A rule-based query language

- **Syntax:** Rules based on first-order **atoms** based on **terms** (constants or variables) and **predicate symbols**
- **Semantics:** Logical semantics based on first-order logic **entailment** from a **database** viewed as a set of **facts**; therefore **set-based**

**Example:** Recursively finding all ancestors of Alice:

\[
\begin{align*}
\text{Parent}(x, y) & : \neg \text{father}(x, y) \\
\text{Parent}(x, y) & : \neg \text{mother}(x, y) \\
\text{Ancestor}(x, y) & : \neg \text{Parent}(x, y) \\
\text{Ancestor}(x, z) & : \neg \text{Parent}(x, y), \text{Ancestor}(y, z) \\
\text{Result}(y) & : \neg \text{Ancestor}(\text{alice}, y)
\end{align*}
\]
Datalog semantics revisited

A more practical definition of semantics is based on “applying” rules:

**Definition 9.1:** A ground substitution \( \sigma \) is a mapping from variables to constants. Given an atom \( A \), we write \( A\sigma \) for the atom obtained by simultaneously replacing all variables \( x \) in \( A \) with \( \sigma(x) \).
A more practical definition of semantics is based on “applying” rules:

**Definition 9.1:** A ground substitution $\sigma$ is a mapping from variables to constants. Given an atom $A$, we write $A\sigma$ for the atom obtained by simultaneously replacing all variables $x$ in $A$ with $\sigma(x)$.

**Definition 9.2:** The immediate consequence operator $T_P$ maps sets of ground facts $I$ to sets of ground facts $T_P(I)$:

$$T_P(I) = \{ H\sigma \mid H \leftarrow B_1, \ldots, B_n \in P \text{ and } B_1\sigma, \ldots, B_n\sigma \in I \}$$

Given a database $D$, we can define a sequence of databases $D^i$ as follows:

$$D_P^1 = D \quad D_P^{i+1} = D \cup T_P(D_P^i) \quad D_P^\infty = \bigcup_{i \geq 0} D_P^i$$

Observations:

- We obtain an increasing sequence $D_1^P \subseteq D_2^P \subseteq D_3^P \subseteq \ldots \subseteq D^\infty_P$ (why?)
- Ground atom $A$ is entailed by $P \cup D$ if and only if $A \in D^\infty_P$.
- Only a finite number of ground facts can ever be derived from $D \cup P$.
- Hence the sequence $D_1^P, D_2^P, \ldots$ is finite and there is some $k \geq 1$ with $D_k^P = D^\infty_P$. 
A more practical definition of semantics is based on “applying” rules:

**Definition 9.1:** A ground substitution $\sigma$ is a mapping from variables to constants. Given an atom $A$, we write $A\sigma$ for the atom obtained by simultaneously replacing all variables $x$ in $A$ with $\sigma(x)$.

**Definition 9.2:** The immediate consequence operator $T_P$ maps sets of ground facts $I$ to sets of ground facts $T_P(I)$:

$$T_P(I) = \{H\sigma \mid H : - B_1, \ldots, B_n \in P \text{ and } B_1\sigma, \ldots, B_n\sigma \in I\}$$

Given a database $D$, we can define a sequence of databases $D^i$ as follows:

$$D_P^1 = D \quad D_P^{i+1} = D \cup T_P(D_P^i) \quad D_P^\infty = \bigcup_{i \geq 0} D_P^i$$

**Observations:**

- We obtain an increasing sequence $D_P^1 \subseteq D_P^2 \subseteq D_P^3 \subseteq \ldots \subseteq D_P^\infty$ (why?)
- Ground atom $A$ is entailed by $P \cup D$ if and only if $A \in D_P^\infty$.
- Only a finite number of ground facts can ever be derived from $D \cup P$.
- Hence the sequence $D_P^1, D_P^2, \ldots$ is finite and there is some $k \geq 1$ with $D_P^k = D_P^\infty$. 
Datalog assumes that databases are given as sets of (relational) facts.

How to apply Datalog to graph data?

Option 1: Properties as binary predicates
- An RDF triple $s$ p o can be represented by a fact $p(s, o)$
- Both predicate names and constants are IRIs
- Datalog "sees" no relation between properties (predicates) and IRIs in subject and object positions

Option 2: Triples as ternary hyperedges
- An RDF triple $s$ p o can be represented by a fact triple $(s, p, o)$
- triple is the only predicate needed to represent arbitrary databases
- IRIs on any triple position can be related in Datalog
Datalog assumes that databases are given as sets of (relational) facts. How to apply Datalog to graph data?

Option 1: Properties as binary predicates

- An RDF triple $s \ p \ o$ can be represented by a fact $p(s, o)$
- Both predicate names and constants are IRIs
- Datalog “sees” no relation between properties (predicates) and IRIs in subject and object positions

Option 2: Triples as ternary hyperedges

- An RDF triple $s \ p \ o$ can be represented by a fact triple$(s, p, o)$
- triple is the only predicate needed to represent arbitrary databases
- IRIs on any triple position can be related in Datalog
Queries beyond SPARQL

Datalog can express many queries that are not expressible in SPARQL.

Example 9.3: The following query expresses parallel s-t-reachability for predicates \( p \) and \( q \) (for triple encoding):

\[
\text{Reach}(x, y) :\neg \text{triple}(x, p, y), \text{triple}(x, q, y)
\]
\[
\text{Reach}(x, z) :\neg \text{Reach}(x, y), \text{Reach}(y, z)
\]
\[
\text{Result}() :\neg \text{Reach}(s, t)
\]

Note the use of a nullary result predicate: this is a boolean query.

Many other forms of recursion are possible:

- Non-regular (context-free) patterns
- Non-linear (e.g., tree-shaped) patterns
- Recursive pattern definitions (e.g., reachability along path of elements that can reach a specific element via some relation)
Fact 9.4: Datalog query answering is

- ExpTime-complete in combined and query complexity
- P-complete in data complexity

See course “Database Theory” for details and proofs.
Fact 9.4: Datalog query answering is

- ExpTime-complete in combined and query complexity
- P-complete in data complexity

See course “Database Theory” for details and proofs.

As with SPARQL “P in data” does not imply that all P-computable problems can be solved with a Datalog query.

Example 9.5: Datalog is monotonic: the more input facts given, the more results derived. Clearly, there are P problems that are not monotonic, e.g., “Check if there is an even number of triples in the database.”
Negation
Negation enables us to ask for the absence of some data or inference.

**Example 9.6:** SPARQL supports negation in the form of the *NOT EXISTS* filter:

```
SELECT ?person WHERE {
  ?person wdt:P19 wd:Q1731 . # born in Dresden
  FILTER NOT EXISTS { ?person wdt:P570 ?date } # no date of death
}
```

To achieve such expressivity in Datalog, we can add a form of logical negation.

**Example 9.7:** Using negation, a query for living people born in Dresden could be expressed as follows:

```
HasDied(x) :- triple(x, wdt:P570, y)

Result(x) :- triple(x, wdt:P19, wd:Q1731), ¬HasDied(x)
```
A negated ground atom $\neg A$ is true over a database $D$ if $A \notin D$. So we can define:

$$T_P(I) = \{ H\sigma \mid H ::= B_1, \ldots, B_n, \neg A_1, \ldots, \neg A_m \in P,\]

$$B_1\sigma, \ldots, B_n\sigma \in I, \text{ and } A_1\sigma, \ldots, A_m\sigma \notin I\}$$

Example 9.8: What is the meaning of the following rule?

Result $(x) : \neg \text{triple}(x, wdt:p19, wd:q1731), \neg \text{triple}(x, wdt:p570, y)$

Observation: If variables appear only in negated atoms, then it is not clear which values they range over (e.g., which bindings for $y$ should be considered in the rule above?).

Definition 9.9: A rule is safe if all of its variables occur in non-negated atoms in its body. It is common to require all rules to be safe, and this does not restrict expressivity (exercise).
A negated ground atom $\neg A$ is true over a database $D$ if $A \notin D$. So we can define:

$$T_p(I) = \{H\sigma \mid H ::= B_1, \ldots, B_n, \neg A_1, \ldots, \neg A_m \in P,$$

$$B_1\sigma, \ldots, B_n\sigma \in I, \text{ and } A_1\sigma, \ldots, A_m\sigma \notin I\}$$

**Example 9.8:** What is the meaning of the following rule?

$$\text{Result}(x) :\text{=} \neg \text{triple}(x, \text{wdt:P19, wd:Q1731}), \text{}\neg\text{triple}(x, \text{wdt:P570, y})$$
A negated ground atom $\neg A$ is true over a database $D$ if $A \notin D$. So we can define:

$$T_P(I) = \{H\sigma \mid H := B_1, \ldots, B_n, \neg A_1, \ldots, \neg A_m \in P, \\
B_1\sigma, \ldots, B_n\sigma \in I, \text{ and } A_1\sigma, \ldots, A_m\sigma \notin I\}$$

**Example 9.8:** What is the meaning of the following rule?

$$\text{Result}(x) :\neg \text{triple}(x, \text{wdt:P19}, \text{wd:Q1731}), \neg\text{triple}(x, \text{wdt:P570}, y)$$

“Find all $x$, such that $x$ is born in Dresden and there is a date $y$, such that $x$ did not die on $y$."

\[903\]
Semantics of negation (1)

A negated ground atom \( \neg A \) is true over a database \( D \) if \( A \notin D \). So we can define:

\[
T_P(I) = \{ H\sigma \mid H := B_1, \ldots, B_n, \neg A_1, \ldots, \neg A_m \in P, \\
B_1\sigma, \ldots, B_n\sigma \in I, \text{ and } A_1\sigma, \ldots, A_m\sigma \notin I \}
\]

**Example 9.8:** What is the meaning of the following rule?

Result(x) := triple(x, wdt:P19, wd:Q1731), \( \neg \) triple(x, wdt:P570, y)

"Find all x, such that x is born in Dresden and there is a date y, such that x did not die on y."

**Observation:** If variables appear only in negated atoms, then it is not clear which values they range over (e.g., which bindings for y should be considered in the rule above?).

**Definition 9.9:** A rule is **safe** if all of its variables occur in non-negated atoms in its body.

It is common to require all rules to be safe, and this does not restrict expressivity (exercise).
The unrestricted use of negation in recursive queries leads to semantic problems:

**Example 9.10:** The following facts and query model a stereotypical gender-binary world view:

<table>
<thead>
<tr>
<th>Human</th>
<th>Male</th>
<th>Female</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evelyn</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Jo</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

\[
\text{Male}(x) : - \text{human}(x), \neg \text{Female}(x) \\
\text{Female}(x) : - \text{human}(x), \neg \text{Male}(x)
\]

What should be the result if Female were the query predicate?
Semantics of negation (2)

The unrestricted use of negation in recursive queries leads to semantic problems:

**Example 9.10:** The following facts and query model a stereotypical gender-binary world view:

```
human(evelyn)  human(jo)

Male(x) :- human(x), ¬Female(x)
Female(x) :- human(x), ¬Male(x)
```

What should be the result if Female were the query predicate?

If we define the sequence $D^i_p$ as before, we obtain:

- $D^1_p = D = \{\text{human(evelyn)}, \text{human(jo)}\}$
- $D^2_p = D \cup T_p(D^1_p) = D \cup \{\text{Male(evelyn)}, \text{Female(evelyn)}, \text{Male(jo)}, \text{Female(jo)}\}$
- $D^3_p = D \cup T_p(D^2_p) = D^1_p$
- $D^4_p = D \cup T_p(D^3_p) = D^2_p = D^\infty$

$\leadsto$ non-monotonic behaviour leads to unfounded conclusions (e.g., that all humans are both male and female)
Stratified negation

**Observation:** Iterative evaluation of rules fails if negation is freely used in recursion

- Initially, when no facts were derived, many negated atoms are true
- However, these initially true atoms can become false when more inferences are computed

Definition 9.11:
Let $P$ be a set of rules with negation. A function $\ell(p)$ that assigns a natural number $\ell(p)$ to every predicate $p$ is a stratification of $P$ if the following are true for every rule $h(t)$:

1. $\ell(h) \geq \ell(p_i)$ for all $i \in \{1, \ldots, n\}$
2. $\ell(h) > \ell(q_i)$ for all $i \in \{1, \ldots, m\}$

Intuition: The function $\ell$ defines the "level" of the rule. By applying rules exhaustively level-by-level, we can avoid non-monotonic behaviour.
Stratified negation

**Observation**: Iterative evaluation of rules fails if negation is freely used in recursion

- Initially, when no facts were derived, many negated atoms are true
- However, these initially true atoms can become false when more inferences are computed

To avoid recursion through negation, one can try to organise rules in “layers” or “strata”:

**Definition 9.11**: Let \( P \) be a set of rules with negation. A function \( \ell \) that assigns a natural number \( \ell(p) \) to every predicate \( p \) is a stratification of \( P \) if the following are true for every rule \( h(t) : \neg p_1(s_1), \ldots, p_n(s_n), \neg q_1(r_1), \ldots, \neg p_m(r_m) \in P \):

1. \( \ell(h) \geq \ell(p_i) \) for all \( i \in \{1, \ldots, n\} \)
2. \( \ell(h) > \ell(q_i) \) for all \( i \in \{1, \ldots, m\} \)

**Intuition**: The function \( \ell \) defines the “level” of the rule. By applying rules exhaustively level-by-level, we can avoid non-monotonic behaviour.
Evaluating stratified rules

**Evaluation of stratified programs:** Let $D$ be a database and let $P$ be a program with stratification $\ell$, with values of $\ell$ ranging from 1 to $m$ (without loss of generality).

- For $i \in \{1, \ldots, m\}$, we define sub-programs for each stratum:
  \[
P_i = \{h(t) : p_1(s_1), \ldots, p_n(s_n), \neg q_1(r_1), \ldots, \neg p_m(r_m) \in P \mid \ell(h) = i\}\]
- Define $D_0^\infty = D$
- Now for $i = 1, \ldots, m$, we define:
  - $D_i^1 = D_{i-1}^\infty$
  - $D_i^{j+1} = D_{i-1}^\infty \cup T_{P_i}(D_i^j)$
  - $D_i^\infty = \bigcup_{j \geq 1} D_i^j$ is the limit of this process
- The evaluation of $P$ over $D$ is $D_m^\infty$. 

Observations:
- For every $i$, the sequence $D_1^i \subseteq D_2^i \subseteq \ldots$ is increasing, since facts relevant for negated body literals are not produced in any $D_j^i$ (due to stratification)
- Such increasing sequences must be finite (since the set of all possible facts is finite)
- The limits $D_i^\infty$ are computed after finitely many steps
Evaluating stratified rules

**Evaluation of stratified programs:** Let $D$ be a database and let $P$ be a program with stratification $\ell$, with values of $\ell$ ranging from 1 to $m$ (without loss of generality).

- For $i \in \{1, \ldots, m\}$, we define sub-programs for each stratum:
  $P_i = \{h(t) :- p_1(s_1), \ldots, p_n(s_n), \neg q_1(r_1), \ldots, \neg p_m(r_m) \in P \mid \ell(h) = i\}$

- Define $D_0^\infty = D$

- Now for $i = 1, \ldots, m$, we define:
  - $D^1_i = D^\infty_{i-1}$
  - $D^{i+1}_i = D^\infty_{i-1} \cup T_{P_i}(D^i_i)$
  - $D^\infty_i = \bigcup_{j \geq 1} D^i_j$ is the limit of this process

- The evaluation of $P$ over $D$ is $D^\infty_m$.

**Observations:**

- For every $i$, the sequence $D^1_i \subseteq D^2_i \subseteq \ldots$ is increasing, since facts relevant for negated body literals are not produced in any $D^i_i$ (due to stratification)

- Such increasing sequences must be finite (since the set of all possible facts is finite)

$\leadsto$ The limits $D^\infty_i$ are computed after finitely many steps
The perfect model

**Summary:** The stratified evaluation of rules terminates after finitely many steps (bounded by the number of possible facts)

What is the set of facts that we obtain from this procedure?
The perfect model

Summary: The stratified evaluation of rules terminates after finitely many steps (bounded by the number of possible facts)

What is the set of facts that we obtain from this procedure?

Fact 9.12: For a database $D$ and stratified program $P$, the set of facts $M$ that is obtained by the stratified evaluation procedure is the least set of facts with the property that

$$M = D \cup T_P(M).$$

In particular, $M$ does not depend on the stratification that was chosen.

$M$ is called perfect model or unique stable model in logic programming.

Intuition: The stratified evaluation is the smallest set of self-supporting true facts that can be derived

- This is not the set of inferences under classical logical semantics! (exercise)
- But it is a good extension of negation in queries to the recursive setting.
Obtaining a stratification

To find a stratification, the following algorithm can be used:

**Input:** program $P$

- Construct a directed graph with two types of edges, $\rightarrow^+$ and $\rightarrow^-$:
  - The vertices are the predicate symbols in $P$
  - $p \rightarrow^+ q$ if there is a rule with $p$ in its non-negated body and $q$ in the head
  - $p \rightarrow^- q$ if there is a rule with $p$ in its negated body and $q$ in the head
- Then $P$ is stratified if and only if the graph contains no directed cycle that involves an edge $\rightarrow^-$
- In this case, we can obtain a stratification as follows:
  1. produce a topological order of the strongly connected components of this directed graph (without distinguishing edge types), e.g., using Tarjan’s algorithm
  2. assign numerical strata bottom-up to all rules with head predicates in each component
Stratified negation is usually sufficient for query answering.
Non-stratified negation is relevant in optimisation and constraint solving.
Outlook: Beyond stratified negation

Stratified negation is usually sufficient for query answering. Non-stratified negation is relevant in optimisation and constraint solving.

Handling non-stratified negation:

- Recursion through negation gives rise to multiple alternative interpretations
- Semantics can be defined in many ways, e.g., stable models (answer set programming), well-founded semantics, and classical semantics
- See various other courses (e.g., “Problem Solving and Search in AI”)
Outlook: Beyond stratified negation

Stratified negation is usually sufficient for query answering.
Non-stratified negation is relevant in optimisation and constraint solving.

Handling non-stratified negation:

- Recursion through negation gives rise to multiple alternative interpretations
- Semantics can be defined in many ways, e.g., stable models (answer set programming), well-founded semantics, and classical semantics
- See various other courses (e.g., “Problem Solving and Search in AI”)

Stratified negation allows us to express non-monotonic queries.
However, not all polynomial-time queries are expressible.
Outlook: Beyond stratified negation

Stratified negation is usually sufficient for query answering. Non-stratified negation is relevant in optimisation and constraint solving.

Handling non-stratified negation:

- Recursion through negation gives rise to multiple alternative interpretations
- Semantics can be defined in many ways, e.g., stable models (answer set programming), well-founded semantics, and classical semantics
- See various other courses (e.g., “Problem Solving and Search in AI”)

Stratified negation allows us to express non-monotonic queries. However, not all polynomial-time queries are expressible.

Capturing PTime:

- To express all polytime queries, in addition to stratified negation, Datalog needs a total order on the domain (defined by special predicates)
- See course “Database Theory” for details
Comparing Datalog and SPARQL
Supported SPARQL features

Datalog with stratified negation captures and extends important parts of SPARQL:

- **Basic Graph Patterns**: are simply conjunctions of triple-atoms
- **Path expressions**: Datalog does not support paths syntactically, but they can be captured in Datalog
- **Union**: disjunction can be expressed in Datalog using several rules (exercise)
- **Minus and Not Exists**: can be expressed with stratified negation in Datalog
- **Values**: can be declared by Datalog facts

**Recall**: Datalog always assumes set semantics (Distinct in SPARQL)

**Example 9.13**: The following rules are an alternative to express the property path pattern `eg:JSBach (ˆeg:hasFather|ˆeg:hasMother)+ ?x:

\[
\text{Result}(x) \leftarrow \text{triple}(x, \text{eg:hasFather}, \text{eg:JSBach})
\]
\[
\text{Result}(x) \leftarrow \text{triple}(x, \text{eg:hasMother}, \text{eg:JSBach})
\]
\[
\text{Result}(x) \leftarrow \text{Result}(y), \text{triple}(x, \text{eg:hasFather}, y)
\]
\[
\text{Result}(x) \leftarrow \text{Result}(y), \text{triple}(x, \text{eg:hasMother}, y)
\]
Many other SPARQL features are not part of Datalog:

- **Filters**: filter conditions (and datatypes) are not part of the pure logical definition of Datalog, but can easily be added as built-in predicates.
- **Bind**: computed functions are not usually found in Datalog but can be added.
- **Optional**: Datalog (and logic in general) does not have a direct way to handle partial result mappings, and there is no equivalent to Optional.
- **Aggregates**: Datalog does not support aggregates, as they introduce non-monotonic behaviour in general; Datalog extensions with restricted aggregation exist.
- **Subqueries**: Datalog cannot express nested limit/offset/order by.
Datalog in practice
Implementations of Datalog

Many implementations of Datalog exist:

• In-Memory systems for query answering and data analysis: Graal, RDFox, Vadalog, VLog4j, . . .
• Answer set programming engines: Clingo, DLV(2), . . .
• Logic programming engines: Prolog implementations
• Database-backed (business) rule engines

→ many use cases; many different implementation approaches
Implementations of Datalog

Many implementations of Datalog exist:

- In-Memory systems for query answering and data analysis: Graal, RDFox, Vadalog, VLog4j, . . .
- Answer set programming engines: Clingo, DLV(2), . . .
- Logic programming engines: Prolog implementations
- Database-backed (business) rule engines

→ many use cases; many different implementation approaches

Compatibility with knowledge graph formats:

- Typically support for RDF and related technologies (IRI, datatypes)
- Most common for in-memory systems: Graal, RDFox, and VLog4j support RDF
- VLog4j also supports SPARQL as source of external data
Rules in VLog4j

VLog4j is a free rule engine that supports extensions of Datalog:

- Download, documentation, and source code online: https://github.com/knowsys/vlog4j
- Java library and command-line client based on the VLog rule engine (C++)
- Support for evaluating Datlog queries over RDF files and SPARQL query results
- Stratified negation and value invention (existential rules)

**Example 9.14:** VLog4j uses a textual syntax for rules, which is slightly different from the one we used so far. Variables are marked by ?, negation is written as ~, and rules end with a full stop:

```
Parent(?x,?y) :- father(?x,?y).
Parent(?x,?y) :- mother(?x,?y).
Ancestor(?x,?y) :- Parent(?x,?y).
Ancestor(?x,?z) :- Parent(?x,?y), Ancestor(?y,?z).
Result(?y) :- Ancestor(alice,?y), ~profession(?y,composer).
```
Facts in VLog4j can be specified as part of the rules, loaded from files (RDF or CSV), or loaded from SPARQL query services. RDF terms and prefixes can be used.

The following example evaluates data from Wikidata with simple rules:

```reason
@prefix wdqs: <https://query.wikidata.org/> .
@source mother[2] : sparql(wdqs:sparql, "child,mother",

% Rules to find maternal ancestors:
% Query for maternal ancestors of Ada Lovelace
Result(?X) :- matAnc(<http://www.wikidata.org/entity/Q7259>, ?X) .
```
Value invention
Existential rules

Especially in data integration, we often need to structurally transform graphs to match each other's encoding.

**Example 9.15:** We have seen previously that the relationship between films, actors, and roles can be encoded using three RDF properties. Now given new data in the form of a table with three columns (e.g., from a CSV file), we would like to transform it into this RDF format.
Existential rules

Especially in data integration, we often need to structurally transform graphs to match each others encoding

**Example 9.15:** We have seen previously that the relationship between films, actors, and roles can be encoded using three RDF properties. Now given new data in the form of a table with three columns (e.g., from a CSV file), we would like to transform it into this RDF format.

**Problem:** encodings like reification require new RDF resources to be created, but Datalog rules can only infer relations about existing elements
Existential rules

Especially in data integration, we often need to structurally transform graphs to match each others encoding.

**Example 9.15:** We have seen previously that the relationship between films, actors, and roles can be encoded using three RDF properties. Now given new data in the form of a table with three columns (e.g., from a CSV file), we would like to transform it into this RDF format.

**Problem:** encodings like reification require new RDF resources to be created, but Datalog rules can only infer relations about existing elements.

**Solution:** use existential quantifiers in rules that “invent new values”
Existential rules syntax

In logic, this is written with the usual existential quantifiers:

**Example 9.16:** The next rule creates a reified graph structure from ternary film-actor-role relations:

\[ \exists v. \text{actorRole}(x, v), \text{actor}(v, y), \text{role}(v, z) :\neg \text{cast}(x, y, z) \]

The \& in the head denotes conjunction.

**Note:** In Datalog, conjunctions in the head are redundant (one can use multiple rules instead), but they are needed here.
Existential rules syntax

In logic, this is written with the usual existential quantifiers:

**Example 9.16:** The next rule creates a reified graph structure from ternary film-actor-role relations:

\[ \exists v. \text{actorRole}(x, v), \text{actor}(v, y), \text{role}(v, z) :\neg \text{cast}(x, y, z) \]

The `, in the head denotes conjunction.

**Note:** In Datalog, conjunctions in the head are redundant (one can use multiple rules instead), but they are needed here.

VLog4j uses `!` to mark existentially quantified head variables:

\[ \text{actorRole}(?x,!v), \text{actor}(!v,?y), \text{role}(!v,?z) :- \text{cast}(?x,?y,?z) . \]

**Note:** In databases, existential rules are known as tuple-generating dependencies.
Existential rules semantics

Mathematically, the meaning of existentially quantified variables is clear: There must be some element that stands in all required relations.

In practice, we need to create new objects when applying rules – we have several options:

- **Oblivious**: create new objects whenever a rule is applied in a new way
- **Restricted**: create new objects only if rule head is not satisfied by existing objects
- Other variants exist.

**Note**: The $T_P$-style bottom-up computation of consequences for existential rules is called chase.
Chase termination

When new objects are invented, termination can no longer be guaranteed.

**Example 9.17:** The following rule set would lead to the creation of infinitely many new elements. The chase would not terminate.

Person(alice).

hasMother(?x,!v), Person(!v) :- Person(?x).

Dealing with potential non-termination:

- Unfortunately, it is undecidable if the chase will terminate on some input.
- We can avoid problems by avoiding recursion in rules with existentials altogether.
- Smarter conditions exist (starting from stratification-like conditions).

See course "Database Theory" for more information.
When new objects are invented, termination can no longer be guaranteed

**Example 9.17:** The following rule set would lead to the creation of infinitely many new elements. The chase would not terminate.

Person(alice).
hasMother(?x, !v), Person(!v) :- Person(?x).

**Dealing with potential non-termination:**

- Unfortunately, it is undecidable if the chase will terminate on some input
- We can avoid problems by avoiding recursion in rules with existentials altogether
- Smarter conditions exist (starting from stratification-like conditions)

See course “Database Theory” for more information.
Summary

Stratified negation is a simple way of adding negation to recursive queries

Datalog can capture and extend many basic features of SPARQL

SPARQL features not in Datalog include many datatypes, aggregates, optional, and multiset semantics

VLog4j is a free RDF-compatible rule engine

What’s next?
- Property graph
- The Cypher query language
- Knowledge Graph quality