



Logic Programming

Chapter 12

Logic Programming

Prolog says:

`?- 1+1 = 2.`

`false.`

so ... keep reading!

Logic Programming

- Algorithm = axioms + control
- Axioms
 - facts and rules
 - supplied by the programmer
- Control
 - computation is deduction
 - supplied by the language
- Given a set of axioms, the user states a theorem, or *goal*, and the language attempts to show that the axioms imply the goal

Logic Programming

- Axioms = Horn clauses

$$Q_1 \wedge Q_2 \wedge \dots \wedge Q_k \rightarrow P$$

or

$$P \leftarrow Q_1 \wedge Q_2 \wedge \dots \wedge Q_k$$

- P is the *head*
- $Q_1 \wedge Q_2 \wedge \dots \wedge Q_k$ is the *body*
- $k \geq 1$: *rule*: if Q_1 and Q_2 and ... and Q_k , then P
- $k = 0$: *fact*: P (also: if `true`, then P)
- The meaning is that if all Q_i 's are true, then we can deduce P

Prolog

- Imperative language:
 - runs in the context of a referencing environment, where various constants and functions have been defined
- Prolog
 - runs in the context of a database where various clauses have been defined
- Clause composed of *terms*:
 - *constants*:
 - *atoms*: id that starts with **lower** case: foo, a , john
 - *numbers*: 0, 2022
 - *variables*: id that starts with **upper** case: Foo, X
 - *structures*: *functor* (atom) and *argument list* (terms)
 - student (john), takes (X, cs3342)
 - arguments can be constants, variables, (nested) structures

Prolog

- structures are interpreted as logical predicates
- predicate: functor + list of arguments
- Syntax:

term \rightarrow *atom* | *number* | *variable* | *struct*

terms \rightarrow *term* | *term* , *terms*

struct \rightarrow *atom* (*terms*)

fact \rightarrow *term* .

rule \rightarrow *term* :- *terms* .

query \rightarrow ?- *terms* .

Prolog

- Rule:

$$P \leftarrow Q_1 \wedge Q_2 \wedge \dots \wedge Q_k$$

- in Prolog:

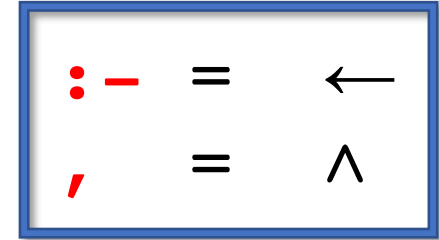
$$P \text{ :- } Q_1, Q_2, \dots, Q_k.$$

- Fact (rule without right-hand side):

$$P \quad (P \leftarrow \text{true})$$

- in Prolog:

$$P.$$



Prolog

- Query (rule without left-hand side)

$$Q_1 \wedge Q_2 \wedge \dots \wedge Q_k$$

- in Prolog:

$$?- Q_1, Q_2, \dots, Q_k.$$

- the negated query is also:

$$\text{false} \leftarrow Q_1 \wedge Q_2 \wedge \dots \wedge Q_k$$

Prolog

- Rules are implicitly universally quantified (\forall)

- Example:

`path(L, M) :- link(L, X), path(X, M).`

- means:

$\forall L, \forall M, \forall X (\text{path}(L, M) \text{ if } (\text{link}(L, X) \text{ and } \text{path}(X, M)))$

or

$\forall L, \forall M (\text{path}(L, M) \text{ if } (\exists X (\text{link}(L, X) \text{ and } \text{path}(X, M))))$

Prolog

- Queries are implicitly existentially quantified (\exists)
- Example:

`?- path(algol60, X), path(X, c).`

- means

$\exists X (\text{path}(\text{algol60}, X) \textbf{ and } \text{path}(X, c))$

Prolog

- Setting up working directory
- Checking working directory:

```
?- working_directory(X, X).  
X = (//).
```

- Changing working directory:

```
?- working_directory(_, '/Users/Lucian/Documents/  
4_myCourses/2021-2022/CS3342b_win2022/my_programs/Prolog').  
true.
```

```
?- working_directory(X, X).
```

```
X = (_, '/Users/Lucian/Documents/4_myCourses/2021-2022/  
CS3342b_win2022/my_programs/Prolog').
```

Prolog

- Facts and rules from a file:
 - reading the file “my_file.pl”
 - must be in the working directory

```
?- consult(my_file).  
true.
```

Prolog

- Example:

```
rainy(seattle).  
rainy(rochester).
```

```
?- rainy(C).  
C = seattle
```

- Type ENTER if done
- Type ‘;’ if you want more solutions

```
C = seattle ;  
C = rochester.
```


Prolog

- Example:

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

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

- only one solution

Prolog

- Example:

```
link(fortran,  algol60).  
link(algol60,  cpl).  
link(cpl,  bcpl).  
link(bcpl,  c).  
link(c,  cplusplus).  
link(algol60,  simula67).  
link(simula67,  cplusplus).  
link(simula67,  smalltalk80).
```

```
path(L, L).
```

```
path(L, M) :- link(L, X), path(X, M).
```

Prolog

- Example:

```
?- link(simula67, X).
```

```
X = cplusplus ;
```

```
X = smalltalk80.
```

```
?- link(algol60, X), link(X, Y).
```

```
X = cpl,
```

```
Y = bcpl ;
```

```
X = simula67,
```

```
Y = cplusplus ;
```

```
X = simula67,
```

```
Y = smalltalk80.
```

Prolog

- Example:

```
?- path(fortran, cplusplus).  
true ;  
true ;  
false.
```

```
?- path(X, cpl).  
X = cpl ;  
X = fortran ;  
X = algol60 ;  
false.
```

Prolog

- Example:

```
?- path(X,Y).  
X = Y ;  
X = fortran,  
Y = algol60 ;  
X = fortran,  
Y = cpl ;  
X = fortran,  
Y = bcpl ;  
X = fortran,  
Y = c ;  
X = fortran,  
Y = cplusplus ; % ... it finds all paths
```


Lists

- $[a, b, c]$ – list
- $[]$ – empty list
- can use a **cons**-like predicate:

$'[|]'(a, '[|]'(b, '[|]'(c, [])))$
means $[a, b, c]$

- *Head | Tail* notation: $[H | T]$
- $[a, b, c]$ can be written as:
 $[a | [b, c]]$
 $[a, b | [c]]$
 $[a, b, c | []]$

Lists

?- [H|T] = [a, b, c].

H = a,

T = [b, c].

?- [H|T] = [[], c | [[a], b, [] | [b]]].

H = [],

T = [c, [a], b, [], b].

?- [H|[X|T]] = [[], c | [[a], b, [] | [b]]].

H = [],

X = c,

T = [[a], b, [], b].

?- [H1,H2|[X|T]] = [[],c | [[a], b, [] | [b]]].

H1 = [],

H2 = c,

X = [a],

T = [b, [], b].

List operations

- Searching an element in a list:

`member(X, [X|_]) .`

`member(X, [_|T]) :- member(X, T) .`

- is a placeholder for a variable not needed anywhere else

List operations

- Searching an element in a list:

```
?- member(a, [b, a, c]).  
true
```

```
?- member(a, [b, d, c]).  
false.
```

```
?- member(a, X).  
X = [a|_14708] ;  
X = [_14706, a|_14714] ;  
X = [_14706, _14712, a|_14720] ;  
X = [_14706, _14712, _14718, a|_14726] ;  
X = [_14706, _14712, _14718, _14724, a|_14732]  
...
```

List operations

- Adding an element to a list:

```
add(X, L, [X|L]).  
?- add(a, [b,c], L).  
L = [a, b, c].
```

- Deleting an element from a list:

```
del(X, [X|T], T).  
del(X, [Y|T], [Y|T1]) :- del(X, T, T1).  
  
?- del(a, [a, b, c, a, b, a, d, a], X).  
X = [b, c, a, b, a, d, a] ;  
X = [a, b, c, b, a, d, a] ;  
X = [a, b, c, a, b, d, a] ;  
X = [a, b, c, a, b, a, d] ;  
false.
```


List operations

- Appending two lists:

```
append([ ], Y, Y).
```

```
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

- Sublists:

```
sublist(S,L) :- append(_,L1,L), append(S,_,L1).
```

List operations

- Example:

```
?- append([a, b, c], [d, e], L).  
L = [a, b, c, d, e].
```

```
?- append(X, [d, e], [a, b, c, d, e]).  
X = [a, b, c]
```

```
?- append([a, b, c], Y, [a, b, c, d, e]).  
Y = [d, e].
```

- Very different from imperative programming: input/output
- In Prolog: no clear notion of input and output
 - Just search for values that make the goal true

List operations

- Subset

`subset([], S).`

`subset([H|T], S) :- member(H, S), subset(T, S).`

- Reversing a list

`reverse([], []).`

`reverse([H|T], R) :- reverse(T, R1), append(R1, [H], R).`

- Permutations

`permute([], []).`

`permute([H|T], P) :- permute(T, P1), insert(H, P1, P).`

Unification

```
path(L, L).
```

```
path(L, M) :- link(L, X), path(X, M).
```

```
?- path(fortran, cplusplus).
```

- *Unification* is a type of pattern matching:

L unifies with `fortran`

M unifies with `cplusplus`

Unification

- Unification *rules*:
- a constant unifies with itself
- two structures unify if and only if:
 - have the same functor
 - have the same arity
 - corresponding arguments unify recursively
- a variable unifies with anything
 - if the other thing has a value, then the variable is instantiated
 - if the other thing is an uninstantiated variable, then the two variables are associated so that if either is given a value later, that value will be shared by both

Unification

- Equality ($=$) is *unifiability*:
 - The goal $=(A, B)$ succeeds iff A and B can be unified
 - $A = B$ – syntactic sugar

- Example:

?- $a = a$.

true.

?- $a = b$.

false.

?- $\text{foo}(a, b) = \text{foo}(a, b)$.

true.

Unification

- Example:

?- $X = a.$

$X = a.$

?- $\text{foo}(a,b) = \text{foo}(X,b).$

$X = a.$

Arithmetic

- arithmetic operators – predicates
- `+(2, 3)` - syntactic sugar `2+3`
- `+(2, 3)` is a two-argument structure; does not unify with 5
 - `?- 1+1 = 2.`
`false.`
- **is**: predicate that unifies first arg. with value of second arg.
 - `?- is(X, 1+1).`
`X = 2.`
 - `?- X is 1+1.`
`X = 2.`

More unification

- Substitution:
 - a function from variables to terms
 - Example: $\sigma = \{X \rightarrow [a,b], Y \rightarrow [a,b,c]\}$
- $T\sigma$ – the result of applying the substitution σ to the term T
 - $X\sigma = U$ if $X \rightarrow U$ is in σ , X otherwise
 - $(f(T_1, T_2, \dots, T_n))\sigma = f(T_1\sigma, T_2\sigma, \dots, T_n\sigma)$

- Example:

$$\sigma = \{X \rightarrow [a,b], Y \rightarrow [a,b,c]\}$$

$$Y\sigma = [a,b,c]$$

$$Z\sigma = Z$$

$$\text{append}([], Y, Y)\sigma = \text{append}([], [a,b,c], [a,b,c])$$

More unification

- A term U is an *instance* of T if $U = T\sigma$, for some substit. σ
- Two terms T_1 and T_2 *unify* if $T_1\sigma$ and $T_2\sigma$ are identical, for some σ ; σ is called a *unifier* of T_1 and T_2
- σ is *the most general unifier* of T_1 and T_2 if, for any other unifier δ , $T_i\delta$ is an instance of $T_i\sigma$
- Example: $L = [a, b \mid X]$
- Unifiers:
 - $\sigma_1 = \{L \rightarrow [a, b \mid X_1], X \rightarrow X_1\}$
 - $\sigma_2 = \{L \rightarrow [a, b, c \mid X_2], X \rightarrow [c \mid X_2]\}$
 - $\sigma_3 = \{L \rightarrow [a, b, c, d \mid X_3], X \rightarrow [c, d \mid X_3]\}$
- σ_1 is the most general unifier

Control Algorithm

- Control algorithm
 - the way Prolog tries to satisfy a query
- Two decisions:
 - *goal order*: choose the leftmost subgoal
 - *rule order*: use the first applicable rule

Control Algorithm

- Control algorithm

start with a query as the current goal

while (the current goal is nonempty) **do**

 choose the *leftmost subgoal*

if (a rule applies to this subgoal) **then**

 select the *first applicable rule* not already used

 form a new current goal

else

if (at the root) **then**

false

else

backtrack

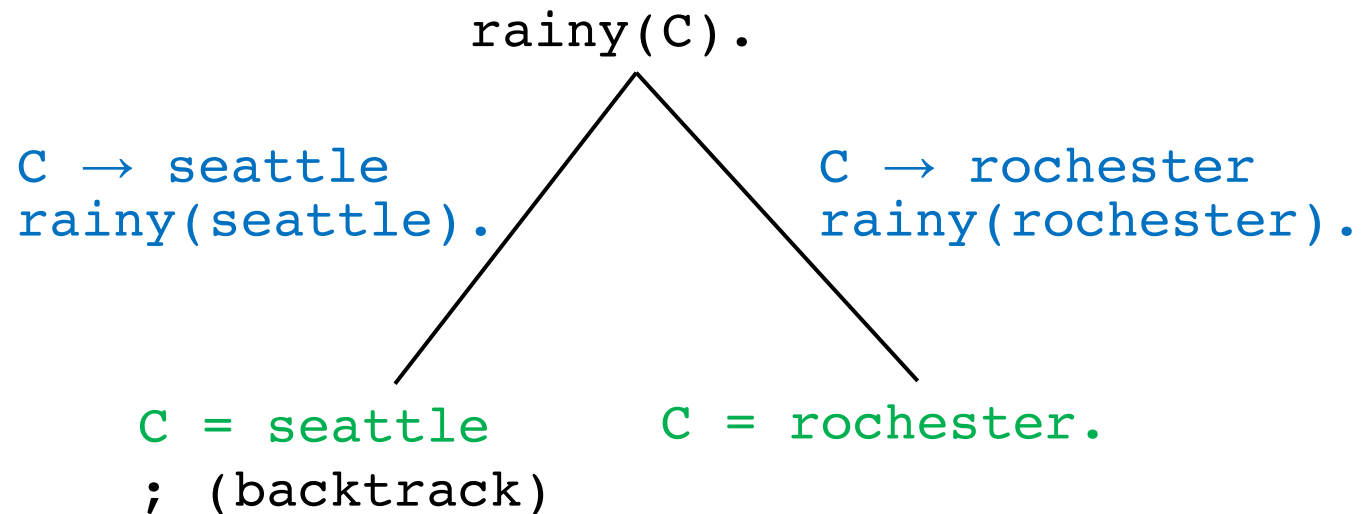
true

Control Algorithm - Example

```
rainy(seattle).  
rainy(rochester).
```

```
?- rainy(C).  
C = seattle ;  
C = rochester.
```

Prolog search tree:



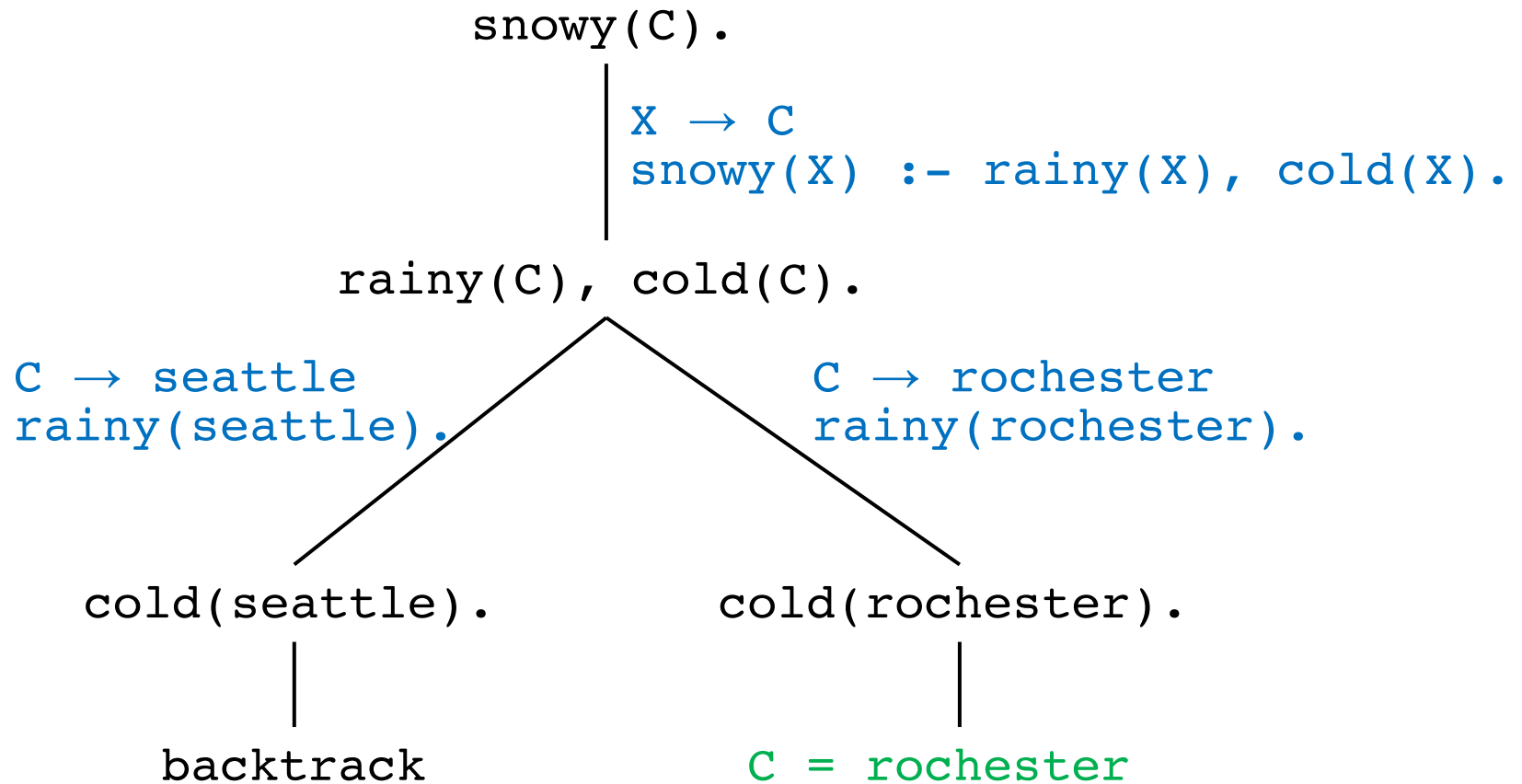
Control Algorithm - Example

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

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

Control Algorithm - Example

Prolog search tree:



Control Algorithm – details

start with a query as the current goal: G_1, G_2, \dots, G_k ($k \geq 0$)

while ($k > 0$) **do** // the current goal is nonempty

 choose the *leftmost subgoal* G_1

if (a rule applies to G_1) **then**

 select *first applicable rule* (not tried): $A :- B_1, \dots, B_j$ ($j \geq 0$)

 let σ be *the most general unifier* of G_1 and A

 the current goal becomes: $B_1\sigma, \dots, B_j\sigma, G_2\sigma, \dots, G_k\sigma$

else

if (at the root) **then**

false // tried all possibilities

else

backtrack // try something else

true // all goals have been satisfied

Control Algorithm - Example

```
append([ ], Y, Y).
```

```
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

```
prefix(P, L) :- append(P, _, L).
```

```
suffix(S, L) :- append(_, S, L).
```

```
?- suffix([a], L), prefix(L, [a, b, c]).
```

```
L = [a]      // that's the obvious solution
```

```
L = [a] ;    // if we ask for more solutions  
              // we get an infinite computation  
              // eventually aborting (out of stack)
```


Control Algorithm - Example

```
?- suffix([a], L), prefix(L, [a, b, c]).
```

```
L = [a] ; // infinite computation
```

- why the infinite computation?
- consider the first subgoal only:

```
?- suffix([a], L).
```

```
L = [a] ;
```

```
L = [_944, a] ;
```

```
L = [_944, _956, a] ;
```

```
L = [_944, _956, _968, a] ; ...
```

- infinitely many solutions, none (but the first) satisfying the second subgoal
- control checks an infinite subtree with no solutions

Control Algorithm - Example

```
append([ ], Y, Y).
```

```
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

```
prefix(P, L) :- append(P, _, L).
```

```
suffix(S, L) :- append(_, S, L).
```

```
?- suffix([b], L), prefix(L, [a, b, c]).
```

```
L = [a, b]    // that's the obvious solution
```

```
L = [a, b] ;// if we ask for more solutions  
              // again, infinite computation
```

Goal order

- Changing the order of subgoals can change solutions:

```
?- suffix([a], L), prefix(L, [a, b, c]).  
L = [a] ;  
// infinite computation
```

- if we change the goal order, then no infinite computation:

```
?- prefix(L, [a, b, c]), suffix([a], L).  
L = [a] ;  
false.
```

Goal order

- The explanation is that the first subgoal now has finitely many solutions:

```
?- prefix(L, [a, b, c]).  
L = [] ;  
L = [a] ;  
L = [a, b] ;  
L = [a, b, c] ;  
false.
```

Rule order

- Changing the order of rules can change solutions:

```
append([ ], Y, Y).
```

```
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

```
?- append(X, [c], Z).
```

```
X = [ ],
```

```
Z = [c] ;
```

```
X = [_576],
```

```
Z = [_576, c] ;
```

```
X = [_576, _588],
```

```
Z = [_576, _588, c] ;
```

```
X = [_576, _588, _600],
```

```
Z = [_576, _588, _600, c] ; ...
```

Rule order

- Changing the order of rules can change solutions:

```
append([H|X], Y, [H|Z]) :- append(X, Y, Z).  
append([], Y, Y).
```

```
?- append(X, [c], Z).  
// infinite computation
```


Cuts

- **!** – cut
- zero-argument predicate
- prevents backtracking, making computation more efficient
- can also implement a form of negation (we'll see later)
- General form of a cut:

$$P \text{ :- } Q_1, Q_2, \dots, Q_{j-1}, \text{ ! }, Q_{j+1}, \dots, Q_k.$$

- Meaning: the control backtracks past

$$Q_{j-1}, Q_{j-2}, \dots, Q_1, P$$

without considering any remaining rules for them

Cuts

- Example:

```
member(X, [X|_]) .  
member(X, [_|T]) :- member(X, T) .  
prime_candidate(X) :- member(X, Candidates), prime(X) .
```

- assume `prime(a)` is expensive to compute
- if `a` is a member of `Candidates` many times, this is slow
- solution:

```
member1(X, [X|_]) :- ! .  
member1(X, [_|T]) :- member1(X, T) .
```

Cuts

```
?- member(a, [a,b,c,a,d,a]).  
true ;  
true ;  
true ;  
false.
```

```
?- member1(a, [a,b,c,a,d,a]).  
true.
```

Negation as failure

- **not** – negation
- Definition:

```
not(X) :- X, !, fail.  
not(_).
```

- `fail` always fails
- the first rule attempts to satisfy `X`
- if `X` succeeds, then `!` succeeds as well, then `fail` fails and `!` will prevent backtracking
- if `X` fails, then `not(X)` fails and, because the cut has not been reached, `not(_)` is tried and immediately succeeds

Negation as failure

- Example:

`?- X=2, not(X=1).`

`X = 2.`

`?- not(X=1), X=2.`

`false.`