Organization of Programming Languages
CS 3200/5200N

Lecture 13

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Logic Programming

- Introduction
- Predicate Calculus
- Inference Rules
- Logic Programming Languages
- Elements of Prolog
- Applications of Logic Programming

Introduction: Logic Programming Languages

- Express programs in a form of **symbolic logic**:
  - **First order predicate calculus**:
    - **Horn clauses**.
- Produce results using a logical inference process:
  - **Resolution**:
    - **Backward chaining** (top-down resolution).
- Sometimes called **declarative** programming languages:
  - *declarative*: specify only properties of the results (***what***).
  - *procedural*: specify procedures for obtaining the results (***how***).

First Order Predicate Calculus

- First Order Logic (FOL) assumes the world contains:
  - Objects: people, houses, numbers, colors, baseball games, wars, …
  - Relations: brother of, bigger than, part of, comes between, …
  - Properties: red, round, prime, multistoried, …
  - Functions: father of, best friend, one more than, plus, …

- Examples:
  - "Evil King John ruled England in 1200."
    - Objects:
    - Relations:
    - Properties:

Syntax of FOL: Basic Elements

- Used to build **terms**:
  - Constants: John, 2, A, B, C, ...
  - Functions: sqrt, cosine, fatherOf, leftLegOf, ...
  - Variables: x, y, a, b, s, ...

- Used to build **sentences**:
  - Predicates: Round, Brother, >, ...
  - Connectives: ¬, ⇒, ∧, ∨, ⇔
  - Equality: =
  - Quantifiers: ∀, ∃

Atomic Sentences

- Every FOL expression is a sentence, which represents a fact.

$$AtomicSentence \rightarrow Predicate\ (Term_1, ..., Term_n)$$
$$\mid\ Term_1 = Term_2$$
$$Term \rightarrow Function\ (Term_1,...,Term_n)$$
$$\mid\ Constant$$
$$\mid\ Variable$$

- Examples:
  - *Brother(John, Richard)*
  - *>(length(leftLegOf(Richard)), length(leftLegOf(John)))*

Complex Sentences

- Complex sentences are made from atomic sentences using connectives:

  $\neg S,\ S_1 \wedge S_2,\ S_1 \vee S_2,\ S_1 \Rightarrow S_2,\ S_1 \Leftrightarrow S_2,$

- Examples:

  Sibling(John,Richard) $\Rightarrow$ *Sibling(Richard,John)*

  $>(1,2) \vee \leq (1,2)$

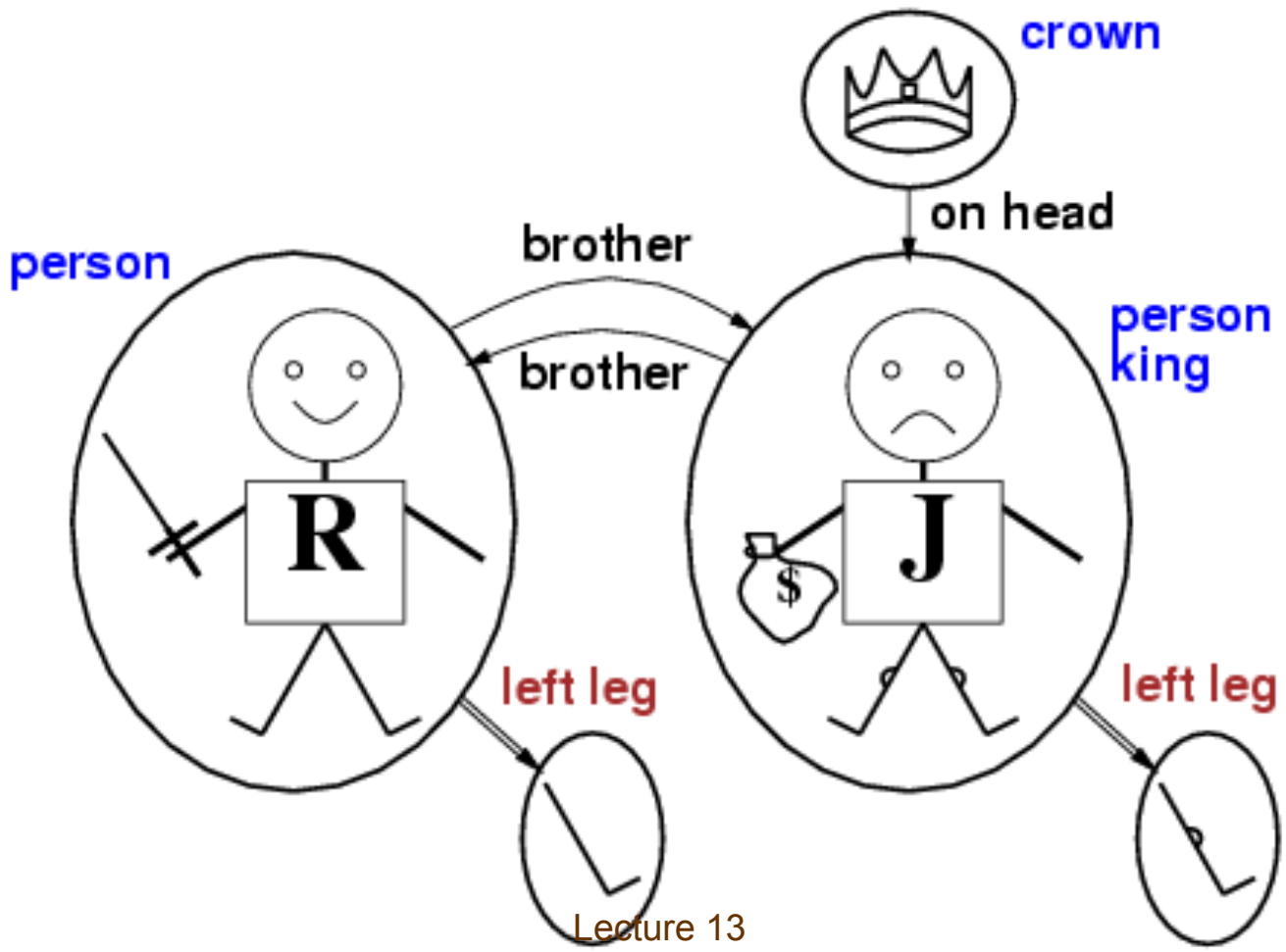
  $>(1,2) \wedge \neg >(1,2)$

Truth in FOL

- Sentences are true/false with respect to a model and an interpretation.
  - Model contains objects and relations among them.
  - Interpretation specifies referents for:
    constant symbols → objects
    predicate symbols → relations
    function symbols → functional relations

- An atomic sentence $Predicate(term_1,...,term_n)$ is true iff the objects referred to by $term_1,...,term_n$ are in the relation referred to by *Predicate*.

Model in FOL: Example



Universal Quantification

- ∀<*variables*> <*sentence*>

  "Everyone at OU is smart": ∀x At(x,OU) ⇒ Smart(x)

- "∀x *P*" is true in a model *M* iff *P* is true for every possible instantiation of *x* with an object in the model.

- Roughly speaking, equivalent to the conjunction of instantiations of *P*:

  At(John, OU) ⇒ Smart(John)
  ∧ At(Richard, OU) ⇒ Smart(Richard)
  ∧ At(OU, OU) ⇒ Smart(OU)
  ∧ ...

Existential Quantification

- ∃ <*variables*> <*sentence*>

  "Someone at OU is smart": ∃*x* At(x,OU) ∧ Smart(x)

- "∃*x P*" is true in a model *M* iff *P* is true with *x* being some possible object in the model.

- Roughly speaking, equivalent to the disjunction of instantiations of *P*:

  At(John, OU) ∧ Smart(John)
  ∨ At(Richard, OU) ∧ Smart(Richard)
  ∨ At(OU, OU) ∧ Smart(OU)
  ∨ …

Properties of Quantifiers

- ∀x ∀y is the same as ∀y ∀x
- ∃x ∃y is the same as ∃y ∃x

- ∃x ∀y is not the same as ∀y ∃x
  - ∃x ∀y Loves(x,y)
    - "There is a person who loves everyone in the world"
  - ∀y ∃x Loves(x,y)
    - "Everyone in the world is loved by at least one person"

- **Quantifier duality**: each can be expressed using the other.
  - ∀x Likes(x, IceCream) <=> ¬∃x ¬Likes(x, IceCream)
  - ∃x Likes(x, Broccoli) <=> ¬∀x ¬Likes(x, Broccoli)

Equality

- $term_1 = term_2$ is true under a given interpretation if and only if $term_1$ and $term_2$ refer to the same object.

- Example: definition of *Sibling* in terms of *Parent*:

$$\forall x,y\ \text{Sibling}(x,y) \Leftrightarrow [\neg(x = y) \wedge \exists m,f \neg (m = f) \wedge \text{Parent}(m,x) \wedge \text{Parent}(f,x) \wedge \text{Parent}(m,y) \wedge \text{Parent}(f,y)]$$

Using FOL: The Kinship Domain

- "Brothers are siblings:

  $\forall x,y\ Brother(x,y) \Rightarrow Sibling(x,y)$

- "One's mother is one's female parent"

  $\forall m,c\ Mother(c) = m \Rightarrow (Female(m) \wedge Parent(m,c))$

- The "Sibling" relation is symmetric:

  $\forall x,y\ Sibling(x,y) \Rightarrow Sibling(y,x)$

Inference in Predicate Logic

- **Inference = Reasoning**: any process by which conclusions are reached.

- **Logical Inference** = a procedure that generate a new sentence α that is necessarily true, given that the old sentences in the KB (Knowledge Base) are true.

- **Entailment**: the relation between KB and the new sentence α.

  $KB \models \alpha$

Logical Inference: Example

- Knowledge Base (KB):

  $\forall x\ \text{King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$
  $\text{King}(\text{John})$
  $\forall y\ \text{Greedy}(y)$
  $\text{Brother}(\text{Richard}, \text{John})$

- Sentence (α):

  $\text{Evil}(\text{John})$

- Entailment: KB $\models \alpha$?

Inference Rules

- **Resolution** (Herbrand 1965):
  - a **complete** procedure for proving sentences by **refutation**.
  - sentences need to be reformulated into a **normal form**.
    - **conjunctive normal form**, or
    - **implicative normal form**.

- **Modus Ponens**:
  - uses **unification** in a **forward-chaining** or **backward-chaining** algorithm; is not complete.
  - sentences need to be reformulated into a **canonical form**:
    - **nonnegated atomic sentences**, or
    - **Horn clauses**: $p_1 \wedge \ldots \wedge p_n \Rightarrow q$, where p_i and q are atoms.

Modus Ponens: Canonical Form

- Convert sentences into canonical form sentences:
  - **Existential Elimination** and **And Elimination**:

    "∃ x Owns (John, x) ∧ Missile (x)" is replaced with:

    "Owns(John, M1)"

    "Missile(M1)"
  - Transform complex sentences into Horn clauses:

    "$\neg P_1 \vee \neg P_2 \Rightarrow Q$" is replaced with $P_1 \wedge P_2 \Rightarrow Q$

- Not all sentences can be transformed into the canonical form for Modus Ponens:

  => Modus Ponens is not a complete inference procedure.

Modus Ponens

Generalized Modus Ponens:

- For atomic sentence p_i, p'_i and q and a substitution of variables θ such that Subst(θ, p_i') = Subst(θ, p_i) for all i:

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

Example:

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*)

Subst(θ, q) is *Evil*(*John*)

Unification

- We can use Modus Ponens 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 Subst(θ, α) = Subst(θ, β)

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

Unification

- We can use Modus Ponens 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 Subst(θ, α) = Subst(θ, β)

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

Unification

- We can use Modus Ponens 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 Subst(θ, α) = Subst(θ, β)

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))	
Knows(John,x)	Knows(x,OJ)	

Unification

- We can use Modus Ponens 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 Subst(θ, α) = Subst(θ, β)

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)	

Unification

- We can use Modus Ponens 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 Subst(θ, α) = Subst(θ, β)

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}

Most General Unifier

- 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

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 "Colonel West is a criminal".

Modus Ponens: Example

... 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 …

$Enemy(Nono,America)$

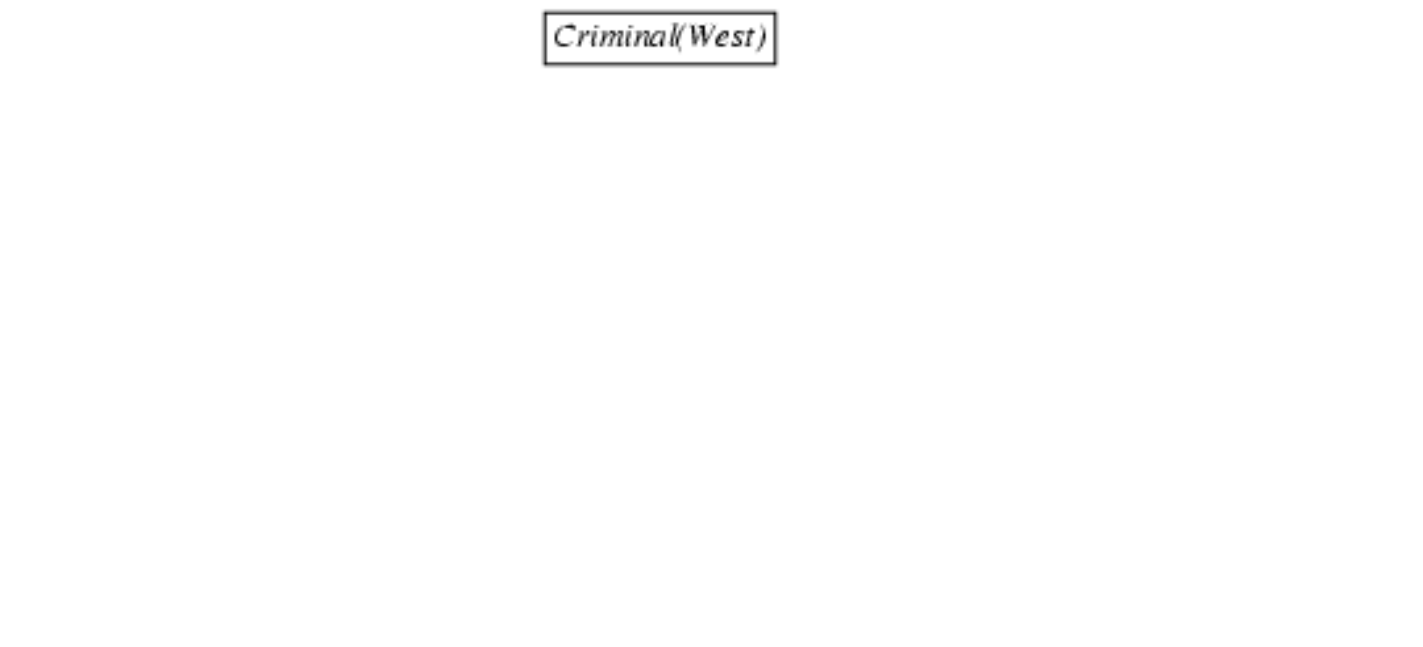
Backward-Chaining Algorithm

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

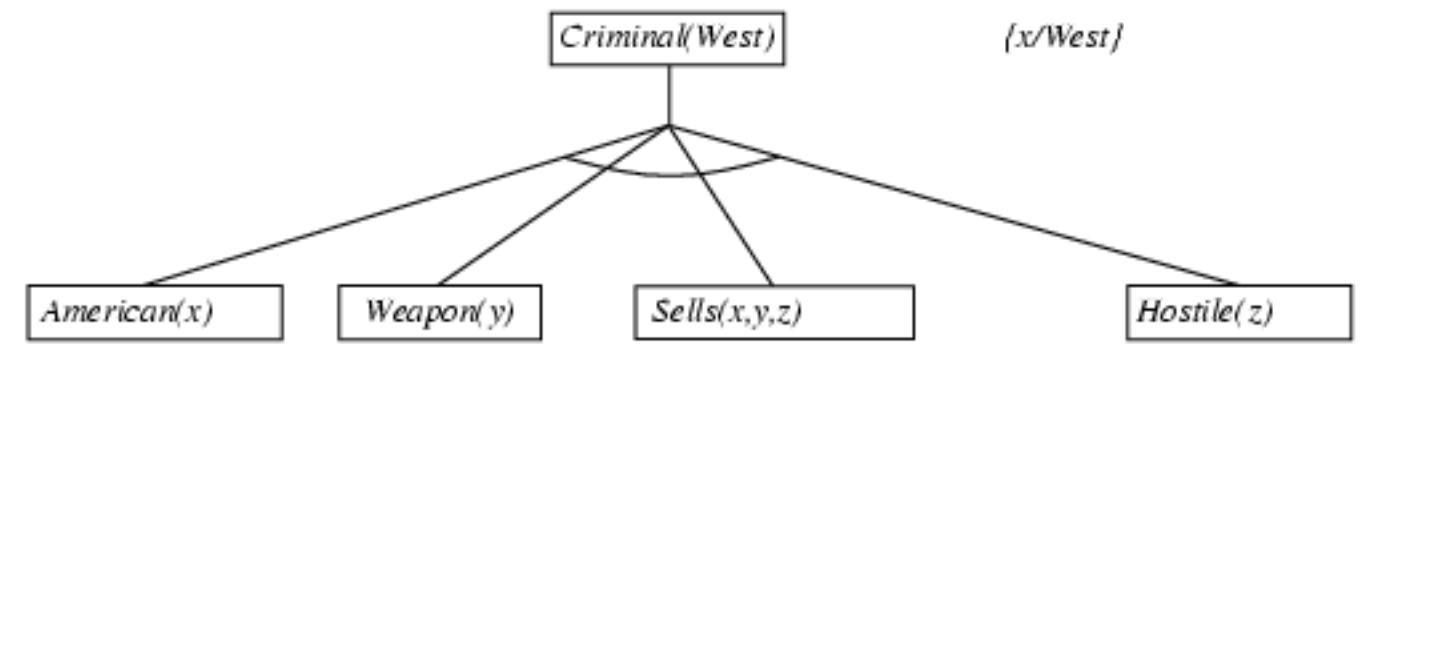
  if $goals$ is empty **then return** $\{\theta\}$
  $q' \leftarrow$ SUBST(θ, FIRST($goals$))
  for each r **in** KB where STANDARDIZE-APART(r) $= (p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
    and $\theta' \leftarrow$ UNIFY(q, q') succeeds
    $ans \leftarrow$ FOL-BC-ASK(KB, $[p_1, \ldots, p_n | \text{REST}(goals)]$, COMPOSE($\theta, \theta'$)) $\cup$ ans
  return ans

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

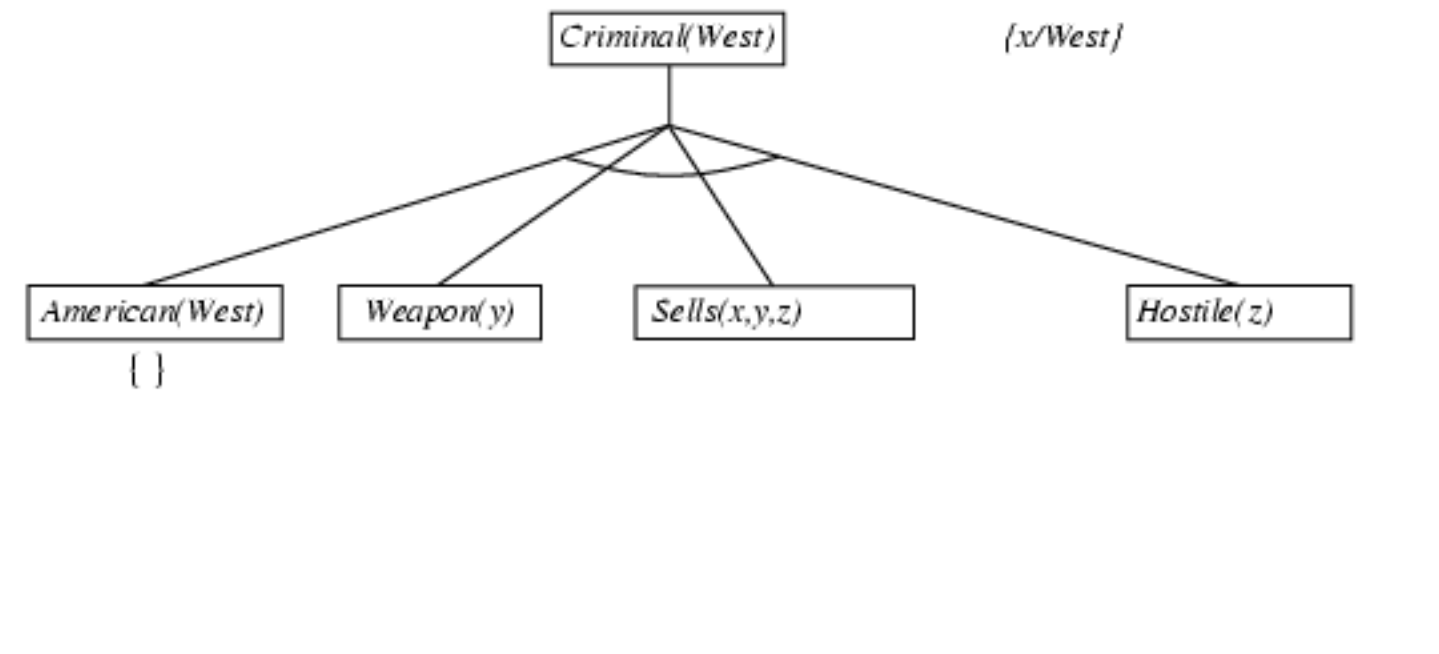
Backward chaining example



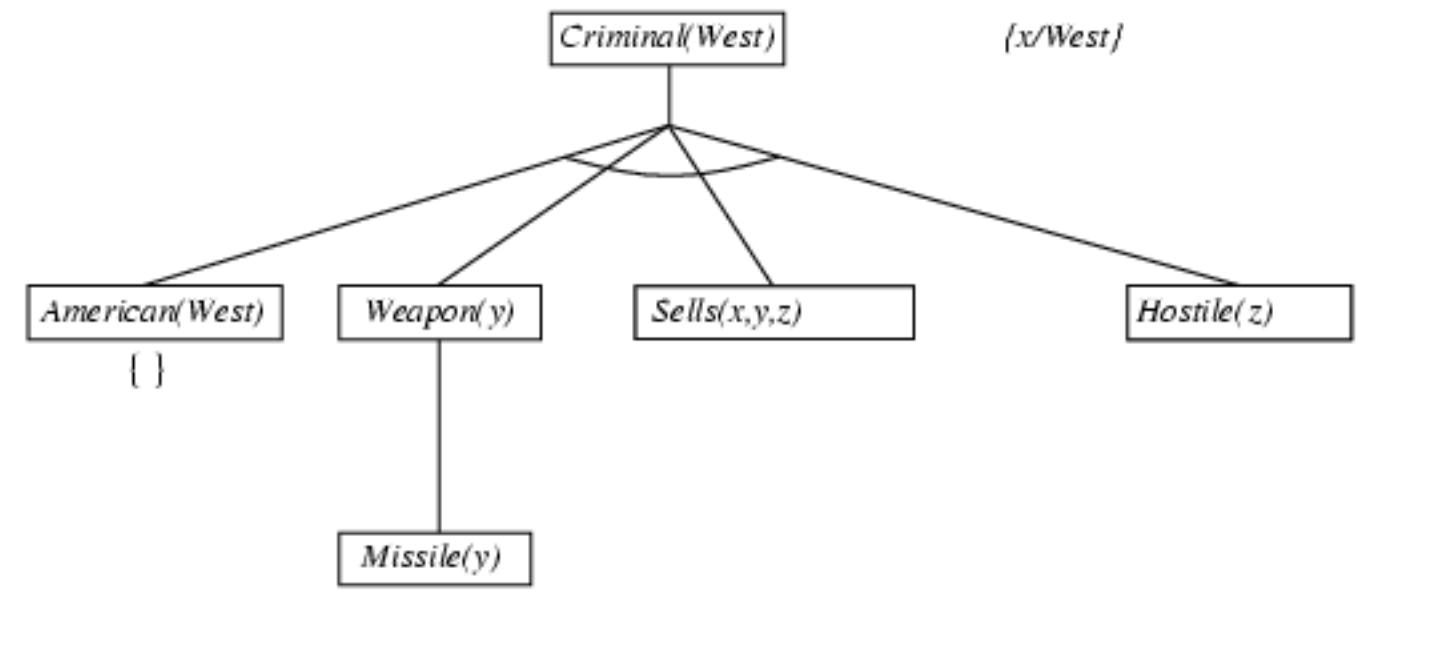
Backward chaining example



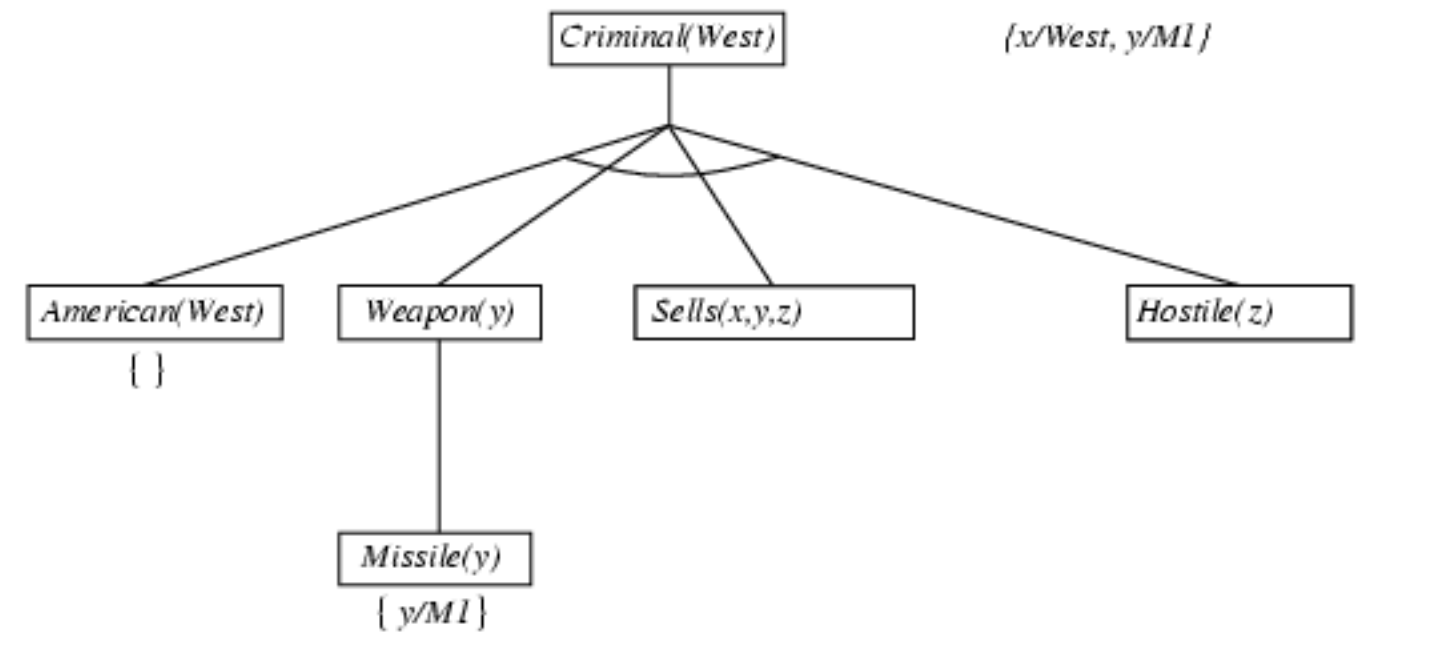
Backward chaining example



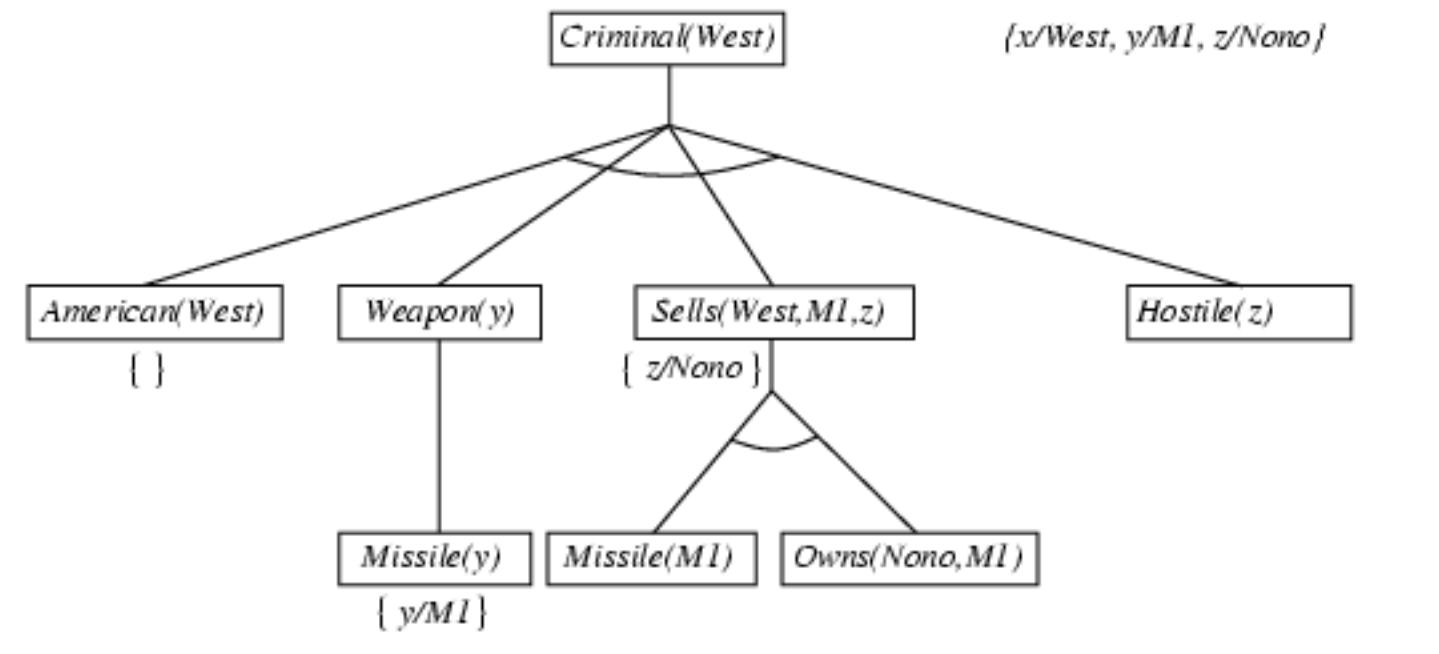
Backward chaining example



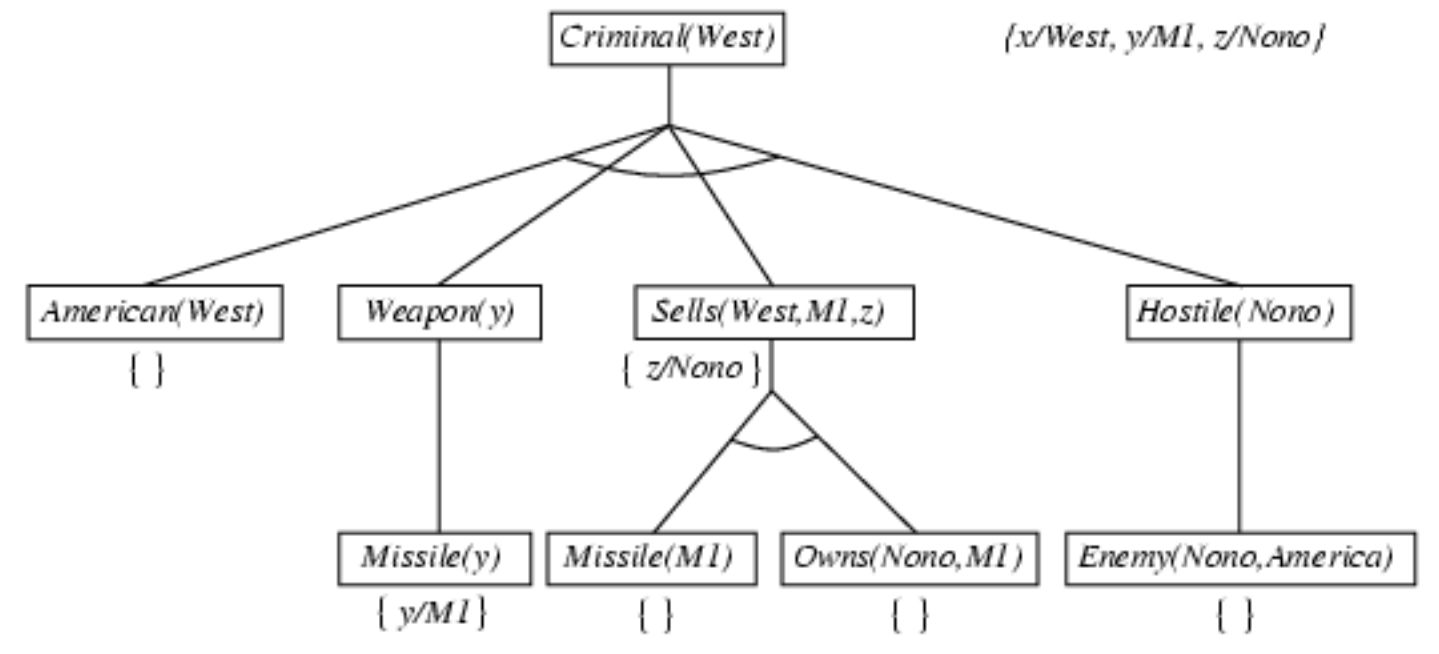
Backward chaining example



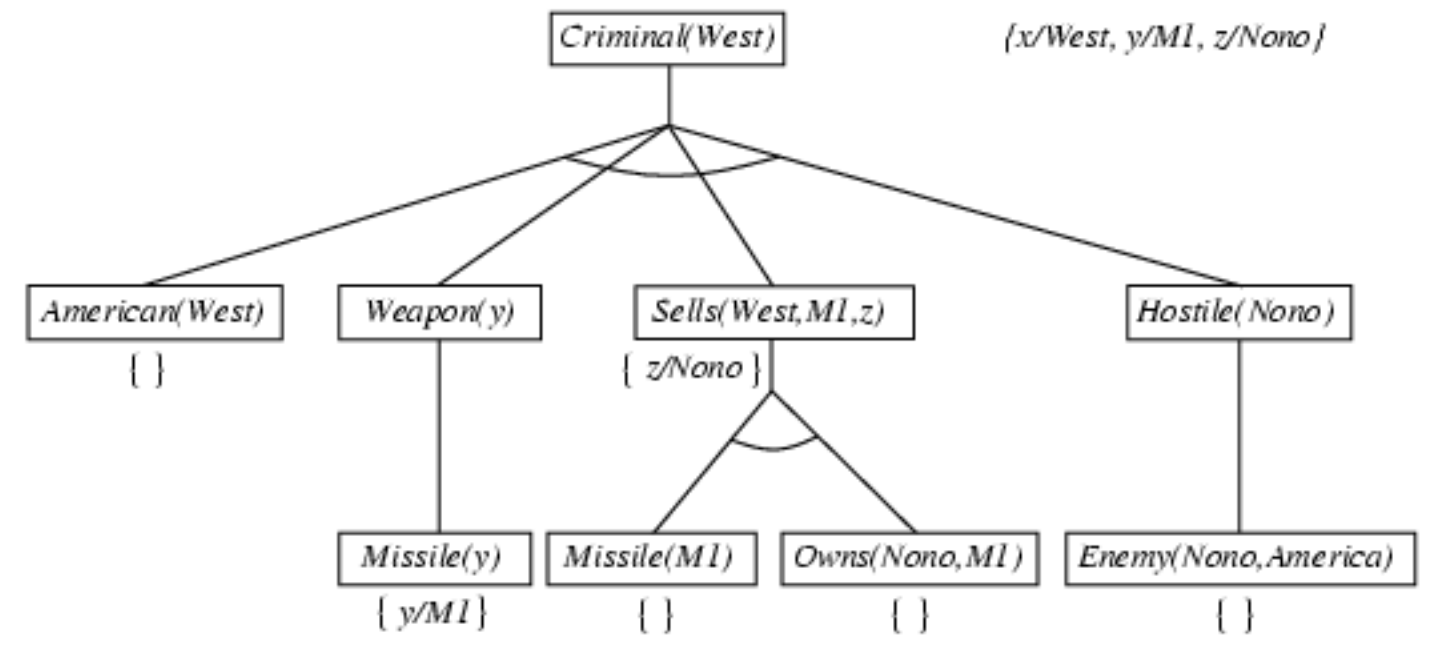
Backward chaining example



Backward chaining example



Backward chaining example



Backward-Chaining

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

Logic Programming

- **Declarative semantics**:
  - There is a simple way to determine the meaning of each statement.
  - The meaning can be concisely determined from the statement itself.
  - The meaning does not depend on how the statement might be used to solve a problem:
    - Sentences in FOL vs. statements in imperative languages.

  ⇒ Simpler than the semantics of imperative languages.
- Programming is **nonprocedural**:
  - Programs do not state how a result is to be computed, but rather the form of the result.

Example: Sorting a List

- Describe the characteristics of a sorted list, not the process of rearranging a list.

  sort(old_list, new_list) $\Leftarrow$ permute (old_list, new_list) $\wedge$ sorted (new_list)

  sorted (list) $\Leftarrow \forall_j$ such that $1 \leq j < n$, list(j) $\leq$ list (j+1)

Logic Programming: Prolog

- University of Aix-Marseille (Colmerauer & Roussel)
  - Natural language processing.
- University of Edinburgh (Kowalski)
  - Automated theorem proving.

- **Algorithm = Logic + Control**
- **Backward-Chaining** with **Horn clauses**
  + bells & whistles.

- Widely used in Europe, Japan (basis of 5th Generation project).

Logic Programming: Prolog

- 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:

  X is Y*Z+3
- Built-in predicates that have side effects:
  - 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

Prolog: Syntax (Edinburgh)

- *Term*: a constant, variable, or structure.
- *Constant*: an atom or an integer.
- *Atom*: symbolic value of Prolog.
  - string of letters, digits, underscores beginning with a lowercase letter.
  - a string of printable ASCII characters delimited by apostrophes.
- *Variable*: any string of letters, digits, and underscores beginning with an uppercase letter.
- *Instantiation*: binding of a variable to a value.
  - Lasts only as long as it takes to satisfy one complete goal
- *Structure*: represents atomic proposition:

  `functor` (*parameter list*)

Prolog: Fact Statements

- Used for the unconditional assertions.

- Headless Horn clauses:

```
female(shelley).
male(bill).
father(bill, jake).
```

Prolog: Rule Statements

- Used for the conditional assertions.

- Headed Horn clause
  - Right side: *antecedent* (***if*** part)
    - May be single term or *conjunction*.
  - Left side: *consequent* (***then*** part)
    - Must be single term.
- *Conjunction*: multiple terms separated by commas.
  - logical AND operator is implied.

Prolog: Example Rules

```
ancestor(mary,shelley):- mother(mary,shelley).
```

- Can use variables (*universal objects*) to generalize meaning:

```
parent(X,Y):- mother(X,Y).
parent(X,Y):- father(X,Y).
grandparent(X,Z):- parent(X,Y), parent(Y,Z).
sibling(X,Y):- mother(M,X), mother(M,Y),
                    father(F,X), father(F,Y).
```

Prolog: Goal Statements

- For theorem proving, theorem is in form of a sentence that we want system to prove or disprove :
  - *goal statement,* or *query.*
- Same format as headless Horn:
  - `man(fred)`
  - Conjunctive sentences and sentences with variables are also legal goals:
    - `father(X,mike)`

Prolog: Simple Arithmetic

- Prolog supports integer variables and integer arithmetic
- `is` operator: takes an arithmetic expression as right operand and variable as left operand

  ```
  A is B / 17 + C
  ```

- Not the same as an assignment statement!

Prolog: Example

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :-  speed(X,Speed),
                  time(X,Time),
                  Y is Speed * Time.
```

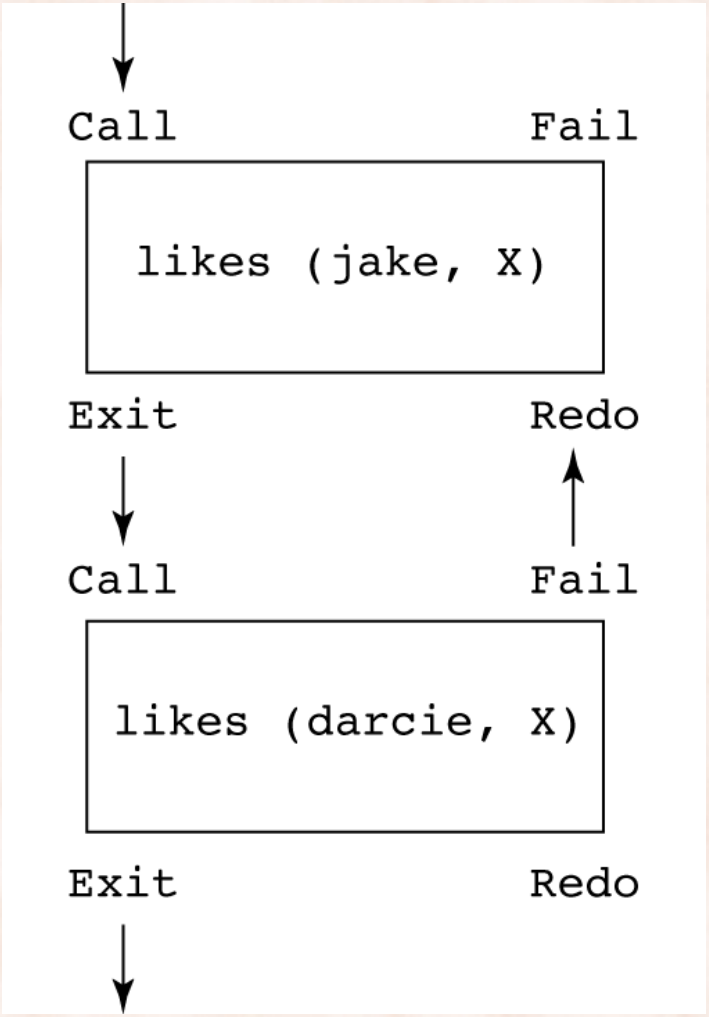
Prolog: Trace

- Built-in structure that displays instantiations at each step.
- *Tracing model* of execution - four events:
  - *Call* (beginning of attempt to satisfy goal)
  - *Exit* (when a goal has been satisfied)
  - *Redo* (when backtrack occurs)
  - *Fail* (when goal fails)

Prolog: Trace Example

```
likes(jake,chocolate).
likes(jake,apricots).
likes(darcie,licorice).
likes(darcie,apricots).

trace.
likes(jake,X),
likes(darcie,X).
```



Prolog: Lists

- Other basic data structure besides atomic sentences.
- *List* is a sequence of any number of elements.
- Elements can be atoms, atomic sentences, or other terms (including other lists)

```
[apple, prune, grape, kumquat]
[]          (empty list)
[X | Y]     (head X and tail Y)
```

Examples: Append and Reverse

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3]) :-
   append (List_1, List_2, List_3).

reverse([], []).
reverse([Head | Tail], List) :-
   reverse (Tail, Result),
   append (Result, [Head], List).
```

Examples: Append and Reverse

- Query: `append(A,B,[1,2]).`

- Answers:

```
A=[]      B=[1,2]
A=[1]     B=[2]
A=[1,2]   B=[]
```

Deficiencies of Prolog

- Resolution order control.
- The closed-world assumption.
- The negation problem.
  - the use of Horn clauses prevents negative conclusions.
- Intrinsic limitations.
  - cannot transform declarative specification of the sorting problem into an efficient sorting algorithm.

Applications of Logic Programming

- Relational database management systems.
- Expert systems.
- Natural language processing.