Introduction to the Semantic Web

Lecture 6: SPARQL Query Evaluation

Basil Ell · Cord Wiljes

AG Semantic Computing

November 21, 2018

CC - Creative Commons Licensing

- This set of slides is based on slides from the lecture „Semantic Web Technologies“ held at Karlsruhe Institute of Technology
- The content of the lecture was prepared by Dr. Andreas Harth based on his book „Introduction to Linked Data“
- The original slides were prepared by Lars Heling

Universität Bielefeld

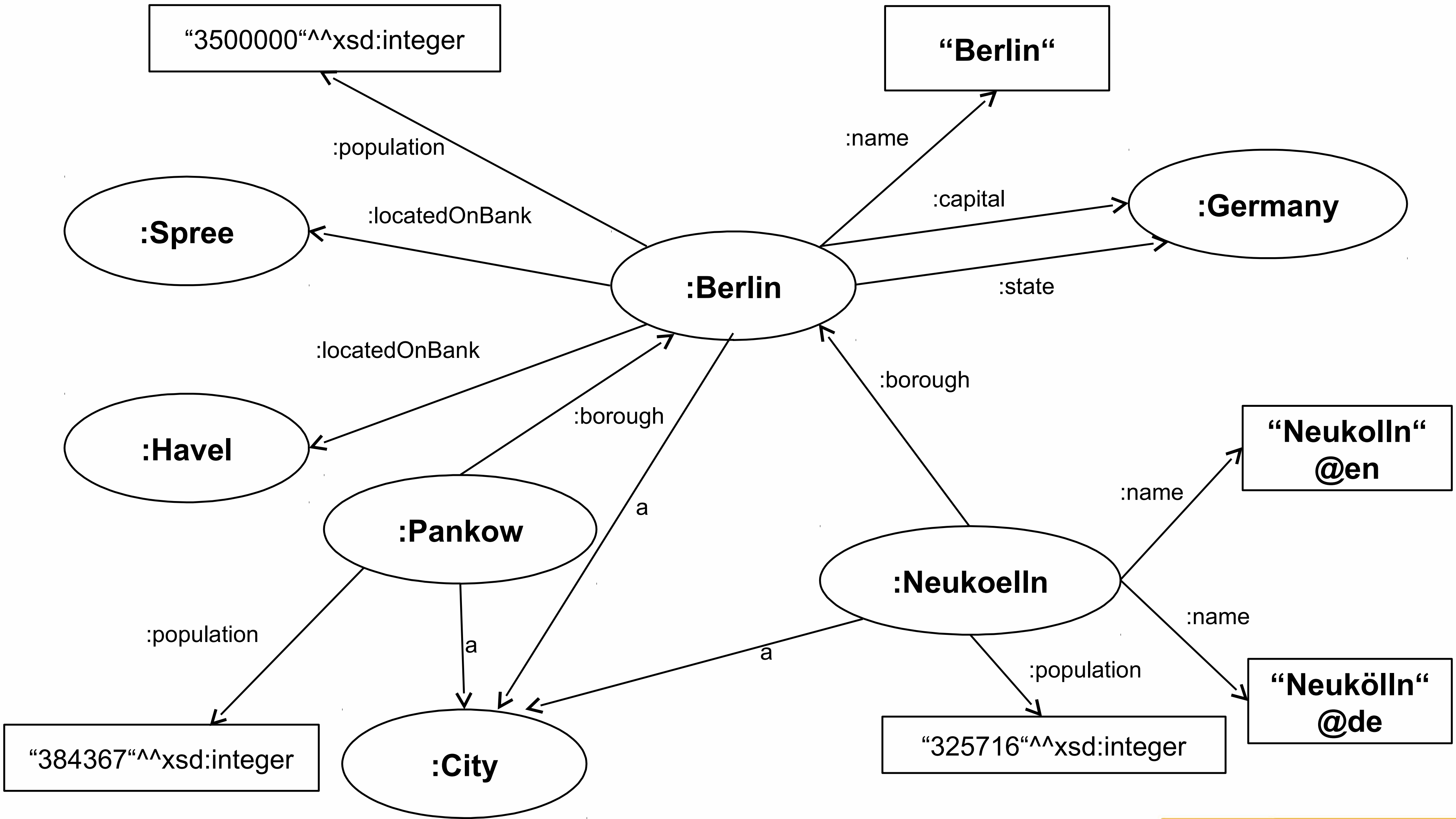
CITEC

An Example Query

```
SELECT ?city WHERE {
    ?city :name ?name .
    ?city :locatedOnBank ?river .
    FILTER ( lang( ?name ) = ”en”)
}
```

How is this query actually executed on the given data graph?

Example Graph

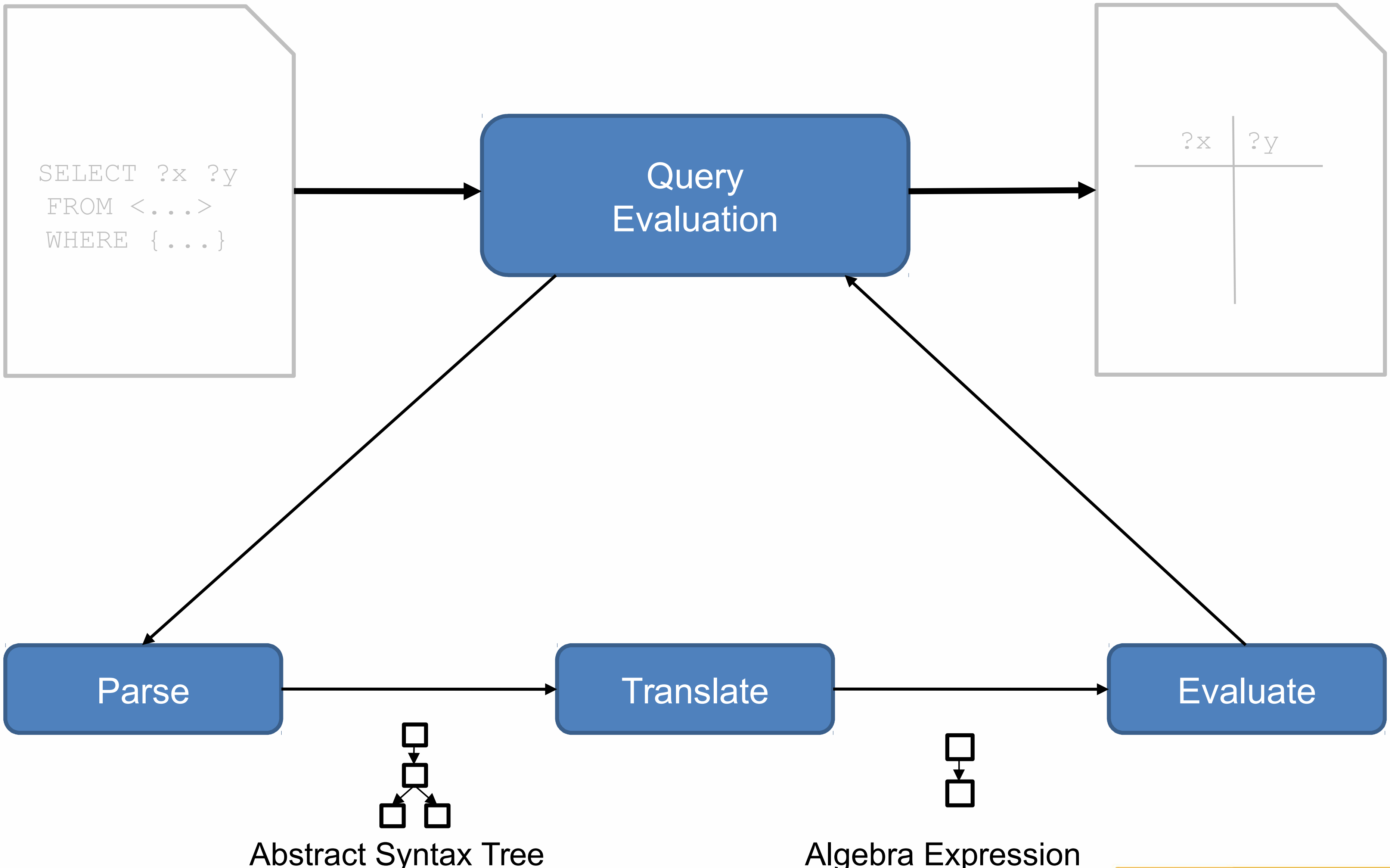


Evaluating a SPARQL Query

- Which steps are necessary to evaluate a SPARQL query expressed as a string on an actual dataset?

- Within this process an algebra expression is derived from the query
- Thereafter, this expression is evaluated on the data

An Overview of the Query Evaluation Process



Agenda

- **Basic Graph Pattern Matching**
- From Queries to Algebra Expressions
- Evaluating Algebra Expressions

Motivation

- As the SPARQL query is evaluated, we need to find mappings for the variables
- Basic Graph Pattern Matching defines this mapping
- Based on the specified rules, we can check whether the mapping of an RDF term to a variable is valid for a triple pattern
- Simple Example:

```
SELECT ?city WHERE {
    ?city :name "Berlin".
}
```

```
:Berlin :name "Berlin".
```

```
?city = :Berlin

?city = :name

?city = "Berlin"
```

Solution Mapping, Solution Sequence and RDF Instance Mapping

- First, we need some basic definitions

Definition 9 (Solution Mapping, Solution Sequence) *A partial function from variables to RDF terms $\mu: \mathcal{V} \mapsto \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ is called a solution mapping. A solution sequence Ω is a set of solutions mappings.*

- Domain of a solution mapping = set of variables *V*
- To take blank nodes into account we introduce the notion of RDF Instance Mapping.

Definition 10 (RDF Instance Mapping) *A partial function from blank nodes to RDF terms $\sigma: \mathcal{B} \mapsto \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ is called an RDF instance mapping.*

Basic Graph Pattern Matching I

- Basic Graph Patterns are the foundation of the SPARQL query evaluation
- The solution of a Basic Graph Pattern is defined via the notion of Basic Graph Pattern Matching

Definition 11 (Basic Graph Pattern Matching) *Let P be a basic graph pattern. A partial function μ is a solution of the expression $BGP(P)$ and the queried graph G if:*

- *the domain of μ is the set of variables in P, and*
- *there exists a mapping σ of blank nodes in P to RDF terms $(\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ in G, so that*
- *the graph $\mu(\sigma(P))$ is a subgraph of G.*

[5] The mapping σ is similar to μ (but for blank nodes instead of variables); σ ensures the correct handling of blank nodes.

Basic Graph Pattern Matching II

- Consider the statement

```
:Berlin        :name  "Berlin" .
:Neukoelln     :name  "Neukolln"@en ,
                      "Neukölln"@de .
```

- And the Basic Graph Pattern

```
?city  :name  ?name .
```

- For all possible mappings the three conditions of the definition need to be verified
- Example:

$$\mu_1(?city) = \text{ :Berlin and } \mu_1(?name) = \text{“Berlin”}$$

Basic Graph Pattern Matching III

- For the example $\mu_1(?city) =$:Berlin and $\mu_1(?name) =$ "Berlin"
- all conditions need to be checked:

- *the domain of μ is the set of variables in P, and*

  ⇒ dom(μ_1) = {?city, ?name} and var(P) = {?city, ?name}

- *there exists a mapping σ of blank nodes in P to RDF terms $(\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ in G, so that*

  ⇒ In this particular example, there are no blank nodes

- *the graph $\mu(\sigma(P))$ is a subgraph of G.*

⇒ The conditions hold, so we can say that μ_1 is a solution for the expression:

```
BGP(?city :name ?name . )
```

Agenda

- Basic Graph Pattern Matching
- **From Queries to Algebra Expressions**
- Evaluating Algebra Expressions

Parsing a SPARQL Query

- **Input:** SPARQL Query
- **Output:** Abstract Syntax Tree (AST)

Introducing the Terms *Parsing* and *Symbols*

- What does *parsing* mean?

Within computational linguistics the term is used to refer to the formal **analysis** by a computer of a **sentence or other string of words** into its constituents, resulting in a **parse tree** showing their syntactic relation to each other, which may also contain semantic and other information.[1]

- What are *symbols*?

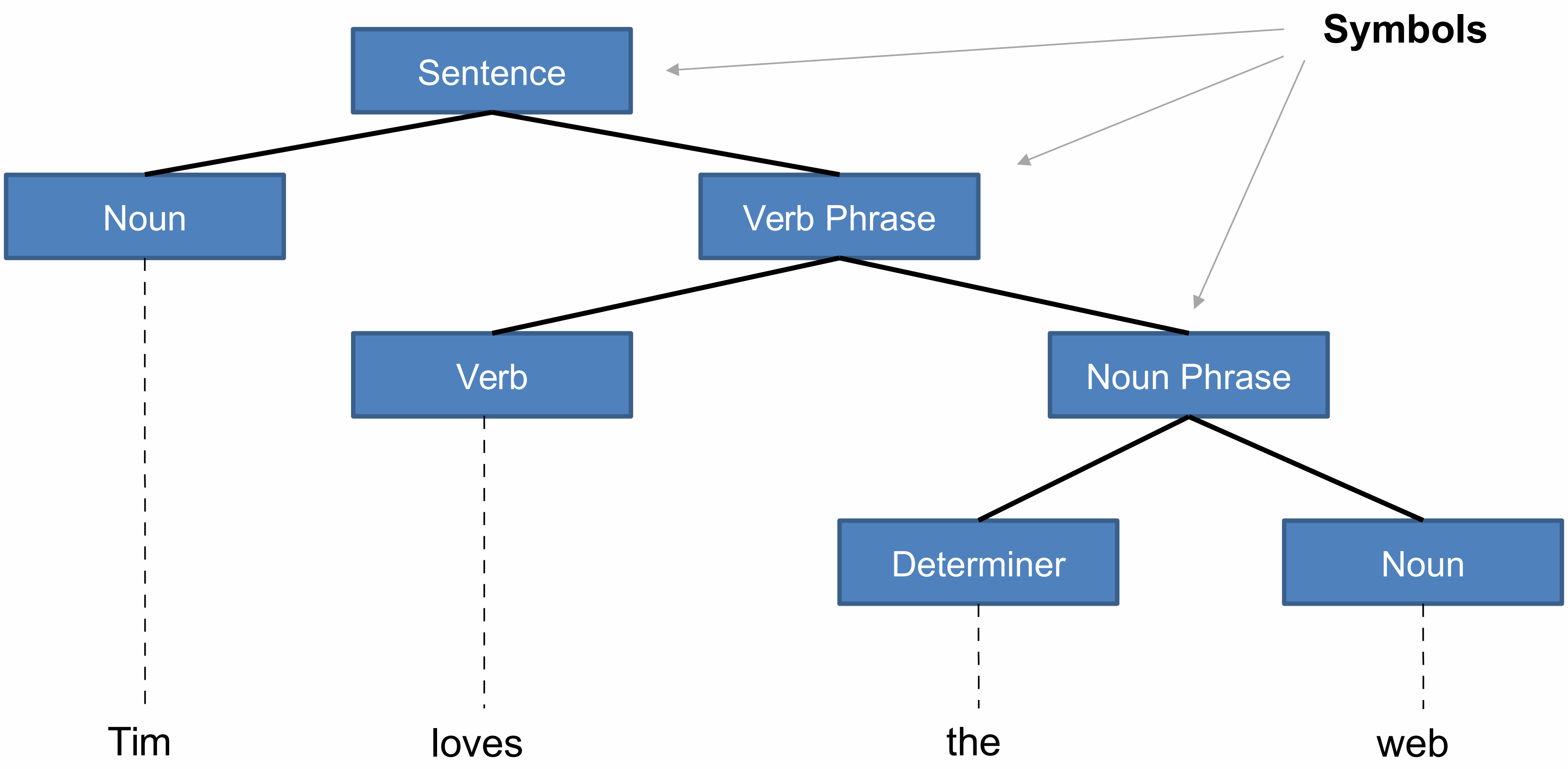
A symbol in computer programming is a primitive datatype whose instances have a unique **human-readable** form.[2]

[1] https://en.wikipedia.org/wiki/Parsing

[2] https://en.wikipedia.org/wiki/Symbol_(programming)

An Example for a Parse Tree

- Example sentences: “Tim loves the web”
- Phrase structure parse tree:



Grammar of SPARQL Queries

- Simplified grammar for SPARQL `SELECT` and `CONSTRUCT` queries:

```
Query ::= Prologue ( SelectQuery | ConstructQuery )
SelectQuery ::= SelectClause DatasetClause∗ WhereClause SolutionModifier
SelectClause ::= ’SELECT’ ( ’DISTINCT’ )? ( Var+ | ’∗’ )
ConstructQuery ::= ’CONSTRUCT’ ConstructTemplate DatasetClause∗ WhereClause SolutionModifier
DatasetClause ::= ’FROM’ ( DefaultGraphClause | NamedGraphClause )
WhereClause ::= ’WHERE’? GroupGraphPattern
GroupGraphPattern ::= TriplesBlock? ( ( GraphGraphPattern | Filter | Bind )  ’.’? TriplesBlock? )∗
GraphGraphPattern ::= ’GRAPH’ VarOrUri GroupGraphPattern
SolutionModifier ::= OrderClause? LimitOffsetClauses?
Filter ::= ’FILTER’ Expression
Bind ::= ’BIND’ ’(’ Expression ’AS’ Var ’)’
```

- Connectives:
  - “A**?**” … matching *A* or nothing
  - “A**+**” … matching one or more *A*
  - “A *****” … matching zero or more *A*
  - “A **|** B” … matching alternatively *A* or *B*

Universität Bielefeld

CITEC

Generating the Abstract Syntax Tree

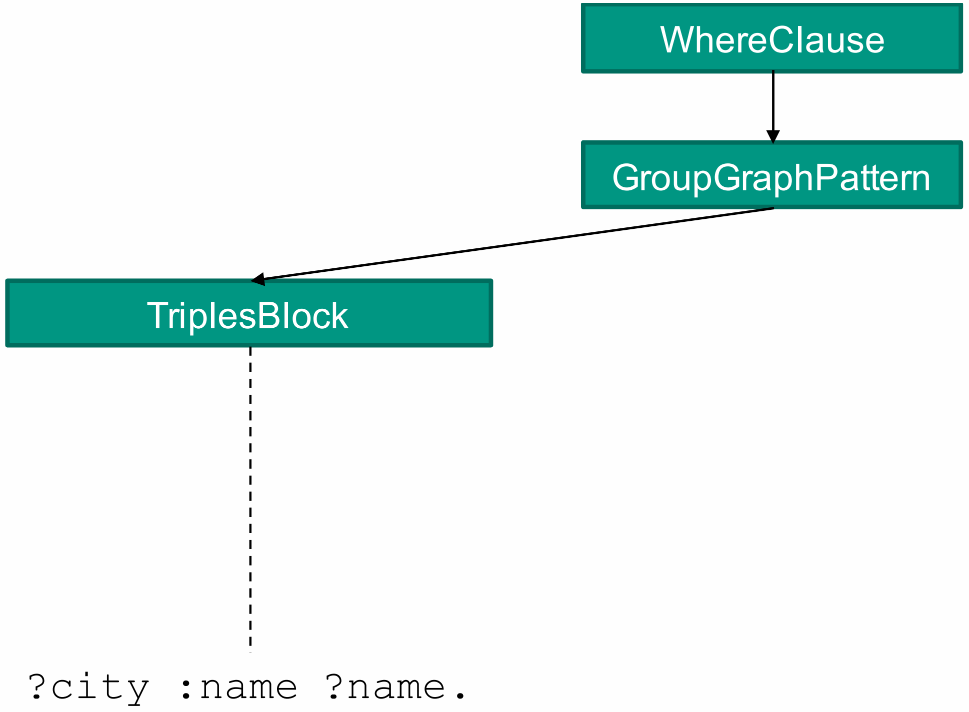
- The query is decomposed such that it only consists of the abstract symbols
- The symbols can only be combined as defined by the connectives
- The AST is the basis for the *translate()*-algorithm

- Example Query:

```
SELECT ?city WHERE {
    ?city :name ?name .
    GRAPH ?g { ?city :locatedOnBank ?river . }
    FILTER ( lang( ?name ) = ″en″)
}
```

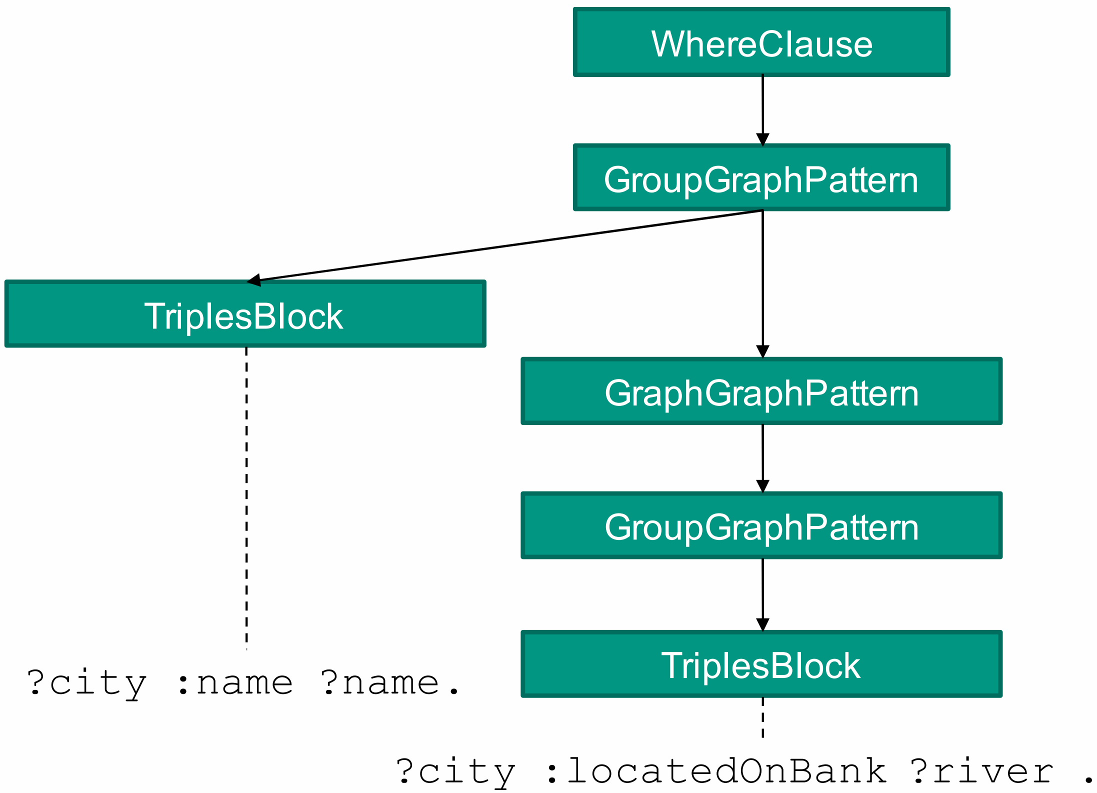
Decomposing the Query String I

```
WHERE {
     ?city :name ?name .
     GRAPH ?g { ?city :locatedOnBank ?river . }
     FILTER ( lang( ?name ) = ”en”)
}
```



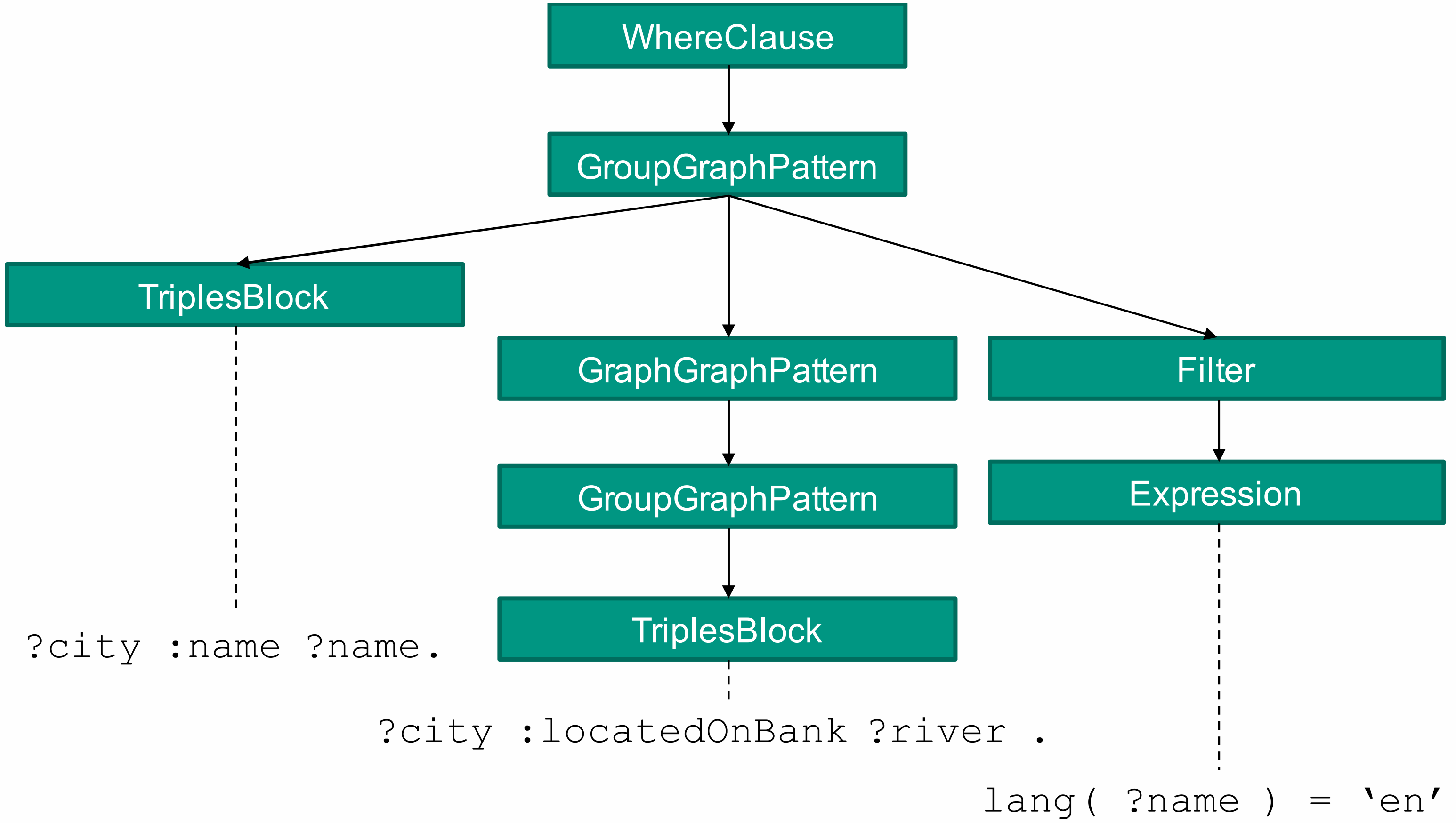
Decomposing the Query String II

```
WHERE {
     ?city :name ?name .
     GRAPH ?g { ?city :locatedOnBank ?river . }
     FILTER ( lang( ?name ) = ”en”)
}
```



Decomposing the Query String III

```
WHERE {
     ?city :name ?name .
     GRAPH { ?g ?city :locatedOnBank ?river . }
     FILTER ( lang( ?name ) = "en")
}
```



Translating the Abstract Syntax Tree

- **Input:** Abstract Syntax Tree
- **Output:** Algebra Expression

Symbols for the Translation

- The given abstract syntax tree is into a SPARQL algebra expression that consists of the following symbols:

Symbol	Description
BGP (triple patterns)	Adjacent triple patterns form a basic graph pattern
Join (M_1, M_2)	Conjunctive combination of expression M_1 and M_2
Union(M_1, M_2)	Alternative combination of expression M_1 and M_2
Graph (U $\cup$ V, M)	Apply the algebra expression M to graphs from U or V
Filter (E, M)	Add expression E to algebra expression M
Extend (M, V, E)	Add expression E to algebra expression M for V

The Translation Process

- The algorithm can be divided into three main steps:

Expand

Expand the shortened syntax of CURIEs and Turtle abbreviations

Translate

Apply the *translate()*-algorithm recursively starting with the GroupGraphPattern

Simplify

Simplify the resulting algebra expression

The *translate()*-Algorithm

- The *translate()*-Algorithm can be divided into 4 subroutines:
  - Translating GroupGraphPattern
  - Translating TriplesBlock
  - Translating GroupOrUnionGraph Pattern
  - Translating GraphGraphPattern

Algorithm: Translating GroupGraphPattern

```
input: AST element GroupGraphPattern
output: SPARQL algebra expression
G := the empty pattern Z
FS := ∅
for each Filter(Constraint) in the GroupGraphPattern:
        FS := FS ∪ Constraint
for each element E in the GroupGraphPattern:
        if E is of the form BIND(expr AS var):
                G := Extend(G, var, expr)
        else:
                if E is of the form TriplesBlock(triple patterns):
                        A := BGP(triple patterns)
                else if E is of the form GraphGraphPattern(URI, GroupGraphPattern):
                        A := Graph(URI, translate(GroupGraphPattern))
                else if E is of the form GraphGraphPattern(Variable, GroupGraphPattern):
                        A := Graph(Variable, translate(GroupGraphPattern))
                G := Join(G, A)
if FS ≠ ∅:
        G := Filter(FS, G)
return G
```

Agenda

- Basic Graph Pattern Matching
- From Queries to Algebra Expressions
- **Evaluating Algebra Expressions**

Evaluating the Algebra Expression

- **Input:** Algebra Expression
- **Output:** Solution Sequence

Evaluating the Expressions

- In order to evaluate the algebra expression, the *eval()*-function is introduced
- The result of this function is a solution sequence consisting of a set of solution mappings
- In the recommendation the *eval()*-function evaluates an algebra expression with respect to a dataset *D* and the active graph *G*:

$$eval(D(G), algebra\ expression)$$

- Analogously to the *translate()*-algorithm, the *eval()*-function is called recursively

Evaluation Functions

- To be able to evaluate the symbols that are the result of the translation process, the according functions need to be defined in a set-theoretic manner
- The actual implementation of how the evaluation functions work is not relevant as long as it adheres to the definition
- In the following, the theoretic foundations for evaluating these functions are presented

Universität Bielefeld

CITEC

Theoretic Foundation – Filter

- The result of the filter can be defined as

$$Filter(\Omega, F) = \{\mu | \mu \in \Omega \text{ and } \mu(F) \text{ is an expression which evaluates to } \texttt{true}\}$$

Theoretic Foundation – Compatibility

- To be able to define the other SPARQL algebra symbols, we need a binary operator which combines two solution mappings
- For this purpose, the concept of compatibility is introduced:

Definition 9 (Compatibility) *Two solution mappings μ_1 and μ_2 are compatible if, for every variable x in $dom(\mu_1)$ and in $dom(\mu_2)$, $\mu_1(x) = \mu_2(x)$.*

➡ In other words: Two solution mappings are *compatible* if variables with the same name are bound to the same RDF term.

Theoretic Foundation – Compatibility

- Example:
  - Given the following information

$$\mu_\emptyset \text{ is the empty mapping}$$
$$\mu_1(\texttt{?x}) = \texttt{"foo"}, \mu_1(\texttt{?y}) = \texttt{<http://example.org/y>}$$
$$\mu_2(\texttt{?y}) = \texttt{<http://example.org/y>}$$
$$\mu_3(\texttt{?x}) = \texttt{"bar"}$$

  - Which solution mappings are compatible?

- Having defined the concept of *compatibility* the remaining functions can be defined as well
- Example:

$$Join(\Omega_1, \Omega_2) = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1, \mu_2 \text{ are compatible}\}$$

All other functions can be defined similarly. These definitions can be found in the book.

Operators of the SPARQL Algebra

Operator	Description
$Join(\Omega_1, \Omega_2)$	$\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ and μ_1, μ_2 are compatible$\}$
$Union(\Omega_1, \Omega_2)$	$\{\mu \mid \mu \in \Omega_1$ or $\mu \in \Omega_2\}$
$Filter(expr, \Omega, D(G))$	$\{\mu \mid \mu \in \Omega$ and $expr(\mu)$ is an expression which evaluates to `true`$\}$
$Extend(\mu, var, expr)$	$\mu \cup \{(var, value) \mid var \notin dom(\mu)$ and $value = expr(\mu)\}$
	μ if $var \notin dom(\mu)$ and $expr(\mu)$ is an error
	undefined when $var \in dom(\mu)$
$Extend(\Omega, var, expr)$	$\{Extend(\mu, var, expr) \mid \mu \in \Omega\}$