4CCS1ELA ELEMENTARY LOGIC WITH APPLICATIONS

LECTURE 10:
PREDICATE LOGIC PROGRAMMING

Objectives for the day

- Consolidate Predicate Definite Clause Programming

- Learn about and work on Backtracking
- Learn about and work on Negation as Failure
- Learn about and work on Recursion
- And all that is *Predicate Logic Programming*

Predicate Definite Clause Programming

Predicate definite clause programs

- A predicate definite clause program is a set of **first order definite clauses**
  - $\forall x_1, \ldots, \forall x_n \ \neg \alpha_1 \vee \ldots \vee \neg \alpha_m \vee \alpha$
- These can be represented as their **equivalent definite rules** and we can program with these rules
  - $\forall x_1, \ldots, \forall x_n \ \alpha_1 \wedge \ldots \wedge \alpha_m \rightarrow \alpha$
  - $\forall x_1, \ldots, \forall x_n \ \alpha_1 , \ldots , \alpha_m \rightarrow \alpha$

First order definite clause programs

- A first order (predicate) definite clause program is a set of **first order definite clauses**
- These can be represented as their **equivalent definite rules** and we can **program** with these rules

definite clauses

1. $loves(mary, john)$
2. $engaged(mary, john)$
3. $\forall x\ \forall y\ \neg loves(x, y) \vee \neg engaged(x, y) \vee marries(x, y)$

definite rules

1. $\rightarrow loves(mary, john)$
2. $\rightarrow engaged(mary, john)$
3. $\forall x\ \forall y\ loves(x, y), engaged(x, y) \rightarrow marries(x, y)$

Querying a predicate logic program

- Let P be a **program** of first order definite rules
- A **query** to P is a PNF formula

$$\exists x_1 \dots \exists x_n \, F$$

where F is a *conjunction* of positive *atoms* and $x_1 \dots x_n$ are the variables in F

$\mathcal{P}_1$

1. $\rightarrow loves(mary, john)$
2. $\rightarrow engaged(mary, john)$
3. $\forall x \; \forall y \; loves(x, y), engaged(x, y) \rightarrow marries(x, y)$

Query $? \; \exists z \exists w \; marries(z, w)$

Querying a predicate logic program

- Choose **rule** with **matching** **head**
- And then **replace** with **body** of rule query

1. $\rightarrow$ **loves(mary,john)**	$\mathcal{P}_1$
2. $\rightarrow$ **engaged(mary,john)**	
3. $\forall x\ \forall y\ loves(x, y), engaged(x, y) \rightarrow \mathbf{marries(x, y)}$	

Query $\quad ?\ \exists z \exists w\ marries(z, w)$

So let's try…

1. $\rightarrow$ ***loves(mary,john)***
2. $\rightarrow$ ***engaged(mary,john)***
3. $\forall x\ \forall y$ *loves(x, y), engaged(x, y)* $\rightarrow$ ***marries(x, y)***

$\mathcal{P}_1$

? $\exists z \exists w$ *marries(z, w)*

- I have a matching head, but it does not look right…
- The head is *marries* ***(x, y)***
- But my query is *marries* ***(z, w)***
- If only I could **substitute** that ***x*** with a ***z*** and that ***y*** with ***w*** …

Substitution

- A substitution S is a finite set $\{ (x_1/t_1), \ldots, (x_n/t_n) \}$ where:
  - $x_1, \ldots, x_n$ are **distinct** variables (**only variables can be substituted**)
  - $t_1, \ldots, t_n$ are terms (**variables, constants** or functions applied to terms)

- **every instance** of variable x_i is **simultaneously** replaced by t_i

Unification through substitution

- We use substitution to make a formula **match** another formula.
  - This is called **unification** of two formulas.
  - In our case here, we are trying to match a query with the head of the rule.

- Head of rule: $marries(\boldsymbol{x}, \boldsymbol{y})$
- Query: $marries(\boldsymbol{z}, \boldsymbol{w})$
- We apply substitution: $\{ (x/z), (y/w) \}$

Unification through substitution

- The goal of the substitution is to make a formula match another formula.
  - This is called **unification** of two formulas.
  - In our case here, we are trying to match a query with the head of the rule.

- $marries(z, w)$
- $marries(z, w)$
- Now the rule and query are unified!

So let's try again…

$\mathscr{P}_1$

1. $\rightarrow$ ***loves(mary,john)***
2. $\rightarrow$ ***engaged(mary,john)***
3. $\forall x\ \forall y\ loves(x, y), engaged(x, y) \rightarrow$ ***marries(x, y)***

I **choose** the left most atom

? ∃z∃w marries(z, w)

marries(x, y)
{ (x/z), (y/w) }

? loves(z, w), engaged(z, w)

loves(mary,john)
{ (z/mary), (w/john) }

? engaged(mary, john)

engaged(mary, john)

$\square$

Recap of predicate **definite clause** programming

Let us formally define the programming procedure:

Input:

- A program P of definite rules
- A query Q,

$Q = \exists x_1 \ldots \exists x_m \, \alpha_1 \wedge \ldots \wedge \alpha_n$

Note that we replace $\wedge$ with ,

Output:

If Q is logical consequence of P

then output ***Yes***, else output ***No***.

Notation conventions

1. $\rightarrow$ ***loves(mary,john)***
2. $\rightarrow$ ***engaged(mary,john)***
3. $\forall x\ \forall y$ *loves(x, y), engaged(x, y)* $\rightarrow$ ***marries(x, y)***

$\mathcal{P}_1$

? ∃z∃w marries(z, w)

- We can simply leave out :
  - the universal quantifiers before **3**.
  - the '→' before ***1.*** and **2**.
  - the existential quantifiers before query *marries*(z, w)

1. ***loves(mary,john)***
2. ***engaged(mary,john)***
3. *loves(x, y), engaged(x, y)* $\rightarrow$ ***marries(x, y)***

$\mathcal{P}_1$

? marries(z, w)

Notation conventions

1. $loves(mary,john)$
2. $engaged(mary,john)$
3. $loves(x, y), engaged(x, y) \rightarrow marries(x, y)$

- 1. and 2. are called **facts**.

  $loves(mary,john)$

  $engaged(mary,john)$

- 3 is called a **rule**
  - if …, then…

  $loves(x, y), engaged(x, y) \rightarrow marries(x, y)$

Another example

Express the following as a (predicate logic) definite clause program:

One can travel between any two cities X and Y, if there is a direct flight from X to Y and there is a ticket available to travel from X to Y.

A ticket between X and Y is available, if it can be bought online or bought over the phone.

Another example

One can travel between any two cities X and Y, **if** *there is a direct flight from X to Y* **and** *there is a ticket available to travel from X to Y*.

A ticket between X and Y is available, if it can be bought online or bought over the phone.

$$direct\_flight(X,Y) \wedge ticket(X,Y) \rightarrow travel\_between(X,Y)$$

$$direct\_flight(X,Y)\ ,\ ticket(X,Y) \rightarrow travel\_between(X,Y)$$

Another example

One can travel between any two cities X and Y, if there is a direct flight from X to Y and there is a ticket available to travel from X to Y.

A ticket between X and Y is available, ***if*** *it can be bought online* **or** *bought over the phone*.

$$direct\_flight(X,Y)\ ,\ ticket(X,Y) \rightarrow travel\_between(X,Y)$$

$$buy\_online(X,Y) \lor buy\_phone(X,Y) \rightarrow ticket(X,Y)$$

But this does not have the form of a rule:

$\alpha_1 \land \ldots \land \alpha_m \rightarrow \alpha$

Can we fix it?

We apply the ‘PNF to definite rules’ process!

One can travel between any two cities X and Y, if there is a direct flight from X to Y and there is a ticket available to travel from X to Y.

A ticket between X and Y is available, ***if*** *it can be bought online* ***or*** *bought over the phone*.

$$direct\_flight(X,Y)\ ,\ ticket(X,Y) \rightarrow travel\_between(X,Y)$$

$$buy\_online(X,Y) \lor buy\_phone(X,Y) \rightarrow ticket(X,Y)$$

$\neg(buy\_online(X,Y) \lor buy\_phone(X,Y)) \lor ticket(X,Y)$

$(\neg buy\_online(X,Y) \land \neg\, buy\_phone(X,Y)) \lor ticket(X,Y)$

$(\neg\, buy\_online(X,Y) \lor ticket(X,Y)) \land (\neg\, buy\_phone(X,Y) \lor ticket(X,Y))$

$\boldsymbol{buy\_online(X,Y) \rightarrow ticket(X,Y)}$ and

$\boldsymbol{buy\_phone(X,Y) \rightarrow ticket(X,Y)}$

Another example

One can travel between any two cities X and Y, if there is a direct flight from X to Y and there is a ticket available to travel from X to Y.

A ticket between X and Y is available, if it can be bought online or bought over the phone.

$$direct\_flight(X,Y)\ ,\ ticket(X,Y) \rightarrow travel\_between(X,Y)$$

$$buy\_online(X,Y) \rightarrow ticket(X,Y)$$

$$buy\_phone(X,Y) \rightarrow ticket(X,Y)$$

Quick guide: representing and and or

- P if A and B and ….and Q

A, B,…., Q $\rightarrow$ P

- P if A or B or ….or Q

A $\rightarrow$ P

B $\rightarrow$ P

:

Q $\rightarrow$ P

Another example

One can travel between any two cities X and Y, if there is a direct flight from X to Y and there is a ticket available to travel from X to Y.

A ticket between X and Y is available, if it can be bought online or bought over the phone.

RULES:

$$direct\_flight(X,Y)\ ,\ ticket(X,Y) \rightarrow travel\_between(X,Y)$$

$$buy\_online(X,Y) \rightarrow ticket(X,Y)$$

$$buy\_phone(X,Y) \rightarrow ticket(X,Y)$$

Let's add the facts

RULES:

$$direct\_flight(X,Y)\ ,\ ticket(X,Y) \rightarrow travel\_between(X,Y)$$

$$buy\_online(X,Y) \rightarrow ticket(X,Y)$$

$$buy\_phone(X,Y) \rightarrow ticket(X,Y)$$

FACTS:

- *There are direct flights from London (lon) to Paris.*
- *There are direct flights from Paris to Athens.*
- *The ticket from London to Paris can be bought online.*
- *The ticket from Paris to Athens can be bought over the phone.*

$$direct\_flight(lon,paris)$$

$$direct\_flight(paris,athens)$$

$$buy\_online(lon,paris)$$

$$buy\_phone(paris,athens)$$

So here is our Program

direct_flight(X,Y) , ticket(X,Y) → travel_between(X,Y)
buy_online(X,Y) → ticket(X,Y)
buy_phone(X,Y) → ticket(X,Y)

direct_flight(lon,paris)
direct_flight(paris,athens)
buy_online(lon,paris)
buy_phone(paris,athens)

$\mathcal{P}_1$

- Let's make a Query:

? travel_between(lon, paris)

Note: in the previous example a query contained variables, but a query can also contain constants

So here is our Program

1) $df(X,Y)\ ,\ t(X,Y) \rightarrow tb(X,Y)$
2) $on(X,Y) \rightarrow t(X,Y)$
3) $ph(X,Y) \rightarrow t(X,Y)$

4) $df(lon,par)$
5) $df(par,ath)$
6) $on(lon,par)$
7) $ph(par,ath)$

$\mathcal{P}_1$

- Let's make a Query:

$?\ tb(lon, par)$

So here is our Program

1) $df(X,Y)\ ,\ t(X,Y) \rightarrow tb(X,Y)$
2) $on(X,Y) \rightarrow t(X,Y)$
3) $ph(X,Y) \rightarrow t(X,Y)$

4) $df(lon,par)$
5) $df(par,ath)$
6) $on(lon,par)$
7) $ph(par,ath)$

$\mathcal{P}_1$

$?\ tb(lon, par)$

1)
{ (X/lon), (Y/par) }

$?\ df(lon, par)\ ,\ t(lon, par)$

4)

I **choose** the left most atom

$?\ t(lon, par)$

Both 2) and 3) match!!
Which one to **choose**?

Choice points and **Backtracking**

Choice point

1) $df(X,Y)\ ,\ t(X,Y) \rightarrow tb(X,Y)$
2) $on(X,Y) \rightarrow t(X,Y)$
3) $ph(X,Y) \rightarrow t(X,Y)$

4) $df(lon,par)$
5) $df(paris,ath)$
6) $on(lon,par)$
7) $ph(par,ath)$

$\mathcal{P}_1$

$?\ tb(lon, par)$

1)
{ (X/lon), (Y/par) }

$?\ df(lon, par)\ ,\ t(lon, par)$

Usual to choose **left most** atom in query

So here is our Program

1) $df(X,Y)\ ,\ t(X,Y) \rightarrow tb(X,Y)$
2) $on(X,Y) \rightarrow t(X,Y)$
3) $ph(X,Y) \rightarrow t(X,Y)$

4) $df(lon,par)$
5) $df(paris,ath)$
6) $on(lon,par)$
7) $ph(par,ath)$

$\mathcal{P}_1$

? $tb(lon, par)$

1) { (X/lon), (Y/par) }

? $df(lon, par)$, $t(lon, par)$

4)

? $t(lon, par)$

Both 2) and 3) match!!
Which one to **choose**?

Choice point

1) $df(X,Y)\ ,\ t(X,Y) \rightarrow tb(X,Y)$
2) $on(X,Y) \rightarrow t(X,Y)$
3) $ph(X,Y) \rightarrow t(X,Y)$

4) $df(lon,par)$
5) $df(paris,ath)$
6) $on(lon,par)$
7) $ph(par,ath)$

$\mathscr{P}_1$

? $tb(lon, par)$

1)
{ (X/lon), (Y/par) }

? $df(lon, par)\ ,\ t(lon, par)$

4)

? $t(lon, par)$

2)
{ (X/lon), (Y/par) }

Choice point

? $on(lon, par)$

6)

$\square$

Choice point

1) $df(X,Y)\ ,\ t(X,Y) \rightarrow tb(X,Y)$
2) $on(X,Y) \rightarrow t(X,Y)$
3) $ph(X,Y) \rightarrow t(X,Y)$

4) $df(lon,par)$
5) $df(paris,ath)$
6) $on(lon,par)$
7) $ph(par,ath)$

$\mathcal{P}_1$

? $tb(lon, par)$

1) { (X/lon), (Y/par) }

? $df(lon, par)\ ,\ t(lon, par)$

4)

? $t(lon, par)$

2) { (X/lon), (Y/par) }

Choice point

? $on(lon, par)$

6)

$\square$

Choose first rule with matching head

Another query

1) *df(X,Y) , t(X,Y) → tb(X,Y)*
2) *on(X,Y) → t(X,Y)*
3) *ph(X,Y) → t(X,Y)*

4) *df(lon,par)*
5) *df(paris,ath)*
6) *on(lon,par)*
7) *ph(par,ath)*

$\mathscr{P}_1$

? tb(par, ath)

1) { (X/par), (Y/ath) }

? df(par, ath) , t(par, ath)

4)

? t(par, ath)

2) { (X/par), (Y/ath) }

Choice point

? on(par, ath)

Failed!

Let's go back to the last choice point and try again…

1) $df(X,Y)\ ,\ t(X,Y) \rightarrow tb(X,Y)$
2) $on(X,Y) \rightarrow t(X,Y)$
3) $ph(X,Y) \rightarrow t(X,Y)$

4) $df(lon,par)$
5) $df(paris,ath)$
6) $on(lon,par)$
7) $ph(par,ath)$

$\mathcal{P}_1$

? $tb(par, ath)$

1) { (X/par), (Y/ath) }

? $df(par, ath)\ ,\ t(par, ath)$

4)

? $t(par, ath)$

3)
{ (X/par), (Y/ath) }

Choice point

? $ph(par, ath)$

7)

$\square$

SUCCESS!

Backtracking to last choice point

1) $df(X,Y)\ ,\ t(X,Y) \rightarrow tb(X,Y)$
2) $on(X,Y) \rightarrow t(X,Y)$
3) $ph(X,Y) \rightarrow t(X,Y)$

4) $df(lon,par)$
5) $df(paris,ath)$
6) $on(lon,par)$
7) $ph(par,ath)$

$\mathcal{P}_1$

? $tb(lon, ath)$

1) { (X/lon), (Y/ath) }

? $df(lon, ath)\ ,\ t(lon, ath)$

4)

? $t(lon, ath)$

3) { (X/lon), (Y/ath) }

Choice point

? $ph(lon, ath)$

6)

$\square$

backtrack to last choice point, and choose next rule

Activity

1) $Q(x,y), T(x,y), S(y) \rightarrow P(x,y)$
2) $Q(a,b)$
3) $Q(a,c)$
4) $Q(a,d)$
5) $T(a,c)$
6) $T(a,d)$
7) $S(d)$

$\mathcal{P}1$

$?\ P(x,y)$

a, b, c, and d are constants
x and y are variables

Show the derivation trees and the order in which they are generated if :

1. The left query atom is always chosen
2. The first listed rule or matching fact is chosen
3. If failure then backtrack to last choice point and choose next listed matching rule or fact

Derivation tree 1

1) $Q(x,y), T(x,y), S(y) \rightarrow P(x,y)$
2) $Q(a,b)$
3) $Q(a,c)$
4) $Q(a,d)$
5) $T(a,c)$
6) $T(a,d)$
7) $S(d)$

$\mathcal{P}_1$

a, b, c, and d are constants
x and y are variables

? $P(x,y)$

1)

? $Q(x,y), T(x,y), S(y)$

2)
{ (x/a), (y/b) }

Choice point

? $T(a,b), S(b)$

Failure, backtrack to last choice point

Derivation tree 2

1) $Q(x,y), T(x,y), S(y) \rightarrow P(x,y)$
2) $Q(a,b)$
3) $Q(a,c)$
4) $Q(a,d)$
5) $T(a,c)$
6) $T(a,d)$
7) $S(d)$

$\mathcal{P}_1$

a, b, c, and d are constants
x and y are variables

? $P(x,y)$

1)

? $Q(x,y), T(x,y), S(y)$

3)
{ (x/a), (y/c) }

Choice point

? $T(a,c), S(c)$

5)

? $S(c)$

■

Backtracking to last choice point

Failure, backtrack again to last choice point

Derivation tree 3

1) $Q(x,y), T(x,y), S(y) \rightarrow P(x,y)$
2) $Q(a,b)$
3) $Q(a,c)$
4) $Q(a,d)$
5) $T(a,c)$
6) $T(a,d)$
7) $S(d)$

$\mathcal{P}_1$

a, b, c, and d are constants
x and y are variables

? $P(x,y)$ — 1)

? $Q(x,y), T(x,y), S(y)$ — **4)** { (x/a), (y/d) } — Choice point

? $T(a,d), S(d)$ — 6)

? $S(d)$ — 7)

$\square$

SUCCESS!

Negation as failure

Negation as failure

- Definite rules do not have negated atoms
- How then would we express the following?

'Someone is innocent if they are not guilty'

- What about:

$\neg\, \boldsymbol{guilty(x) \rightarrow innocent(x)}$

- **But this is not a definite rule!**

Negation as failure

‘Someone is innocent if they are not guilty’

$\neg\, \boldsymbol{guilty(x)} \rightarrow \boldsymbol{innocent(x)}$

- But in the eyes of the law, someone is innocent if the prosecution **FAILED to PROVE** that he/she is guilty!

Negation as failure

- It simply means that "**not** p" **must hold** if we **failed** to **prove** "p"!
  - **every** attempt to prove p failed, that is, there is **no successful derivation tree for p**.
- *'we cannot prove "p" to be true, so "not p" must be true!'*
- But note: saying 'not p *is true' is not strictly the same as saying '*$\neg p$ *is true', because the latter would require us to prove* $\neg p$

- What we cannot *show/prove* to be true must be false (this is called *closed world assumption*).

Example

1) *bird(x)*, not *can_fly(x)* → *flightless_bird(x)*
2) *bird(eagle)*
3) *can_fly(eagle)*
4) *bird(chicken)*

$\mathscr{P}_1$

every attempt to prove *can_fly(chicken)* failed (that is, there is **no successful derivation tree for** *can_fly(chicken)*). So **not** ***can_fly(chicken)*** must be true.

? *flightless_bird (chicken)*

1) { (x/chicken) }

? *bird(chicken)*, not *can_fly(chicken)*

4)

? not *can_fly(chicken)*

? *can_fly(chicken)*

■

□

Example: another query

1) *bird(x)*, not *can_fly(x)* → *flightless_bird(x)*
2) *bird(eagle)*
3) *can_fly(eagle)*
4) *bird(chicken)*

$\mathcal{P}_1$

? flightless_bird (eagle)

1) { (x/eagle) }

? *bird(eagle)*, not *can_fly(eagle)*

2)

? not *can_fly(eagle)*

? *can_fly(eagle)*

3)

□

■

Our **attempt** to prove *can_fly(eagle)* succeeded (that is, there **was** a **successful derivation tree for** *can_fly(eagle)*). So **not** *can_fly(eagle)* must be false.

From definite clause programming to **logic programming**

Predicate logic programming =

Definite Clause Programming +

1) **Control** (procedural) features with selection of leftmost query atom and topmost program rule or fact
2) **Backtracking** to choice points
3) **Negation** as Failure (Closed World Assumption)

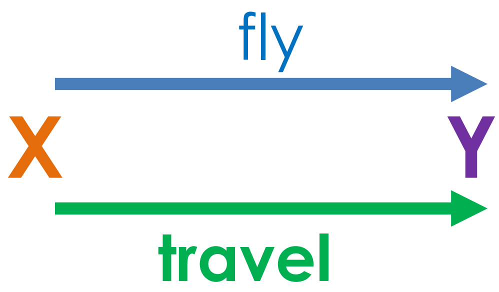
A predicate logic programming derivation includes all derivation trees obtained on backtracking in order to prove query and all trees attempting to prove $\boldsymbol{p}$ given a query 'not $\boldsymbol{p}$'

Recursion

Predicate Logic Programming Example of Recursion

- Specify the following as a predicate logic program:

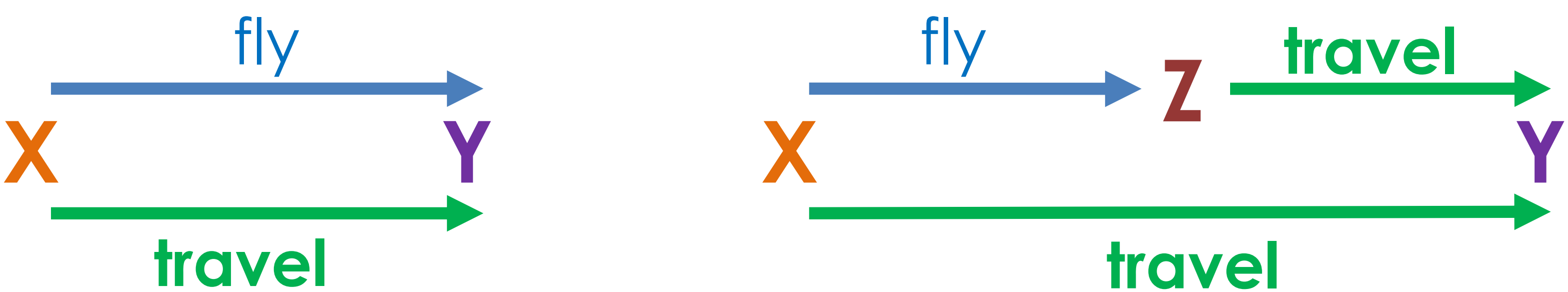
One can travel from city X to Y if there is a direct flight from X to Y. *Otherwise, one can travel from city X to Y, if one can get a direct flight from X to Z and then travel from city Z to Y*



Predicate Logic Programming Example of Recursion

- Specify the following as a predicate logic program:

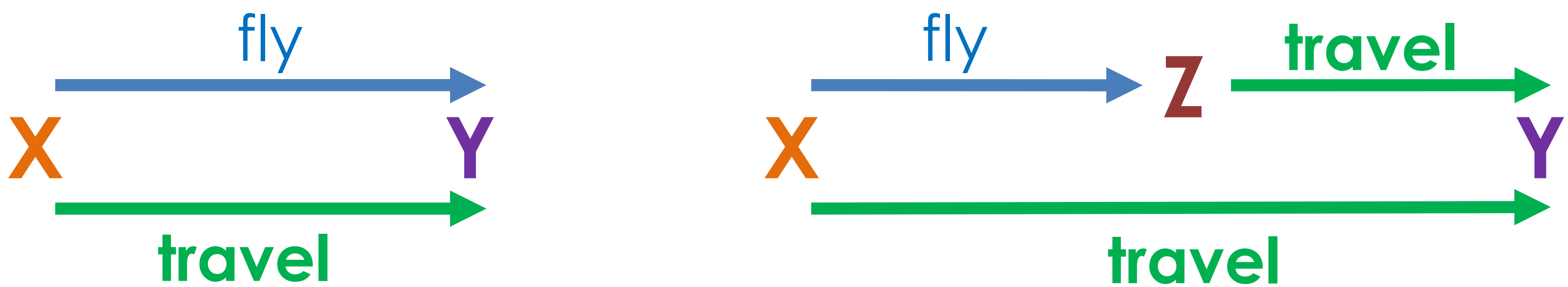
One can travel from city X to Y if there is a direct flight from X to Y. ***Otherwise, one can travel from city X to Y, if one can get a direct flight from X to Z and then travel from city Z to Y***



Solving the problem of whether one can travel from X to Y depends on solving a ***smaller version of the same*** *problem – whether can one travel from Z to Y.*

Predicate Logic Programming Example of Recursion

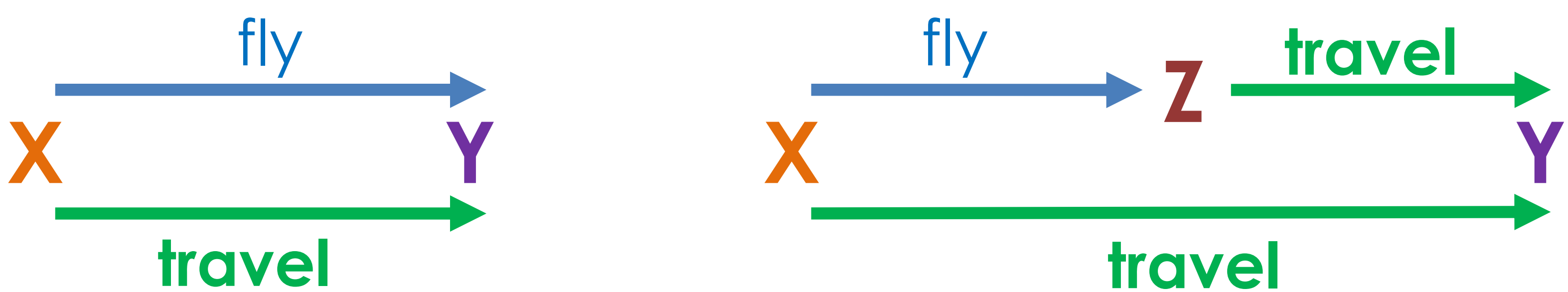
One can travel from city X to Y if there is a direct flight from X to Y. *Otherwise, one can travel from city X to Y, if one can get a direct flight from X to Z and then travel from city Z to Y*



$$direct\_flight(x,y) \rightarrow travel\_from(x,y)$$

Predicate Logic Programming
Example of Recursion

One can travel from city X to Y if there is a direct flight from X to Y. ***Otherwise, one can travel from city X to Y, if one can get a direct flight from X to Z and then travel from city Z to Y***



$$direct\_flight(x,y) \rightarrow travel\_from(x,y)$$
$$direct\_flight(x,z)\ ,\ travel\_from(z,y) \rightarrow travel\_from(x,y)$$

Predicate Logic Programming Example of Recursion

1) *direct_flight(x,y) → travel_from(x,y)*
2) *direct_flight(x,z) , travel_from(z,y) → travel_from(x,y)*
3) *direct_flight(london, paris)*
4) *direct_flight(paris, athens)*

$\mathcal{P}1$

? travel_from(london,athens)

1)
{ (x/london), (y/athens) }

Choice point

? direct_flight(london,athens)

■

Predicate Logic Programming
Example of Recursion

1) $direct\_flight(x,y) \rightarrow travel\_from(x,y)$
2) $direct\_flight(x,z)\ ,\ travel\_from(z,y) \rightarrow travel\_from(x,y)$
3) $direct\_flight(london, paris)$
4) $direct\_flight(paris, athens)$

$\mathscr{P}1$

? $travel\_from(london,athens)$

1)
{ (x/london), (y/athens)}

Choice point

? $direct\_flight(london,athens)$

Backtrack to last choice point

Predicate Logic Programming
Example of Recursion

1) direct_flight(x,y) → travel_from(x,y)
2) direct_flight(x,z) , travel_from(z,y) → travel_from(x,y)
3) direct_flight(london, paris)
4) direct_flight(paris, athens)

$\mathscr{P}1$

? travel_from(london,athens)

2)
{ (x/london), (y/athens) }

Choice point

? direct_flight(london,z), travel_from(z,athens)

3)
{ (z/paris) }

? travel_from(paris,athens)

1)
{ (x/paris), (y/athens) }

? direct_flight(paris,athens)

4)

$\square$

Recursion

- **When something is defined in terms of smaller versions of itself**
- Generally in CS, recursion is a method where solving a problem depends on solving smaller versions of the same problem
- A recursive program consists of:
  - **a base case:** the rule that cannot be expressed in terms of smaller versions of itself

$$direct\_flight(x,y) \rightarrow travel\_from(x,y)$$

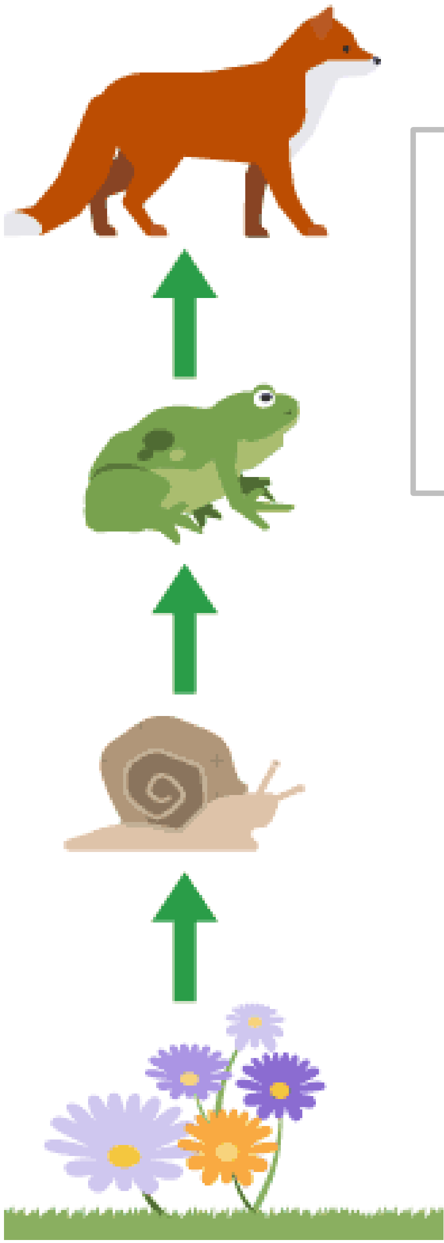
the problem of whether one can travel from X to Y is when there is a direct flight from X to Y

  - **a recursive case:** the rule that can be expressed in terms of smaller versions of itself

$$direct\_flight(x,z)\ ,\ travel\_from(z,y) \rightarrow travel\_from(x,y)$$

Solving the problem of whether one can travel from X to Y depends on solving a smaller version of the same problem – whether can one travel from Z to Y.

Another Example



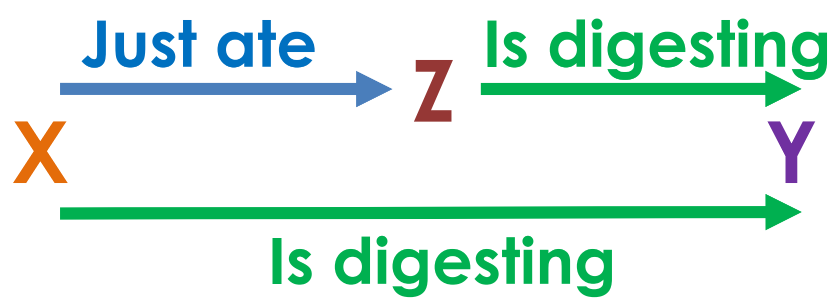
justAte(X,Y) → isDigesting(X,Y)

justAte(X,Z), isDigesting (Z,Y) → isDigesting (X,Y)

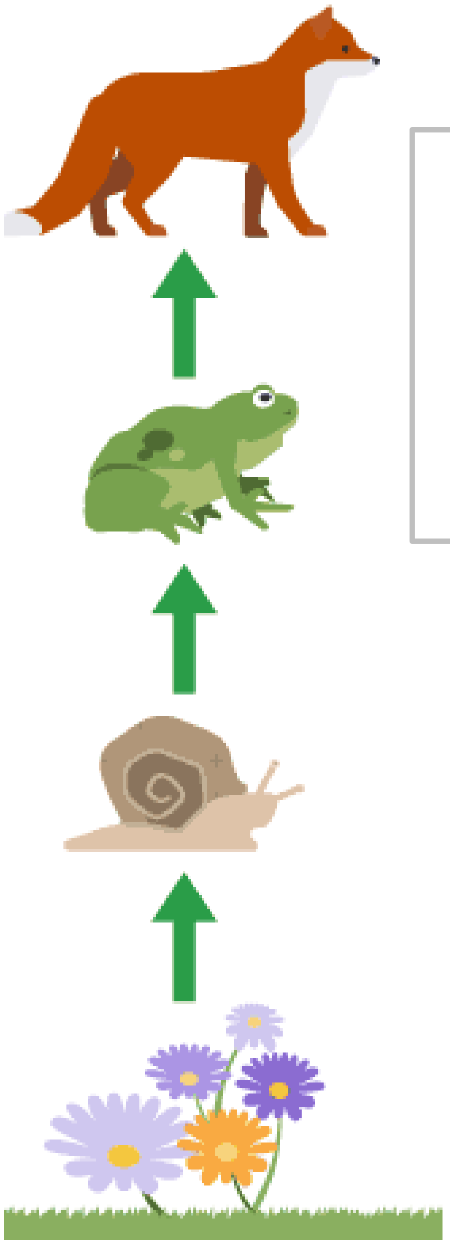
justAte(snail, flower)

justAte(frog,snail)

justAte(fox,frog)



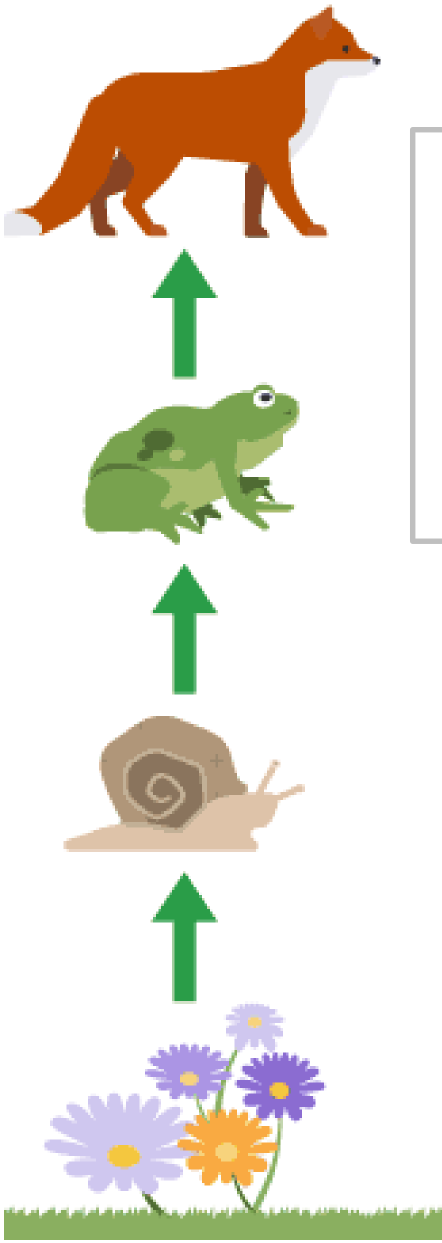
Another Example



1) *justAte(X,Y) → isDigesting(X,Y)*
2) *justAte(X,Z), isDigesting (Z,Y) → isDigesting (X,Y)*
3) *justAte(snail, flower)*
4) *justAte(frog,snail)*
5) *justAte(fox,frog)*

? isDigesting(fox,flower)

Another Example



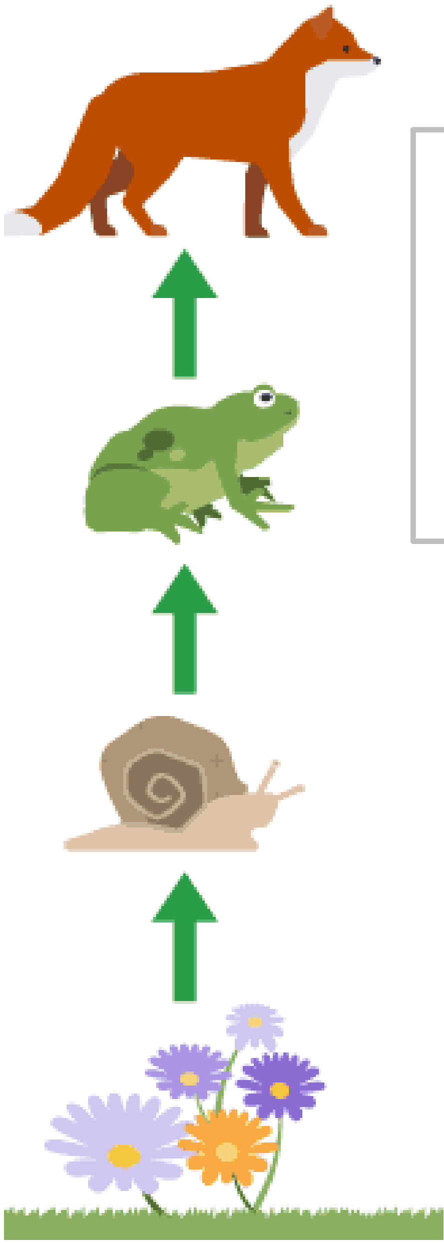
1) *justAte(X,Y) → isDigesting(X,Y)*
2) *justAte(X,Z), isDigesting (Z,Y) → isDigesting (X,Y)*
3) *justAte(snail, flower)*
4) *justAte(frog,snail)*
5) *justAte(fox,frog)*

? isDigesting(fox,flower)

1)
{ (X/fox), (Y/flower) }

? justAte(fox,flower)

Another Example



1) *justAte(X,Y) → isDigesting(X,Y)*
2) *justAte(X,Z), isDigesting (Z,Y) → isDigesting (X,Y)*
3) *justAte(snail, flower)*
4) *justAte(frog,snail)*
5) *justAte(fox,frog)*

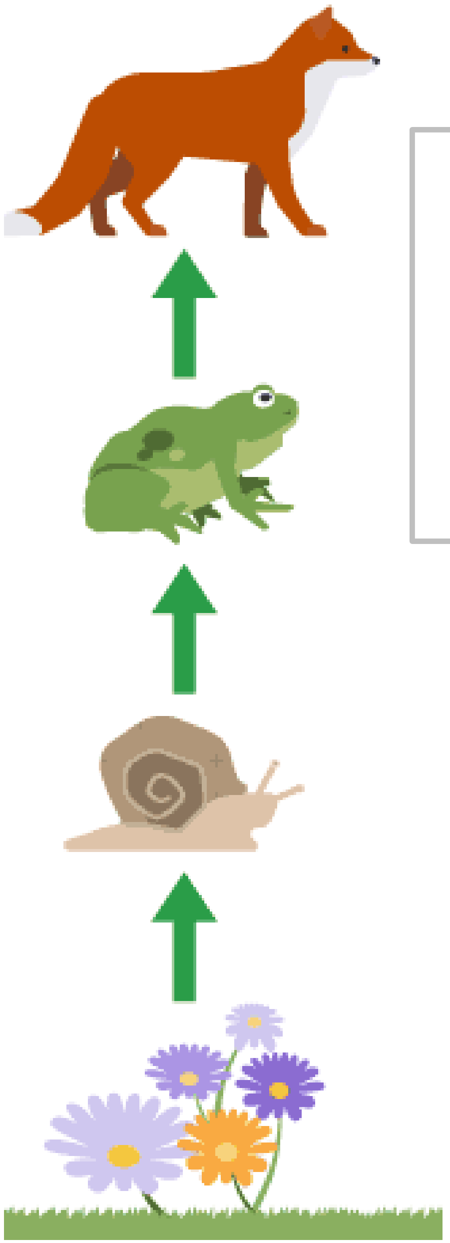
? isDigesting(fox,flower)

1)
{ (X/fox), (Y/flower) }

? justAte(fox,flower)

Backtrack to last choice point

Another Example



1) $justAte(X,Y) \rightarrow isDigesting(X,Y)$
2) $justAte(X,Z), isDigesting(Z,Y) \rightarrow isDigesting(X,Y)$
3) $justAte(snail, flower)$
4) $justAte(frog, snail)$
5) $justAte(fox, frog)$

? *isDigesting(fox,flower)*

2)
{ (X/fox), (Y/flower) }

? *justAte(fox, Z), isDigesting (Z, flower)*

5)
{ (Z/frog) }

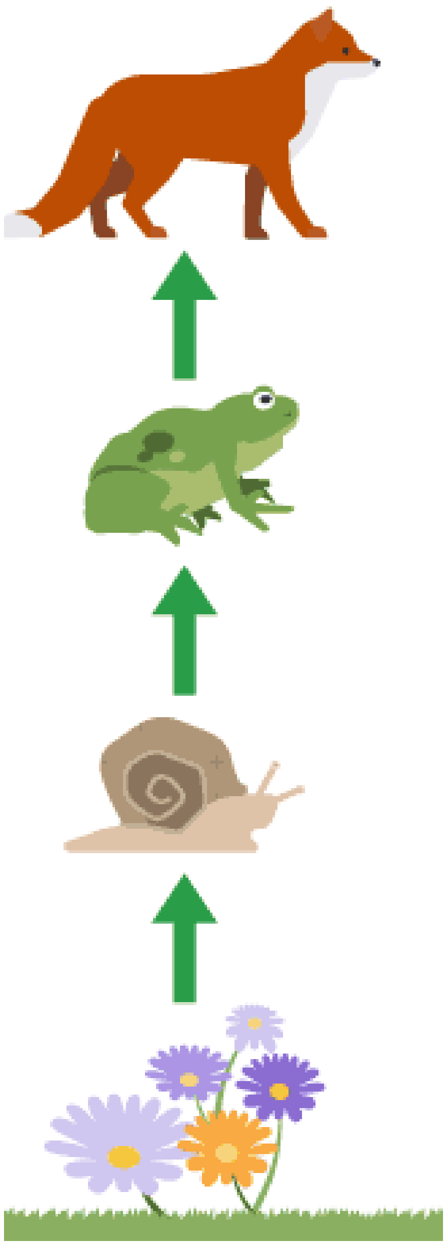
? *isDigesting (frog, flower)*

1)
{ (X/frog), (Y/flower) }

? *justAte (frog, flower)*

Backtrack to last choice point

Another Example



1) *justAte(X,Y) → isDigesting(X,Y)*
2) *justAte(X,Z), isDigesting (Z,Y) → isDigesting (X,Y)*
3) *justAte(snail, flower)*
4) *justAte(frog,snail)*
5) *justAte(fox,frog)*

? isDigesting(fox,flower)

2) { (X/fox), (Y/flower) }

? justAte(fox, Z), isDigesting (Z, flower)

5) { (Z/frog) }

? isDigesting (frog, flower)

2) { (X/frog), (Y/flower) }

? justAte(frog, Z), isDigesting (Z, flower)

4) { (Z/snail) }

? isDigesting (snail, flower)

1) {(X/frog), (Y/snail) }

? justAte (snail, flower)

3)

□

Another example

1) *direct_flight(x,y) → travel_from(x,y)*
2) *direct_flight(x,z) , travel_from(z,y) → travel_from(x,y)*
3) *direct_flight(london, paris)*
4) *direct_flight(paris, athens)*
5) *direct_flight(athens,cairo)*

Draw only the successful derivation tree for the query:

? travel_from(london,cairo)

Another example

1) *direct_flight(x,y) → travel_from(x,y)*
2) *direct_flight(x,z) , travel_from(z,y) → travel_from(x,y)*
3) *direct_flight(london, paris)*
4) *direct_flight(paris, athens)*
5) *direct_flight(athens,cairo)*

? travel_from(london,cairo) — 2) { (x/london), (y/cairo) }

? direct_flight(london,z) , travel_from(z,cairo) — 3) { (z/paris)}

? travel_from(paris,cairo) — 2) {(x/paris), (y/cairo)

? direct_flight(paris,z) , travel_from(z,cairo) — 4) { (z/athens)}

? travel_from(athens,cairo) — 1) { ((x/athens), (y/cairo) }

? direct_flight(athens,cairo) — 5)

□

Tutorials and Next Lecture

- **Large Group Tutorial:**
  - Questions 1 and 2a) are in slides; make sure you can do them yourself!
  - *Tutorial questions 2b), 3 and 4* not in slides
- **Small Group Tutorials:**
  - You can complete all questions.

- **Next Lecture:**
  - Revision
- **Next Year:** 5CCS2PLD you will revisit and start 'doing' logic programming