Logic Programming and Prolog

Read: Scott, Chapter 12

Lecture Outline

- Quiz 2
- Logic programming
- Prolog
    - Language constructs: facts, rules, queries
    - Search tree, unification, backtracking, backward chaining

Prolog

- Download and install SWI Prolog on laptop
  - Write your Prolog program and save in **`.pl`** file, e.g., **`snowy.pl`**
  - Run **`swipl`** (Prolog interpreter) on command line
  - Load your file: **`?- [snowy].`**
  - Issue query at prompt: **`?- snowy(C).`**

- J.R.Fisher's Prolog Tutorial:

http://www.cpp.edu/~jrfisher/www/prolog_tutorial/contents.html

Why Study Prolog?

- Declarative programming and logic programming
- Prolog is useful in a variety of applications
  - Rule-based reasoning
  - Natural-language processing
  - Database systems
    - Prolog and SQL have a lot in common
- Practice of important concepts such as first-order logic

Logic Programming

- Logic programming is declarative programming
- Logic program states what (logic), not how (control)

G [nodes
edges

- Programmer declares axioms
  - In Prolog, facts and rules

path(x,y) if path(x,z), path(z,y)

path(a,b)?

- Programmer states a theorem, or a goal (the what)
  - In Prolog, a query
- Language implementation determines how to use the axioms to prove the goal

Logic Programming

- Logic programming style is characterized by
  - Database of facts and rules that represent logical relations. Computation is modeled as search (queries) over this database
  - Use of lists and use of recursion, which turns out very similar to the functional programming style

Logic Programming Concepts

- A Horn Clause is: $H \leftarrow B_1, B_2, \ldots, B_n$
  - Antecedents (B's): conjunction of zero or more terms in predicate calculus; this is the body of the horn clause
  - Consequent (H): a term in predicate calculus
- Resolution principle: if two Horn clauses

$$A \leftarrow B_1, B_2, B_3, \ldots, B_m$$

$$C \leftarrow D_1, D_2, D_3, \ldots, D_n$$

$$\frac{\begin{array}{c} A \\ C \leftarrow A, B \end{array}}{C \leftarrow B} \quad \Big\| \quad \frac{\begin{array}{c} A \leftarrow B \\ B \end{array}}{A}$$

are such that A matches D_1,

then we can replace D_1 with $B_1, B_2, B_3, \ldots, B_m$

$$C \leftarrow \underline{B_1, B_2, B_3, \ldots, B_m}, D_2, D_3, \ldots, D_n$$

Lecture Outline

- Logic programming
- Prolog
  - Language constructs: facts, rules, queries
  - Search tree, unification, backtracking, backward chaining

Horn Clauses in Prolog

- In Prolog, a Horn clause is written
  `h :- b1,...,bn.` (h :- b_1,...,b_n.)
- Horn Clause is called clause
- Consequent is called goal or head
- Antecedents are called subgoals or tail

- Horn Clause with no tail is a fact
  - E.g., `rainy(seattle).` Depends on no other conditions
- Horn Clause with a tail is a rule

```
snowy(X) :- rainy(X),cold(X).
```

Horn Clauses in Prolog

- Clause is composed of terms
  - Constants
    - Number, e.g., 123, etc.
    - Atoms e.g., `seattle`, `rochester`, `rainy`, `foo`

    In Prolog, atoms begin with a lower-case letter!
  - Variables
    - `X`, `Foo`, `My_var`, `etc.`

    In Prolog, variables begin with upper-case letter!
  - Structures
    - E.g., `rainy(seattle)`, `snowy(X)`
    - Consists of an atom, called a functor and a list of arguments

Horn Clauses in Prolog

- Variables may appear in the tail and head of a rule:

  c(x,y) :- h(x,y)

  c(x1,y1).

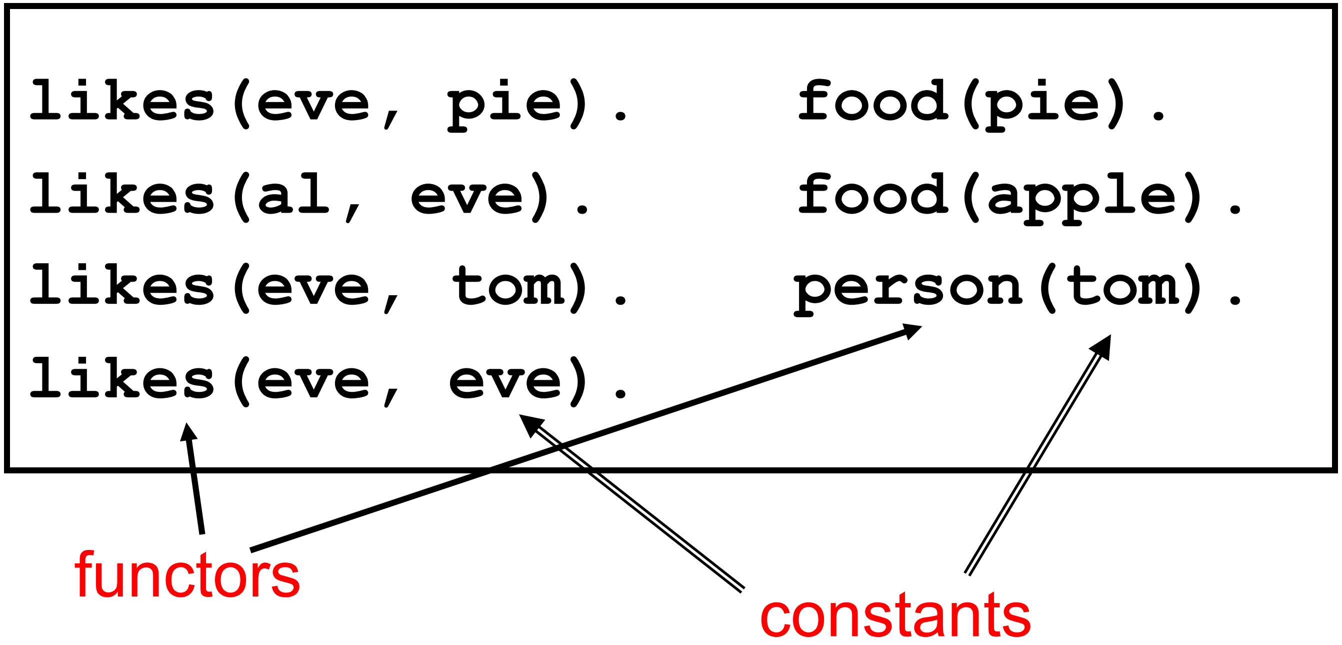
  - `c(X) :- h(X,Y).`
    For all values of `X`, `c(X)` is true if there exist a value of `Y` such that `h(X,Y)` is true
  - Call `Y` an auxiliary variable. Its value will be bound to make consequent true, but not reported by Prolog, because it does not appear in the head

Prolog

- Program has a database of clauses i.e., facts and rules; the rules help derive more facts
- We add simple queries with constants, variables, conjunctions or disjunctions

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X),cold(X).
? – [snowy].
? - rainy(C).
? – snowy(C).
```

Facts



```
likes(eve, pie).       food(pie).
likes(al, eve).        food(apple).
likes(eve, tom).       person(tom).
likes(eve, eve).
```

functors

constants

The combination of the functor and its arity (i.e., its number of arguments) is called a predicate.

Queries

```
likes(eve, pie).        food(pie).
likes(al, eve).         food(apple).
likes(eve, tom).        person(tom).
likes(eve, eve).
```

query

```
?-likes(al,eve).
true.
```

answer

```
?-likes(al,pie).
false.
?-likes(eve,al).
false.
```

variable

```
?-likes(al,Who).
Who=eve.
```

answer with variable binding

```
?-likes(eve,W).
W=pie ;
W=tom ;
W=eve .
```

force search for more answers

Question

```
likes(eve, pie).      food(pie).
likes(al, eve).       food(apple).
likes(eve, tom).      person(tom).
likes(eve, eve).
```

```
?-likes(eve,W).
W = pie ;
W = tom ;
W = eve .
```

Prolog gives us the answer precisely in this order: first **`W=pie`** then **`W=tom`** and finally **`W=eve`**.
Can you guess why?

Harder Queries

```
likes(eve, pie).     food(pie).
likes(al, eve).      food(apple).
likes(eve, tom).     person(tom).
likes(eve, eve).
```

and

```
?-likes(al,V) , likes(eve,V).
V=eve.
?-likes(eve,W) , person(W).
W=tom
?-likes(A,B).
A=eve,B=pie ; A=al,B=eve ; A=eve,B=tom ;
A=eve,B=eve.
?-likes(D,D).
D=eve.
```

Harder Queries

```
likes(eve, pie).      food(pie).
likes(al, eve).       food(apple).
likes(eve, tom).      person(tom).
likes(eve, eve).
```

same binding

```
?-likes(eve,W),likes(W,V).
W=eve,V=pie ; W=eve,V=tom ; W=eve,V=eve.

?-likes(eve,W),person(W),food(V).
W=tom,V=pie ; W=tom,V=apple
```

or

```
?-likes(eve,V),(person(V);food(V)).
V=pie  ; V=tom
```

Rules

```
likes(eve, pie).      food(pie).
likes(al, eve).       food(apple).
likes(eve, tom).      person(tom).
likes(eve, eve).
```

Add a rule to the database:

```
rule1:-likes(eve,V),person(V).
```

```
?-rule1.
true
```

Rules

```
likes(eve, pie).      food(pie).
likes(al, eve).       food(apple).
likes(eve, tom).      person(tom).
likes(eve, eve).
rule1 :- likes(eve,V),person(V).
 rule2(V) :- likes(eve,V),person(V).
```

```
?-rule2(H).
H=tom
?-rule2(pie).
false.
```

rule1 and rule2 are just like any other predicate!

Queen Victoria Example

cf Clocksin and Mellish

```
male(albert).
male(edward).
female(alice).
female(victoria).
parents(edward,victoria,albert).
parents(alice,victoria,albert).
```

Put all clauses in file family.pl

```
?- [family].
```

Loads file family.pl

```
  true.
  ?- male(albert).
```

A query

```
  true.
  ?- male(alice).
  false.
  ?- parents(edward,victoria,albert).
  true.
  ?- parents(bullwinkle,victoria,albert).
  false.
```

Queen Victoria Example

```
?-female(X).    a query
X = alice ;     ; asks for more answers
X = victoria.
```

- Variable **X** has been unified to all possible values that make **female(X)** true.
- Variables are upper-case, constants are lower-case!

Queen Victoria Example

- Facts alone do not make interesting programs. We need variables and deductive rules.

```
sister_of(X,Y) :- female(X),parents(X,M,F),
                  parents(Y,M,F).
```

```
?- sister_of(alice, Y).
Y = edward ;
Y = alice.
```

```
?- sister_of(alice, Y).
Y = edward ;
```

<enter>: not asking for more answers

```
?- sister_of(alice, victoria).
false.
```

Another Prolog Program

snowy.pl

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X),cold(X).

?- [snowy].   ✓
?- rainy(C).
?- snowy(C).
```

Lecture Outline

- Logic programming
- Prolog
  - Language constructs: facts, rules, queries
  - Search tree, unification, rule ordering, backtracking, backward chaining

Logical Semantics

- Prolog program consists of facts and rules

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X):-rainy(X),cold(X).
```

Rules like `snowy(X):- rainy(X),cold(X).`

correspond to logical formulas:

∀X[snowy(X) ← rainly(X) ^ cold(X)]

/* For every X, X is snowy, if X is rainy and X is cold */

Logical Semantics

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X):-rainy(X),cold(X).
```

- A query such as `?- rainy(C).` triggers resolution. Logical semantics does not impose restriction in the order of application of resolution rules

`C = seattle`	`C = rochester`
`C = rochester`	`C = seattle`

Procedural Semantics

```
?- snowy(C).
```

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X):-rainy(X),cold(X).
```

cold(seattle)

Find the first clause in the database whose head matches the query. In our case this is clause snowy(X):-

```
snowy(X) :- rainy(X),cold(X)
```

Then, find a binding for **X** that makes **rainy(X)** true; then, check if **cold(X)** is true with that binding

- If yes, report binding as successful
- Otherwise, backtrack to the binding of **X**, unbind and consider the next binding

- Prolog's computation is well-defined procedurally by search tree, rule ordering, unification, backtracking, and backward chaining

Question

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X):-rainy(X),cold(X).
snowy(troy).
```

What does this query yield?

```
?- snowy(C).
```

Answer:

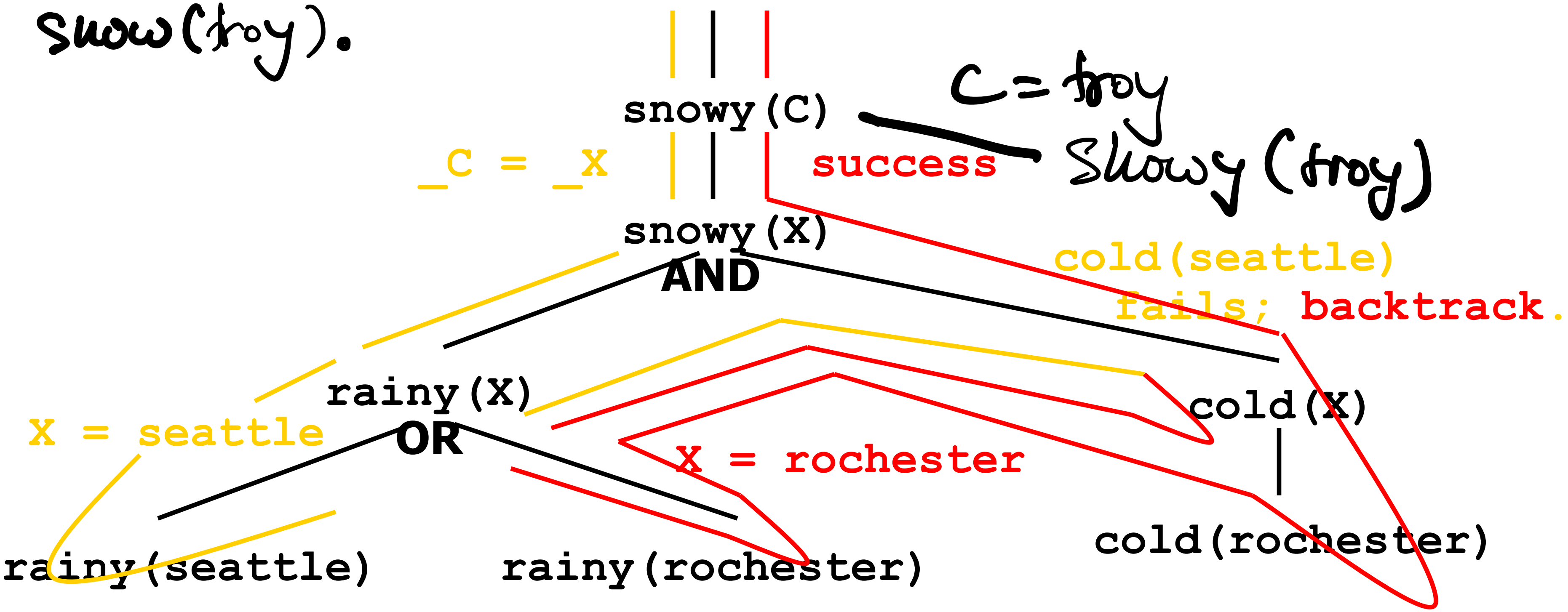
```
C = rochester ;
C = troy.
```

Procedural Semantics

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X),cold(X).
```

snow(troy).

C = rochester ;
C = troy.



snowy(C)

_C = _X

success

C = troy

snowy(troy)

snowy(X)

AND

cold(seattle) fails; backtrack.

rainy(X)

OR

X = seattle

X = rochester

cold(X)

rainy(seattle)

rainy(rochester)

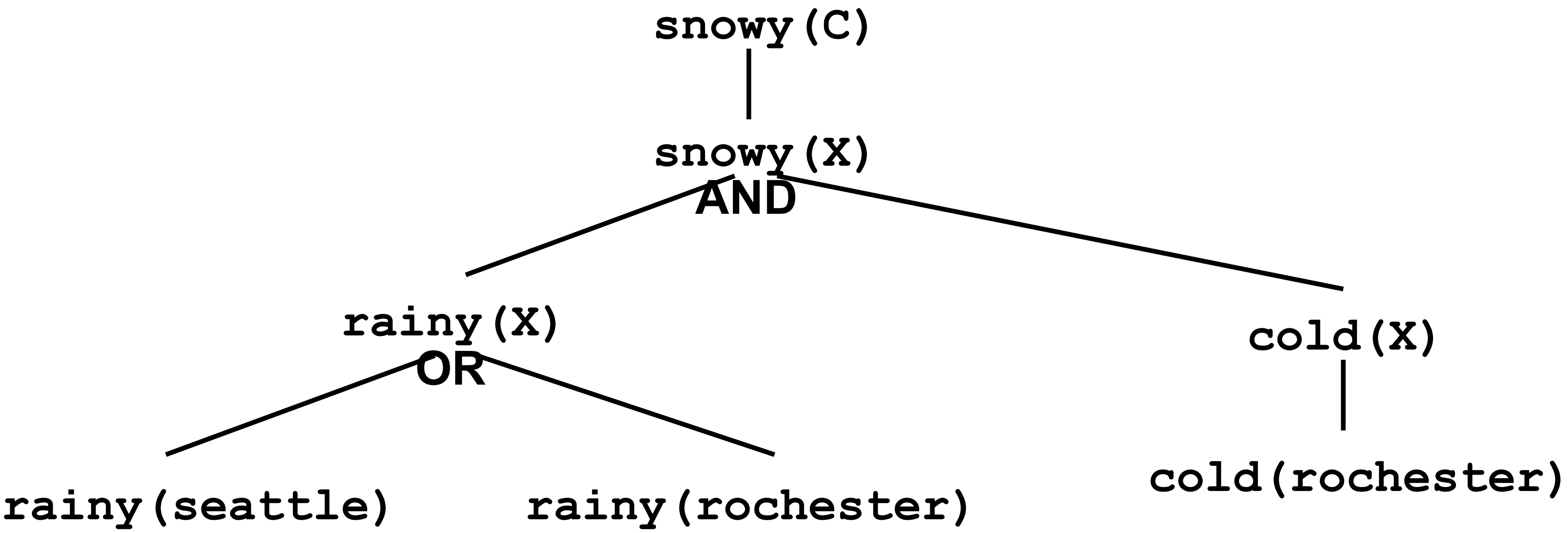
cold(rochester)

Prolog Concepts: Search Tree

OR levels:
- parent: goal (e.g., **rainy(X)**)
- children: heads-of-clauses (**rainy(…)**)
  - **ORDER:** from left to right

AND levels:
- parent: goal (e.g., **snowy(X)**)
- children: subgoals (**rainy(X), cold(X)**)
  - **ORDER:** from left to right

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X):-rainy(X),cold(X).
Snowy(troy).
?- snowy(C).
```



snowy(C)

snowy(X)

AND

rainy(X)

OR

cold(X)

rainy(seattle)

rainy(rochester)

cold(rochester)

Prolog Concepts: Unification

- At **OR** levels Prolog performs unification
  - Unifies parent (goal), with child (head-of-clause)
- E.g.,
  - `snowy(C) = snowy(X)`
    - success, `_C = _X` ✓
  - `rainy(X) = rainy(seattle)`
    - success, `X = seattle`
  - `parents(alice,M,F) = parents(edward,victoria,albert)`
    - fail
  - `parents(alice,M,F) = parents(alice,victoria,albert)`
    - success, `M = victoria, F = albert`

In Prolog, = denotes unification, not assignment!

Prolog Concepts: Unification

- A constant unifies only with itself
  - E.g., **`alice=alice`**, but **`alice=edward`** fails
- Two structures unify if and only if (i) they have the same functor, (ii) they have the same number of arguments, and (iii) their arguments unify recursively
  - E.g., **`rainy(X) = rainy(seattle)`**

unbound

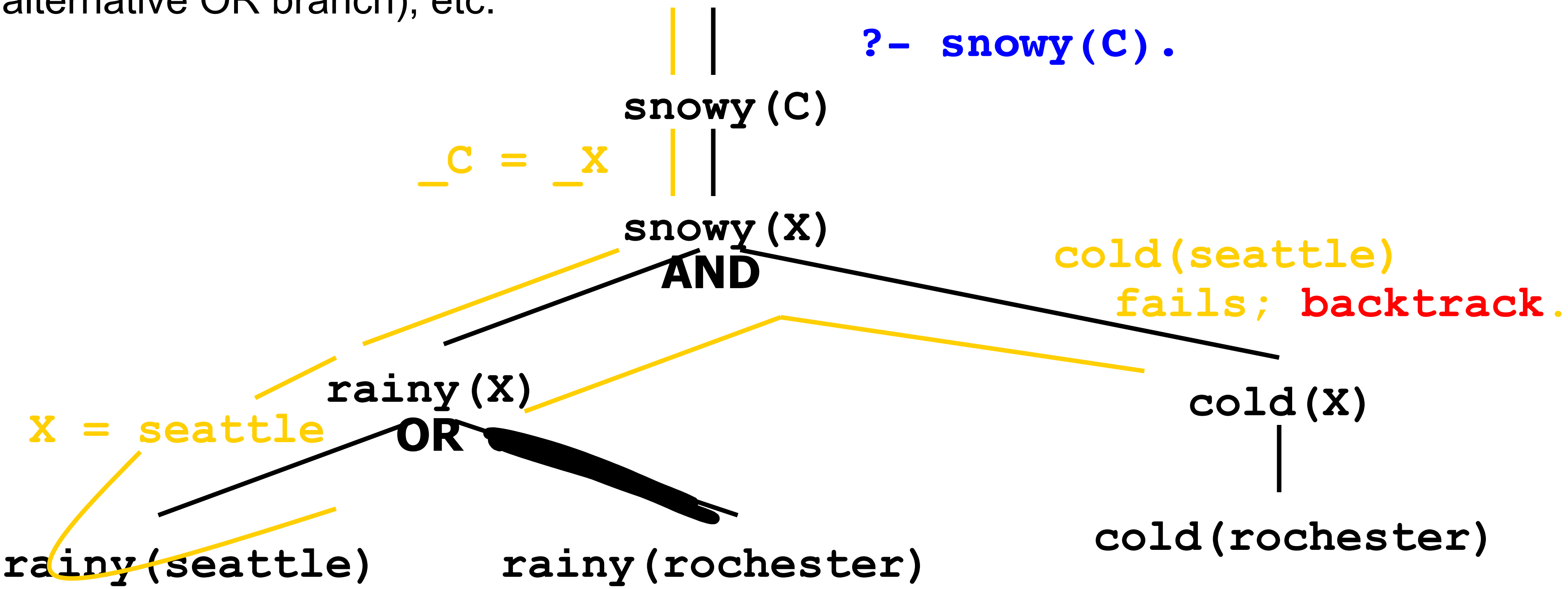
- A variable unifies with anything. If the other thing has a value, then variable is bound to that value. If the other thing is an unbound variable, then the two variables are associated and if either one gets a value, both do

Prolog Concepts: Backtracking

If at some point, a goal fails, Prolog backtracks to the last goal (i.e., last unification point) where there is an untried binding, undoes current binding and tries new binding (an alternative OR branch), etc.

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X):-rainy(X),cold(X).

?- snowy(C).
```



Prolog Concepts: Backward Chaining

- Backward chaining: starts from goal, towards facts

```
? – snowy(rochester).

snowy(rochester):-
    rainy(rochester),
    cold(rochester)
rainy(rochester)
----------------------
snowy(rochester):-
    cold(rochester)
cold(rochester)
----------------------
snowy(rochester).
```

- Forward chaining: starts from facts towards goal

```
? – snowy(rochester).

rainy(rochester)
snowy(rochester):-
    rainy(rochester),
    cold(rochester)
-----------------------------
cold(rochester)
snowy(rochester):-
    cold(rochester)
-----------------------------
snowy(rochester).
```

Exercise

```
takes(jane, his).
takes(jane, cs).
takes(ajit, art).
takes(ajit, cs).
classmates(X,Y):-takes(X,Z),takes(Y,Z).

?- classmates(jane,C).
```

Draw search tree for query.

What are the bindings for **C**?

The End