

Semantic and Declarative Technologies

László Kabódi, Péter Tóth, Péter Szeredi

kabodil@gmail.com
peter@toth.dev
szeredi@cs.bme.hu

Aquincum Institute of Technology

Budapest University of Technology and Economics
Department of Computer Science and Information Theory

2025 Autumn Semester

Course information

- Course layout
 - Introduction to Logic Weeks 1–2
 - Declarative Programming
 - Prolog – Programming in Logic Weeks 3–7
 - Constraint Programming Weeks 8–12
 - Semantic Technologies
 - Logics for the Semantic Web Weeks 13–14
- Requirements
 - 2 assignments (150 points each) 300 points
 - 2 tests (mid-term and final, 200 points each) 400 points total
 - many small exercises + class activity 300 points total
- Course webpage: <https://ait-sdt.hu/>
- Course rules: <https://ait-sdt.hu/course-rules.pdf>

(AIT)

Semantic and Declarative Technologies

2025 Autumn Semester

2 / 424

Part I: Declarative Programming – the Prolog language

- Prolog – PROgramming in LOGic
 - The program = statements in **simplified** first-order logic (FOL)
 - Program execution is a **very simple** reasoning process explainable as pattern-based procedure invocation with backtracking
- Dual semantics: **declarative** and **procedural**
 - Slogan: **WHAT** rather than **HOW**
(focus on the **logic** first, but then think over Prolog **execution**, too).

Part II: Declarative Programming with Constraints

- The CLP(\mathcal{X}) schema

Prolog or some other programming language, e.g. C++

 +

“strong” reasoning capabilities on a **restricted domain** \mathcal{X} involving specific constraint (relation) and function symbols.
- Examples for the domain \mathcal{X} :
 - $\mathcal{X} = \mathbb{Q}$ or \mathbb{R} (rationals or reals)
constraints: linear equalities and inequalities
reasoning techniques: Gauss elimination and the simplex method
 - $\mathcal{X} = \mathbb{B}$ (Boole values `true` and `false`, or 1 and 0)
constraints: propositional operators (\wedge, \vee, \neg , etc.)
reasoning techniques: SAT solvers, using e.g. Binary Decision Diagrams.
 - $\mathcal{X} = \text{FD}$ (Finite Domains, e.g. of integers)
constraints: arithmetic, logic, and combinatorial relationships
reasoning techniques: those developed for Constraint Satisfaction Problems (CSPs)
- Main topic: Finite domain constraints (base constraints, reification, combinatorial and user defined constraints, disjunctions, modelling)

(AIT)

Semantic and Declarative Technologies

2025 Autumn Semester

3 / 424

(AIT)

Semantic and Declarative Technologies

2025 Autumn Semester

4 / 424

Part III: The Semantic Web

- The goal: making the information on the web processable by computers
- Slogan: machines should be able to **understand** the web, not only read it
- To achieve the vision of the Semantic Web one has to:
 - add formal meta-information to web pages (ABox), e.g.
 - (AIT hasLocation Budapest) (AIT hasType University)
 - (Paul hasParent Peter) (Paul hasType Parent)
 - formalise background knowledge – build terminologies (TBox)
 - hierarchies of notions, e.g.
 - a **Father** is a (subconcept of) **Parent**,
 - the **hasFather** relationship is contained in the **hasParent** relation
 - definitions and axioms, e.g.
 - a **Father** is a **Male Person** having at least one **child** (i.e. having at least one object in **hasChild** relationship).
 - develop **reasoning** algorithms and tools
- Main topics: Description Logics, from \mathcal{AL} to $\mathcal{SROIQ}(\mathbf{D})$, reasoning tasks, ABoxes; a brief overview of reasoning algorithms; OWL (Web Ontology Language) – the language of the Semantic Web

(AIT)

Semantic and Declarative Technologies

2025 Autumn Semester

5 / 424

Course overview

Part I – *practical* mathematical logic

Propositional Logic

- Basic Boolean functions (bitwise ops in C, Python, etc.)
 - and: \wedge (&)
 - or: \vee (|)
 - not: \neg (~)
 - implies: \rightarrow $A \rightarrow B$ (A implies B) is the same as $(\neg A \vee B)$
- The puzzle below is cited from “What Is The Name Of This Book?” by Raymond M. Smullyan, chapter “From the cases of Inspector Craig”
- Puzzles in this chapter involve suspects of a crime, named A, B, etc. Some of them are guilty, some innocent.
- Example:

An enormous amount of loot had been stolen from a store. The criminal (or criminals) took the heist away in a car. Three well-known criminals A, B, C were brought to Scotland Yard for questioning. The following facts were ascertained:

 - 1 No one other than A, B, C was involved in the robbery.
 - 2 C never works without A (and possibly others) as an accomplice.
 - 3 B does not know how to drive.

Is A innocent or guilty? Can you answer the same question for B or C?

Semantic and Declarative Technologies

2025 Autumn Semester

6 / 424

Course overview

Inspector Craig puzzle – transforming to formal logic

- Let's recall the facts
 - 1 No one other than A, B, C was involved in the robbery.
 - 2 C never works without A (and possibly others) as an accomplice.
 - 3 B does not know how to drive.
- Transform each statement into a formula involving the letters **A**, **B**, **C** as atomic propositions. Proposition **A** stands for “A is guilty”, etc.
 - 1 A is guilty or B is guilty or C is guilty: $A \vee B \vee C$
 - 2 If C is guilty then A is guilty: $C \rightarrow A$
 - 3 It cannot be the case that only B is guilty: $B \rightarrow (A \vee C)$
- Transform each propositional formula into conjunctive normal form (CNF), then show the clauses in simplified form:

Original formula	CNF	Simplified clausal form
1 $A \vee B \vee C$	$A \vee B \vee C$	$+A +B +C.$
2 $C \rightarrow A$	$\neg C \vee A$	$-C +A.$
3 $B \rightarrow (A \vee C)$	$\neg B \vee A \vee C$	$-B +A +C.$
- A clause is a **set** of signed atomic propositions, called *literals*

Inspector Craig puzzle – resolution proof

- Collect the clauses, giving each a reference number:

(1)	$+A +B +C.$	Only A, B, C was involved in the robbery.
(2)	$-C +A.$	C never works without A as an accomplice.
(3)	$-B +A +C.$	B does not know how to drive.
- A resolution step requires two input clauses which have **opposite** literals e.g. literal 3 of clause (1) is $+C$ while lit 1 of clause (2) is $-C$
- The resolution step creates a new clause, called the resolvent. It takes the union of the literals in the inputs and removes a single pair of opposite literals, e.g. resolving (1) lit 3 with (2) lit 1 results in $+A +B$
- The resolvent follows from (is a consequence of) the input clauses, as $(U \vee V) \wedge (\neg U \vee W) \rightarrow (V \vee W)$ always holds (is a tautology)
- A sample resolution proof:

	resolve (1) lit 2 with (3) lit 1 resulting in (4)
(4)	$+A +C.$
	resolve (4) lit 2 with (2) lit 1 resulting in (5)
(5)	$+A.$
- We deduced that **A** is true, so the solution of the puzzle is: **A** is guilty

Clauses in First Order Logic (FOL)

- Example: There is an island where some people are optimistic (opt)
- Given the facts below, can you deduce that someone is optimistic?
 - 1 Those having an opt parent are bound to be opt.
 - 2 Those having a non-opt friend are also bound to be opt.
 - 3 Susan's mother has Susan's father as a friend.
- To formalize this in FOL we introduce some task-specific symbols:
 - X has a parent $Y \rightarrow \text{hasP}(X, Y)$; X has a friend $Y \rightarrow \text{hasF}(X, Y)$
 - X is opt $\rightarrow \text{opt}(X)$; s, f, m stand for Susan, her father and her mother, resp.
- The **FOL form** and the **clausal form** of the above statements:
 - 1 For all X and Y , X is opt if X has a parent Y and Y is opt:
 $\forall X, Y. (\text{opt}(X) \leftarrow \text{hasP}(X, Y) \wedge \text{opt}(Y))$
 $\text{+opt}(X) \text{ -hasP}(X, Y) \text{ -opt}(Y).$
 - 2 For all X and Y , X is opt if X has a friend Y and Y is not opt:
 $\forall X, Y. (\text{opt}(X) \leftarrow \text{hasF}(X, Y) \wedge \neg \text{opt}(Y))$
 $\text{+opt}(X) \text{ -hasF}(X, Y) \text{ +opt}(Y).$
 - 3 $\text{hasP}(s, m) \text{ hasP}(s, f) \text{ hasF}(m, f)$
 $\text{+hasP}(s, m). \text{+hasP}(s, f). \text{+hasF}(m, f).$
- You will learn a reasoning algorithm **FOL resolution**, capable of finding an optimistic person. Prolog execution is based on FOL resolution.

Part II – Prolog

Example 1: checking if an integer is a prime

- A Prolog program consists of predicates (functions returning a Boolean)
- Let's write a predicate, which is true if and only if the argument is a prime
- Programming by specification: first describe when the predicate is true, then transform the description to Prolog code

```

prime(P) :-                                % P is a prime if
integer(P), P > 1,                          % P is an integer and P > 1 and
P1 is P-1,                                  % P1 = P-1 and
\+ (                                         % it is not the case that
    between(2, P1, I),                     % (there exists an integer I such that)
    P mod I == 0,                          % 2 <= I <= P1 and
    P is divisible by I                    % P is divisible by I
).

```

Are you convinced of the correctness of the code? :-)

Example 2: append - multiple uses of a single predicate

- $\text{app}(L1, L2, L3)$ is true if $L3$ is the concatenation of lists $L1$ and $L2$.

```

app([], L, L).                               % appending an empty list with L gives L.
app([H|L1], L2, [H|L3]) :-                  % appending a list composed of
    % head H and tail L1 with a list L2
    % gives a list with head H and tail L3 if
    app(L1, L2, L3).                         % appending L1 and L2 gives L3.

```

- app can be used, for example,
 - to check whether the relation holds:
 $| \text{?- app}([1,2], [3], [1,2,3]). \Rightarrow \text{yes}$
 - to append two lists:
 $| \text{?- app}([1,2], [3,4], L). \Rightarrow L = [1,2,3,4] \text{ ? ; no}$
 - to split a list into two:
 $| \text{?- app}(L1, L2, [1,2,3]). \Rightarrow L1 = [], L2 = [1,2,3] \text{ ? ;}$
 $L1 = [1], L2 = [2,3] \text{ ? ;}$
 $L1 = [1,2], L2 = [3] \text{ ? ;}$
 $L1 = [1,2,3], L2 = [] \text{ ? ; no}$
- Predicate app is available as a built-in: $\text{append}/3$ (append with 3 args)

Example 3: A number puzzle

- An arithmetic expression is **simple** if it uses the four basic operations only
- Let's write a Prolog program for solving the following task:
 Given a set of integers, e.g. $\{1, 3, 4, 6\}$, and a target integer n , e.g. 14, build a simple arithmetic expression that contains each element of the given set exactly once, and evaluates to n
- Some further clarification:
 - you cannot "glue" together integers to form larger ones, e.g. forming 13 from 1 and 3 is **not** allowed
 - each operation can be used 0 or more times
 - parentheses can be used freely
- Examples: $1 + 6 * (3 + 4) = 43$, $(1 + 3)/4 - 6 = -5$
- The list of integers contained within an expression (in order of occurrence) is called its **list of leaves**, e.g. the list of leaves of $6 * (3 + 4)$ is $[6, 3, 4]$
- A fairly hard task is to construct an expression that evaluates to 24, using integers $\{1, 3, 4, 6\}$

The number puzzle in Prolog

Blue/orange color indicates built-in/library predicates

```
% Expr uses all integers in L and evaluates to Val.
leaves_value_expr(L, Val, Expr) :-
    permutation(L, PL),      % PL is a permutation of L,
    leaves_expr(PL, Expr),    % PL is a list of leaves of Expr,
    catch(Expr := Val, _,    % Expr evaluates to Val, if any error
        fail).               % occurs (e.g. division by 0), simply fail

% Expr is an (arbitrary) expression having a given list of leaves L.
leaves_expr(L, Expr) :-
    L = [Expr].              % If L is a singleton, Expr is the element

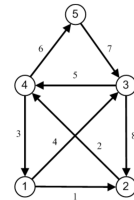
leaves_expr(L, Expr) :-
    append(L1, L2, L),       % Split L into L1 ⊕ L2
    L1 \= [], L2 \= [],      % so that neither L1, nor L2 is empty ([])
    leaves_expr(L1, E1),      % Let E1 be an arbitrary expr with leaves L1
    leaves_expr(L2, E2),      % Let E2 be an arbitrary expr with leaves L2
    member(Op, [+,-,*,/]),    % Let Op be one of the four allowed operations
    Expr =.. [Op,E1,E2].      % Let Expr be a binary expression
                                % with operation Op and operands E1 and E2
```

Example 4: Finding an Euler walk in a graph

- From Wikipedia (https://en.wikipedia.org/wiki/Eulerian_path):

An Eulerian trail, or Euler walk, in an undirected graph is a walk that uses each edge exactly once.

Euler's conjecture: *For the existence of an Euler walk it is necessary that zero or two vertices have an odd degree.* This holds for the “House of Santa Claus” graph on the right: only vertices 1 and 2 have odd degrees.



- An edge of a graph between A and B is represented by a Prolog term A-B
- A graph is represented by a list of edges, e.g. [1-2,1-3,1-4,2-3,2-4,3-4,3-5,4-5] represents the graph above
- Another representation is [1-2,2-4,4-1,1-3,3-4,4-5,5-3,3-2], which corresponds to the above Euler walk (see the numbers along the edges)
- Let's count the number of representations of a graph with n ($= 8$) edges:
 - each edge can be written in two ways, giving 2^n (256) arrangements
 - the n edges can be arranged in $n!$ ($8! = 40320$) ways
 - the number of representations is $2^n \cdot n!$, for $n = 8$ this is ~ 10 million

Euler walks – a simple solution

```
% draw0(G, W): W is a walk representing the same graph as G.
draw0(G0, W) :-
    same_graph0(G0, W), % G0 and W represent the same graph and
    walk(W).             % W is a walk

% walk(G): graph G is an Euler walk.
walk([_ _]).            % A graph consisting of a single edge is a walk.
walk(G) :-
    G = [_-Q|T],         % the endpoint Q of its first edge is the same
    T = [Q-_|_],         % as the starting point of the second edge, and
    walk(T).             % the tail of G, T, is a walk.

% same_edge(E0, E): E0 and E represent the same (undirected) edge
same_edge(A-B, A-B).     same_edge(A-B, B-A).

% same_graph0(G0, G): G0 and G represent the same graph
same_graph0(G0, G) :-
    maplist(same_edge, % G1 is obtained from G0 by reversing some
        G0, G1),       % (0 or more) edges, and
    permutation(G1, G). % G is obtained from G1 by permuting the edges
                        % (permutation/2 is defined in library(lists))
```

- draw0 finds all solutions in 2.7s (run on a processor i5-6300U @ 2.40GHz)

Euler walks – merging edge reversal and permutation

- Permutation relies on another library predicate: select/3:


```
% select(E, L, RestL): E occurs in list L, and list RestL is obtained
%                       by removing this occurrence of E from L.
%                       Concisely: selecting E from L leaves Rest
```
- This is one of the ways to implement permutation:


```
% permutation(L, P): list L has a permutation P.
permutation([], []). % [] has a permutation [].
permutation(L, [First|Perm]) :- % L has a permutation [First|Perm] if
    select(First, L, Rest),      % selecting First from L leaves Rest,
    permutation(Rest, Perm).     % Rest has a permutation Perm.
```
- Inserting edge reversal into the above code yields this predicate:


```
% same_graph1(G0, G): G0 and G represent the same graph
same_graph1([], []).
same_graph1(L, [E|Perm]) :- % L is the same graph as [E|Perm] if
    select(First, L, Rest),  % selecting First from L leaves Rest,
    same_edge(First, E),     % First represents the same edge as E,
    same_graph1(Rest, Perm). % Rest is the same graph as Perm.

draw1(G0, W) :- same_graph1(G0, W), walk(W). % All solutions in 3.5s :-).
```

Euler walks – breakthrough

- To make the code faster we reorder subgoals and add a redundant goal:


```
draw1(GO, W) :-
    /* generate-and-test */
    same_graph1(GO, W),
    walk(W).

draw2(GO, W) :-
    same_length(GO, W), % GO and W are of
    walk(W),           % equal length
    same_graph1(GO, W).
/* constrain-and-generate */
```
- draw2 completes in 0.43 msec, **4 magnitudes faster** than draw1/draw0
- same_length/2 is a library predicate from library(lists). It ensures that the two arguments are lists of the same length:


```
| ?- same_length([1,2,3], L).    => L = [_A,_B,_C] ? ; no
```
- If one of the arguments is a variable (as L above), it will be instantiated to a list of appropriate length, containing distinct variables
- What happens when walk/1 is called with such an argument?


```
| ?- G = [_,_], walk(G).        => G = [_A-_B,_B-_C,_C-_D] ?
```
- Variable G obtains a “walk pattern”: the endpoint of the first edge (_B) is the same variable as the starting point of the second edge, etc. Whenever one of these variable occurrences gets instantiated, the other occurrence gets the same value, **reducing the search space** considerably.

Prolog extensions: coroutines (Prolog II)

- Wikipedia: Coroutines are computer program components that allow execution to be suspended and resumed, generalizing subroutines for cooperative multitasking. Coroutines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes.
- A typical example of coroutines, the Hamming problem: Generate, in increasing order, the sequence of all positive integers divisible by no primes other than 2, 3, 5.
- We implement a simplified version: the only divisors allowed are 2 and 3, using predicates times/3 and merge/3 in dataflow programming style
- Merge two sorted lists into a single sorted list


```
% merge(As, Bs, Cs): Sorted list Cs is obtained by
% collating sorted lists As and Bs, removing duplicates
```
- Multiply each element of a list by a number:


```
% times(As, M, Bs): List Bs is obtained from number list As by
% multiplying each list element by M.
```

Example 5: Solving the Hamming problem via coroutines

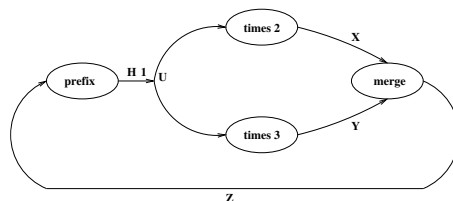
- For this we add the block declaration


```
:- block times(-, ?, ?).
```

 Meaning: suspend pred. times if the first arg. is an unbound variable
- Also, suspend pred. merge if the first **or** second arg is unbound


```
:- block merge(-, ?, ?), merge(?, -, ?).
```
- % U is the list of the first N (2,3)-Hamming numbers*

```
hamming(N, U) :-
    U = [1|_], times(U, 2, X), times(U, 3, Y), merge(X, Y, Z),
    prefix_length([1|Z], U, N). % A predicate from library(lists)
    % prefix_length(L, P, N): L has a prefix P of length N
```



Part III – Constraint technology

Example 6: The 711 problem (David Gries, May 1982)

<https://www.cs.cornell.edu/gries/TechReports/82-493.pdf>

One day, a customer bought four items at a 711 store (a chain of stores in the US). The cashier bagged them and said:

- That will be \$7.11, please.*
- The customer asked: *Is it \$7.11 because this is a 711 store?*
- No*, replied the cashier, *I multiplied the prices together and got \$7.11.*
- But you are supposed to **add** them, not multiply them*, said the customer.
- Oh, you're right!* exclaimed the cashier
- Let me recalculate ... that will be \$7.11.*

Can you find out the price of each of the four items, based on the above conversation?

Note: calculations are assumed to be exact, no rounding!

We will use library(c1pfd): Constraint Logic Programming over Finite Domains