Logic Programming and Prolog [part 1]

Daniel S. Fava

In part based on slides from Gerardo Schneider, which where in turn based on John C. Mitchell's

Prolog

Declarative programming language

Program: relations represented as facts and rules.

```
cat(tom).
animal(X) :- cat(X).
```

Computation: a query over these relations.

```
?- animal(tom).
yes
```

Leads to a form of *interactive programming*

Example

Facts and rules in a *database file*, often with `.pl` extension

`example001.pl`

```
cat(tom).
animal(X) :- cat(X).
```

On the prolog prompt: import databases then make queries

```
?- [example001].
...
?- animal(tom).
yes
```

History

1930s, Gödel, Herbrand: Computation as deduction

- *Computation:* Computing `f(x)` and obtaining `c`
- *Deduction:* Proving that the equation `f(x)=c` holds

1965, Robinson: Resolution and unification

- (un)satisfiability of predicate and first-order logic

1970s, Kowalski: Logic programs with a restricted form of resolution

Applications of logic programming

- Theorem proving
- Expert systems (eg. IBM's Watson)
- Term rewriting
- Type systems
- Automated planning
  - For example: strategies for autonomous robots
- Natural language processing

Basics

Clauses

- facts, rules

Data types

- atoms, numbers, variables, compound terms

Impure predicates

- predicates with side effects
- for example: printing a value to the screen

Clauses

Facts: clauses with empty bodies

```
cat(tom).
```

```
cat(tom) :- true.
```

Rules: head is true if body is true

```
Head :- Body.
```

```
animal(X) :- cat(X).
```

A rule's body consists of calls to predicates, which are called the rule's goals.

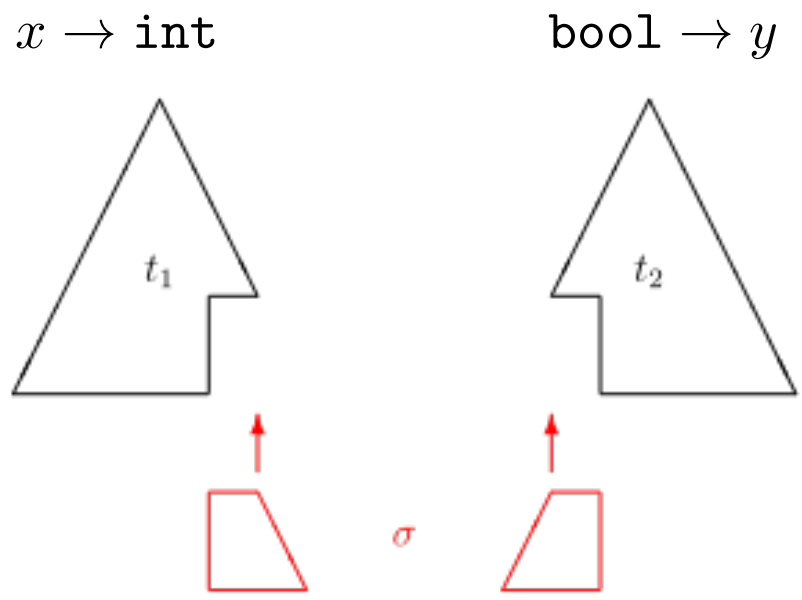
Queries

Variables: start with Upper case letter or _

```
?- cat(X).
X = tom
yes
```

Unification [ML lecture 3]

We have seen unification in the context of type inference.



$\sigma(t_1) = x \mapsto \texttt{bool}$ $\sigma(t_2) = y \mapsto \texttt{int}$

Unification, Prolog

Unification for solving queries

- instantiate a query's variables to match facts/rules

Example

```
fact:          child(anne,sofia)
query:         child(X,sofia)
unification:   X:= anne .
```

Note: Syntactic equality. Meaning, 1+2 and 3 do not unify

Composite queries

Comma is the logical *and*

```
siblings(X,Y) :- child(X,Z), child(Y,Z), X \== Y.
```

Semicolon is the logical *or*

```
eitherMember(X,Y,L) :- member(X,L); member(Y,L).
```

```
eitherMember(a,b,[a,b,c,d]).
eitherMember(a,b,[  b,c,d]).
eitherMember(a,b,[a  ,c,d]).
eitherMember(a,b,[    c,d]).
```

Relations vs. Functions

In the previous example, we used a relation to represent "`X` is a child of `Y`":

```
child(X,Y)
```

as opposed to a function from parent to child:

```
child(Y) = X
```

Note: There are no functions in Prolog! Only relations.

A function can be turned into a relation:

$$f(a) = b \text{ becomes } f'(a, b)$$

Recursive rules

```
descendant(X,Y) :- child(X,Y).
descendant(X,Y) :- child(X,Z), descendant(Z,Y).
```

Note the order of rule definitions:

- Non-recursive rule first
- Recursive goal after

Example:

```
?- descendant(anne, X).     % All ancestors
?- descendant(X, sofia).    % All descendants
```

Lists

- Unification
- Membership
- Append
  - append to split a list

Unification on lists

Question. If/How do the following terms unify?

```
[a,b,c] ?= [Head | Tail]
```

```
[a]     ?= [H | T]
```

```
[a,b,c] ?= [a | T]
```

```
[a,b,c] ?= [b | T]
```

```
[]      ?= [H|T]
```

```
[]      ?= []
```

Unification on lists

Assume the following fact:

```
p([H | T], H, T).
```

Compute:

```
?- p([a,b,c], X, Y).
```

```
?- p([a], X, Y).
```

```
?- p([], X, Y).
```

List membership

```
member(X, [X|Rest]).
member(X, [H | Tail]) :- member(X, Tail).
```

```
member(2,[1,2,3]) ? -> member(2,[2,3]) ? -> yes
```

Appending two lists

```
?- append([1,2,3],[4,5,6],Xs).
Xs = [1,2,3,4,5,6]
yes
```

Can use `append` to split a list in all possible ways:

```
?- append(Xs, Ys, [first, second, third, fourth]).
Action (; for next solution,
          a for all solutions,
          RET to stop) ?
```