MI-PDB, MIE-PDB: **Advanced Database Systems**

http://www.ksi.mff.cuni.cz/~svoboda/courses/2015-2-MIE-PDB/

Lecture 11:

RDF, SPARQL

3. 5. 2016

Lecturer: **Martin Svoboda**
svoboda@ksi.mff.cuni.cz

Author: **Martin Svoboda**
Faculty of Mathematics and Physics, Charles University in Prague
Course NSWI144: **Linked Data**

RDF

Resource Description Framework

Introduction

- RDF
  - **Resource Description Framework**
    - Language for representing information about resources in the World Wide Web
  - W3C recommendations
    - Concepts and abstract syntax
      - http://www.w3.org/TR/rdf-concepts/
    - Semantics
      - http://www.w3.org/TR/rdf-mt/
    - …

Statements

- RDF
  - Idea: statements about resources
  - **Resource**
    - Anything that is identifiable by an IRI
      - Usually things identified by standard URLs…
      - … but also things that may not be directly retrievable
  - **Statements**
    - Triples inspired by natural languages
    - **Subject Predicate Object**
      - `http://example.cz/is#student358`
        `http://example.cz/is#name`
        `"John"`

Statements

- Components of triples
  - **Subject**
    - Describes the thing the statement is about
  - **Predicate**
    - Describes the property or characteristic of the subject
  - **Object**
    - Describes the value of that property

Statements

- Resource identifiers
  - **IRIs**
    - IRI with an optional fragment identifier
      - `http://example.cz/is#student358`
      - `mailto:svoboda@ksi.mff.cuni.cz`
      - `urn:issn:0167-6423`
    - Unicode characters
  - **Qualified names**
    - Similar idea as prefixes for namespaces in XML
      - `ex: = http://example.cz/is#`
      - `ex:student358`

Statements

- Domains for triple components
  - **Identifiers**
    - **IRIs**
    - **Blank node identifiers**
      - Special and only locally valid identifiers (within a file etc.)
      - `_:a185`
  - **Literals**
    - Plain or typed values with or without language tags
      - `"John"`
      - `"John"^^xsd:string`
      - `"Praha"@cs`
      - `"Prague"@en`

Statements

- Allowed structure of triples
  - **Subject**
    - *IRI* or *blank node identifier*
  - **Predicate**
    - *IRI*
  - **Object**
    - *IRI* or *blank node identifier* or *literal*

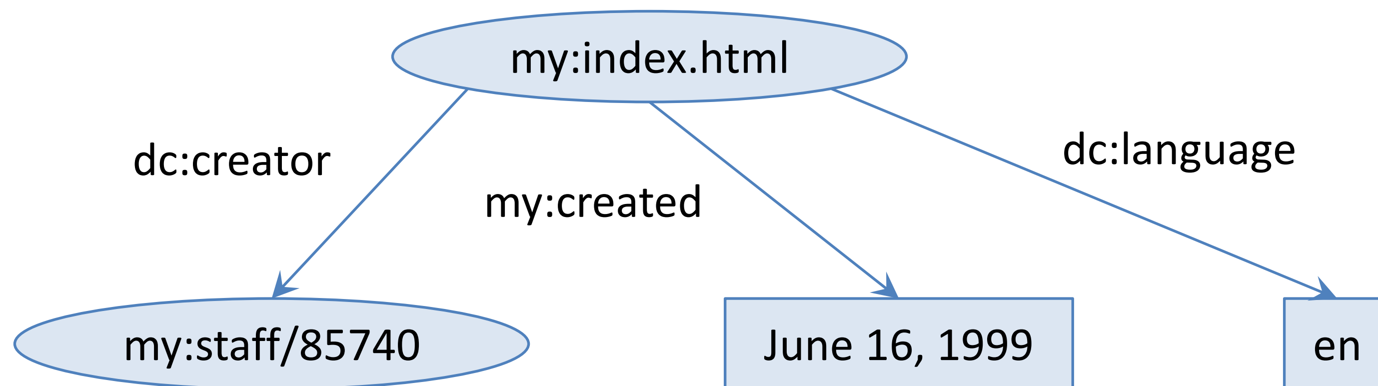
Data Model

- RDF data model
  - **Directed labeled multigraph**
    - Vertices for subjects and objects
    - Labeled directed edges for particular triples

Data Model

- Example

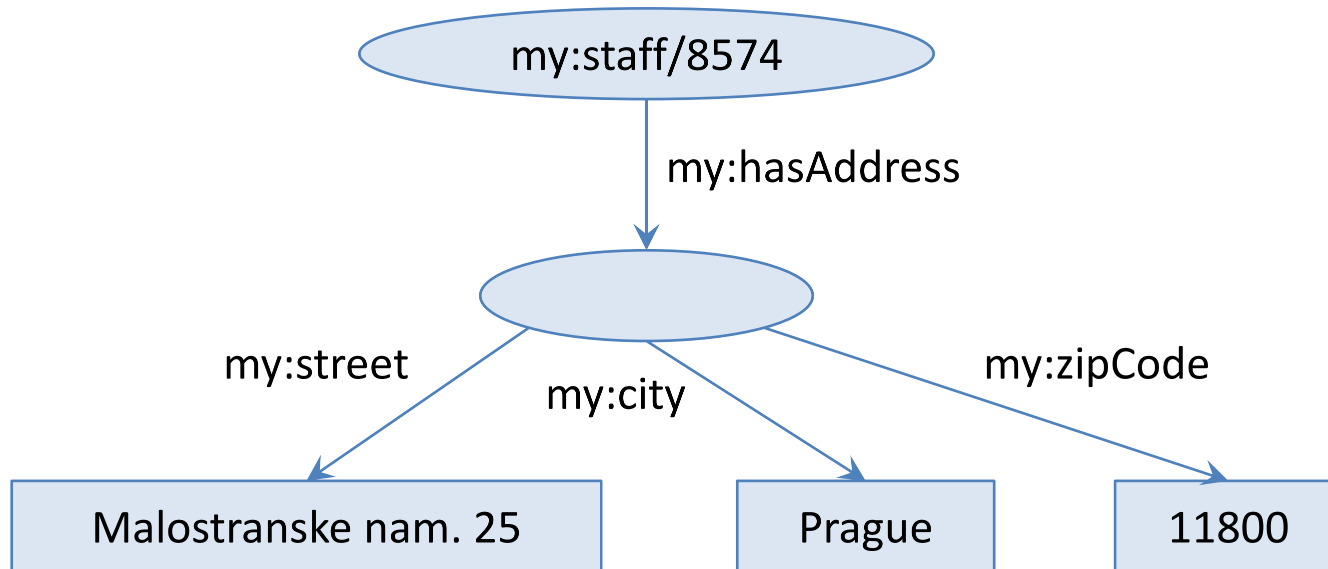
```
my:index.html dc:creator my:staff/85740 .
my:index.html my:created "June 16, 1999" .
my:index.html dc:language "en" .
```



Blank Nodes

- Example

```
my:staff/8574 my:hasAddress _:a185 .
_:a185 my:street "Malostranske nam. 25" .
_:a185 my:city "Prague" .
_:a185 my:zipCode "11800" .
```



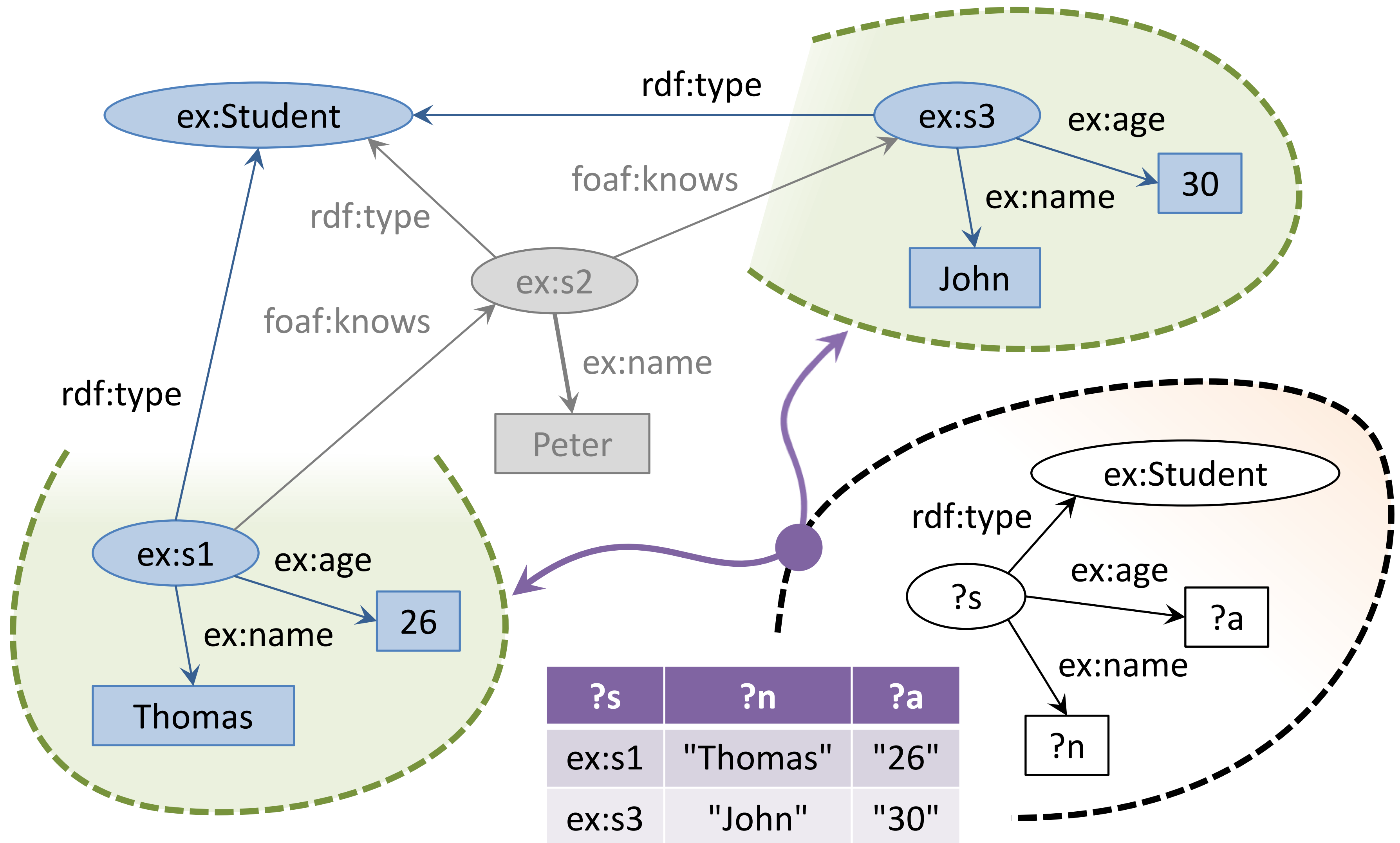
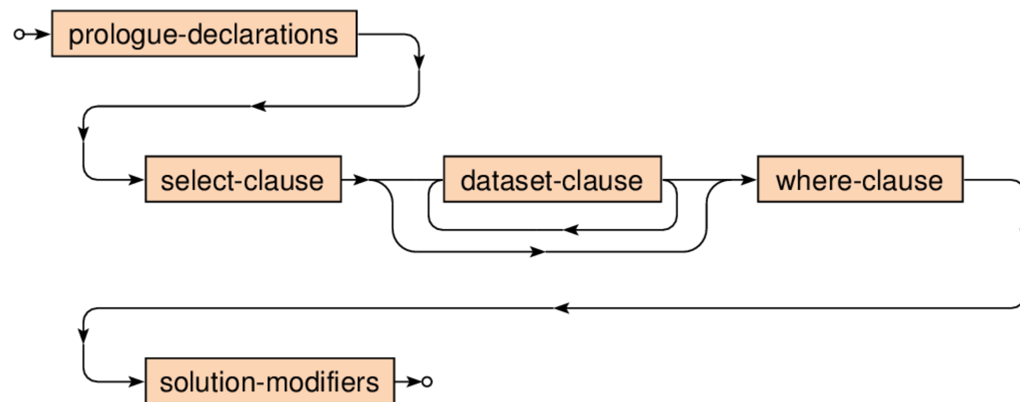
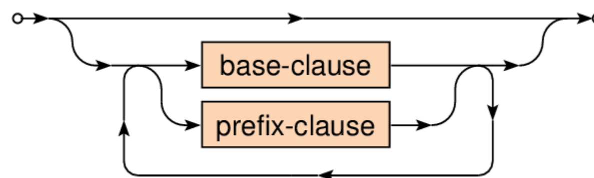
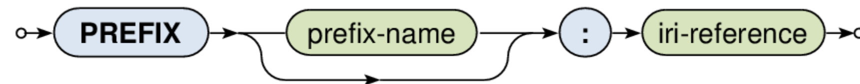
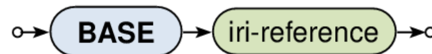
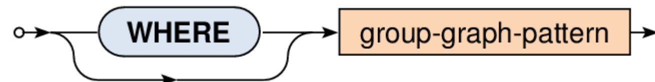
SPARQL

Query Language for RDF

Introduction

- **SPARQL Protocol and RDF Query Language**
  - RDF query language
    - Required and optional graph patterns, their conjunctions and disjunctions, subqueries, negation, aggregation, value constructors, …
  - W3C recommendations
    - **Version 1.0** (2008)
      - https://www.w3.org/TR/rdf-sparql-query/
    - **Version 1.1** (2013)
      - 11 recommendations: query language, update facility, federated queries, protocol, result formats, …
      - https://www.w3.org/TR/sparql11-query/

Sample Data

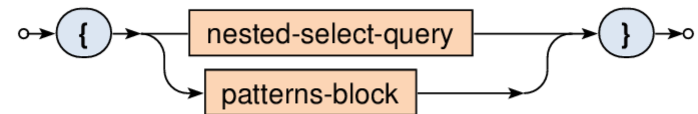
- Data
  - ```
    @prefix ex: <http://example.cz/is#> .
    @prefix foaf: <http://xmlns.com/foaf/0.1/> .
    ex:s1 rdf:type ex:Student ;
          ex:name "Thomas" ;
          ex:age "26" .
    ex:s2 rdf:type ex:Student ;
          ex:name "Peter" .
    ex:s3 rdf:type ex:Student ;
          ex:name "John" ;
          ex:age "30" .
    ex:s1 foaf:knows ex:s2 .
    ex:s2 foaf:knows ex:s3 .
    ```
    ```

Sample Query

- Query expression

```
PREFIX ex: <http://example.cz/is#>
SELECT ?n ?a
WHERE {
  ?s rdf:type ex:Student ;
     ex:name ?n ;
     ex:age ?a .
}
```

- Query result

?n	?a
"Thomas"	"26"
"John"	"30"

Graph Pattern Matching

- **Graph pattern expressions**
  - Based on ordinary triples
    - **Subject, predicate and object** components
    - IRI references, blank nodes, literals and **variables**
      - `?name` or `$name`
  - We are attempting to find **subgraphs of the data graph that are matched by the query patterns**
    - Based on **substitution of variables**
    - However, SPARQL is not just a simple graph matching!

Graph Pattern Matching



?s	?n	?a
ex:s1	"Thomas"	"26"
ex:s3	"John"	"30"

Graph Pattern Matching

- Graph pattern expressions
  - **Basic graph pattern** as a set of triples
  - … and other more complex patterns
    - E.g. group, optional, …
- How the matching works?
  - **Basic graph pattern matches a subgraph** of the RDF data graph **when terms** from that subgraph **may be substituted for the variables** and the result is RDF graph equivalent to the subgraph

Graph Pattern Matching

- Equivalency of **literals**
  - **Language tags**
    - When tags are specified, they must be identical
      - `"Praha"`
      - `"Praha"@cs`
      - `"Prague"@en`
  - **Typed literals**
    - When types are specified, they must be identical
    - Shortcuts available for literals of common types…
      - `1 = "1"^^xsd:integer`
      - `1.5 = "1.5"^^xsd:decimal`
      - `true = "true"^^xsd:boolean`

Graph Pattern Matching

- Equivalency of **blank nodes**
  - … in a data graph
    - Distinct nodes within a given document scope
  - … in a query pattern
    - Blank nodes act as non-selectable variables
      - **Blank node labels in a query expression cannot be expected to correspond to blank nodes from a source data graph!**
  - … in a query result
    - Distinct nodes within a given result scope
      - **Blank node labels in a query result may not correspond to blank nodes from the source graph and nor the query!**

Query Result

- **Variable binding**
  - `(?n, "Thomas")`

?n
"Thomas"

- **Solution**
  - Set of variable bindings
    - Represents one way how query variables can be substituted
      - Not all variables need to be bound!
    - `{ (?n, "Thomas"), (?a, "26") }`

?n	?a
"Thomas"	"26"

- **Solution sequence**
  - Ordered multiset of solutions
    - `{ (?n, "Thomas"), (?a, "26") },`
      `{ (?n, "John"), (?a, "30") }`

?n	?a
"Thomas"	"26"
"John"	"30"

Query Result

- **Solution compatibility**
  - Two solutions are mutually compatible if and only if for each variable they share both the corresponding variable bindings are equivalent
  - Examples
    - `{ (?s, ex:s1), (?a, "26") }`
      and
      `{ (?s, ex:s1), (?n, "Thomas") }`
      are compatible solutions
    - `{ (?s, ex:s1), (?n, "Thomas"), (?a, "26") }`
      and
      `{ (?s, ex:s1), (?n, "John") }`
      are not compatible solutions (since there is at least one conflict)

Select Queries

- **Basic query structure**
  - Optional prologue declarations
    - PREFIX, BASE
  - Selection clauses
    - **SELECT** – list of output variables
    - **FROM** – dataset to be queried
    - **WHERE** – required graph pattern and filtering expressions
  - Optional solution modifiers
    - GROUP BY, HAVING
    - ORDER BY, LIMIT, OFFSET

Select Queries

- **Basic query structure**



  - Both the prologue declarations and solution modifiers are optional

- Prologue declarations: **BASE** and **PREFIX** clauses



Prologue Declarations

- **PREFIX** clause
  - Definition of prefix labels for IRI references



  - Example
    - `PREFIX my: <http://example.cz/>`
    - Then a prefixed name `my:x` corresponds to `<http://example.cz/x>`

Prologue Declarations

- **BASE** clause
  - Usage of relative IRI references



  - Example
    - `BASE <http://example.cz/>`
    - Then a relative IRI `<x>` corresponds to `<http://example.cz/x>`

Graph Patterns

- **WHERE** clause



  - Graph patterns
    - **Basic**… when a set of triple patterns must all match
    - **Group**… when a set of graph patterns must all match
    - **Optional**… when additional patterns may extend solutions
    - **Alternative**… when two or more possible patterns are tried
    - **Graph**… when a particular dataset should be queried
  - Inductive construction
    - Patterns can be combined into more complex ones

Graph Patterns

- **Basic** graph pattern
  - **... when a set of triple patterns must all match**
  - Syntax
    - Ordinary **triples** separated by dots
    - … and their abbreviated forms inspired by Turtle notation
      - Object lists using `,`
      - Predicate-object lists using `;`
      - Blank nodes using `[]`
      - Collections using `()`
  - Examples
    - `s p1 o1 . s p1 o2 . s p2 o3`
    - `s p1 o1 , o2 ; p2 o3`

Graph Patterns

- Example
  - ```
    PREFIX ex: <http://example.cz/is#>
    SELECT ?n ?a
    WHERE {
      ?s rdf:type ex:Student .
      ?s ex:age ?a .
      ?s ex:name ?n .
    }
    ```

?s
ex:s1
ex:s2
ex:s3

?s	?a
ex:s1	"26"
ex:s3	"30"

?s	?n
ex:s1	"Thomas"
ex:s2	"Peter"
ex:s3	"John"

?s	?n	?a
ex:s1	"Thomas"	"26"
ex:s3	"John"	"30"
    ```

Graph Patterns

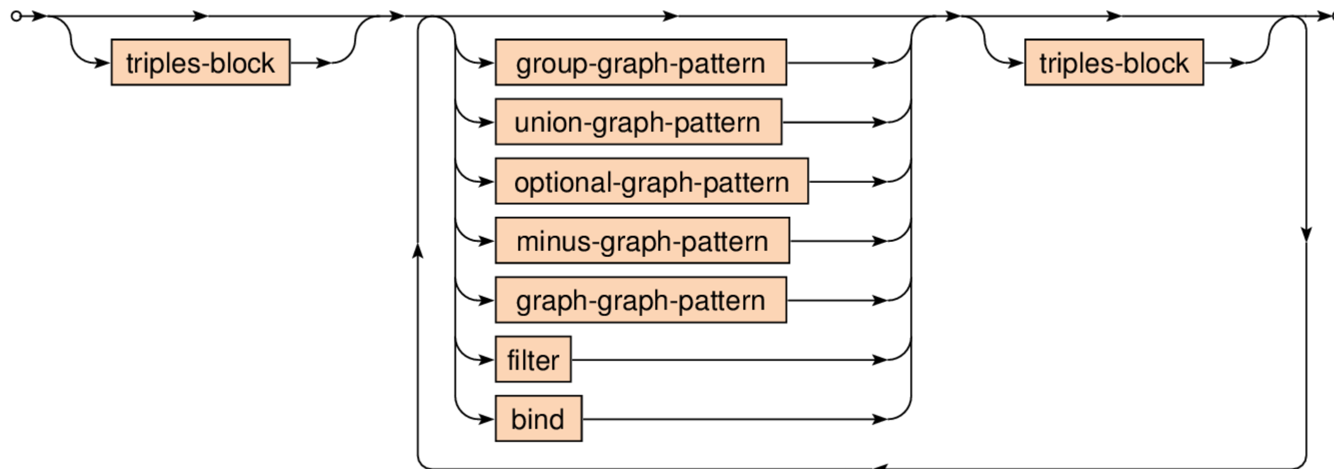
- **Basic** graph pattern
  - Interpretation
    - **All the involved triple patterns must match**
    - I.e. we combine them as if they were in **conjunction**
    - Note that all variables need to be bound
      - I.e. if any of the involved variables cannot be bound, then the entire basic graph pattern cannot be matched!

Graph Patterns

- **Group** graph pattern
  - **… when a set of graph patterns must all match**
  - Syntax – two alternatives:



  - Nested select query
    - SELECT clause, WHERE clause, and solution modifiers
  - **Block with a set of graph patterns**:

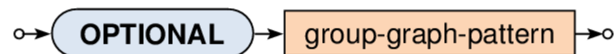


Graph Patterns

- **Group** graph pattern
  - Notes
    - Empty group patterns `{ }` are also allowed
  - Interpretation
    - **All the involved patterns must match**
    - I.e. we combine them as if they were in **conjunction**

Graph Patterns

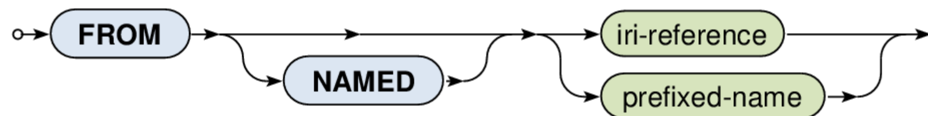
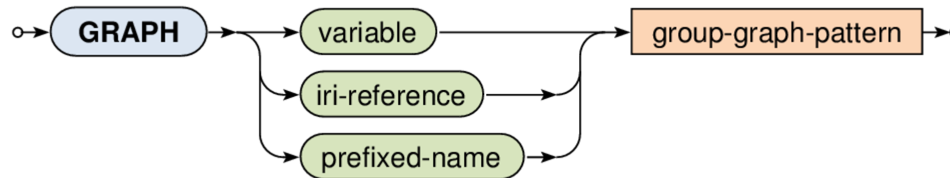
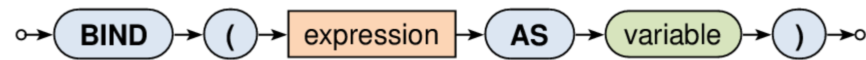
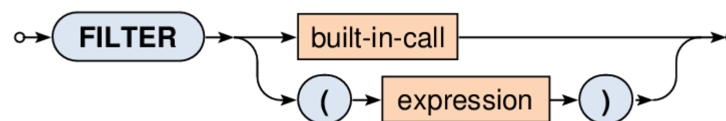
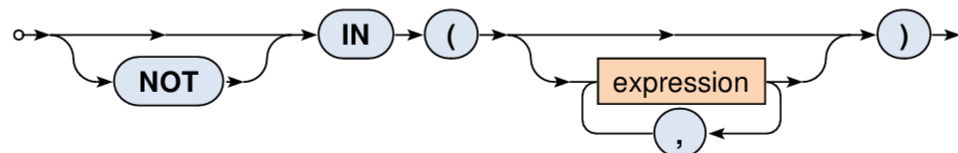
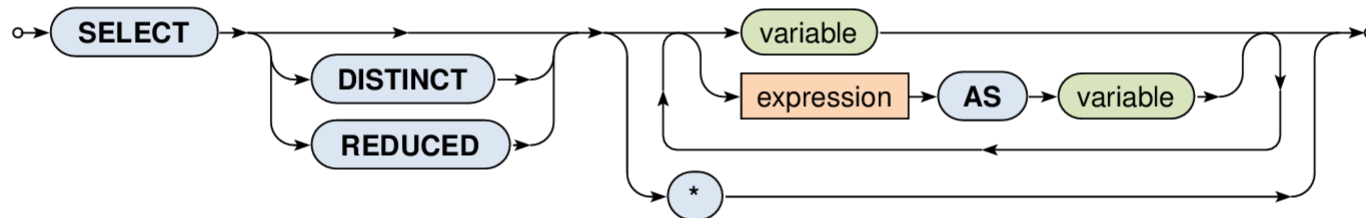
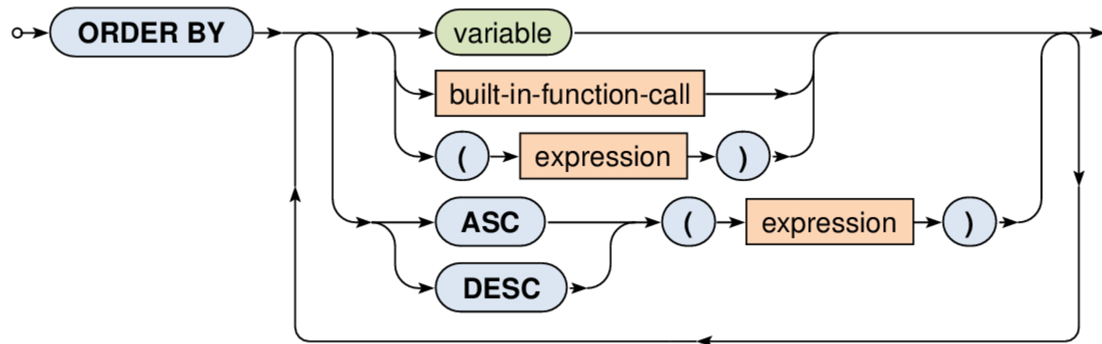
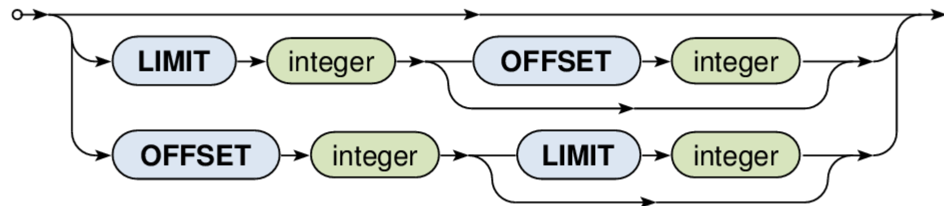
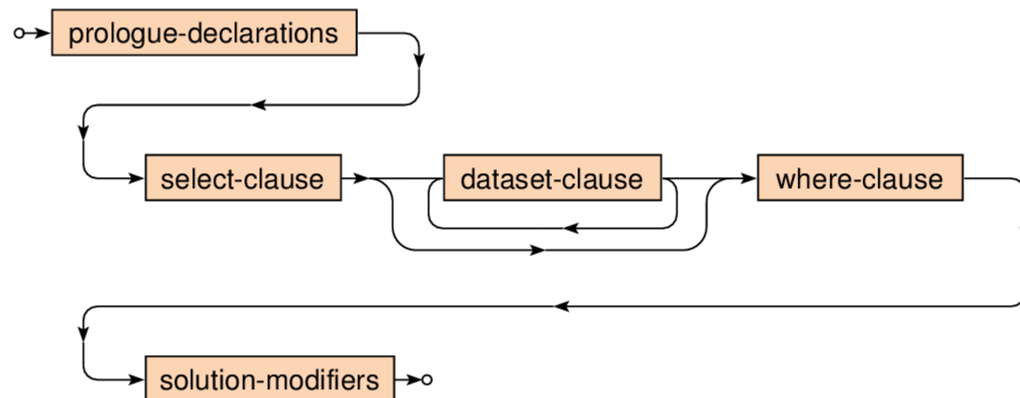
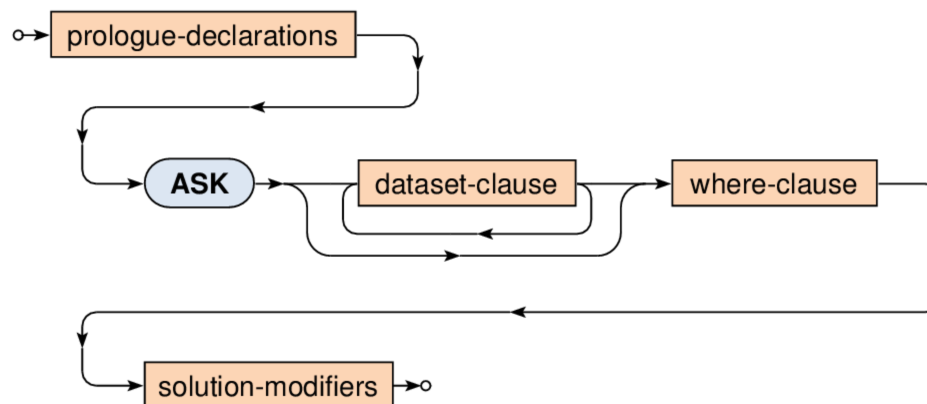
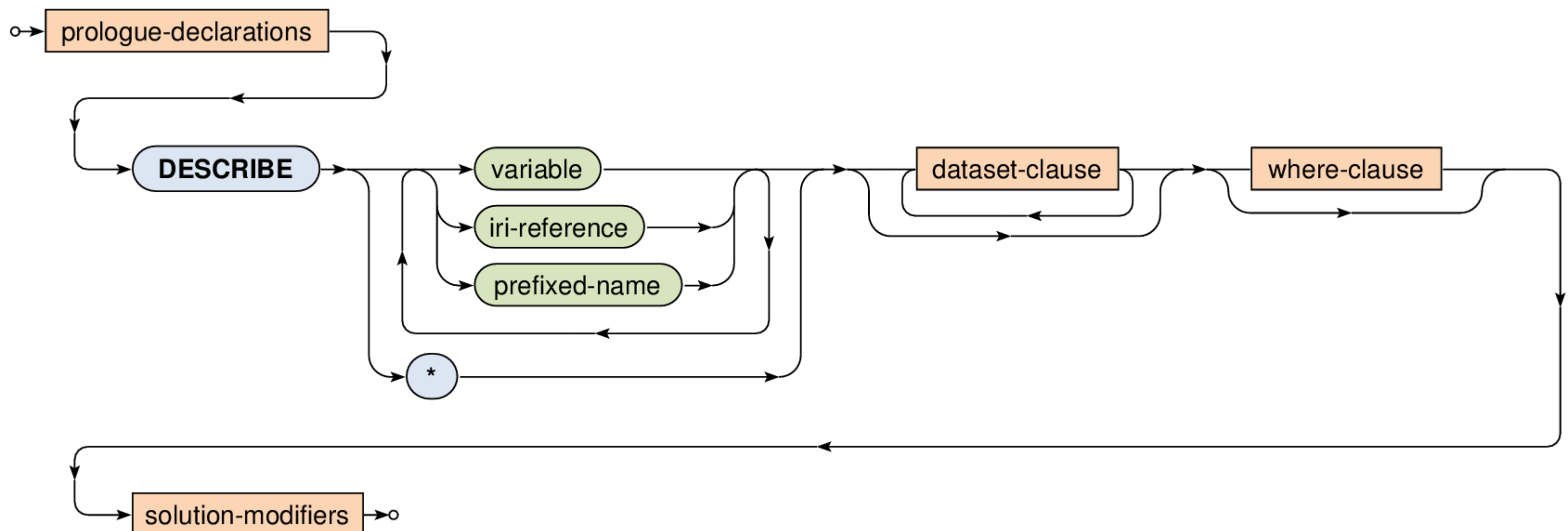
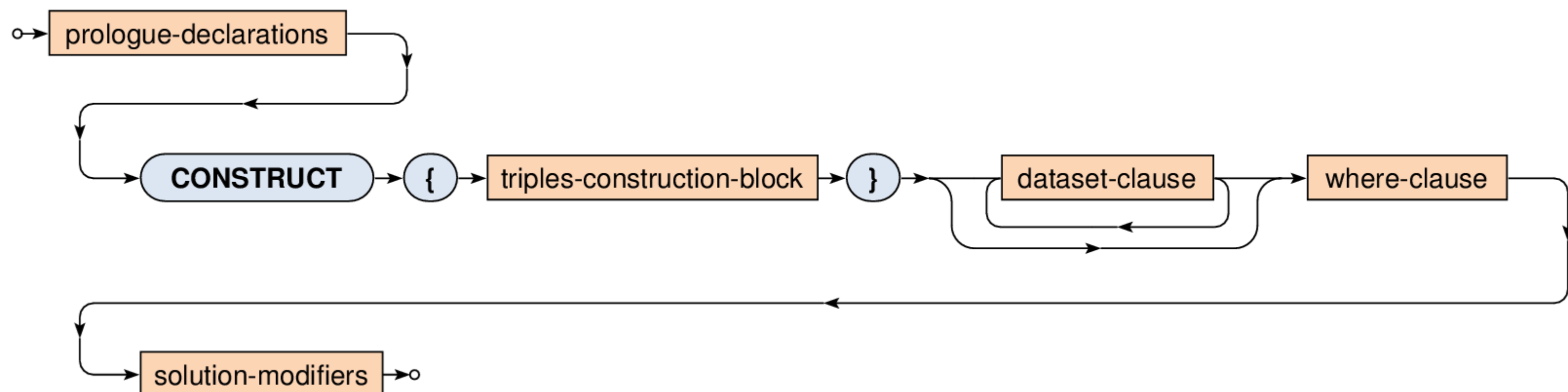
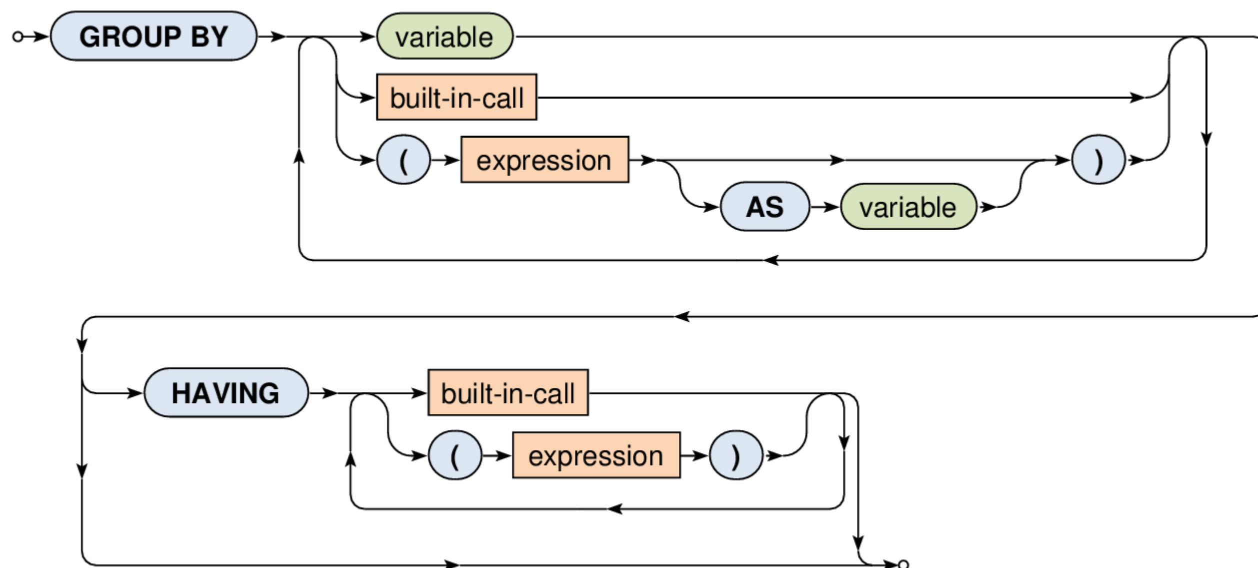
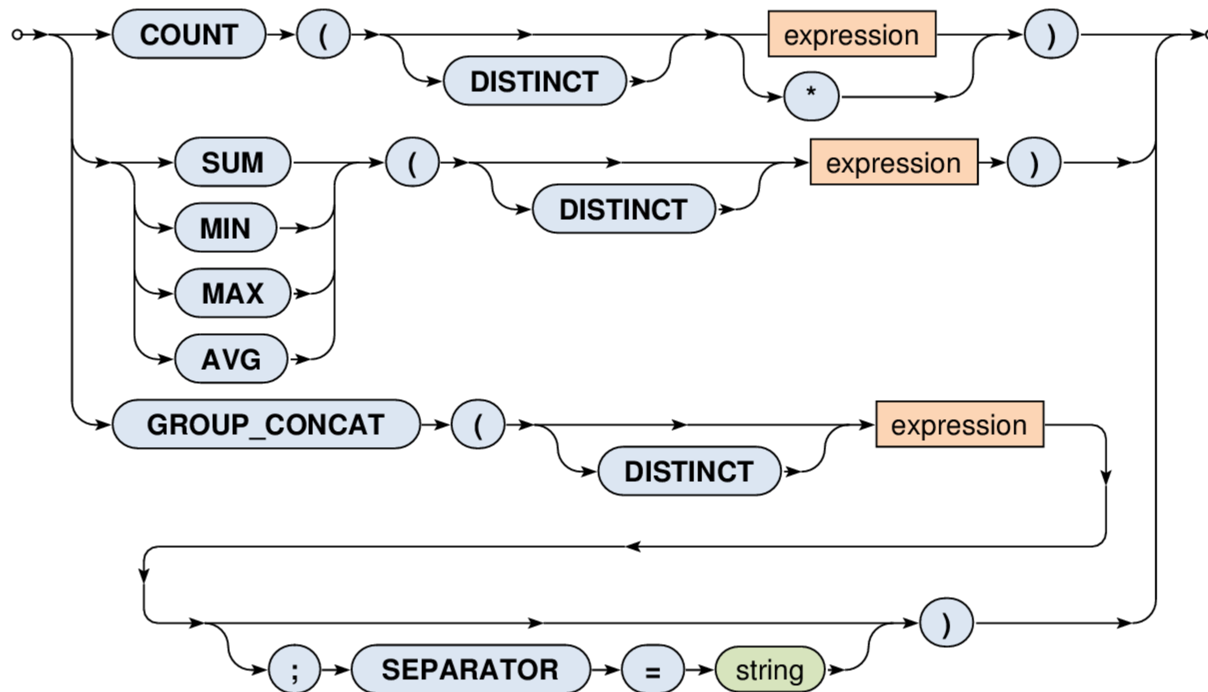
- **Optional** graph pattern
  - **… when additional patterns may extend the solution**
  - Syntax



  - Interpretation
    - **If the optional part does not match,**
      it creates no bindings but does not eliminate the solution

Graph Patterns

- **Optional** graph pattern
  - Example
    - ```
      PREFIX ex: <http://example.cz/is#>
      SELECT ?n ?a
      WHERE {
        ?s rdf:type ex:Student ; ex:name ?n .
        OPTIONAL { ?s ex:age ?a . }
      }
      ```

?n	?a
"Thomas"	"26"
"Peter"	
"John"	"30"
      ```

Graph Patterns

- **Alternative** graph pattern
  - **… when two or more possible patterns are tried**
  - Syntax

- Interpretation
  - Standard **union of multisets of solutions**

Dataset Clause

- **Dataset**
  - Collection of graphs to be queried
  - It contains…
    - one **default graph**,
    - and zero or more **named graphs**
  - Each of these graphs is indentified by an IRI
- **Active graph**
  - Particular graph used for the evaluation
    - We can switch the default graph to a selected named graph

Dataset Clause

- **FROM** clause
  - Definition of all the graphs to be used in a query



    - **Default graph** = merge of all the declared unnamed graphs
      - It is empty, when no unnamed FROM clause is available
  - Examples
    - **FROM** <http://example.cz/students>
    - **FROM** <http://example.cz/teachers>
    - **FROM NAMED** <http://example.cz/courses>

Graph Patterns

- **Graph** graph pattern
  - **… when a particular named graph should be queried**
  - Syntax



- Examples
  - **GRAPH** `<http://example.cz/courses> { … }`
    - Sets the specified named graph as the active one
  - **GRAPH** `?g { … }`
    - Ranges over all the named graphs defined in the dataset

Variable Assignments

- **Bind** "graph pattern"
  - Assigns a value to a (not yet bound!) variable
  - Syntax



Filter Constraints

- **Filter** "graph pattern"
  - Motivation
    - Impose constraints on variables and their values
    - Preserves only solutions that satisfy a given condition
  - Example
    - `FILTER ( BOUND(?age) && (?age < 20) )`
  - Usage
    - Boolean expressions with operators and functions
    - Filters are applied on the entire group graph patterns



Filter Constraints

- **Boolean expressions**
  - Logical connectives
    - `!` `&&` `||`
  - 3 value logic
    - True, false, error

Filter Constraints

- **Relational expressions**
  - Comparisons
    - `<  <=  >=  >`
      - Unbound variable < blank node < IRI < literal
    - `=  !=`
  - Set membership tests
    - `IN`
    - `NOT IN`



Filter Constraints

- **Numeric expressions**
  - Arithmetic operators
    - Unary + –
    - Binary + – * /
- **Primary expressions**
  - Literals: numeric, Boolean, RDF
  - Variables
  - Built-in calls
  - Parenthesized expression

Filter Constraints

- **Built-in calls**
  - Term accessors
    - `STR` – lexical form of IRI or literal
    - `LANG` – language tag of a literal
    - `DATATYPE` – data type of a literal
  - Variable tests
    - `BOUND` – whether a variable is assigned a value
    - `isIRI`, `isBLANK`, `isLITERAL`
  - Existence tests
    - `EXISTS` and `NOT EXISTS`
  - …

Select Clause

- **SELECT** clause
  - Specification of output variables



    - Asterisk * selects all the variables
  - Solution modifiers
    - **DISTINCT**
      - Removes all duplicates from the solution sequence
    - **REDUCED**
      - Permits elimination of some non-unique solutions

Solution Modifiers

- Query structure
  ```
  PREFIX …
  SELECT DISTINCT | REDUCED …
  FROM …
  WHERE { … }
  ORDER BY … LIMIT … OFFSET …
  ```
- Objective
  - Modification of the entire sequence of solutions

Solution Modifiers

- **ORDER BY** clause
  - Mutually orders solutions in the solutions sequence



  - Behavior
    - **ASC**(…) = ascending (default), **DESC**(…) = descending
    - Unbound variable < blank node < IRI < literal
  - Example
    - `ORDER BY ?name DESC(?age)`

Solution Modifiers

- **LIMIT** clause
  - Limits the number of solutions in the result
    - (Always) should be preceded by ORDER BY modifier
      - Otherwise the order of solutions is undefined
  - Example
    - `ORDER BY ?name LIMIT 10`
- **OFFSET** clause
  - Index of the first reported item from the sequence
  - Example
    - `ORDER BY ?name LIMIT 10 OFFSET 20`

Solution Modifiers

- **LIMIT** and **OFFSET** clauses



Query Forms

- Forms
  - **SELECT** – standard solutions sequence
  - **ASK** – test for a solution existence
  - **DESCRIBE** – retrieval of a graph about resources
  - **CONSTRUCT** – construction of a graph from a pattern

Query Forms

- **SELECT** query form
    - SPARQL querying considered so far…
  - Result
    - **Solutions sequence** as an ordered multiset of solutions
  - Syntax



Query Forms

- **ASK** query form
    - – Checks whether at least one solution exists
  - Result
    - – `true` or `false`
  - Syntax



Query Forms

- **DESCRIBE** query form
  - Retrieves an RDF graph with data about selected resources
  - Result
    - Non-deterministic **implementation-dependent behavior**



Query Forms

- **DESCRIBE** query form
  - Examples
    - ```
      DESCRIBE <http://example.cz/is#s1>
      ```
    - ```
      PREFIX ex: <http://example.cz/is#>
      DESCRIBE ?s
      FROM <http://example.cz/is>
      WHERE { ?s rdf:type ex:Student }
      ```

Query Forms

- **CONSTRUCT** query form
  - Construction of new graphs from solutions sequences
  - Result
    - RDF graph constructed from a template
    - Illegal triples (unbound or invalid) are thrown away



Query Forms

- **CONSTRUCT** query form
  - Example

```
PREFIX ex: <http://example.cz/is#>
CONSTRUCT
   { ?s ex:name concat(?n1, " ", ?n2) . }
FROM <http://example.cz/is>
WHERE
  { ?s ex:firstName ?n1 ; ex:lastName ?n2 . }
```

Advanced Constructs

- SPARQL 1.1
  - Constructs that were not available in SPARQL 1.0
  - Aggregation
    - **GROUP BY** and **HAVING** clauses (solution modifiers)
  - Negation
    - **NOT EXSITS** constraint
    - **MINUS** graph pattern
  - Property paths

Aggregation

- Motivation
  - Standard aggregation of solution sequences (tables) inspired by relational databases and SQL
- **GROUP BY + HAVING** clauses



Aggregation

- **Aggregates**



Aggregation

- Example
  - Total capacity of all rooms in each building

```
PREFIX ex: <http://example.cz/is#>
SELECT ?b (SUM(?c) AS ?capacity)
FROM <http://example.cz/faculty/>
WHERE
  { ?r ex:inBuilding ?b ; ex:capacity ?c . }
GROUP BY ?b
```

Negation

- **EXISTS** constraint
  - **… when the existence of solutions should be tested**
  - Syntax

- Notes
  - **<u>Does not generate</u> any additional bindings**

Negation

- **MINUS** graph pattern
  - **… when compatible solutions should be removed**
  - Syntax

  - Idea
    - Solutions of the left pattern are preserved if and only if they are <u>not compatible</u> with <u>any</u> solution from the right pattern
      - I.e. note that this minus graph pattern does not correspond to a traditional set minus operator!

Conclusion

- **SPARQL**
  - Model
    - **Graph pattern matching** and **substitution of variables**
      - Result = **ordered multiset of solutions**
      - Solution = **set of variable bindings**
  - Syntax
    - ```
      PREFIX …
      SELECT DISTINCT | REDUCED …
      FROM …
      WHERE { … }
      GROUP BY … HAVING …
      ORDER BY … LIMIT … OFFSET …
      ```
      ```