# Logic Programming and Prolog [part 2]

Daniel S. Fava

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

## Prolog, recap

words starting with lower-case letters

- Constants: “anne”, “sofia”
- Relations: “person”

words starting with upper-case letter or _

- Variables

# Prolog

*Closed world assumption:* not known to be true means false

For example:

- If `cat(tom)` is not in the database,
- then the query `?-cat(tom)` evaluates to false

*Negation as failure:* false if cannot prove true

For example:

```
legal(X) :- \+ illegal(X).
```

- Attempt to prove `illegal(X)`,
- if proof can be found, then `legal(X)` fails
- if proof cannot be found, then `legal(X)` succeeds

# Prolog, some operators

`A = B` are `A` and `B` unifiable?

```
?- 1 = 1.
yes
?- 2 = 1+1.
no
?- X = 1.
X=1
yes
```

# Prolog, some operators

`A == B` are `A` and `B` syntactically equal?

```
?- 1 == 1.
yes
?- 2 == 1+1.
no
?- X == 1.
no
```

## Prolog, some operators

`A =:= B` are `A` and `B`'s values equal (after computation)?

```
?- 1 =:= 1.
yes
?- 2 =:= 1+1.
yes
?- X =:= 1.
uncaught exception: error(instantiation_error,(=:=)/2)
```

**Question.** How to say that `X` is the result of `3+1`?

`=` Answers `yes` but does not evaluate `3+1`

```
?- X = 3+1.
X = 3+1
yes
```

`==` Answers `no`

```
?- X == 3+1.
no
```

`=:=` Gives out an error

```
?= X =:= 3+1.
uncaught exception: error(instantiation_error,(=:=)/2)
```

**Question.** How to say that `X` is the result of `3+1`?

Use builtin predicate `is`

```
?- X is 3+1.
X = 4
yes
```

Example: factorial

```
factorial(0,1).
factorial(N,F) :- N>0, N1 is N-1,
                  factorial(N1,F1),
                  F is N*F1.
```

Queries

```
?- factorial(5,X).
X = 120
Yes

?- factorial(X,120).
uncaught exception: error(instantiation_error,(>)/2)
```

Example: ordered

```
ordered([]).
ordered([X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

Queries

```
?- ordered([3,4,67,8]).
no
?- ordered([3,4,67, 88]).
yes
?- ordered([3,4,X,88]).
uncaught exception: error(instantiation_error,(=<)/2)
```

## Prolog & arithmetic

Issues

- Operations of arithmetic are functional, not relational
- Arithmetic compromises Prolog's "declarativeness"

```
?- factorial(X,120).
uncaught exception: error(instantiation_error,(>)/2)

?- ordered([3,4,X,88]).
uncaught exception: error(instantiation_error,(=<)/2)
```

## Prolog & efficiency: Cut

Cut is a control flow abstraction

It is a goal that always succeeds and cannot be backtracked

Added for efficiency reasons; for example,
to prevent finding more solutions

```
head :- body.      % Finds all solutions
```

```
head :- body,!.    % Finds one solution
```

In general, no backtracking of B once it succeeds

```
H :- A, B, !, C, D
```

## Prolog & efficiency: Cut

Issues

- programs become harder to understand
- easy to introduce mistakes
- “destroys declarativeness”

# Prolog & I/O

Various predicates for input/output

```
?- print(f(a)).                % print term
?- display('Hello World').     % print string
```

# Prolog & I/O

Issue: Does not work well with backtracking

```
io_problem1 :- print(one), fail.
io_problem1 :- print(two).
```

```
?- io_problem1.
onetwo
```

```
io_problem2 :- fail, print(one).
io_problem2 :- print(two).
```

```
?- io_problem1.
two
```

Wait... conjunction should be commutative!

## Other logic programming languages / paradigms

- Mercury
- Curry
- Constraint Logic Programming
- Answer Set Programming
- Datalog, Overlog, Dedalus, and BLOOM
- Maude and rewriting logic

**Mercury**

To address issues in Prolog

Mercury is

- Compiled
- Strict typed
- Has a module system
- Disallows cut
- Has clean integration of IO
- Includes functional features

**Curry**

- Research language
- Functional / logic programming language
- Based on Haskell

lazy functional programming: demand-driven evaluation

\+ logic programming: non-deterministic operations

= more efficient search strategies

## Constraint Logic Programming

Logic programming + constraints in the body of clauses

```
A(X,Y) :- X+Y>0, B(X), C(Y)
```

Logic programming

- Interpreter starts from the goal and
  recursively scans the clauses trying to prove the goal

Constraint Logic Programming

- Constraints encountered when trying to prove a goal
  are placed in a set
- Constraint solver is called
- If constraints are unsatisfiable:
  interpreter backtracks, tries other clauses

Example applications:

- Civil and mechanical engineering
- Digital circuit verification
- Air traffic control

## Answer Set Programming

Declarative programming

Prolog-style query evaluation

Application:

- solving NP-hard search problems
  - worst case exponential time,
    no known polynomial time algorithm

## Datalog

- Subset of Prolog
- Not Turing complete
- Used as a query language for deductive databases
- Derivatives: Overlog, Dedalus, Bloom, etc

## Overlog

- Originally for *declarative networking*,
  then used to prototype distributed systems.
  For example: Berkeley Orders of Magnitude (BOOM)
  - Reimplementation of HDFS and MapReduce
  - Hadoop scheduler in ~10x fewer lines of code

Incorporate lessons from BOOM project

Fix pain points from Overlog

**Dedalus**

- Dedalus = Datalog + notion of time
- For reasoning, but not necessarily programming

**BLOOM**

- Based on Dedalus
- For distributed and cloud programming

## Rewriting logic and Maude

Prolog database

```
Head(H) :- Body(H).
```

*Rewriting system:* Set of rewriting rules.

```
Body1(H) -> Head1
Body2(H) -> Head2
...
```

Rewrite left- into right-hand-side. For example:

```
2 * 2 -> 4
```

```
pop(push(E,S)) -> S
```

Unify a term with the left-hand-side of a rewriting rule

```
pop(push('tomato', empty))    // term
pop(push(E,S)) -> S           // rule
```

The term above unifies with LHS of the rule
with `E` mapping to `'tomato'` and `S` to `empty`

Therefore, rewrite step:

```
pop(push('tomato', empty)) -> empty
```

More than one match possible. Can ask questions like:

*Is value* $v$ *a possible result from executing program* `P`*?*

Application:

- model execution of a program; support for verification