Logic Programming

Part 1: First steps – Datalog and Prolog

Programming Paradigms, Autumn 2020

Aalborg University

Learning goals for this session

- To understand the central notions of logic programming
- To understand the underlying terminology: Clauses, facts, rules, atoms, queries
- To be able to describe certain problems as Datalog programs
- To understand that there is a difference between Datalog and Prolog

What is logic programming?

- Logic programming is a ***declarative*** approach to programming.
- In a declarative language we do not describe ***how*** a computation proceeds; we describe ***what*** is to be computed. We do not run the program but ***perform queries***.
- This is a familiar distinction in computer science: Regular expressions and context-free grammars are declarative ways of describing languages. Finite-state automata and pushdown automata are operational ways of describing languages.

What is a logic program?

- A declaration of facts and proof rules.
- A relational database with rules for populating the database.
- A collection of formulas in first-order predicate logic.

All of these ways of thinking of a logic program are valid and useful – and are equivalent in a very precise way.

First-order predicate logic

First-order predicate logic is a logic over the alphabet

- The Boolean connectives $\wedge, \vee, \neg$
- The universal quantifier $\forall$
- The existential quantifier $\exists$
- Relation symbols R
- Function symbols f
- Variables $x, y, z, \ldots$

First-order predicate logic

In first-order predicate logic we can write formulas such as

$$\forall x. \forall y. [\exists z. E(x,z) \wedge E(z,y)] \Rightarrow P(x,y)$$

Its intended meaning is that

"For every x and y, if there exists a z such that x and z are related wrt. the relation E and z and y are related wrt. the relation E, then x and y are related wrt. the relation P"

So we can quantify over individuals *(not sets or functions)*. Programs in logic programming let us write formulas that look this way.

A first example

We know that

1. Certain people are born royal.
2. If you are married to a royal person, you are also royal.
3. Certain people are married to each other.
4. If X is married to Y, then Y is also married to X.

Mary and Frederik

Marie and Joachim

A program representing this

```
royal(frederik).
royal(joachim).
royal(X) :- married(X,Y), royal(Y).

married(frederik,mary).
married(joachim,marie).
married(X,Y) :- married(Y,X).
```

This is a Datalog program! (It is also a Prolog program)

It began with Prolog

- Prolog dates back to the 1970s where Alain Colmerauer and his colleagues at Université de Marseille wondered if it would be possible to use logic for representing knowledge and for writing programs.
- An important contribution is the work of Kowalski (1974) on resolution and the procedural interpretation of Horn clauses
- Prolog uses a subset of first-order predicate logic and draws its structure from work by earlier logicians such as Herbrand (1930) and Robinson (1965) on automated theorem proving.

Datalog is a database language

- Datalog is a sublanguage of Prolog that saw the light of day in the database community.
- Datalog has some really nice computational properties.
- On the other hand, Datalog is **not** Turing-complete.
- Prolog is Turing-complete and has data structures (such as lists) and arithmetic.

Applications of Datalog

Relational databases

Declarative networking

Distributed computing

Information extraction

Planning

Program analysis

Network monitoring

…

Program analysis

- Bug finders
- Code completion
- Optimizations
- Refactoring tools
- Security

Techniques in program analyis

- Call graph construction
- Dataflow analysis
- Points-to analysis
- Information flow analysis

Applications of Prolog

All of the above

plus

Natural language processing

Machine intelligence

Program semantics

Rule-based systems

and much more …

Language rules

- Representing context-free grammars
- Representing parse trees

Program semantics and type systems

- Representing abstract syntax
- Representing transition rules
- Representing type rules

The Fifth-Generation Project

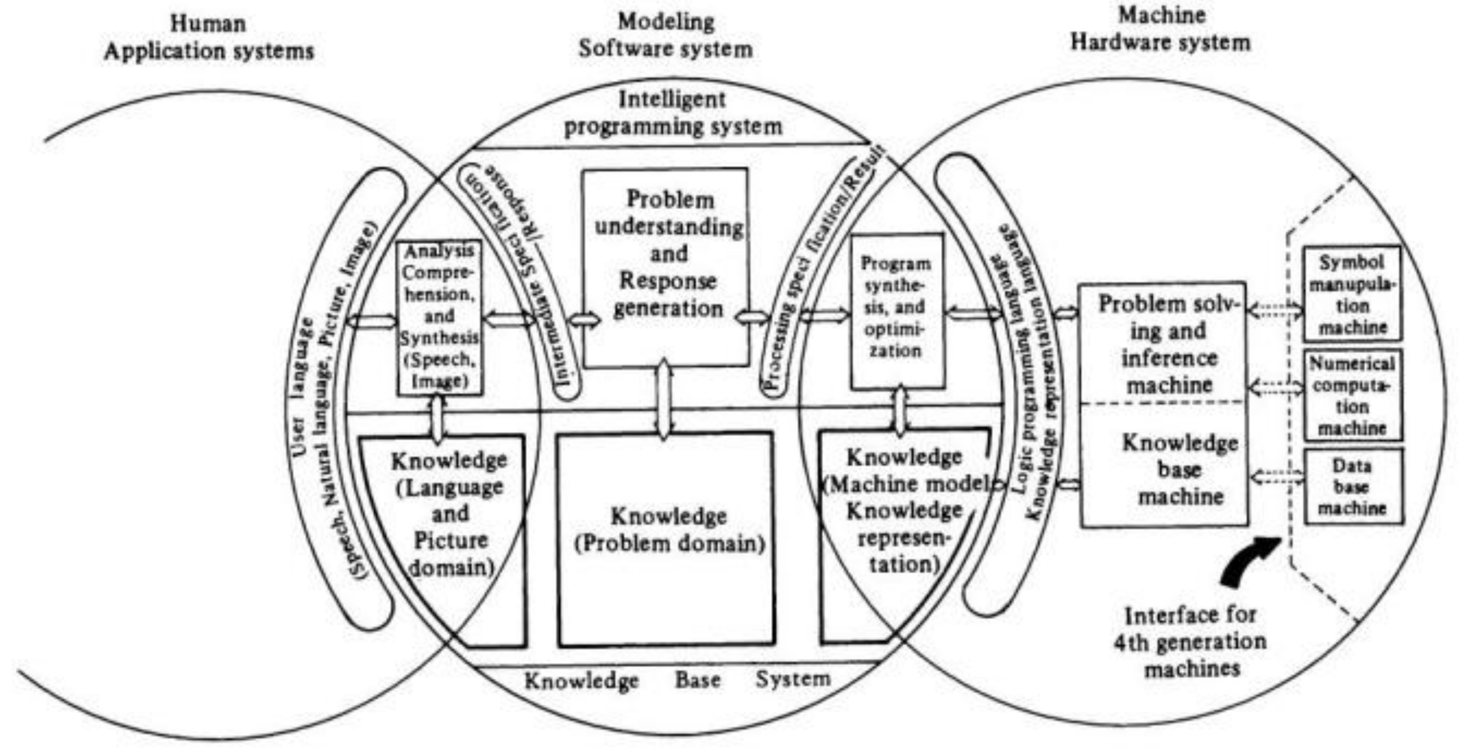


Fig. 4-1 Conceptual diagram of a fifth generation computer system as viewed from the standpoint of programming

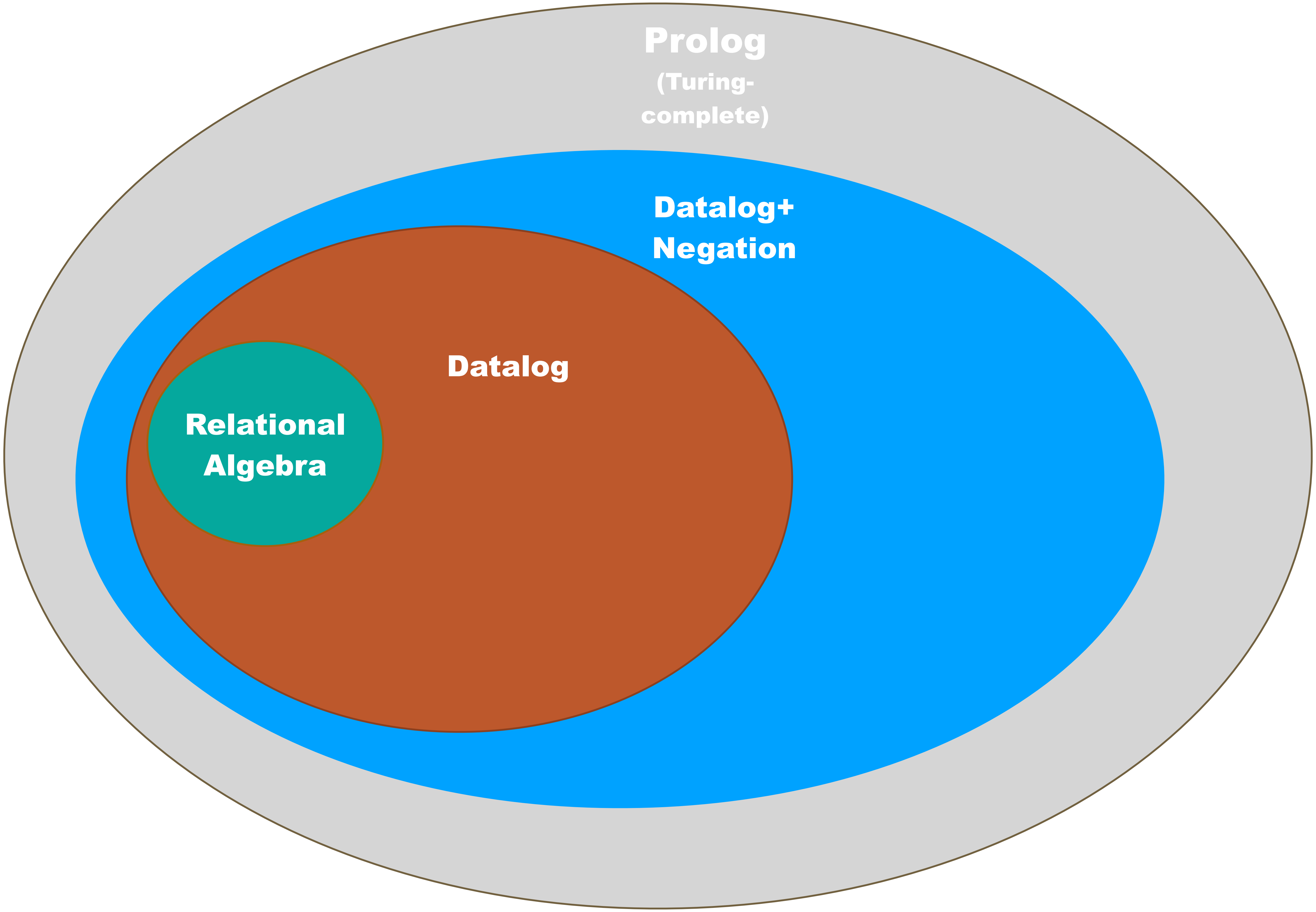
The Fifth-Generation Project

The Japanese *Fifth Generation Project* began in 1982. It was a collaborative effort of the Japanese computer industry coordinated by the Japanese government. The goal was to update the hardware technology of computers but alleviate the problems of programming by creating AI operating systems that would ferret out what the user wanted and then do it. The project wanted to use **Prolog** as the machine language of a new parallel computer.

The project did not go so well. The computer industry shifted so rapidly that what seemed a good idea in 1982 turned out to be at odds with the computer industry's direction ten years later. In particular, the personal computer had arrived.

Prolog is not “wrong”

- Prolog is probably not a good choice for a machine language; we will see why, when we encounter the semantics of Prolog. (There were also LISP machines in the 1970s.)
- But Prolog is extremely well-suited for solving problems that can be expressed in a declarative fashion. We will see why!
- Every programming paradigm has its strengths and weaknesses.


Prolog
(Turing-complete)
Datalog+ Negation
Datalog
Relational Algebra

Horn clauses

- In Datalog and Prolog, rules are ***Horn clauses*** (named after the logician Alfred Horn) of the form

$$P_0(t) \Leftarrow P_1(t), \ldots, P_n(t)$$

- The P's are *predicates*, and the t's are *terms.* The clause should be read as

  If *the predicates* $P_1(t), \ldots, P_n(t)$ *all hold for the terms* t*,* ***then*** $P_0(t)$ *holds*

- Here is an example:

```
royal(X) :- married(X,Y), royal(Y).
```

- The predicates are **`royal`** and **`married`.**

Example: Paths in a directed graph

Given a directed graph, is there a path between two vertices?

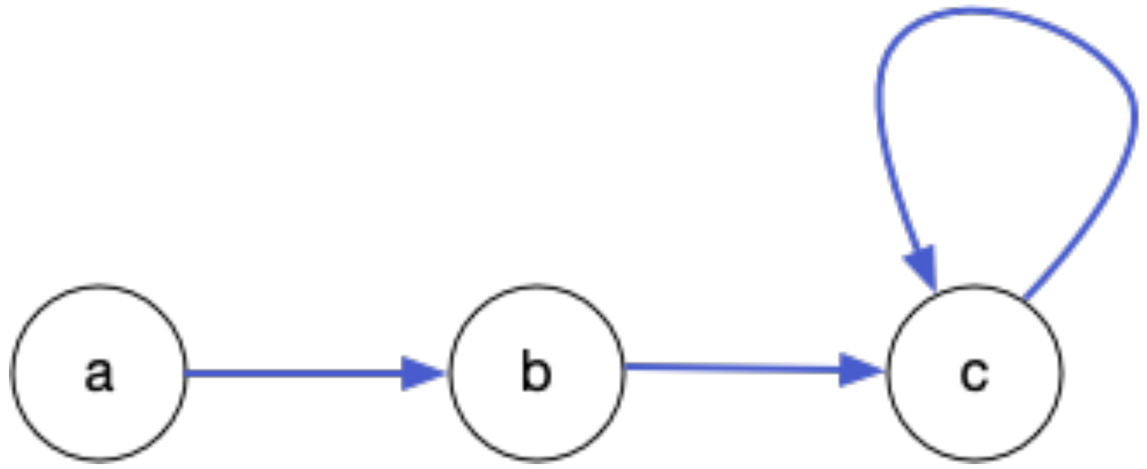
```
path(X,Y) :- path(X,Z), edge(Z,Y).

path(X,Y) :- edge(X,Y).
```

Example: Paths in a directed graph

```
edge(a,b).
edge(b,c).
edge(c,c).

path(X,Y) :- path(X,Z),
edge(Z,Y).
path(X,Y) :- edge(X,Y).
```

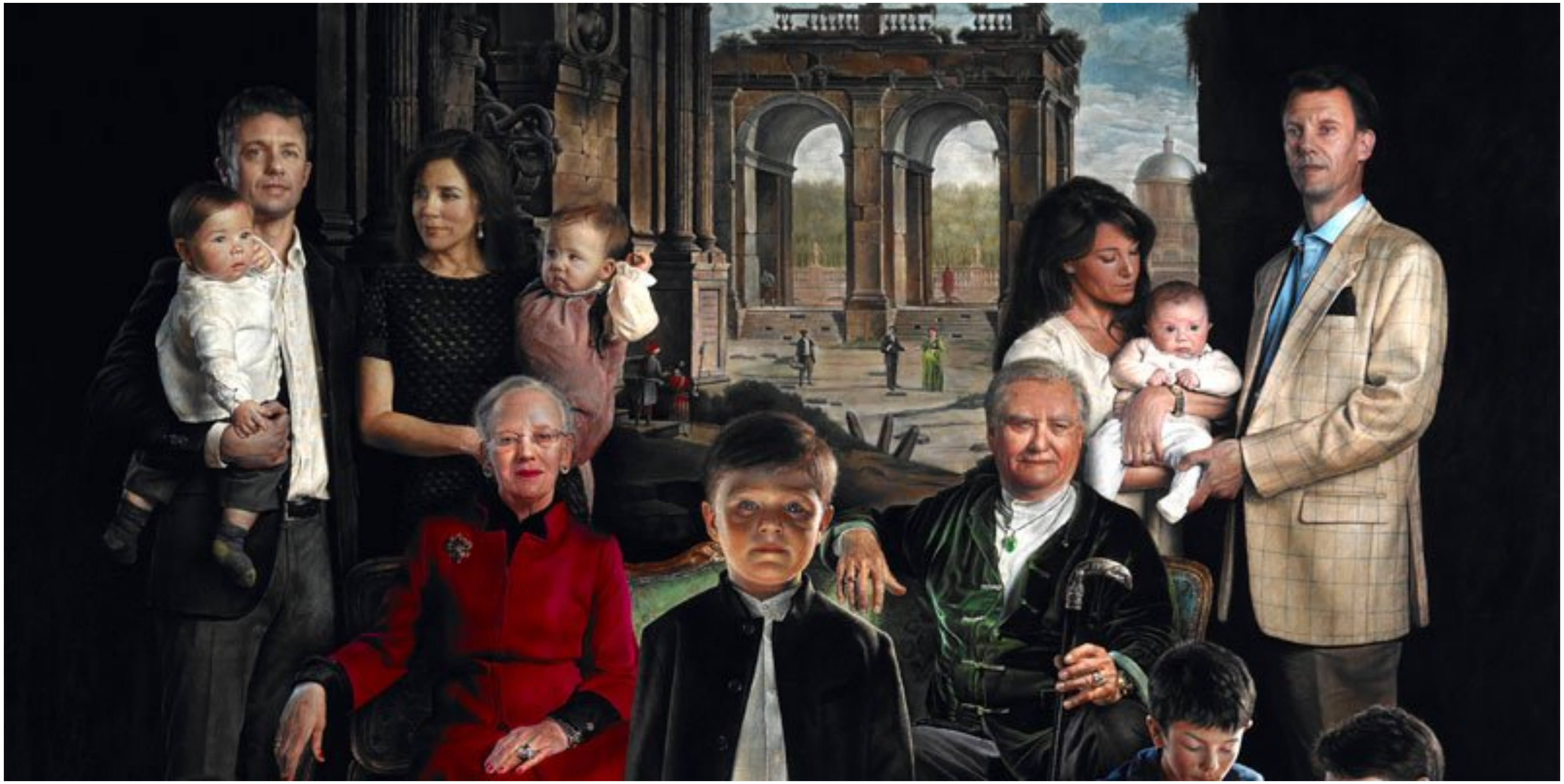


The graph is represented as a collection of facts.

Example: Royal uncles

An uncle is a person who is the brother of one of my parents.

```
uncle(X,Y) :- brother(X,Z),parent(Z,Y).
```



Painting by Thomas Kluge (https://kunsten.nu/journal/galleries/en-fremmedgjort-kongefamilie/)

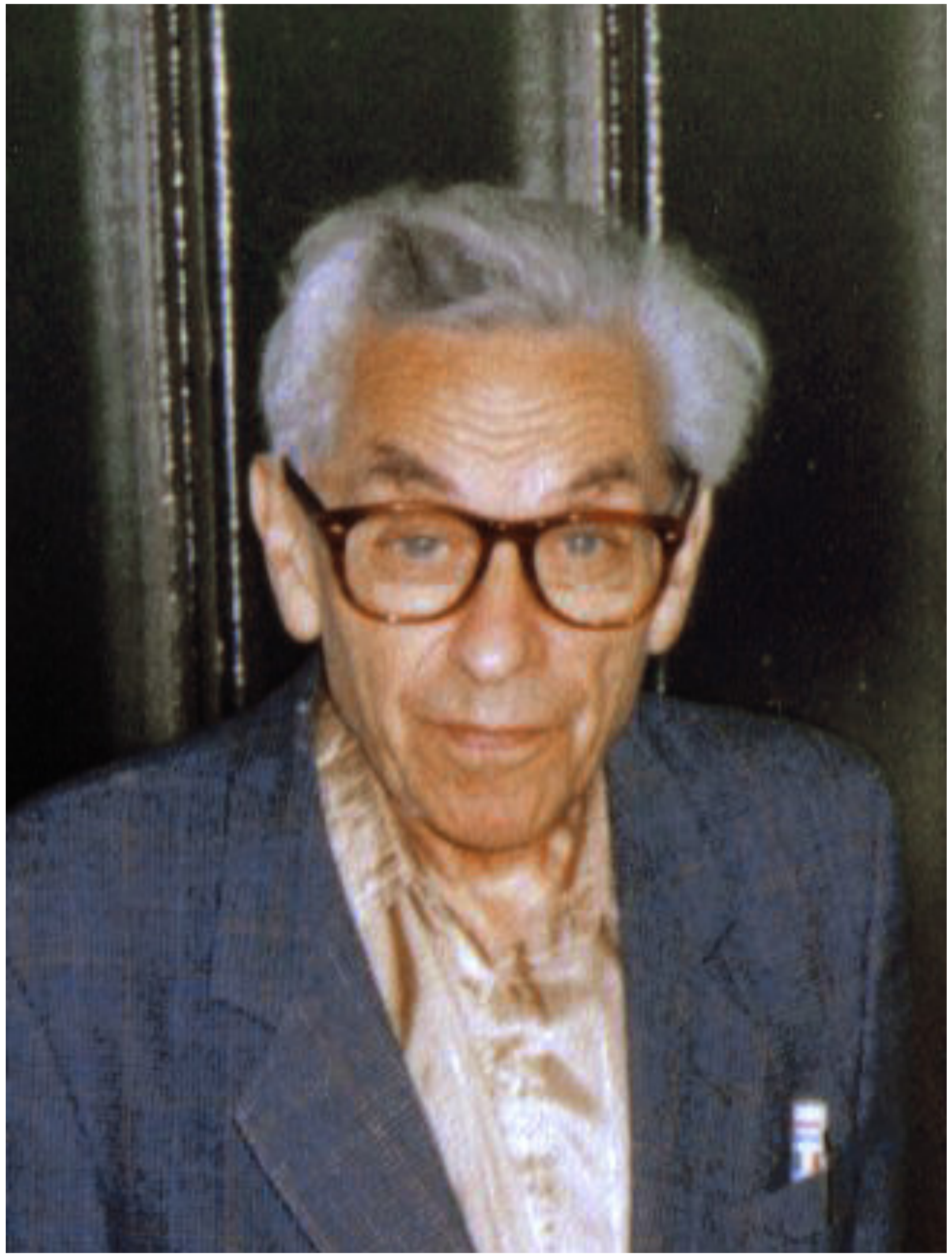
Example: Royal aunts and uncles

```
parent(frederik,christian).
parent(frederik,isabella).
parent(frederik,vincent).
parent(frederik,josephine).
parent(joachim,henrik).
parent(joachim,athena).
brother(frederik,joachim).
brother(joachim,frederik).

uncle(X,Y) :- brother(X,Z),parent(Z,Y).
```

Example: Erdös numbers

- Paul Erdös (1913-1996) was a Hungarian mathematician mainly active in the areas of combinatorics, number theory and graph theory.
- He travelled all over North America and Europe and collaborated with lots of people in the mathematical science. He has published more research papers than any other mathematician (about 1500 papers in total).
- In maths, the notion of *Erdös number* exists. It tells us how far one is from having collaborated with Erdös.
- My own Erdös number is 5.



Example: Erdös numbers in Datalog

```
collaborate(erdos,turan).

collaborate(erdos,bollobas).

erdos1(X):- collaborate(erdos,X).

erdos2(X) :- erdos1(Z),collaborate(Z,X).

…

erdos5(X):- erdos4(Z),collaborate(Z,X).
```

This is a clumsy representation! We need to have a relation for every Erdös number. (We need Prolog to find a nicer approach to programming this.)

The syntax of Datalog - programs and clauses

A Datalog **program** is a finite sequence of Horn **clauses**.

$$P \in \textbf{Program} \qquad P ::= C_1 \ldots C_m$$

$$C \in \textbf{Clause} \qquad C ::= A_0 \Leftarrow A_1, \ldots, A_n$$

The order of clauses does not matter. If there are none, the program is empty.

A_0 is called the *head* of the clause, and $A_1, \ldots, A_n$ are called the *body*.

If $n < 1$ we call the clause a *fact*. $A_0, \ldots, A_n$ are called **atoms**.

The syntax of Datalog - atoms and terms

A Datalog **atom** is a **predicate** symbol applied to a sequence of **terms**.

$$A \in \mathbf{Atom} \qquad A ::= p(t_1, \ldots, t_n)$$

A predicate symbol is an identifier. As for terms,

$$t \in \mathbf{Term} \qquad t ::= k \mid x$$

Here k ranges over the set of *constants* (including strings and numerals), and x ranges over the set of *variables*.

Some unfortunate confusion

- Some implementations of Datalog write variables in lowercase. Many implementations of Prolog write variables in uppercase.
- Some implementations of Datalog write predicates in uppercase. Many implementations of Prolog write predicates in lowercase!!
- Datalog from DrRacket and SWI-Prolog are in agreement; they both use uppercase for variables and lowercase for predicates.
- *So that is what I will use here as well.*

Some unfortunate confusion

- Some authors use a different terminology!

Us	Them	Explanation
Clauses, rules, facts	Constraints	We say that a clause is either a fact or rule.
Atoms	Literals	An atom is a predicate symbol along with its terms.

Ground atoms and ground rules

- An atom is a *ground atom* if it does not contain variables.

  `married(frederik,marie)` is ground, but
  `married(frederik,X)` is not ground.

- A rule is a *ground rule* if it does not contain variables.

  ```
  royal(marie) :- married(marie,mary), royal(mary).
  ```
  is ground, but
  ```
  royal(X) :- married(X,Y), royal(Y).
  ```
  is not.

Extensional and intensional databases

- In Datalog, we speak of the ***extensional*** and ***intensional*** *(with s!!)* ***databases***.
- The extensional database (EDB) is the set of facts already in our program.
- The intensional database (IDB) is the set of facts ***derivable*** from the clauses in our program.

Examples of extensional and intensional databases

Consider the program

```
edge(a,b).
edge(b,c).
edge(c,c).

path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).
```

The ***extensional*** database of this program is the set of facts

EDB = **`{ edge(a,b), edge(b,c), edge(c,c)}`**

The ***intensional*** database of this program contains the facts in EDB and the set of derivable facts, so

IDB = EDB U{ **`path(a,b), path(b,c),path(c,c),path(a,c)}`**

Queries

- Given a Datalog or Prolog program, a ***query*** is a question given as an atom.

  Consider the program

```
edge(a,b).
edge(b,c).
edge(c,c).

path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).
```

  An example of a query is **`path(a,c)?`** (Datalog syntax) or **`path(a,c).`** (Prolog syntax).

  The query asks if **`path(a,c)`** can be derived, that is, if it is an element of the intensional database of the program.

Safety in Datalog

- We would like the intensional database to be finite for every Datalog program.
- A Datalog program is safe if:

  1. Every fact is ground.
  2. Every variable that occurs in the head of a rule also occurs in its body.

     Why is this a good definition?

Safety in Datalog

Here is a program that is *not safe*; its intensional database is *infinite:*

```
edge(a,b)
edge(X,Y).
path(X,Y):- edge(X,Y.
path(X,Y):- path(a,b).
```

We have that **`edge(a,b)`** but also that (for instance) **`edge(a,d)`** and **`edge(e,f)`**. This is the case, since **`edge(X,Y)`** will hold for every **`X`** and **`Y`**.

Even worse, **`path(X,Y)`** does not depend on **`X`** or **`Y`**.

Substitutions

When Datalog or Prolog evaluates a query, it tries to build a substitution.

A substitution is an instantiation of variables to constants.

$$\sigma : \mathbf{Variables} \to \mathbf{Atoms}$$

A substitution turns atoms into ground atoms and rules into ground rules.

Substitutions

Let $\sigma = [X \mapsto a, Y \mapsto b]$

and consider the rule (which is not ground)

```
path(X,c) :- path(X,Y),edge(Y,c)
```

If we apply the substitution, we get the ground rule

```
path(a,c) :- path(a,b),edge(b,c)
```

What happens when a query is evaluated?

- **Datalog** has some nice properties:
  - Every query to a Datalog program will terminate! The semantics of Datalog is fairly straightforward.
  - The answer to any query will be found *in polynomial time*!

Datalog is “the polynomial-time programming language”!

- Recall the decision problem

  “Given a directed graph G, and vertices a and b , is there a path from a to b ?”

  The corresponding language PATH is in P

- Every polynomial-time algorithm can be expressed in Datalog. This is the case, because PATH is *P-complete*. Every language in P polynomial-time reduces to PATH. We already know that the path problem can be expressed in Datalog.

What happens when a query is evaluated?

- But this also implies that Datalog is not Turing-complete.
- **Prolog** is Turing-complete, but that also means that Prolog is less forgiving than Datalog – not all queries will terminate. The semantics of Prolog is more involved.

But… why are things that way?

A precise answer to all this is given by the semantics of Datalog and Prolog, respectively. We also need a precise explanation of the difference between the two languages.

We return to all this in the next session!