Week 1

Magnus Madsen

Friday 14th March, 2025 at 15:00

Week 1: Outline

Tuesday

Lecture (45min)

- Introduction to Declarative Logic Programming
- Introduction to Datalog
- Getting Started with Datalog in Flix

Exercises (45min)

- Work on the assignment alone or together in small groups.

Thursday

Lecture (45min)

- Model-Theoretic Semantics
- Fixpoint Semantics
- Stratified Negation

Exercises (45min)

- Work on the assignment alone or together in small groups.

Quote of the Day

"To know a second language, is to have a second soul."

— Charlemagne

Pull Requests are Welcome

You can improve the course material!

- Exercises are in `src/weekX.md`
- Slides are in `slides/weekX.tex`

PRs can be submitted on GitHub:

`https://github.com/magnus-madsen/advprog/`

Introduction to
Declarative Logic Programming

Programming Paradigms

- Imperative Programming
- Object-Oriented Programming
- Functional Programming
- Logic Programming

Declarative Programming

What is a **declarative** programming language?

> *"Denoting high-level programming languages which can be used to solve problems without requiring the programmer to specify an exact procedure to be followed."*
>
> — *The Oxford Dictionary*

Declarative Programming

What is a **declarative** programming language?

> *"Denoting high-level programming languages which can be used to solve problems without requiring the programmer to specify an exact procedure to be followed."*
>
> — *The Oxford Dictionary*

*"The **what**, not the **how**."*

Declarative Programming Languages

Examples:

- Hypertext Markup Language (HTML)
- Cascading Style Sheets (CSS)
- Structured Query Language (SQL)
- Regular Expressions

Example: Regular Expressions

A ***regular expression*** is a declarative description of a set of strings.

For example, the regular expression r:

$$r = (ab)^\star + c$$

Describes the set of strings consisting of any number of ab's or a single c.

Example: Regular Expressions

A ***regular expression*** is a declarative description of a set of strings.

For example, the regular expression r:

$$r = (ab)^{\star} + c$$

Describes the set of strings consisting of any number of ab's or a single c.

We may want to ask: Is the string "aba" in the language of r?

We can compute the answer to this question in multiple ways:

- We can construct a finite state automaton (FA) and run the string on it.
- We can write a regular expression interpreter and run the string on it.

Why does it matter?

In high-school you may have seen complex equations of the form:

$$2x = 6$$

We can compute the solution to such an equation by various means.

Why does it matter?

In high-school you may have seen complex equations of the form:

$$2x = 6$$

We can compute the solution to such an equation by various means.

- Guess! (Yes, why not?)

Why does it matter?

In high-school you may have seen complex equations of the form:

$$2x = 6$$

We can compute the solution to such an equation by various means.

- Guess! (Yes, why not?)
- Use algebraic simplifications (subtract x on both sides, and so on).

Why does it matter?

In high-school you may have seen complex equations of the form:

$$2x = 6$$

We can compute the solution to such an equation by various means.

- Guess! (Yes, why not?)
- Use algebraic simplifications (subtract x on both sides, and so on).
- Rewrite the equation to a system of linear equalities and use Gauss-Jordan Elimination (reduction to row echelon form).

Why does it matter?

In high-school you may have seen complex equations of the form:

$$2x = 6$$

We can compute the solution to such an equation by various means.

- Guess! (Yes, why not?)
- Use algebraic simplifications (subtract x on both sides, and so on).
- Rewrite the equation to a system of linear equalities and use Gauss-Jordan Elimination (reduction to row echelon form).
- Rewrite the equation to a system of linear inequalities and use Fourier–Motzkin Elimination.

Why does it matter?

In high-school you may have seen complex equations of the form:

$$2x = 6$$

We can compute the solution to such an equation by various means.

- Guess! (Yes, why not?)
- Use algebraic simplifications (subtract x on both sides, and so on).
- Rewrite the equation to a system of linear equalities and use Gauss-Jordan Elimination (reduction to row echelon form).
- Rewrite the equation to a system of linear inequalities and use Fourier–Motzkin Elimination.

Upshot: We agree on the meaning of the equation and we can *check* whether a proposed solution is a *valid* solution.

Logic Programming

What is a **logic** programming language?

> *"Logic programming is a type of programming paradigm which is largely based on formal logic. Any program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain."*
>
> — *Wikipedia*

Declarative Logic Programming

The programmer writes a collection of **logic constraints**.

Declarative Logic Programming

The programmer writes a collection of **logic constraints**.

The compiler and runtime **computes the solution** to the constraints.

- It freely chooses the algorithms and data structures required to do so.
  - For example, it might solve the constraints in parallel.

Declarative Logic Programming

The programmer writes a collection of **logic constraints**.

The compiler and runtime **computes the solution** to the constraints.

- It freely chooses the algorithms and data structures required to do so.
  - For example, it might solve the constraints in parallel.

Declarative logic programming offers several benefits:

Declarative Logic Programming

The programmer writes a collection of **logic constraints**.

The compiler and runtime **computes the solution** to the constraints.

- It freely chooses the algorithms and data structures required to do so.
  - For example, it might solve the constraints in parallel.

Declarative logic programming offers several benefits:

- no side-effects + no explicit control-flow
  - Programs are easy to understand.
  - Programs are easy to modify and extend.
  - Programs can be structured in any order.

Declarative Logic Programming

The programmer writes a collection of **logic constraints**.

The compiler and runtime **computes the solution** to the constraints.

- It freely chooses the algorithms and data structures required to do so.
    - For example, it might solve the constraints in parallel.

Declarative logic programming offers several benefits:

- no side-effects + no explicit control-flow
    - Programs are easy to understand.
    - Programs are easy to modify and extend.
    - Programs can be structured in any order.
- Strong guarantees about termination.

Declarative Logic Programming

The programmer writes a collection of **logic constraints**.

The compiler and runtime **computes the solution** to the constraints.

- It freely chooses the algorithms and data structures required to do so.
  - For example, it might solve the constraints in parallel.

Declarative logic programming offers several benefits:

- no side-effects + no explicit control-flow
  - Programs are easy to understand.
  - Programs are easy to modify and extend.
  - Programs can be structured in any order.
- Strong guarantees about termination.

Challenge: Logic programming requires a different mindset.

Introduction to Datalog

Datalog

Datalog is a simple, yet powerful declarative logic programming language.

Datalog

Datalog is a simple, yet powerful declarative logic programming language.

- Research on Datalog goes back to the 1970s in the fields of artificial intelligence, deductive databases, and knowledge representation.

Datalog

Datalog is a simple, yet powerful declarative logic programming language.

- Research on Datalog goes back to the 1970s in the fields of artificial intelligence, deductive databases, and knowledge representation.
- Datalog (and Prolog) are cornerstones of classical A.I. based on symbolic reasoning — before the golden age of machine learning.

Datalog

Datalog is a simple, yet powerful declarative logic programming language.

- Research on Datalog goes back to the 1970s in the fields of artificial intelligence, deductive databases, and knowledge representation.
- Datalog (and Prolog) are cornerstones of classical A.I. based on symbolic reasoning — before the golden age of machine learning.

A Datalog program is essentially a collection of *Horn clauses*:

$$\forall x_1, \cdots, x_n.\, P_0(t \cdots) \Leftarrow P_1(t \cdots), \cdots, P_m(t \cdots).$$

which allow us to derive new knowledge from existing knowledge.

Datalog

Datalog is a simple, yet powerful declarative logic programming language.

- Research on Datalog goes back to the 1970s in the fields of artificial intelligence, deductive databases, and knowledge representation.
- Datalog (and Prolog) are cornerstones of classical A.I. based on symbolic reasoning — before the golden age of machine learning.

A Datalog program is essentially a collection of *Horn clauses*:

$$\forall x_1, \cdots, x_n.\, P_0(t \cdots) \Leftarrow P_1(t \cdots), \cdots, P_m(t \cdots).$$

which allow us to derive new knowledge from existing knowledge.

Datalog and Prolog are closely related, but should not be confused.

Real-World Applications

Datalog has been successfully used in a range of applications:

- in large-scale points-to analysis of Java programs.
- as an alternative foundation for the Rust borrow checker.
- to identify misconfigurations or security vulnerabilities in AWS networks.

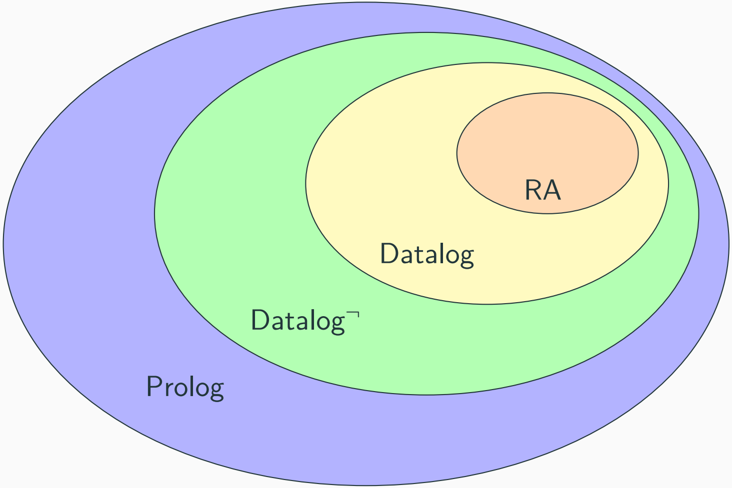
Real-World Applications

Datalog has been successfully used in a range of applications:

- in large-scale points-to analysis of Java programs.
- as an alternative foundation for the Rust borrow checker.
- to identify misconfigurations or security vulnerabilities in AWS networks.

Datalog is a surgical instrument: You use it when the problem calls for it.

Expressive Power



Example

Find a **one-way** trip from Toronto to Billund with the same airline.

```
Route(x, airline, y) :- Leg(x, airline, y).
Route(x, airline, z) :-
    Route(x, airline, y),
    Leg(y, airline, z).

OneWay(airline) :- Route("YYZ", airline, "BLL").
```

Example

Find a **one-way** trip from Toronto to Billund with the same airline.

```
Route(x, airline, y) :- Leg(x, airline, y).
Route(x, airline, z) :-
    Route(x, airline, y),
    Leg(y, airline, z).

OneWay(airline) :- Route("YYZ", airline, "BLL").
```

Find a **round-trip** from Toronto to Billund with the same airline.

```
TwoWay(airline) :-
    Route("YYZ", airline, "BLL"),
    Route("BLL", airline, "YYZ").
```

Datalog Programs

A Datalog **program** p is a finite sequence of constraints:

$$p \in Program = c_1 \cdots c_n$$

Datalog Programs

A Datalog **program** p is a finite sequence of constraints:

$$p \in Program = c_1 \cdots c_n$$

The order of constraints is immaterial.

Datalog Programs

A Datalog **program** p is a finite sequence of constraints:

$$p \in Program = c_1 \cdots c_n$$

The order of constraints is immaterial.

Note: The shortest Datalog program is the empty sequence of constraints.

Datalog Constraints

A Datalog **constraint** c consists of a **head** and a **body**:

$$c \in Constraint = A_0 \Leftarrow A_1, \cdots, A_n.$$

Datalog Constraints

A Datalog **constraint** c consists of a **head** and a **body**:

$$c \in Constraint = A_0 \Leftarrow A_1, \cdots, A_n.$$

Each A_i is an atom. The atom A_0 is the head. The atoms $A_1, \cdots, A_N$ are the body.

Datalog Constraints

A Datalog **constraint** c consists of a **head** and a **body**:

$$c \in \mathit{Constraint} = A_0 \Leftarrow A_1, \cdots, A_n.$$

Each A_i is an atom. The atom A_0 is the head. The atoms $A_1, \cdots, A_N$ are the body.

The sequence of body atoms may be empty.

Datalog Constraints

A Datalog **constraint** c consists of a **head** and a **body**:

$$c \in Constraint = A_0 \Leftarrow A_1, \cdots, A_n.$$

Each A_i is an atom. The atom A_0 is the head. The atoms $A_1, \cdots, A_N$ are the body.

The sequence of body atoms may be empty.

A **fact** is a constraint with an empty body.

A **rule** is a constraint with a non-empty body.

Datalog Atoms and Terms

A Datalog atom A is a predicate symbol and a finite sequence of terms:

$$A \in Atom = p(t_1, \cdots, t_n)$$

A predicate symbol p is an identifier, i.e. a name.

Datalog Atoms and Terms

A Datalog atom A is a predicate symbol and a finite sequence of terms:

$$A \in Atom = p(t_1, \cdots, t_n)$$

A predicate symbol p is an identifier, i.e. a name.

A term t is either a constant k or a variable x:

$$t \in Term = k \mid x.$$

A constant k is a primitive value, e.g. a number of string.

Datalog Grammar

The complete grammar for Datalog is:

$$
\begin{aligned}
p \in \mathit{Program} &= c_1 \cdots c_n \\
c \in \mathit{Constraint} &= A_0 \Leftarrow A_1, \cdots, A_n. \\
A \in \mathit{Atom} &= p(t_1, \cdots, t_n) \\
t \in \mathit{Term} &= k \mid x.
\end{aligned}
$$

$$
\begin{aligned}
p \in \mathit{Predicates} &= \text{is a finite set of predicate symbols.} \\
x \in \mathit{Variables} &= \text{is a finite set of variable symbols.} \\
k \in \mathit{Constants} &= \text{is a finite set of constants.}
\end{aligned}
$$

Example

$$\overbrace{\underbrace{\text{OneWay(airline)}}_{\textbf{Head}} \Leftarrow \underbrace{\text{Route("YYZ", airline, "BLL").}}_{\textbf{Body}}}^{\textbf{Rule}}$$

$$\overbrace{\underbrace{\text{Route}}_{\textbf{Predicate}}(\underbrace{\text{"YYZ"}}_{\textbf{Const}}, \underbrace{\text{airline}}_{\textbf{Var}}, \underbrace{\text{"BLL"}}_{\textbf{Const}}).}^{\textbf{Atom}}$$

Ground Atoms and Rules

An **atom** is said to be **ground** if it does not contain a variable.

A **rule** is said to be **ground** if it all of its atoms are ground.

For example:

```
A(1, 2, 3).                  // Ground Atom
A(1, 2, 3) :- B(2), C(3).    // Ground Rule
```

Safety

A Datalog program P is **safe** if:

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

For example:

```
A(1, x).               // unsafe, violates (1)
A(x, y) :- B(x).       // unsafe, violates (2)
A(x, _) :- B(x).       // unsafe, violates (2)
A(1, x) :- C(x, y).    // OK
```

[1] This is sometimes called the *range restriction property*.

Theoretical Properties

Datalog has several important theoretical properties:

- Every Datalog program has a unique solution.
- Every Datalog program eventually terminates.
- Every polynomial time algorithm can be expressed in Datalog.

Theoretical Properties

Datalog has several important theoretical properties:

- Every Datalog program has a unique solution.
- Every Datalog program eventually terminates.
- Every polynomial time algorithm can be expressed in Datalog.

Upshot: Debugging is easy!

A Larger Example (1/2)

```
Friend("Cartman", "Kyle").
Friend("Cartman", "Stan").
Friend("Kyle", "Cartman").
Friend("Kyle", "Stan").
Friend("Stan", "Cartman").
Friend("Stan", "Kyle").
Friend("Stan", "Wendy").
Friend("Wendy", "Stan").


Interest("Cartman", "Politics").
Interest("Cartman", "Guitar Hero").
Interest("Kyle", "Guitar Hero").
Interest("Stan", "Guitar Hero").
Interest("Wendy", "Politics").
```

A Larger Example (2/2)

```
Friend("Cartman", "Kyle").
Friend("Cartman", "Stan").
Friend("Kyle", "Cartman").
Friend("Kyle", "Stan").
Friend("Stan", "Cartman").
Friend("Stan", "Kyle").
Friend("Stan", "Wendy").
Friend("Wendy", "Stan").


Interest("Cartman", "Politics").
Interest("Cartman", "Guitar Hero").
Interest("Kyle", "Guitar Hero").
Interest("Stan", "Guitar Hero").
Interest("Wendy", "Politics").
```

```
FriendOfFriend(x, z) :-
    Friend(x, y),
    Friend(y, x),
    Friend(y, z),
    if x != z.

ShareInterest(x, y) :-
    Interest(x, i),
    Interest(y, i),
    if x != y.

FriendSuggestion(x, y) :-
    FriendOfFriend(x, y),
    ShareInterest(x, y),
    not Friend(x, y).
```

Getting Started with Datalog in Flix

Theory vs. Practice

We study Datalog in its purest form: as a minimal calculus.

- A bit like the lambda calculus of logic programming.
- In real life, no one writes functional programs in the pure lambda calculus.
- Similarly, no one writes logic programs in pure Datalog.

Theory vs. Practice

We study Datalog in its purest form: as a minimal calculus.

- A bit like the lambda calculus of logic programming.
- In real life, no one writes functional programs in the pure lambda calculus.
- Similarly, no one writes logic programs in pure Datalog.

In practice, we want a *programming language* with amenities like:

- extensions that increase the expressive power.
- type systems to prevent mistakes.
- IDE support.
- ... and more ...

Datalog Dialects and Implementations (1/2)

There are many object-oriented languages:

- E.g. Java, C#, JavaScript, Python, Smalltalk, ...

Datalog Dialects and Implementations (1/2)

There are many object-oriented languages:

- E.g. Java, C#, JavaScript, Python, Smalltalk, ...

There are many relational database management systems:

- E.g. MSSQL, MySQL, Oracle DBMS, IBM DB2, SQLite, ...

Datalog Dialects and Implementations (1/2)

There are many object-oriented languages:

- E.g. Java, C#, JavaScript, Python, Smalltalk, ...

There are many relational database management systems:

- E.g. MSSQL, MySQL, Oracle DBMS, IBM DB2, SQLite, ...

In the same vein, there are also many Datalog dialects and solvers:

- DLV is an established commercial Datalog engine `https://www.dlvsystem.it/`
- Logica is an open source Datalog engine released by Google `https://logica.dev/`
- Souffle is a open source and highly scalable Datalog engine `https://souffle-lang.github.io/`

Datalog Dialects and Implementations (2/2)

In this course, we shall use the **Flix programming language**:

- Flix is fully-blown functional, logic, and imperative programming language.
- A unique feature of Flix is its support for Datalog as a strongly-typed deeply embedded domain specific language (EDSL).
- Flix is developed by researchers from several universities, including Aarhus University, the University of Waterloo (Canada), the University of Copenhagen, and the University of Tubingen (Germany).

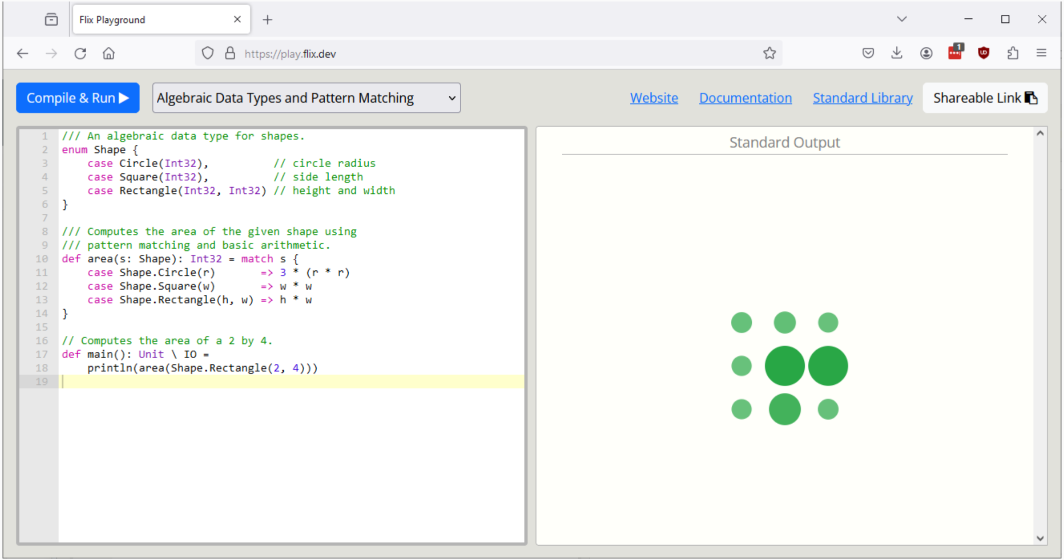
The Flix Playground (1/2)

Flix has an online playground available at:

`https://play.flix.dev/`

Note: The playground runs on a shared server and may be slow.

The Flix Playground (2/2)



The Flix VSCode Extension (1/2)

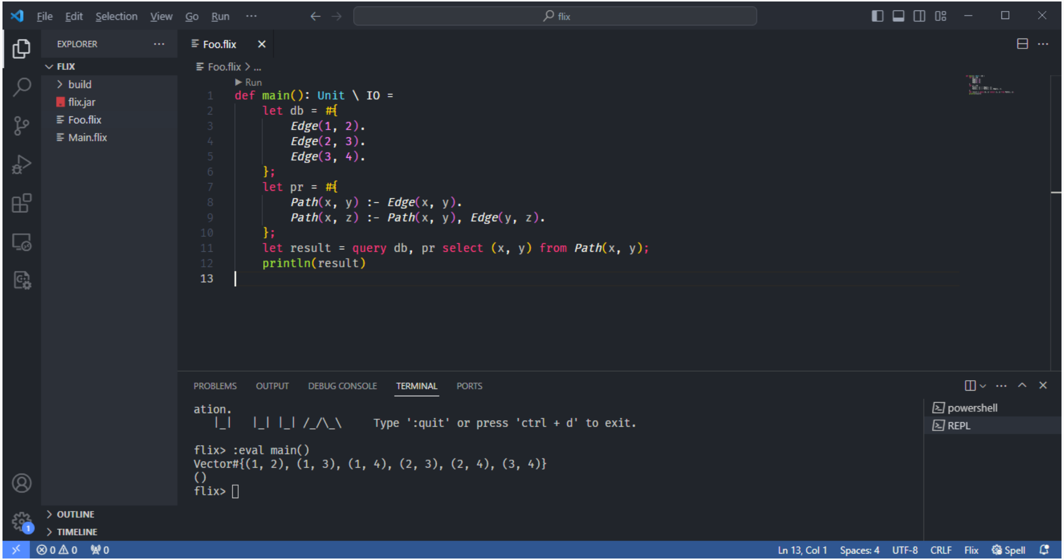
Flix has a fully-featured Visual Studio Code (VSCode) extension.

To run Flix on your machine:

- Ensure that you have Visual Studio Code installed.
- Ensure that you have Java 21 (or later) installed.
  - `https://adoptium.net/`
- Follow the instructions at:
  - `https://flix.dev/get-started/`

Note: VSCode must be used in project mode, i.e. "File -> Open Folder".

The Flix VSCode Extension (2/2)



Flix – An Example to Get You Started

Here is a simple example program you can copy-and-paste to get started:

```
def main(): Unit \ IO =
    let db = #{
        Edge(1, 2).
        Edge(2, 3).
        Edge(3, 4).
    };
    let pr = #{
        Path(x, y) :- Edge(x, y).
        Path(x, z) :- Path(x, y), Edge(y, z).
    };
    let result = query db, pr select (x, y) from Path(x, y);
    println(result)
```

Summary

Declarative Programming

- the ***what***, not the ***how***.

Summary

Declarative Programming

- the ***what***, not the ***how***.

Logic programming

- programs as logic constraints: ***facts*** and ***rules***.
- infer new knowledge from existing knowledge.

Summary

Declarative Programming

- the ***what***, not the ***how***.

Logic programming

- programs as logic constraints: ***facts*** and ***rules***.
- infer new knowledge from existing knowledge.

Datalog is a simple, yet powerful *declarative logic* programming language.

- a Datalog program is a collection of facts and rules.
- every Datalog program has a unique and efficiently computable solution.

Week 1: Outline

Tuesday

Lecture (45min)

- Introduction to Declarative Logic Programming
- Introduction to Datalog
- Getting Started with Datalog in Flix

Exercises (45min)

- Work on the assignment alone or together in small groups.

Thursday

Lecture (45min)

- Model-Theoretic Semantics
- Fixpoint Semantics
- Stratified Negation

Exercises (45min)

- Work on the assignment alone or together in small groups.

Quote of the Day

“A language that doesn’t affect the way you think about programming, is not worth knowing.”

— Alan Perlis

Pull Requests are Welcome

You can improve the course material!

- Exercises are in `src/weekX.md`
- Slides are in `slides/weekX.tex`

PRs can be submitted on GitHub:

`https://github.com/magnus-madsen/advprog/`

Model-Theoretic Semantics

Extensional vs. Intensional

Given a Datalog program P:

- The ***extensional database (EDB)*** is the set of facts already in P.
- The ***intensional database (IDB)*** is the set of facts derivable from P.

Extensional vs. Intensional

Given a Datalog program P:

- The ***extensional database (EDB)*** is the set of facts already in P.
- The ***intensional database (IDB)*** is the set of facts derivable from P.

An **extensional** definition defines an object by **enumeration**.

- E.g. a fruit is an apple, or an apricot, or an avocado, or a banana, or …

Extensional vs. Intensional

Given a Datalog program P:

- The ***extensional database (EDB)*** is the set of facts already in P.
- The ***intensional database (IDB)*** is the set of facts derivable from P.

An **extensional** definition defines an object by **enumeration**.

- E.g. a fruit is an apple, or an apricot, or an avocado, or a banana, or ...

An **intensional** definition defines an object by its **necessary and sufficient conditions**.

- E.g. a fruit is the sweet and fleshy product of a tree or other plant that contains seed and can be eaten as food.

Model-theoretic Semantics (1/2)

The **model-theoretic** semantics define the meaning of a Datalog program in terms of interpretations and models. Briefly,

- An interpretation is a set of facts.
- A model is an interpretation that satisfy all constraints in the program.
- The minimal model, which is unique, is smaller than all other models.
  - We think of the minimal model as the solution to a Datalog program.

Model-theoretic Semantics (1/2)

The **model-theoretic** semantics define the meaning of a Datalog program in terms of interpretations and models. Briefly,

- An interpretation is a set of facts.
- A model is an interpretation that satisfy all constraints in the program.
- The minimal model, which is unique, is smaller than all other models.
  - We think of the minimal model as the solution to a Datalog program.

The model-theoretic semantics describes the ***what***, not the ***how***.

Model-theoretic Semantics (1/2)

We will need to learn several new definitions and concepts:

- Herbrand Base and Herbrand Universe
- Interpretations
- Truth
- Models
- Minimality

But fear not, these definitions and concepts are not too difficult.

Running Example

We will use the following simple Datalog program P:

```
GrandParent(x, z) :- Parent(x, y), Parent(y, z).

Parent("Bart", "Homer").
Parent("Lisa", "Homer").
Parent("Homer", "Grampa").
```

Herbrand Universe

The ***Herbrand Universe*** $\mathcal{U}$ of a Datalog program P is **the set of all constants** appearing anywhere in P.

For example, the Herbrand Universe of P is the set:

$$U = \{\text{"Bart"}, \text{"Lisa"}, \text{"Homer"}, \text{"Grampa"}\}$$

Herbrand Base

The Herbrand Base $\mathcal{B}$ of a Datalog program P is the set of all ground atoms with predicates symbols drawn from P and terms drawn from the Herbrand Universe $\mathcal{U}$.

For our example, the Herbrand Base of P is the set:

$$B = \left\{ \begin{array}{ll} \text{Parent}(\text{"Bart"}, \text{"Bart"}), & \text{GrandParent}(\text{"Bart"}, \text{"Bart"}), \\ \text{Parent}(\text{"Bart"}, \text{"Lisa"}), & \text{GrandParent}(\text{"Bart"}, \text{"Lisa"}), \\ \text{Parent}(\text{"Bart"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Homer"}), \\ \text{Parent}(\text{"Bart"}, \text{"Grampa"}), & \text{GrandParent}(\text{"Bart"}, \text{"Grampa"}), \\ \text{Parent}(\text{"Lisa"}, \text{"Bart"}), & \text{GrandParent}(\text{"Lisa"}, \text{"Bart"}), \\ \text{Parent}(\text{"Lisa"}, \text{"Lisa"}), & \text{GrandParent}(\text{"Lisa"}, \text{"Lisa"}), \\ \text{Parent}(\text{"Lisa"}, \text{"Homer"}), & \text{GrandParent}(\text{"Lisa"}, \text{"Homer"}), \\ \text{Parent}(\text{"Lisa"}, \text{"Grampa"}), & \text{GrandParent}(\text{"Lisa"}, \text{"Grampa"}), \\ \ldots & \ldots \\ \text{Parent}(\text{"Grampa"}, \text{"Grampa"}), & \text{GrandParent}(\text{"Grampa"}, \text{"Grampa"}), \end{array} \right\}$$

Interpretations

An **interpretation** $I \subseteq \mathcal{B}$ is a subset of the Herbrand Base.

For example,

$$I = \left\{ \begin{array}{ll} \text{Parent}(\text{"Bart"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Grampa"}), \\ \text{Parent}(\text{"Lisa"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Lisa"}) \end{array} \right\}$$

is an interpretation.

Truth w.r.t. an Interpretation

Given an interpretation I we can determine the truth of a constraint:

- A ground atom $A = p(k_1, \cdots, k_n)$ is true w.r.t. an interpretation I if $A \in I$.
- A conjunction of ground atoms $A_1, \cdots, A_n$ is true w.r.t. an interpretation I if each atom A_i is true in the interpretation.
- A ground rule $A_0 \Leftarrow A_1, \cdots, A_n$ is true w.r.t. an interpretation if the body conjunction $A_1, \cdots, A_n$ is false or the head atom A_0 is true.

Example

Given the interpretation:

$$I = \left\{ \begin{array}{ll} \text{Parent}(\text{"Bart"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Grampa"}), \\ \text{Parent}(\text{"Lisa"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Lisa"}) \end{array} \right\}$$

Example

Given the interpretation:

$$I = \left\{ \begin{array}{ll} \text{Parent}(\text{"Bart"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Grampa"}), \\ \text{Parent}(\text{"Lisa"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Lisa"}) \end{array} \right\}$$

The ground atom:

$$\text{Parent}(\text{"Lisa"}, \text{"Homer"})$$

is true.

Example

Given the interpretation:

$$I = \left\{ \begin{array}{ll} \text{Parent}(\text{"Bart"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Grampa"}), \\ \text{Parent}(\text{"Lisa"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Lisa"}) \end{array} \right\}$$

The ground atom:

$$\text{Parent}(\text{"Lisa"}, \text{"Homer"})$$

is true.

Moreover, the ground rule:

$$\text{GrandParent}(\text{"Lisa"}, \text{"Lisa"}) \Leftarrow \text{Parent}(\text{"Bart"}, \text{"Homer"}), \text{Parent}(\text{"Homer"}, \text{"Grampa"}).$$

is true.

Models

A **model** M of a Datalog program P is an interpretation I that makes each ground instance of each constraint in P true.

Models

A **model** M of a Datalog program P is an interpretation I that makes each ground instance of each constraint in P true.

A *ground instance* of a rule is obtained by replacing every variable in a rule with a constant from the Herbrand universe. For example, the interpretation:

$$M = \left\{ \begin{array}{ll} \text{Parent}(\text{"Bart"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Grampa"}), \\ \text{Parent}(\text{"Lisa"}, \text{"Homer"}), & \text{GrandParent}(\text{"Lisa"}, \text{"Grampa"}), \\ \text{Parent}(\text{"Homer"}, \text{"Grampa"}) & \end{array} \right\}$$

is a model of the program.

Minimal Models (1/2)

A model M is **minimal** if there is no other model M' such that $M' \subset M$.

Minimal Models (1/2)

A model M is **minimal** if there is no other model M' such that $M' \subset M$.

For example, the interpretation on the previous slide was a minimal model.

Minimal Models (1/2)

A model M is **minimal** if there is no other model M' such that $M' \subset M$.

For example, the interpretation on the previous slide was a minimal model.

On other hand, the interpretation:

$$M = \left\{ \begin{array}{ll} \text{Parent}(\text{"Bart"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Grampa"}), \\ \text{Parent}(\text{"Lisa"}, \text{"Homer"}), & \text{GrandParent}(\text{"Lisa"}, \text{"Grampa"}), \\ \text{Parent}(\text{"Homer"}, \text{"Grampa"}), & \text{GrandParent}(\text{"Homer"}, \text{"Homer"}) \end{array} \right\}$$

is a **model**, but it is **not minimal**.

Minimal Models (1/2)

A model M is **minimal** if there is no other model M' such that $M' \subset M$.

For example, the interpretation on the previous slide was a minimal model.

On other hand, the interpretation:

$$M = \left\{ \begin{array}{ll} \text{Parent}(\text{"Bart"}, \text{"Homer"}), & \text{GrandParent}(\text{"Bart"}, \text{"Grampa"}), \\ \text{Parent}(\text{"Lisa"}, \text{"Homer"}), & \text{GrandParent}(\text{"Lisa"}, \text{"Grampa"}), \\ \text{Parent}(\text{"Homer"}, \text{"Grampa"}), & \text{GrandParent}(\text{"Homer"}, \text{"Homer"}) \end{array} \right\}$$

is a **model**, but it is **not minimal**.

Intuition: A model satisfies the constraints, but may contain superfluous facts.

Minimal Models (2/2)

Theorem: Given two models M_1 and M_2 of a Datalog program P the intersection $M_1 \cap M_2$ is also a model of P.

Theorem: The minimal model is the intersection of all models.

Upshot: The minimal model is unique!

Fixpoint Semantics

Computing Minimal Models (1/4)

We now have the mathematical foundations to answer questions such as:

- When is an interpretation a model?
- When is a model minimal?
- What is the solution to a Datalog program?

Computing Minimal Models (1/4)

We now have the mathematical foundations to answer questions such as:

- When is an interpretation a model?
- When is a model minimal?
- What is the solution to a Datalog program?

What we *lack* is method to **compute** the minimal model of a program.

- We need the ***how***. Enter the fixpoint semantics.

Computing Minimal Models (2/4)

Assume that I is an interpretation of a Datalog program P.

We define the ***immediate consequence operator*** T_P as the head atoms of each ground rule instance satisfied by I. For example, if we have the interpretation:

$$I = \left\{ \text{Parent}(\text{"Bart"}, \text{"Homer"}), \text{Parent}(\text{"Homer"}, \text{"Grampa"}) \right\}$$

We can *derive* the fact:

$$\text{GrandParent}(\text{"Bart"}, \text{"Grampa"})$$

Intuitively, we can think of T_P as computing the set of facts that can be inferred in one step from the interpretation I, i.e. its direct consequences.

Computing Minimal Models (3/4)

We can use the immediate consequence operator T_P to compute the minimal model of a Datalog program as the sequence:

$$\text{Iteration } 1 = T_P(\emptyset)$$
$$\text{Iteration } 2 = T_P(T_P(\emptyset))$$
$$\text{Iteration } 3 = T_P(T_P(T_P(\emptyset)))$$
$$\text{Iteration i} = T_P^i(\emptyset) = T_P(T_P^i(\emptyset))$$

That is, we repeatedly apply T_P, starting from the empty set, and until we do not infer any new facts.

Formally, we compute the **least fixpoint** of T_P.

Computing Minimal Models (4/4)

Theorem: The least fixpoint of the immediate consequence operator T_P is equivalent to the minimal model.

Computing Minimal Models (4/4)

Theorem: The least fixpoint of the immediate consequence operator T_P is equivalent to the minimal model.

Using the immediate consequence operator to compute the minimal model of a Datalog program is an example of **bottom-up evaluation**.

Computing Minimal Models (4/4)

Theorem: The least fixpoint of the immediate consequence operator T_P is equivalent to the minimal model.

Using the immediate consequence operator to compute the minimal model of a Datalog program is an example of **bottom-up evaluation**.

Using T_P to compute the minimal model is called **naïve evaluation**.

Computing Minimal Models (4/4)

Theorem: The least fixpoint of the immediate consequence operator T_P is equivalent to the minimal model.

Using the immediate consequence operator to compute the minimal model of a Datalog program is an example of **bottom-up evaluation**.

Using T_P to compute the minimal model is called **naïve evaluation**.

A better strategy, used in practice, is called **semi-naïve evaluation**.

- We shall not discuss it further, but the core idea is to propagate delta sets (i.e. set differences) which is faster than propagating full sets.

Stratified Negation

Negation

What if we had the program:

```
Path(x, y) :- Edge(x, y).
Path(x, z) :- Path(x, y), Edge(y, z).
```

Negation

What if we had the program:

```
Path(x, y) :- Edge(x, y).
Path(x, z) :- Path(x, y), Edge(y, z).
```

And we wanted to compute the pairs (x, y) which are not connected by a path? We can achieve this by using negation:

```
Unconnected(x, y) :- Vertex(x), Vertex(y), not Path(x, y).
```

Note: We must bind x and y by using Vertex.

Datalog Grammer Extended with Negation

We extend the grammar of Datalog to allow negated *body* atoms:

$$
\begin{aligned}
p \in \mathit{Program} &= c_1 \cdots c_n \\
c \in \mathit{Constraint} &= A_0 \Leftarrow B_1, \cdots, B_n. \\
A \in \mathit{HeadAtom} &= p(t_1, \cdots, t_n) \\
B \in \mathit{BodyAtom} &= p(t_1, \cdots, t_n) \mid \textbf{not}\ p(t_1, \cdots, t_n) \\
t \in \mathit{Term} &= k \mid x.
\end{aligned}
$$

$$
\begin{aligned}
p \in \mathit{Predicates} &= \text{is a finite set of predicate symbols.} \\
x \in \mathit{Variables} &= \text{is a finite set of variable symbols.} \\
k \in \mathit{Constants} &= \text{is a finite set of constants.}
\end{aligned}
$$

Safety for Datalog Programs with Negation

A Datalog program P which uses negation is **safe** if:

1. Every fact in P is ground.
2. Every variable x that occurs in the head of a rule also occurs in its body.
3. Every variable that occurs in a *negative* body atom also occurs in a *positive* body atom.

Safety for Datalog Programs with Negation

A Datalog program P which uses negation is **safe** if:

1. Every fact in P is ground.
2. Every variable x that occurs in the head of a rule also occurs in its body.
3. Every variable that occurs in a *negative* body atom also occurs in a *positive* body atom.

For example:

```
A(x) :- not B(x).          // unsafe, violates (3)
A(x) :- B(x), not C(x).   // OK
```

Problems with Unrestricted Negation

Unfortunately, *unrestricted* negation causes problems. Consider the program:

$$P(x) \Leftarrow \textbf{not}\ Q(x).$$

$$Q(x) \Leftarrow \textbf{not}\ P(x).$$

Problems with Unrestricted Negation

Unfortunately, *unrestricted* negation causes problems. Consider the program:

$$P(x) \Leftarrow \textbf{not}\ Q(x).$$
$$Q(x) \Leftarrow \textbf{not}\ P(x).$$

Assume that the program contains the constant 42.

Now this program has **two** models:

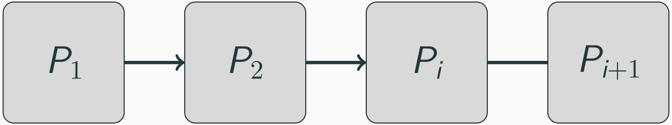
$$M_1 = \{P(42)\} \qquad M_2 = \{Q(42)\}$$

Neither of which is minimal! Yikes!

Stratified Negation

We *side-step* these difficulties with **stratified** Datalog programs which *disallow recursion through negation*.

The idea is that we take a Datalog program P, with negation, and view it as a sequence of programs $P_1, \cdots, P_n$:



The computed facts (the IDB) of P_i become the facts (the EDB) of P_{i+1}.

- Critically, we must partition the predicate symbols such that if p depends on q then q occurs in an earlier or the same program.

Example

The Datalog program:

```
Path(x, y) :- Edge(x, y).
Path(x, z) :- Path(x, y), Edge(y, z).
Unconnected(x, y) :- Vertex(x), Vertex(y), not Path(x, y).
```

Example

The Datalog program:

```
Path(x, y) :- Edge(x, y).
Path(x, z) :- Path(x, y), Edge(y, z).
Unconnected(x, y) :- Vertex(x), Vertex(y), not Path(x, y).
```

is stratified as shown by the partition:

$$P_0 = \{\text{Edge}, \text{Path}, \text{Vertex}\} \qquad \text{and} \qquad P_1 = \{\text{Unconnected}\}$$

Precedence Graph

Given a Datalog program P, we define the precedence graph $\mathcal{PG}$:

- If there is a rule $A \Leftarrow \cdots, B, \cdots$ then there is an edge $A \xleftarrow{+} B$.
- If there is a rule $A \Leftarrow \cdots, \textbf{not}\, B, \cdots$ then there is an edge $A \xleftarrow{-} B$.

Precedence Graph

Given a Datalog program P, we define the precedence graph $\mathcal{PG}$:

- If there is a rule $A \Leftarrow \cdots, B, \cdots$ then there is an edge $A \overset{+}{\leftarrow} B$.
- If there is a rule $A \Leftarrow \cdots, \textbf{not}\, B, \cdots$ then there is an edge $A \overset{-}{\leftarrow} B$.

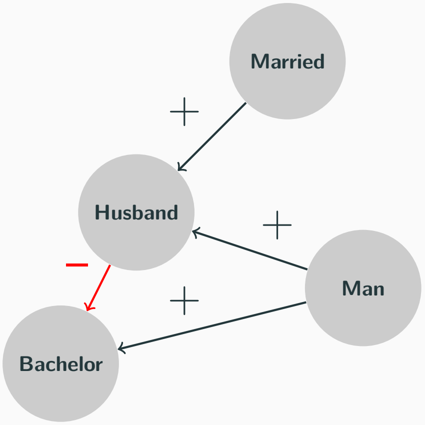
Theorem. A Datalog program P is stratifiable if and only if its precedence graph $\mathcal{PG}$ contains no cycle with an edge labeled $-$.

Example

The Datalog program:

```
Husband(x)   :- Man(x), Married(x).
Bachelor(x)  :- Man(x), not Husband(x).
```

is stratified with the graph on the right.

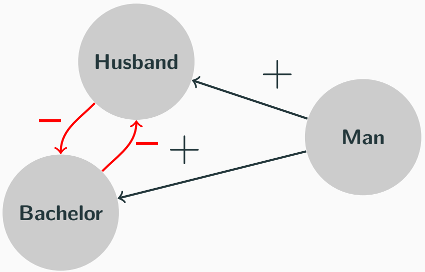


Example

The Datalog program:

```
Husband(x)  :- Man(x), not Bachelor(x).
Bachelor(x) :- Man(x), not Husband(x).
```

is *not* stratified with the graph on the right.



Computing the Strata

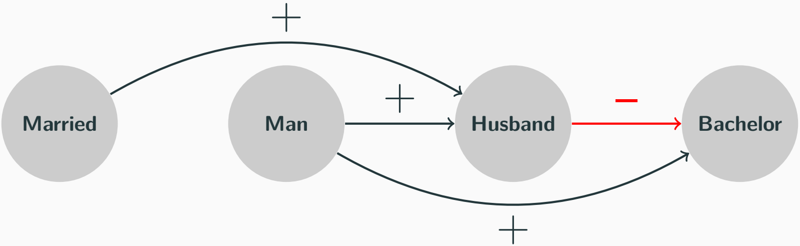
We can use the precedence graph $\mathcal{PG}$ to compute the strata:

1. Compute the precedence graph $\mathcal{PG}$.
2. Compute the strongly connected components of $\mathcal{PG}$.
3. Compute a topological sort of the strongly connected components to determine an ordering of the strata.

Computing the Strata

We can use the precedence graph $\mathcal{PG}$ to compute the strata:

1. Compute the precedence graph $\mathcal{PG}$.
2. Compute the strongly connected components of $\mathcal{PG}$.
3. Compute a topological sort of the strongly connected components to determine an ordering of the strata.



Stratified Negation

We don't actually have to compute the precedence graph or any stratification.

- Any half-decent Datalog engine will automatically stratify the program for us.
- However, we must understand stratification, to understand when Datalog programs with negation are actually meaningful.

Summary

Declarative Programming

- the ***what***, not the ***how***.

Summary

Declarative Programming

- the ***what***, not the ***how***.

Logic programming

- programs as logic constraints: ***facts*** and ***rules***.

Summary

Declarative Programming

- the ***what***, not the ***how***.

Logic programming

- programs as logic constraints: ***facts*** and ***rules***.

Datalog is a simple, yet powerful *declarative logic* programming language.

- Model-Theoretic Semantics
- Fixpoint Semantics
- Stratified Negation