EECS 3401 — AI and Logic Prog. — Lectures 4 & 5

Adapted from slides of Prof. Yves Lesperance

Vitaliy Batusov
vbatusov@cse.yorku.ca

York University

September 23, 2020

Intro to Prolog

- Today: **Prolog: Core Concepts and Notation**
- Required reading: Clocksin & Mellish, C.S., *Programming in Prolog*, 5^{th} edition, Springer Verlag, New York, 2004.
- Chapters 1, 2, 3.1–3.3, 8

Declarative/logic programming

Key idea: the program is a logical theory

- Axiomatize a domain of interest, and then query it
- Most popular language — Prolog
- Core constructs, terms, and statements come from FOL

Terms

- Prolog statements express relationships between *terms*
- Prolog **terms** = a generalization of FOL terms
  constants, variables, functions with other terms as arguments
- Examples:

`john`	constant
`john_smith`	constant
`X`	variable
`Node`	variable
`_person`	variable
`fatherOf(paul)`	unary function
`date(23,09,2020)`	3-ary function

For complex terms:

$$\underbrace{\texttt{fatherOf}}_{\text{functor}}\underbrace{\texttt{(paul)}}_{\text{arity is 1}}$$

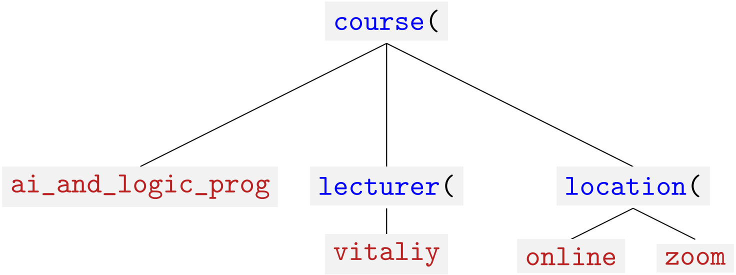
$$\underbrace{\texttt{date}}_{\text{functor}}\underbrace{\texttt{(23,09,2020)}}_{\text{arity is 3}}$$

Terms

- Variables begin with upper-case character or the underscore "_"
- Constants and functors begin with a lower-case character
- As always, **terms denote objects**
- Compound terms are called **structures** (or structs) and are used to represent complex, structured data

```
course(ai_and_logic_prog, lecturer(vitaliy),
       location(online, zoom))
```

- Terms usually have a tree structure:



Facts

- **Facts** are like atomic formulas in FOL
- Syntax is exactly like that of terms, **but** facts are stand-alone parts of the program
- Each fact ends with a period.

```
fatherOf(paul, henry).
```

```
mortal(socrates).
```

```
likes(X, iceCream).
```

$$\underbrace{\texttt{likes(mary, }\underbrace{\texttt{brotherOf(helen)}}_{\text{term}}\texttt{).}}_{\text{fact}}$$

- Variables are implicitly universally quantified

`likes(X, iceCream).` means $\forall X(\mathit{likes}(X, \mathit{iceCream}))$

Rules

- **Rules** are conditional statements

$$\texttt{mortal(X)} \underbrace{\texttt{:-}}_{\text{“if”}} \texttt{human(X).}$$

$$\forall X(\mathit{human}(X) \rightarrow \mathit{mortal}(X))$$

- The left-hand side is an atom/fact, called the **head**
- The right-hand side is the **body** of the rule

Rules

The body of the rule can be a **conjunction**:

```
daughter(X, Y) :- father(Y, X), female(X).
```

The comma "`,`" means "and".
Equivalently in FOL:

$$\forall X \forall Y (father(Y, X) \land female(X) \rightarrow daughter(X, Y))$$

Rules: more examples

```
ancestor(X, Y) :- father(X,Z), ancestor(Z,Y).
```

- Recursive rule
- In FOL:

$$\begin{aligned} & \forall X \forall Y \forall Z (\mathit{father}(X, Z) \land \mathit{ancestor}(Z, Y) \rightarrow \mathit{ancestor}(X, Y)) \\ \equiv\ & \forall X \forall Y (\exists Z (\mathit{father}(X, Z) \land \mathit{ancestor}(Z, Y)) \rightarrow \mathit{ancestor}(X, Y)) \end{aligned}$$

- Thus, consider variables which appear only in the body as *existentially quantified*
- Interestingly, this kind of statement doesn't actually work in FOL — it's an instance of *transitive closure* — but does work in Prolog due to some semantic differences (later)

Queries

- A **query** is a question posed to a Prolog program
- More generally, a **goal** is a statement to be proved. Query = user-issued goal.
- Program = KB, query = formula
- "Does the KB entail this formula?"
- Let the program be

```
mortal(X) :- human(X).
human(ulyssus).  human(penelope).
god(zeus).
```

- When the program is queried with

```
?- mortal(ulyssus).
```

the Prolog interpreter derives the answer `Yes`.

Open Queries

- Can have variables in the query, e.g.,

```
?- mortal(X).
```

- Consider the variables in queries to be existentially quantified
- The interpreter tries to find a binding for the variables for which the query is true with respect to the program
- Can query the interpreter for all possible bindings

```
?- mortal(X).
X = ulyssus ;
X = penelope
Yes
```

Open Queries

- Can have conjunctive queries

```
?- mortal(X), mortal(Y), not (X=Y).
X = ulyssus,
Y = penelope ;

X = penelope,
Y = ulyssus ;

false.
```

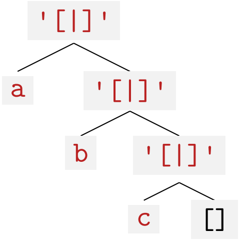
Lists

- A **list** is a special kind of term
- An arbitrary-length ordered sequence of elements
- Due to their usefulness, lists get special syntax, but are regular terms under the hood

  `[]` is the empty list

  `[a, b, c]` is a list of three components, namely `a`, `b`, and `c`
- Can peek under the hood using the query `display(X)`:

```
?- display([a,b,c]).
'[|]'(a,'[|]'(b,'[|]'(c,[])))
```



Lists

- Standard syntax: `[First | Rest]`
- Used to gain access to:

`First`	The **head** — the first element of the list
`Rest`	The **tail** — the remainder of the list

Deconstructing a list:

```
?- [cat, dog, monkey] = [X | Y].
X = cat,
Y = [dog, monkey].
```

Constructing a list:

```
?- X = cat, Y = [dog, monkey], Z = [X | Y].
X = cat,
Y = [dog, monkey].
Z = [cat, dog, monkey].
```

Unification

- **Unification** — an essential operation in Prolog
- *Matching* of one structure with another, instantiating variables as necessary
- Achieved using the operator `=`
  Remember: "$=$" is neither numerical equality nor identity
- So when we issue a query like `[cat, dog, monkey] = [X | Y]`, we are asking the interpreter to find a binding for all variables such the left-hand side and the right-hand side are identical.

Building a knowledge base

- To be used in computation, facts and rules must be stored in the dynamic database (internal to the interpreter)
- Facts and rules get into the database through **assertion** and **consultation**
- Consultation loads facts and rules from a file

Assertion

- *Assert the fact* `human(ulyssus)`

```
?- assert(human(ulyssus)).
```

This adds it to the dynamic knowledge base of the interpreter.

- We can now issue a query

```
?- human(X).
```

and the interpreter will reply with `X = ulyssus`.

- Similarly, the special predicate *retract* removes facts and rules from the dynamic KB.
- Avoid `assert` and `retract` whenever possible (they are meta-predicates which change the state)

Consult

- This loads facts and rules from a file `family.prolog`:

```
?- consult('family.prolog').
```

- Synonym:

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

Semantics of Prolog, intuitively

- A Prolog program defines a set of relations — i.e., specifies which tuples of objects/terms belong to a particular relation
- In other words, a prolog program defines a *model* (in the sense of FOL)
- Note: a Prolog constant (e.g., `cat`) is a *literal*. In FOL, two distinct constants can be mapped to the same domain object. In Prolog, distinct literals are interpreted as distinct objects. Same goes for all other symbols.
- Thus, Prolog merges the notion of terms and domain objects into one.
- Declarative programming generally avoids state changing operations. Once written, the datastructures are immutable, and all the useful work is done in the process of proving some goal from the existing facts and rules.

Semantics of Prolog, cont.

- Consider the program

```
fatherOf(john,paul).
fatherOf(mary,paul).
motherOf(john,lisa).
parentOf(X,Y) :- fatherOf(X,Y).
parentOf(X,Y) :- motherOf(X,Y).
```

- This specifies `fatherOf/2` as the relation $\{\langle$ `john` , `paul` $\rangle, \langle$ `mary` , `paul` $\rangle\}$.
- Similarly for `motherOf/2` , `parentOf/2`

Rules as Procedures

- Recall: a rule has the form

```
head :- body.
```

- The **head** is like the name of a procedure
- The **body** is like the body of the procedure — a sequence of sub-goals that have to be proved to show that the head's goal holds
- The sub-goals are proved in the left-to-right order; if in the process a variable is bound to something, the binding persists for the subsequent sub-goals
- The rule succeeds if all sub-goals succeed

Passing values

- Calling a goal can instantiate its variables
- A sub-goal's success can bind a variable, also binding the same variable in the goal
- Akin to passing values in or out of a procedure

Example

- A program:

```
motherOf(john,lisa).
parentOf(X,Y) :- motherOf(X,Y).
```

- Queries:

```
?- parentOf(john, X).
X = lisa.

?- parentOf(X, lisa).
X = john.

?- parentOf(X, Y).
X = john,
Y = lisa.
```

Relational Thinking

- No functions per se in Prolog (except in arithmetics)
- When writing a program, try formulating statements about function values as relational facts
- Example: factorial

```
factorial(0, 1).
factorial(N, M) :- K is N-1, factorial(K, L),
                   M is N * L.
```

- To compose functions as in $Y = f(g(X))$, you must name intermediate results:

```
fg(X, Y) :- g(X, Z), f(Z, Y).
```

Asmost Everything is a Term

- **Syntactically**, almost everything is a term in Prolog
- Lists are terms (recall an earlier slide)
- Rules are terms

```
grandfather(X,Y) :- father(X,Z), father(Z,Y).
```

  What is the functor here?

- Queries are terms

Arithmetics in Prolog

- For convenience, Prolog retains arithmetic functions as actual functions:[1]

```
?- X is exp(1).
X = 2.718281828459045.

?- X is (4 + 2) * 5.
X = 30.
```

- Meaning of `is`: **evaluate** the right-hand side and **unify** the result with the left-hand side.
- In contrast: the unification operator `=` will **not** evaluate the term on RHS, try it.

[1] `exp(N)` means e^N

Operators

- Some functors are represented by *infix* or *prefix* or *postfix* operators

prefix	$F\,ab$
infix	$a\,F\,b$
postfix	$ab\,F$

- Some infix operators: `is`, `=`, `+`, `*`, `/`, `mod`, `>`, `>=`, `:-`, `,`, etc.
- `+` and `-` are both prefix and infix: `+(1,2)` is the same as `1 + 2`
- `:-` as prefix is a comand used for declarations
- Operators have precedence
- You can easily define your own operators

Getting help

- Built-in predicate `help`:[2]

```
?- help(reverse).
reverse(?List1, ?List2)
    Is true when the elements of List2 are
    in reverse order compared to List1.
```

- Notation here:

`+Arg`	Means `Arg` should be instantiated (input)
`-Arg`	Means `Arg` can be a new variable; will be unified with the result (output)
`?Arg`	Means `Arg` can be either input or output

- `https://www.swi-prolog.org/pldoc/doc_for?object=manual`

[2] On Linux, need to install docs as a separate package, e.g. `pl-doc` on Fedora

Getting help

- Built-in predicate `apropos`:

```
?- help(comparison).
Warning: No help for comparison.
Warning: Use ?- apropos(query). to search for candidates.

?- apropos(comparison).
% SEC 'cpp-plterm-comparison'                Comparison
% SEC 'foreign-compare'                      Term Comparison
% SEC collate                                Language-specific compa
% SEC 'cql-compare-null'                     Comparisons with NULL
% SEC 'cql-where-arith'                      WHERE with arithmetic c
% SEC unifyspecial                           Special unification and
% SEC compare                                Comparison and Unificat
% SWI collation_key/2                        Create a Key from Atom
% LIB rdf_compare/3                          True if the RDF terms L
true.
```

Some examples

`append/3` is an example of a reversible if **steadfast** predicate

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

?- append([a,b],X,Y).
Y = [a, b|X].

?- append(X,Y,Z).
X = [],
Y = Z ;
X = [_111616],
Z = [_111616|Y] ;
X = [_111616, _112730],
Z = [_111616, _112730|Y] .
```

Reversible Programming

- Good predicates are steadfast
- They work correctly even if unusual values are supplied — e.g., variables for inputs, constants for outputs
- Non-steadfast predicates require specific arguments to be instantiated

Unification

- Prolog matches terms by **unifying** them. Specifically, it apples *the most general unifier*
- Instantiates variables as little as possible to make them match

```
?- X = f(Y,b,Z), X = f(a,V,W).
X = f(a, b, W),
Y = a,
Z = W,
V = b.
```

Family Relations Example

- The program:

```
parent(Parent, Child) :- mother(Parent, Child).
parent(Parent, Child) :- father(Parent, Child).
father('George', 'Elizabeth').
father('George', 'Margaret').
mother('Mary', 'Elizabeth').
mother('Mary', 'Margaret').
```

- Observe: implicitly, there is disjunction.

Using ";" to get all solutions

```
?- parent(P, C).
P = 'Mary',
C = 'Elizabeth' ;

P = 'Mary',
C = 'Margaret' ;

P = 'George',
C = 'Elizabeth' ;

P = 'George',
C = 'Margaret'.
```

How Prolog finds solutions

Use predicate `trace/0` to enable derivation info in interactive mode (useful for debugging). Use `notrace/0` to turn off.

```
[debug]  ?- trace.
true.

[trace]  ?- parent(Parent, Child1), parent(Parent, Child2),
            not(Child1 = Child2).
   Call: (11) parent(_35824, _35826) ? creep
   Call: (12) mother(_35824, _35826) ? creep
   Exit: (12) mother('Mary', 'Elizabeth') ? creep
   Exit: (11) parent('Mary', 'Elizabeth') ? creep
   Call: (11) parent('Mary', _35832) ? creep
   Call: (12) mother('Mary', _35832) ? creep
   Exit: (12) mother('Mary', 'Elizabeth') ? creep
   Exit: (11) parent('Mary', 'Elizabeth') ? creep
^  Call: (11) not('Elizabeth'='Elizabeth') ? creep
     ...
```

How Prolog finds solutions (cont.)

```
     ...
^  Fail: (11) not(user:('Elizabeth'='Elizabeth')) ? creep
   Redo: (12) mother('Mary', _35832) ? creep
   Exit: (12) mother('Mary', 'Margaret') ? creep
   Exit: (11) parent('Mary', 'Margaret') ? creep
^  Call: (11) not('Elizabeth'='Margaret') ? creep
^  Exit: (11) not(user:('Elizabeth'='Margaret')) ? creep
Parent = 'Mary',
Child1 = 'Elizabeth',
Child2 = 'Margaret' .
```

The Query-Answering Process

- A **query** is a conjunction of terms
- If all terms succeed, the answer to query is `Yes`
- A term in a query **succeeds** if
  - it matches a fact in the database
  - it matches the head of a rule whose body succeeds
- The substitution used to unify the term and the fact/head is applied to the rest of the query
- Query terms are processed in the **left-to-right order**
- Database facts/rules are tried in **top-to-bottom order**

Generating permutations

- Intuitively: a permutation is a rearrangement
- Recursive thinking: a permutation `P` of a list `L` is a list
  - whose first element is some arbitrary element `E` from `L` and
  - whose remainder is a permutation of `L` with `E` removed
- Special/base case: `[]` is a permutation of `[]`
- Program:

```
permutation([],[]).
permutation(L, [E|Tail]) :- select(E,L,Rest),
                            permutation(Rest,Tail).
```

Selecting an Element From a List

To select an element from a list, we can either

- select the first leaving the rest, or
- select some element from the rest and leave the first plus the unselected elements from the rest

```
select(X,[X|R],R).
select(X,[Y|R],[Y|RS]):- select(X,R,RS).
```

Sorting

Find a permutation that is ordered:

```
mysort(L,P):- permutation(L,P), ordered(P).

ordered([]).
ordered([_]).
ordered([E1,E2|R]) :- E1 =< E2, ordered([E2|R]).
```

```
?- mysort([8,3,5,6,3,3,6,1],X).
X = [1, 3, 3, 3, 5, 6, 6, 8] ;
```

This is an example of the *generate-and-test* pattern

Reverse

`reverse(List, ReversedList)` holds if `ReversedList` is a list with the components of `List` in the reverse order

A recursive implementation:

```
reverse([],[]).
reverse([F|R],RL):- reverse(R,RR), append(RR, [F], RL).

append([],L,L).
append([F|R],L,[F|RL]):- append(R,L,RL).
```

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

Reverse with tail recursion

- **Tail recursion**: save the recursive call till the end to avoid flooding the call stack
- A tail-recursive definition of `reverse/2`:

```
reverse(L,RL):- reverse(L,[],RL).  /* Alias */

reverse([], Acc, Acc).         /* Tertiary! */
reverse([F|R],Acc,RL) :- reverse(R,[F|Acc],RL).
```

Solving a Logical Puzzle with Prolog

The Zebra Puzzle

> *There are five houses, occupied by five gentlemen of five different nationalities, who all have different coloured houses, keep different pets, drink different drinks, and smoke different brands of cigarettes.*
> *The Englishman lives in a red house.*
> *The Spaniard keeps a dog.*
> *The owner of the green house drinks coffee.*
> *...*
> *The ivory house is just to the left of the green house.*
> *...*
> *The Chesterfields smoker lives next to a house with a fox.*

Who owns the zebra? Who drinks water?

Zebra in Prolog

- Represent the 5 houses by a structure of 5 terms `house(Colour, Nationality, Pet, Drink, CigBrand)`
- Create a partial structure using variables, to be instantiated in the process of solving
- Specify constraints to instantiate variables

Zebra: Building Houses

- Let's build the (incomplete) houses:

```
makehouses(0,[]).

makehouses(N,[house(Col, Nat, Pet, Drk, Cig)|List]) :-
                  N>0, N1 is N - 1, makehouses(N1,List).
```

- Or, more cleanly, using anonymous variables:

```
makehouses(N,[house(_, _, _, _, _)|List]) :-
                  N>0, N1 is N - 1, makehouses(N1,List).
```

The Empty Houses

```
?- makehouses(5, Houses).
Houses = [house(_10398, _10400, _10402, _10404, _10406),
          house(_10416, _10418, _10420, _10422, _10424),
          house(_10434, _10436, _10438, _10440, _10442),
          house(_10452, _10454, _10456, _10458, _10460),
          house(_10470, _10472, _10474, _10476, _10478)]
```

Constraints

- The Englishman lives in a red house[3]
  ```
  house(red, englishman, _, _, _) on Houses,
  ```
- The Spaniard keeps a dog
  ```
  house(_, spaniard, dog, _, _) on Houses,
  ```
- The owner of the green house drinks coffee
  ```
  house(green, _, _, coffee, _) on Houses,
  ```
- The ivory house is just to the left of the green house
  ```
  sublist2([house(ivory, _, _, _, _),
            house(green, _, _, _, _)], Houses),
  ```
- The smoker of Chesterfields lives next to a house with a fox
  ```
  nextto([house(_, _, _, _, chesterfields),
            house(_, _, fox, _, _)], Houses),
  ```

[3] Wait till next slide regarding `on`

Defining an Operator

- `on` is a user-defined infix operator that is a version of `member/2`.
- Definition:

```
:- op(100, zfy, on).
X on List :- member(X, List).
```

- This amounts to

```
X on [X|_].
X on [_|R] :- X on R.
```

Helper Predicates Used Above

- "just left of", "lives next to"?
- Define `sublist2/2` as

```
sublist2([S1, S2], [S1, S2 | _]) .
sublist2(S, [_ | T]) :- sublist2(S, T).
```

- Define `nextto/3` as

```
nextto(H1, H2, L) :- sublist2([H1, H2], L).
nextto(H1, H2 ,L) :- sublist2([H2, H1], L).
```

Finding the Zebra

- Who owns the zebra and who drinks water?

```
find(ZebraOwner, WaterDrinker) :-
    makehouses(5, Houses),
    house(red, englishman, _, _, _)   on Houses,
    ... /* all other constraints here */
    house( _, WaterDrinker, _, water, _)  on Houses,
    house( _, ZebraOwner, zebra, _, _)  on Houses.
```

- The solution is generated and queried in the same clause
- Neither water nor zebra are mentioned in the constraints

Solving the Puzzle

```
?- [zebra].
true.

?- find(ZebraOwner, WaterDrinker).
ZebraOwner = japanese,
WaterDrinker = norwegian ;
false.
```

How Prolog Finds the Solution

After 8 constraints we have:

```
Houses = [
    house(red, englishman, snail, _G251, old_gold),
    house(green, spaniard, dog, coffee, _G264),
    house(ivory, ukrainian, _G274, tea, _G276),
    house(green, _G285, _G286, _G287, _G288),
    house(yellow, _G297, _G298, _G299, kools)]
```

The next constraint is "the owner of the third house drinks milk", which can't be done with the current instantiation of `Houses`. Prolog will **backtrack** to latest point of choice and try another.

Full Solution

The complete solution is unique:

```
Houses = [
    house(yellow, norwegian, fox, water, kools),
    house(blue, ukrainian, horse, tea, chesterfields),
    house(red, englishman, snail, milk, old_gold),
    house(ivory, spaniard, dog, orange, lucky_strike),
    house(green, japanese, zebra, coffee, parliaments)]
```

End of lecture

Next time: More Formal Logic (Inference in FOL)