What happened so far

- We learned how Prolog executes derivations
- Simple predicates like transitive closure can lead to non-termination
- Thinking about the set of answers / number of substitutions may explain non-termination
- Sometimes reordering goals helps
- Sometimes we reformulate the problem
Overview

1. Arithmetic

2. Declarative Arithmetic (Finite Domain Constraints)

3. Meta-logical Predicates

4. Non-logical Predicates
Outline

1. Arithmetic

2. Declarative Arithmetic (Finite Domain Constraints)

3. Meta-logical Predicates

4. Non-logical Predicates
Unification is not computation:

?- X = 1+(2*3).
X = 1+(2*3).

is/2 relates ground arithmetic expressions to evaluation:

?- X is 1+(2*3).
X = 7.

Other predicates: <, >, <=, >=, =:=
Problem:

?- 7 is 1+(X*Y).
ERROR: is/2: Arguments are **not** sufficiently instantiated

?- X < 10.
ERROR: </2: Arguments are **not** sufficiently instantiated

Computation must be possible when goal is encountered

Declarative properties (commutativity of goals) are lost:

?- X < 10, X=1.
ERROR: </2: Arguments are **not** sufficiently instantiated

?- X=1, X < 10.
X = 1.
Outline

1. Arithmetic

2. Declarative Arithmetic (Finite Domain Constraints)

3. Meta-logical Predicates

4. Non-logical Predicates
Reasoning with constraints over integer domain:

?- 7 #= 1+(X*Y).
X in -6.. -1\1..6,
X*Y#=6,
Y in -6.. -1\1..6.

“The equation holds under the conditions:
X \in \{-6\ldots -1\} \cup \{1\ldots 6\}
Y \in \{-6\ldots -1\} \cup \{1\ldots 6\}
X \times Y = 6”
Finite Domain Constraints – CLP(FD)

- Concrete substitutions require enumeration of variables:

```prolog
?- 7 #= 1+(X*Y), labeling([], [X,Y]).
X = -6,
Y = -1 ;
% ...
```
Finite Domain Constraints – CLP(FD)

- Arithmetic Relations: \#=, \#\= \#<,#>, \#=<, \#>=
- Domain Relations:
  - Var in Lower .. Upper: \( X \in \{Lower \ldots Upper\} \)
    Lower, Upper must be numbers or inf / sup
  - [A,B,C] ins Lower .. Upper: like in but for lists of variables
- Labeling: label/1 expects a list of variables to label
Problem

Given a 3x3 square of fields, assign each of the numbers 1 to 9 to the fields such that the sums of each row, the sums of each column and the sums of the diagonals amount to the same value.
Example: Magic Squares

:- use_module(library(clpfd)).

rows_sum([A1,A2,A3,B1,B2,B3,C1,C2,C3], Sum):-
    A1+A2+A3 #= Sum,
    B1+B2+B3 #= Sum,
    C1+C2+C3 #= Sum.

cols_sum([A1,A2,A3,B1,B2,B3,C1,C2,C3], Sum):-
    A1+B1+C1 #= Sum,
    A2+B2+C2 #= Sum,
    A3+B3+C3 #= Sum.
Example: Magic Squares

diag_sum([A1, _A2, A3, _B1, B2, _B3, C1, _C2, C3], Sum):-
    A1 + B2 + C3 #= Sum,
    A3 + B2 + C1 #= Sum.

magicsquare(Sum, [A1, A2, A3], [B1, B2, B3], [C1, C2, C3]) :-
    % define domain variables
    Zs = [A1, A2, A3, B1, B2, B3, C1, C2, C3],
    Zs ins 1..9,
    % core predicates
    all_distinct(Zs),
    rows_sum(Zs, Sum),
    cols_sum(Zs, Sum),
    diag_sum(Zs, Sum),
    % labeling
    label([Sum | Zs]).
The structure of a CLP(FD) program

- Define a list Zs of finite domain variables to label
- Set the domain for the variables
- Add constraints on core predicates
  - *Core predicates do not label on their own!*
- Finally: Label Zs
Why label only in the end?

- CLP(FD) maintains a set of constraints
- Adding new constraints allows constraint propagation:

```prolog
?- X in 2..6, X #> 3.
X in 4..6.
```

- Labeling grounds the constraint, barely any propagation.
  Compare

```prolog
?- time((X in 1..1000, label([X]), X #> 950, false)).
% 68,119 inferences, 0.006 CPU in 0.006 seconds (99% CPU, 11166330 Lips) false.
```

  to

```prolog
?- time((X in 1..1000, X #> 950, label([X]), false)).
% 3,527 inferences, 0.001 CPU in 0.001 seconds (94% CPU, 4958534 Lips) false.
```
Why label at all?

- Constraints are not guaranteed to be satisfiable
- Only labeling guarantees their satisfiability
- Example:

```
?- X #< Y, Y #< X.
Y#=<X+ -1,
X#=<Y+ -1.
```

but there are not X,Y s.t. $X < Y \land Y < X$. 

Martin Riener (Manchester) Lecture 6 Prolog Programming Techniques February 2019 11 / 27
Labeling strategies

- `labeling/2`:
  Like `label/1` but first argument has list of options
  - Variable selection:
    leftmost (order of appearance), ff (order by domain size), ffc (ff, prefer by number of occurrences), min (order by smallest lower bound), max (order by largest upper bound),
  - Value order
    up (ascending order), down (descending order)
  - Branching strategy:
    step (distinguish equal / different from picked value), enum (distinguish all possible values at the same time), bisect (divide search space along middle point)
Improving performance

- Try different labeling strategies
- Reduce the number of solutions!
Improving performance

- Try different labeling strategies
- Reduce the number of solutions!
  Magic squares example:
Improving performance

- Try different labeling strategies
- Reduce the number of solutions!

Magic squares example:
  - Calculate $\text{Sum}$:
    - Formula for $n \times n$: $\text{Sum} = \frac{n^3+n}{2}$
    - 15 for $n=3$
Improving performance

- Try different labeling strategies
- Reduce the number of solutions!

Magic squares example:

- Calculate $Sum$:
  - Formula for $n \times n$: $Sum = \frac{n^3+n}{2}$
  - $15$ for $n=3$

```
?- time((magicsquare(N, R1, R2, R3),false)).
% 1,523,920 inferences, 0.151 CPU in 0.152 seconds
false.

?- time((magicsquare(15, R1, R2, R3),false)).
% 341,873 inferences, 0.049 CPU in 0.050 seconds
false.
```
Improving performance

- Try different labeling strategies
- Reduce the number of solutions!

Magic squares example:
  - Add symmetry breaking constraints:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>original</th>
<th>horizontally flipped</th>
<th>A1 &lt; C1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>vertically flipped</th>
<th>diagonally flipped</th>
<th>B1 &lt; A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Add symmetry breaking constraints:
Improving performance

- Try different labeling strategies
- Reduce the number of solutions!

Magic squares example:
- Add symmetry breaking constraints:

```prolog
symmetries([A1,A2,A3,B1,_B2,_B3,C1,_C2,_C3]) :-
    A1 #< A3,
    A1 #< C1,
    A2 #< B1.
```
Improving performance

- Try different labeling strategies
- Reduce the number of solutions!

Magic squares example:

- Add symmetry breaking constraints:

```prolog
symm_magicsquare(N, [A1,A2,A3],[B1,B2,B3],[C1,C2,C3]) :-
    % ...
    symmetries(Zs),
    % ...
```
Improving performance

- Try different labeling strategies
- Reduce the number of solutions!

Magic squares example:
- Add symmetry breaking constraints:

```prolog
?- time((magicsquare(15, R1, R2, R3),false)).
% 341,873 inferences, 0.032 CPU in 0.032 seconds (100% CPU, 10587328 Lips) false.

?- time((symm_magicsquare(15, R1, R2, R3),false)).
% 96,382 inferences, 0.010 CPU in 0.010 seconds (100% CPU, 9494353 Lips) false.
```
Improving performance

- Try different labeling strategies
- Reduce the number of solutions!

Lesson

A little thinking makes the program $\geq 10\times$ faster!
Hic sunt dragones!
You need to be aware of non-logical predicates but you need **not** be skilled in their use.
Non-logical Predicates

Some predicates usually destroy the declarative properties of Prolog

- Cut
- Negation-as-failure
- If-then-else
- Input / Output
Non-logical Predicates

Some predicates usually destroy the declarative properties of Prolog

- Cut
- Negation-as-failure
- If-then-else
- Input / Output

...but you will encounter them in practice.

This lecture will only explain them and how to avoid them. If you want to learn about them properly, read R. O’Keefe: *The Craft of Prolog*. 
Outline

1. Arithmetic

2. Declarative Arithmetic (Finite Domain Constraints)

3. Meta-logical Predicates

4. Non-logical Predicates
Meta-logical Predicates

- Meta-logical predicates go beyond the expressitivity of FOL:
  - Using terms as predicates
  - Querying if a term is a variable / atom / ground etc.
  - Generating the list of all solutions of a predicate

- Meta-logical predicates may destroy the declarative meaning of a program
Meta-Calls

- We can create terms programmatically with \( =../2 \):

  ```prolog
  ?- P =.. [isa_list, X].
P = isa_list(X).
  ```

- We can use such a term as a goal:

  ```prolog
  ?- P =.. [isa_list, X], call(P).
P = isa_list([]),
X = [] ;
P = isa_list([_4302]),
X = [_4302] ;
% ... just like calling ?- isa_list(X). directly.
  ```
Typechecks

- `var/1`: true if argument is a variable
- `nonvar/1`: true if argument is not a variable
- `atom/1`: true if argument is a constant (no variable, no function)
- `ground/1`: true if argument does not contain variables
- `==/2`: true if arguments are identical (no unification!)
- `
==/2`: true if terms are not identical (no unification!)
- `\=/2`: true if no substitution makes the terms equal
What are they useful for?

- Program transformation of Prolog programs,
- Writing your own unification predicate
- Writing a Prolog interpreter in Prolog
Typechecks

- What are they useful for?
  - Program transformation of Prolog programs,
  - Writing your own unification predicate
  - Writing a Prolog interpreter in Prolog

- What makes them problematic? – They destroy commutativity of conjunction!

?- var(X), X = something.
X = something.

?- X = something, var(X).
false.
Aggregating all solutions of a predicate

- when setof(Pattern, Goal, List) succeeds, List contains each answer substitution to Goal applied to Pattern
  Variables in Pattern and Goal must not occur elsewhere!
Aggregating all solutions of a predicate

- when setof(Pattern, Goal, List) succeeds, List contains each answer substitution to Goal applied to Pattern Variables in Pattern and Goal must not occur elsewhere!

- Example: create a list of all elements of \{1, 2, 3\} \times \{2, 3, 5\}

```prolog
?- setof(X-Y, ( member_of(X, [1,2,1,3]), member_of(Y, [3,2,5])), Xs).
Xs = [1-2, 1-3, 1-5, 2-2, 2-3, 2-5, 3-2, 3-3, ... - ...].
```
Variables that do not occur in the pattern lead to backtracking:

?- setof(X, ( member_of(X, [1,2,1,3]), member_of(Y, [3,2,5])), Xs).
Y = 2,
Xs = [1, 2, 3] ;
Y = 3,
Xs = [1, 2, 3] ;
Y = 5,
Xs = [1, 2, 3].
If we want to ignore the value of Y, we have to add an existential quantifier Goal:

?- setof(X, Y ^ ( member_of(X, [1,2,1,3]), member_of(Y, [3,2,5])), Xs).
Xs = [1, 2, 3].
Aggregating all solutions of a predicate

- Only useful for terminating predicates!

?- setof(X, isa_list(X), Xs).
% does not terminate, exhausts memory really fast
The cut operator ! cuts off derivation branches
The cut operator ! cuts off derivation branches

Example:

```prolog
nondet(a,c).
nondet(b,d).

nocut(X) :-
    nondet(X, _).
nocut(X) :-
    nondet(_, X).
```
The cut operator ! cuts off derivation branches

Example:

```prolog
nondet(a,c).
nondet(b,d).

% extract first argument of nondet/2
withcut(Y) :-
  !, % never backtrack past this point
  nondet(Y, _).

% extract second argument of nondet/2
withcut(Y) :-
  !, % never backtrack past this point
  nondet(_, Y).
```
The cut operator ! cuts off derivation branches

Example:

```prolog
nondet(a,c).
nondet(b,d).

% extract second argument of nondet/2
withcut2(Y) :-
    !, % never backtrack past this point
    nondet(_, Y).
% extract first argument of nondet/2
withcut2(Y) :-
    !, % never backtrack past this point
    nondet(Y, _).
```
The cut operator cuts off derivation branches

Comparison:

?- nocut(X).
X = a ;
X = b ;
X = c ;
X = d.

?- withcut(X).
X = a ;
X = b.

?- withcut2(X).
X = c ;
X = d.
The cut operator ! cuts off derivation branches
	nocut/1 has not the same solution set as withcut/1 and withcut2/1

The order of rules changed the set of solutions between withcut/1 and withcut2/1!

Red cuts change the solutions of a program,
Green cuts prune derivations but keep the solution set intact,
Blue cuts are green cuts that the compiler should optimize automatically
Why use it then?
Why use it then?
  - Speed up a program

  Writing green cuts is difficult – correctness over speed!

- Needed to implement negation-as-failure and if-then-else
Why cuts are problematic

Problem

Implement a predicate max/3 such that the third argument is equivalent to the maximum of the first two arguments.
Why cuts are problematic

Solution without cuts:

```prolog
max(X,Y,X) :-
    X >= Y.
max(X,Y,Y) :-
    X < Y.
```
Solution with a blue cut:

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

The two branches are mutually exclusive
Why cuts are problematic

Temptation: Let’s remove the second guard!
If \( X \geq Y \) then \( X < Y \) must hold, after all...

```
max_red(X,Y,X) :-
    X >= Y,
    !.
max_red(X,Y,Y).
```
Why cuts are problematic

max_red(X,Y,X) :-
    X >= Y,
    !.
max_red(X,Y,Y).

?- max_red(9,0,0).
true.

this is not proper mathematics...
max_red(X,Y,X) :-
    X >= Y,
    !.
max_red(X,Y,Y).

?- max_red(9,0,0).
true.

this is not proper mathematics... what happened?

- max_red(9,0,0) and rule head max_red(X,Y,X) are not unifiable!
- Prolog immediately tries the second rule but we deleted the guard
Why cuts are problematic

max_red(X,Y,X) :-
    X >= Y,
    !.
max_red(X,Y,Y).

?- max_red(9,0,0).
true.

this is not proper mathematics... what happened?

- max_red(9,0,0) and rule head max_red(X,Y,X) are not unifiable!
- Prolog immediately tries the second rule but we deleted the guard
- This particular predicate can be fixed by making it steadfast – see chapter 3.11 in *The Craft of Prolog*.
Why cuts are problematic

Lesson 1: Using cuts for efficiency is error prone
As long as we can achieve magnitudes of speedups by cleverly restating the problem, why use cuts?

Lesson 2: Cut is rarely necessary
In most cases, we can get by without cut. In the rare cases we need it, there are slightly safer predicates.
Negation as failure

- Inference rule: if we can not derive \( \text{pred} \) then conclude \( \neg \text{pred} \).

- Implementation (uses cut):

\[
\text{\texttt{\+\( (\text{Goal}) \)} :-}
\]
\[
\text{\texttt{\quad \texttt{call(\text{Goal}), } \% \text{ call to \text{Goal} \}}}
\]
\[
\text{\texttt{\quad \texttt{!}. \quad \% \text{ we have derived \text{Goal}, cut the other branch}}}
\]
\[
\text{\texttt{\quad \texttt{false. \quad \% \ldots \text{ and fail}}}}
\]
\[
\text{\texttt{\texttt{\+\( (\_\text{Goal}) \)} :- \}}
\]
\[
\text{\texttt{\quad \texttt{true. \quad \% \text{ we could not derive \text{Goal}, succeed}}}}
\]
Consider the following program:

continent(antarctica).
continent(america).
continent(asia).
continent(australia).
continent(europe).

land(X) :- % if X is a continent, X is on land
    continent(X).

water(X) :- % if X is not a continent, X is in the ocean
    \+ land(X).
Negation as failure and the closed world assumption

?- land(X).
X = antarctica ;
X = america ;
X = asia ;
X = australia ;
X = europe.

so far, so good!
Negation as failure and the closed world assumption

?- water(pacific_ocean).
thue.

?- water(saturn).
thue.

?- water(minnie_mouse).
thue.

it’s getting stranger... but that’s due to the closed world assumption
?- water(X).
false.

...There is no water?
?- water(X).
false.

... There is no water?

- According to our definition, whenever \text{land}(X) succeeds, \text{water}(X) fails.
  This is not classical logic! In FOL we have \( p(t) \rightarrow \exists x \ p(x) \)!
Negation by failure coincides with FOL if
- the query is ground
- the negated goal terminates

Everything else is tricky
The built-in predicates are fast but only compute
Constraint logic programming over finite domains provides declarative integer arithmetic
CLP(FD) predicates...
  - assign variables to domain
  - compose constraints with core predicates
  - need to label the variables to find the solutions
Non-logical predicates...
  - destroy the declarative reading of Prolog
  - are useful for special cases (negation-as-failure, if-then-else, meta-programming)
  - should only be used when absolutely necessary
That's all for today!