CS339: Abstractions and Paradigms for Programming

Logic Paradigm (Cont.)

Manas Thakur
CSE, IIT Bombay

Autumn 2024

Recall the problem that we had ended with

- Inefficiency in mutually exclusive cases:

```
max(X, Y, Y) :- X =< Y.
max(X, Y, X) :- X > Y.
```

- Unexpected semantics in some others:

```
factorial(N, 1) :- N = 0.
factorial(N, Result) :- M is N - 1,
                        factorial(M, SubRes),
                        Result is N * SubRes.
```

- What we want: A way to avoid further exploration.

Cuts

- Interrupt execution if a subgoal is satisfied.
- Improved *max* program:

```
max(X, Y, Y) :- X =< Y, !.
max(X, Y, X) :- X > Y.
```

- How to correct the problem with removing N > 0 then?

```
factorial(N, 1) :- N = 0.
factorial(N, Result) :- ~~N > 0,~~ M is N - 1,
                        factorial(M, SubRes),
                        Result is N * SubRes.
```

- Logic is independent of order; cuts make Prolog programs impure but efficient.

Green cuts and Red cuts

- Green cuts: Do not change program semantics.

```
max(X, Y, Y) :- X =< Y, !.
max(X, Y, X) :- X > Y.
```

  - Main purpose: Efficiency.

- Red cuts: May change program semantics for certain inputs.

```
factorial(N, 1) :- N = 0, !.
factorial(N, Result) :- M is N - 1,
                        factorial(M, SubRes),
                        Result is N * SubRes.
```

- What do we actually “cut”?

Negation

```
factorial(N, 1) :- N = 0, !.
factorial(N, Result) :- M is N - 1,
                        factorial(M, SubRes),
                        Result is N * SubRes.
```

- Yet another way:

```
factorial(N, 1) :- N = 0.
factorial(N, Result) :- not(N = 0), M is N - 1,
                        factorial(M, SubRes),
                        Result is N * SubRes.
```

- Though using negations might look easier than cuts, there is a big catch with negations in Prolog!

Negation as failure

- Try this:

```
bachelor_student(X) :- not(married(X)), student(X).
student(joe).
married(john).

?- bachelor_student(john).
?- bachelor_student(joe).
```

- What should be the outcome of this?

```
?- bachelor_student(X).
```

- **Insight:** Negation is actually implemented as follows:

```
not(Goal) :- call(Goal), !, fail.
not(Goal).
```

- **PCQ:** How can you correct this to be logically consistent?

Puzzles in Prolog

Who lives where

- Anirudh, Krish, Mrinal, Nilabha and Parthiv live in a five-room corridor in the H17 hostel, starting from 1 to 5.
- Anirudh doesn't live in the fifth room and Krish doesn't live in the first.
- Mrinal doesn't live in the first or the last rooms, and he is not in a room adjacent to Parthiv or Krish.
- Nilabha lives in some room numbered after that of Krish.

- Who lives in which room?

Who lives where (Cont.)

- Say the solution should look like this:
  - `[room(_,5), room(_,4), room(_,3), room(_,2), room(_,1)]`
- The five variables to be assigned rooms are: `A`, `K`, `M`, `N` and `P`.
- Say the structure `rooms(Rooms)` contains the solution list.
- The first constraint can be encoded as:
  - `member(room(anirudh, A), Rooms), A \= 5.`
- We can similarly add the other constraints to give the complete *program*.

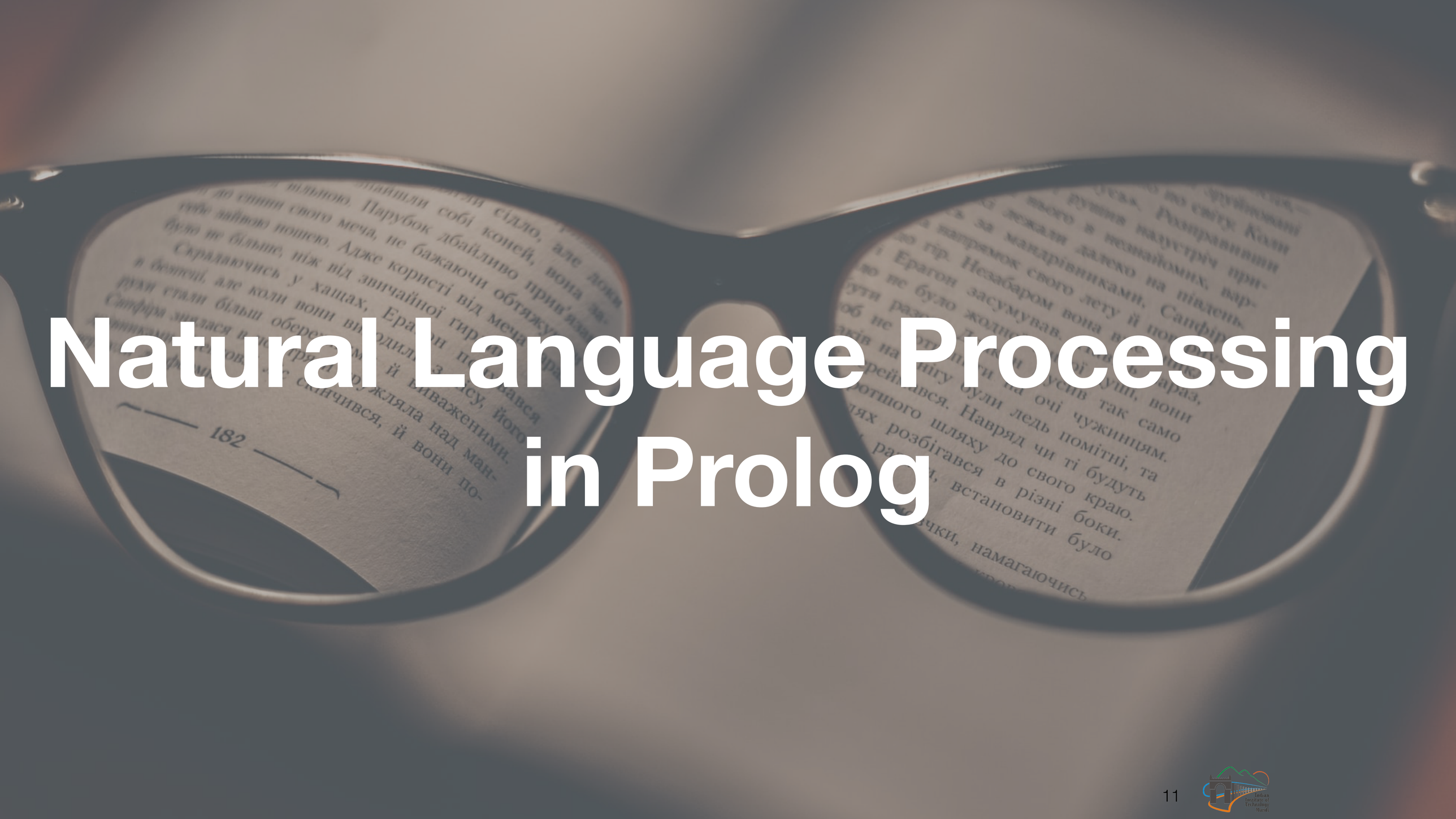
Who lives where (Cont.)

```
rooms([room(_,5),room(_,4),room(_,3),room(_,2),room(_,1)]).
hostel(Rooms) :- rooms(Rooms),
    member(room(anirudh, A), Rooms), A \= 5,
    member(room(krish, K), Rooms), K \= 1,
    member(room(mrinal, M), Rooms), M \= 1, M \= 5,
    member(room(parthiv, P), Rooms),
    not(adjacent(M, P)), not(adjacent(M, K)),
    member(room(nilabha, N), Rooms), N > K,
    print_rooms(Rooms).
```

- where `adjacent` and `print_rooms` are defined as follows:

```
adjacent(X, Y) :- X =:= Y+1.
adjacent(X, Y) :- X =:= Y-1.
print_rooms([A | B]) :- write(A), nl, print_rooms(B).
print_rooms([]).
```

Solution: `?- hostel(X).`



Natural Language Processing in Prolog

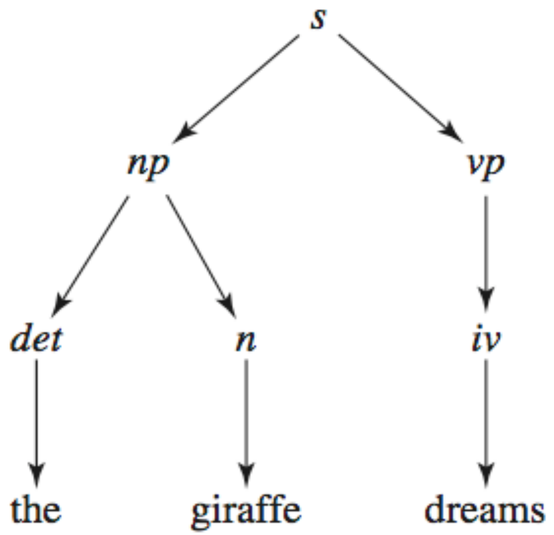
NLP in Prolog

- Natural languages (say English) can be defined using *Backus Naur Form (BNF)* — a grammar notation.
- A Prolog program can model a BNF grammar.
- A Prolog list can model a sentence.
- Example: `[this, teacher, rocks]`
- Thus, running the program can parse an English sentence!

Parse Trees

➤ Consider the following grammar:

$s \rightarrow np\ vp$
$np \rightarrow det\ n$
$vp \rightarrow tv\ np$
$\rightarrow iv$
$det \rightarrow$ this
$n \rightarrow$ teacher
$\rightarrow$ student
$iv \rightarrow$ rocks
$tv \rightarrow$ dreams



Here, *s*, *np*, *vp*, *det*, *n*, *iv*, and *tv* denote "sentence," "noun phrase," "verb phrase," "determiner," "noun," "intransitive verb," and "transitive verb", respectively.

Naive Prolog Encoding

s → *np vp*
np → *det n*
vp → *tv np*
→ *iv*
det → this
n → teacher
→ student
iv → rocks
tv → dreams

```
s(X, Y) :- np(X, U), vp(U, Y).
np(X, Y) :- det(X, U), n(U, Y).
vp(X, Y) :- iv(X, Y).
vp(X, Y) :- tv(X, U), np(U, Y).
det([this | Y], Y).
n([teacher | Y], Y).
n([student | Y], Y).
iv([rocks | Y], Y).
tv([dreams | Y], Y).
```

- **First rule:**
  - List X is a sentence leaving tail Y if X is a noun phrase leaving tail U and U is a verb phrase leaving tail Y.
  - Successful parse from s: Y should be empty.

Definitive Clause Grammars (DCGs)

```
s(X, Y) :- np(X, U), vp(U, Y).
```

```
s --> np, vp.
```

```
s(X, Y) :- np(X, U), vp(U, Y).
np(X, Y) :- det(X, U), n(U, Y).
vp(X, Y) :- iv(X, Y).
vp(X, Y) :- tv(X, U), np(U, Y).
det([this | Y], Y).
n([teacher | Y], Y).
n([student | Y], Y).
iv([rocks | Y], Y).
tv([dreams | Y], Y).
```

```
s --> np, vp.
np --> det, n.
vp --> iv.
vp --> tv, np.
det --> [this].
n --> [teacher].
n --> [student].
iv --> [rocks].
tv --> [dreams].
```