses + bells & whistles
- Program = set of clauses = `head :- literal`$_1$`,` … `literal`$_n$.

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

- Depth-first, left-to-right backward chaining
- Built-in predicates for arithmetic etc., e.g., `X is Y*Z+3`
- Built-in predicates that have side effects (e.g., input and output
- predicates, assert/retract predicates)
- Closed-world assumption ("negation as failure")
  - e.g., given `alive(X) :- not dead(X).`
  - `alive(joe)` succeeds if `dead(joe)` fails

Logic in the real world

- Encode information formally in web pages
- Business rules
- Airfare pricing

Airfare Pricing

- Ignore, for now, finding the best itinerary
- Given an itinerary, what's the least amount we can pay for it?
- Can't just add up prices for the flight legs; different prices for different flights in various combinations and circumstances

Fare Restrictions

- Passenger under 2 or over 65
- Passenger accompanying someone paying full fare
- Doesn't go through an expensive city
- No flights during rush hour
- Stay over Saturday night
- Layovers are legal
- Round-the-world itinerary that doesn't backtrack
- Regular two phase round-trip
- No flights on another airline
- This fare would not be cheaper than the standard price

Ontology

- What kinds of things are there in the world?
- What are their properties and relations?

> Ontology is the science of something and of nothing, of being and not-being, of the thing and the mode of the thing, of substance and accident.

> The Role of Ontological Engineering in B2B Net Markets

Leibniz

6.034 – Spring 03 • 9

Airfare Domain Ontology

- passenger
- flight
- city
- airport
- terminal
- flight segment (list of flights, to be flown all in one "day")
- itinerary (a passenger and list of flight segments)
- list
- number

Representing Properties

- Object P is red
  - Red(P)
  - Color(P, Red)
  - color(P) = Red
  - Property(P, Color, Red)

- All the blocks in stack S are the same color

$$\exists c.\ \forall b.\ \mathrm{In}(b, S) \rightarrow \mathrm{Color}(b, c)$$

- All the blocks in stack S have the same properties

$$\forall p.\ \exists v.\ \forall b.\ \mathrm{In}(b, S) \rightarrow \mathrm{Property}(b, p, v)$$

Basic Relations

- Age(passenger, number)
- Nationality(passenger, country)
- Wheelchair(passenger)
- Origin(flight, airport)
- Destination(flight, airport)
- Departure_Time(flight, number)
- Arrival_Time(flight, number)
- Latitude(city, number)
- Longitude(city, number)
- In_Country(city, country)
- In_City(airport, city)
- Passenger(itinerary, passenger)
- Flight_Segments(itinerary, passenger, segments)
- Nil
- cons(object,list) => list

Age(Fred, 47)
Nationality(Fred, US)
~Wheelchair(Fred)

Defined Relations

- Define complex relations in terms of basic ones
- Like using subroutines

$$\forall i.\ P(i) \wedge Q(i) \rightarrow \text{Qualifies37}(i)$$

- Implication rather than equivalence
  - easier to specify definitions in pieces

$$\forall i.\ R(i) \wedge S(i) \rightarrow \text{Qualifies37}(i)$$

  - can't use the other direction

$$\text{Qualifies37}(i) \rightarrow ?$$

  - if you need it, write the equivalence

$$\forall i.\ (P(i) \wedge Q(i)) \vee (R(i) \wedge S(i)) \leftrightarrow \text{Qualifies37}(i)$$

Infant Fare

$$\forall i, a, p.\ \mathrm{Passenger}(i, p) \wedge \mathrm{Age}(p, a) \wedge a < 2 \rightarrow \mathrm{InfantFare}(i)$$

Rules and Logic Programming

- Language of logic is extremely powerful.
- Say what's true, not how to use it.
  - $\forall x, y\ (\exists z\ Parent(x,z) \wedge Parent(z,y)) \leftrightarrow GrandParent(x,y)$
  - Given parents, find grandparents
  - Given grandparents, find parents
- But, resolution theorem-provers are too inefficient!
- To regain practicality:
  - Limit the language
  - Simplify the proof algorithm
- Rule-Based Systems
- Logic Programming

Horn Clauses

- A clause is Horn if it has at most one positive literal
  - $\neg P_1 \vee \ldots \vee \neg P_n \vee Q$ (Rule)
  - Q (Fact)
  - $\neg P_1 \vee \ldots \vee \neg P_n$ (Consistency Constraint)
- We will not deal with Consistency Constraints
- Rule Notation
  - $P_1 \wedge \ldots \wedge P_n \rightarrow Q$ (Logic)
  - If $P_1 \ldots P_n$ Then Q (Rule-Based System)
  - Q :- $P_1, \ldots, P_n$ (Prolog)
- P_i are called antecedents (or body)
- Q is called the consequent (or head)

Limitations

- Cannot conclude negation
  - $P \rightarrow \neg Q$
  - $\neg P \vee \neg Q$: Consistency constraint
  - $\neg P$: Consistency constraint
- Cannot conclude (or assert) disjunction
  - $P_1 \wedge P_2 \rightarrow Q_1 \vee Q_2$
  - $Q_1 \vee Q_2$
  - These are not Horn

Inference: Backchaining

- To "prove" a literal C
  - Push C and an Ans literal on a stack
  - Repeat until stack only has Ans literal or no actions available.
    - Pop literal L off of stack
    - Choose [with backup] a rule (or fact) whose consequent unifies with L
      - Push antecedents (in order) onto stack
      - Apply unifier to entire stack
      - Rename variables on stack
    - If no match, fail [backup to last choice]

Backchaining and Resolution

- Backchaining is just resolution
- To prove C (propositional case)
  - Negate C ⇒ ¬ C
  - Find rule $\neg P_1 \lor \ldots \lor \neg P_n \lor C$
  - Resolve to get $\neg P_1 \lor \ldots \lor \neg P_n$
  - Repeat for each negative literal
- First order case introduces unification but otherwise the same.

Proof Strategy

- Depth-First search for a proof
- Order matters
  - Rule order
    - try ground facts first
    - then rules in given order
  - Antecedent order
    - left to right
- More predictable, like a program, less like logic

Example

```
1.  Father(A,B)          ; ground fact
2.  Mother(B,C)          ; ground fact
3.  GrandP(?x,?z) :- Parent(?x,?y),Parent(?y,?z)
4.  Parent(?x,?y) :- Father(?x,?y)
5.  Parent(?x,?y) :- Mother(?x,?y)
```

Example

```
1. Father(A,B)          ; ground fact
2. Mother(B,C)          ; ground fact
3. GrandP(?x,?z):- Parent(?x,?y),Parent(?y,?z)
4. Parent(?x,?y):- Father(?x,?y)
5. Parent(?x,?y):- Mother(?x,?y)
```

- **Prove:**
  GrandP(?g,C), Ans(?g)

Example

1. Father(A,B) ; ground fact
2. Mother(B,C) ; ground fact
3. GrandP(?x,?z) :- Parent(?x,?y),Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

- Prove:
  GrandP(?g,C), Ans(?g)
  - [3,?x/?g,?z/C; ?y⇒$?y_1$,?g⇒$?g_1$]
- Parent($?g_1$,$?y_1$), Parent($?y_1$,C), Ans($?g_1$)
  - [4,?x/$?g_1$,?y/$?y_1$; $?y_1$⇒$?y_2$,$?g_1$⇒$?g_2$]
- Father($?g_2$,$?y_2$), Parent($?y_2$,C), Ans($?g_2$)
  - [1,$?g_2$/A,$?y_2$/B]
- Parent(B,C), Ans(A)
  - [4,?x/B,?y/C]
- Father(B,C), Ans(A)
- <fail>

Example

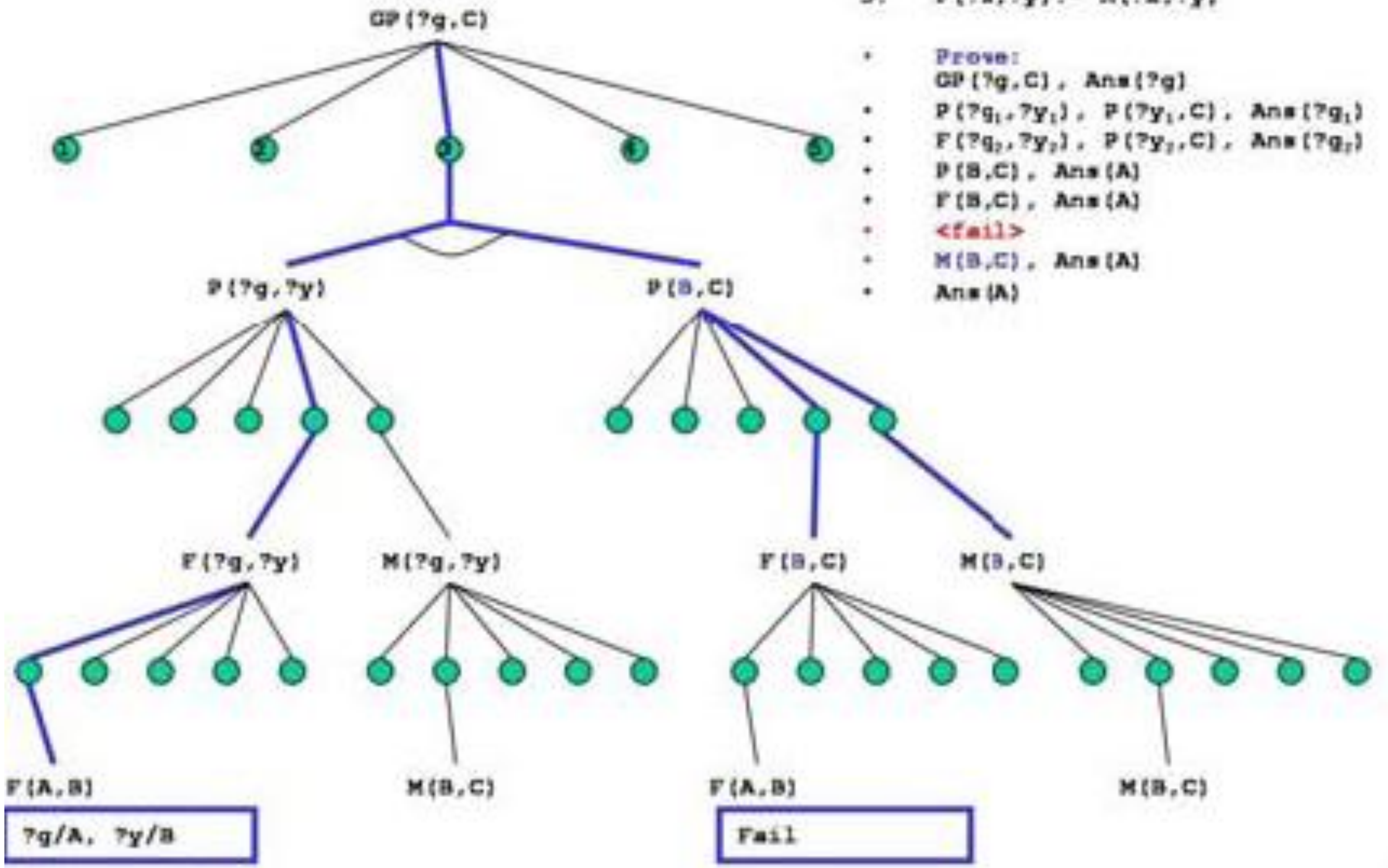
1. Father(A,B) ; ground fact
2. Mother(B,C) ; ground fact
3. GrandP(?x,?z) :- Parent(?x,?y),Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

- Prove:
  GrandP(?g,C), Ans(?g)
  - [3,?x/?g,?z/C; ?y$\Rightarrow$?y_1,?g$\Rightarrow$?g_1]
- Parent(?g_1,?y_1), Parent(?y_1,C), Ans(?g_1)
  - [4,?x/?g_1,?y/?y_1; ?$y_1\Rightarrow$?y_2,?$g_1\Rightarrow$?g_2]
- Father(?g_2,?y_2), Parent(?y_2,C), Ans(?g_2)
  - [1,?g_2/A,?y_2/B]
- Parent(B,C), Ans(A)
  - [4,?x/B,?y/C]
- Father(B,C), Ans(A)
- <fail>
  - [5,?x/B,?y/C]
- Mother(B,C), Ans(A)
  - [2]
- Ans(A)

Proof Tree

1. F(A,B)
2. M(B,C)
3. GP(?x,?z) :- P(?x,?y), P(?y,?z)
4. P(?x,?y) :- F(?x,?y)
5. P(?x,?y) :- M(?x,?y)

- Prove:
  GP(?g,C), Ans(?g)
- P(?g_1,?y_1), P(?y_1,C), Ans(?g_1)
- F(?g_2,?y_2), P(?y_2,C), Ans(?g_2)
- P(B,C), Ans(A)
- F(B,C), Ans(A)
- <fail>
- M(B,C), Ans(A)
- Ans(A)



Relations not Functions

1. Father(A,B) ; ground fact
2. Mother(B,C) ; ground fact
3. GrandP(?x,?z) :- Parent(?x,?y),Parent(?y,?z)
4. Parent(?x,?y) :- Father(?x,?y)
5. Parent(?x,?y) :- Mother(?x,?y)

- Prove:
  GrandP(A,?f), Ans(?f)
  - [3,?x/A,?z/?f; ?y⇒?y_1,?f⇒?f_1]
- Parent(A,?y_1), Parent(?y_1,?f_1), Ans(?f_1)
  - [4,?x/A,?y/?y_1; ?y_1⇒?y_2,?f_1⇒?f_2]
- Father(A,?y_2), Parent(?y_2,?f_2), Ans(?f_2)
  - [1,?y_2/B; ?f_2⇒?f_3]
- Parent(B,?f_3), Ans(?f_3)
  - [4,?x/B,?y/?f_3; ?f_3⇒?f_4]
- Father(B,?f_4), Ans(?f_4)
- <fail>
  - [5,?x/B,?y/?f_3; ?f_3⇒?f_4]
- Mother(B,?f_4), Ans(?f_4)
  - [2,?f_4/C]
- Ans(C)

Order Revisited

- Given
  1. parent(A,B)
  2. parent(B,C)
  3. ancestor(?x,?z) :- parent(?x,?z)
  4. ancestor(?x,?z) :- parent(?x,?y), ancestor(?y,?z)
  - Prove:
    ancestor(?x,C), Ans(?x)
  - …
  - Ans(A)
- How about:
  1. parent(A,B)
  2. parent(B,C)
  3. ancestor(?x,?z) :- ancestor(?y,?z), parent(?x,?y)
  4. ancestor(?x,?z) :- parent(?x,?z)
  - Prove:
    ancestor(?x,C), Ans(?x)
  - …
  - <error: stack overflow>
- Clauses examined top to bottom and literals left to right.
  This is not logic!

Logic Programming

- So far, not much like programming
- But, this framework can be used as the basis of a general purpose programming language
- Prolog is the most widely used logic programming language
- For example:
  - Gnu Prolog http://www.gnu.org/software/prolog/prolog.html
  - SWI Prolog http://www.swi-prolog.org/
  - SICStus Prolog http://www.sics.se/sicstus/
  - Visual Prolog http://www.visual-prolog.com/
  - ...