Abfragen auf Relationswerte

```
?p a owl:Class;
        rdfs:subClassOf pg:Programm;
        rdfs:subClassOf ?restriction.
            ?restriction owl:onProperty pg:verwendetProgrammiersprache.
            ?restriction owl:hasValue pg:Deklarativ.

    ?p a owl:Class;
        rdfs:subClassOf pg:Programm;
        rdfs:subClassOf ?restriction.
            ?restriction owl:onProperty pg:bestehtAus.
            ?restriction owl:someValuesFrom pg:Axiom.
```

Alle Programmiersprachen, welche von einem Programm verwendet werden, welches aus Axiomen besteht

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX pg:
<http://www.semanticweb.org/sosterwalder/ontologies/2014/9/programming#>
SELECT *
WHERE {
        ?ps pg:wirdVerwendetVon ?o.
        ?o pg:bestehtAus pg:axiom.

#pg:Programm ?xx ?object
}
```

Alle Einsatzgebiete welche in Programmen zum Einsatz kommen, welche aus Axiome bestehen:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX pg:   <http://www.semanticweb.org/sosterwalder/ontologies/2014/9/programming#>
SELECT *
        WHERE {
        ?es a owl:Class;
                rdfs:subClassOf pg:Einsatzgebiet.
        ?r a owl:Restriction;
                owl:onProperty pg:hatEinsatzgebiet;
                owl:allValuesFrom ?es.
        ?x a ?es.
        ?p a pg:Programm;
                a ?r;
                pg:bestehtAus pg:axiom.
}
```

Alle Einsatzgebiete welche in deklarativen Programmen zum Einsatz kommen :

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX pg:   <http://www.semanticweb.org/sosterwalder/ontologies/2014/9/programming#>
SELECT Distinct ?x
        WHERE {
        ?es a owl:Class; rdfs:subClassOf pg:Einsatzgebiet.
        ?r a owl:Restriction; owl:onProperty pg:hatEinsatzgebiet; owl:allValuesFrom ?es.
        ?x a ?es.
        ?p a pg:Programm;
                a ?r.
                #pg:bestehtAus pg:axiom.
        FILTER  (regex(str(?p), "deklarativesProgramm","i"))
}
```

Wie ist Prolog aufgebaut?

Aus was besteht Prolog?

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX pl: <http://www.semanticweb.org/mira/ontologies/2014/9/untitled-ontology-7#>
SELECT * WHERE {
        ?x rdfs:subClassOf pl:Programmiersprache.
        ?r a owl:Restriction;
                owl:onProperty pl:hatSyntaxElement;
                owl:someValuesFrom ?t.
        ?e rdfs:subClassOf ?t.
}
```

Aus was bestehen logische Elemente?

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX pl: <http://www.semanticweb.org/mira/ontologies/2014/9/untitled-ontology-7#>
SELECT * WHERE {
        ?x rdfs:subClassOf pl:Programmiersprache.
        ?r a owl:Restriction;
                owl:onProperty pl:hatSyntaxElement;
                owl:someValuesFrom ?t.
        ?e rdfs:subClassOf ?t.
        ?el a ?e.
}
```

Was für sprachliche Elemente verwendet Prolog?

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX pl: <http://www.semanticweb.org/mira/ontologies/2014/9/untitled-ontology-7#>
SELECT * WHERE {
        ?x rdfs:subClassOf pl:Programmiersprache.
        ?r a owl:Restriction;
```

```
            owl:onProperty pl:hatSyntaxElement;
            owl:someValuesFrom ?t.
        ?e rdfs:subClassOf ?t.
        ?el rdfs:subClassOf ?e.
        ?xa a ?el.
}
```

Wie funktioniert Unifkation?

Erwartete Antwort: In Prolog wird versucht mit Hilfe von Unifikation Anfragen mit Regeln oder Fakten identisch zu machen.

Da wir von der Fragestellung her wissen, dass wir die Klasse *Unifikation* wollen, geben wir diese bereits als Filter an.

Schritt 1: Beziehungen der Klasse *Unifikation* nach oben herausfinden

```
SELECT DISTINCT * WHERE {
        ?u a owl:Class.
        ?p ?b ?u.
        ?t ?a ?p.

        FILTER  (?u=pl:Unifikation)
}
```

u	p	b	t	a
Unifikation	verwendet some Unifikation	someValuesFrom	Prolog	subClassOf

Schritt 2: Beziehungen der Klasse *Unifikation* nach unten herausfinden

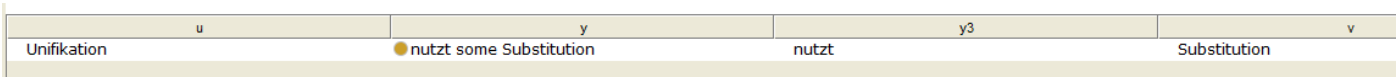
```
SELECT DISTINCT * WHERE {
        ?u a owl:Class.
        ?u rdfs:subClassOf ?y.

        FILTER  (?u=pl:Unifikation)
}
```

u	y
Unifikation	nutzt some Substitution
Unifikation	wirdAngewendetAuf value Anfrage
Unifikation	setztGleich value Fakt
Unifikation	setztGleich value Regel

Schritt 3: Details der Beziehungen anzeigen

```
SELECT distinct * WHERE {
        ?u a owl:Class.
        ?u rdfs:subClassOf ?y.
        ?y owl:onProperty ?y3.
```

```
        ?y owl:someValuesFrom ?v.
        FILTER  (?u=pl:Unification)
}
```



u	y	y3	v
Unifikation	nutzt some Substitution	nutzt	Substitution

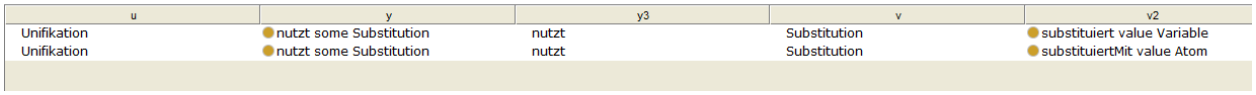
Feststellung: Die Beziehungen *setztGleich* und *wirdAngewendetAuf* scheinen keine weiteren Relationen zu haben.

Zwischenfazit

Mit den Schritten 1 bis 3 haben wir festgestellt, dass Unifikation auf Anfragen angewendet wird und diese Fakten und Regeln gleich setzt.
Als Verfahren scheint dabei Substitution zum Einsatz zu kommen. Dies muss aber in weiteren Schritten genauer analysiert werden.

Schritt 4: Beziehungen der Klasse *Substitution* herausfinden

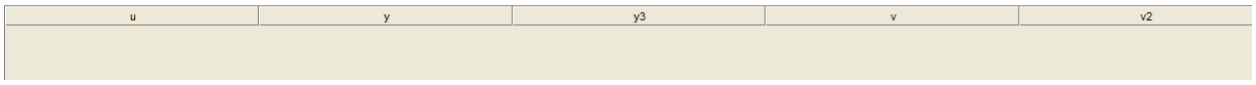
```
SELECT distinct * WHERE {
        ?u a owl:Class.
        ?u rdfs:subClassOf ?y.
        ?y owl:onProperty ?y3.
        ?y owl:someValuesFrom ?v.
        ?v rdfs:subClassOf ?v2.
        FILTER  (?u=pl:Unification)
}
```



u	y	y3	v	v2
Unifikation	nutzt some Substitution	nutzt	Substitution	substituiert value Variable
Unifikation	nutzt some Substitution	nutzt	Substitution	substituiertMit value Atom

Schritt 5: Details der Beziehungen anzeigen

```
SELECT distinct * WHERE {
        ?u a owl:Class.
        ?u rdfs:subClassOf ?y.
        ?y owl:onProperty ?y3.
        ?y owl:someValuesFrom ?v.
        ?v rdfs:subClassOf ?v2.
        ?v2 owl:onProperty ?y3.
        FILTER  (?u=pl:Unification)
}
```



u	y	y3	v	v2

Feststellung: Die Beziehungen *substituiert* und *substituiertMit* scheinen keine weiteren Relationen zu haben.

Zwischenfazit

Mit den Schritten 4 und 5 haben wir festgestellt, dass Substitution Variablen mit Atomen substituiert.

Schritt 6: Individuen extrahieren

```
SELECT distinct * WHERE {
        ?u a owl:Class.
        ?u rdfs:subClassOf ?y.
        ?y owl:onProperty ?y3.
        ?y owl:someValuesFrom ?v.
        ?v rdfs:subClassOf ?v2.
        ?v2 owl:hasValue ?v4.

        FILTER  (?u=pl:Unifikation )
}
```

u	y	y3	v	v2	v4
Unifikation	nutzt some Substitution	nutzt	Substitution	substituiert value Variable	Variable
Unifikation	nutzt some Substitution	nutzt	Substitution	substituiertMit value Atom	Atom

Fazit

Dadurch, dass keine weitere Beziehungen vorhanden sind, haben wir somit alle Beziehungen/Informationen der Unifikation erhalten.

Es stellt sich nun die Frage, wie weitere Informationen zu den Individuen (Variable, Atom, Fakt, Regel und Anfrage) abgerufen werden können.

Was sind Atome?

Erwartete sparql'sche Antwort: Bei den Atomen handelt es sich um einfache Tokens, wobei diese wiederum Sprachelemente sind. Sprachelemente sind Tokens, woraus Prolog besteht.

Erwartete menschliche Antwort: Atome sind Sprachelemente von Prolog, welche mit einem Kleinbuchstaben oder einem Apostrophen beginnen. Atome sind einfache Tokens.

Da wir von der Fragestellung her wissen, dass es sich bei *Atom* um ein sog. *owl:NamedIndividual* handelt, selektieren wir dieses direkt per Filter.

Schritt 1: Atom-Objekt selektieren

```
SELECT distinct * WHERE {
        ?i a owl:NamedIndividual.

        FILTER  (regex(str(?i),"atom","i") )
}
```

i
Atom

Schritt 2: Klasse/Typ des Individuums herausfinden

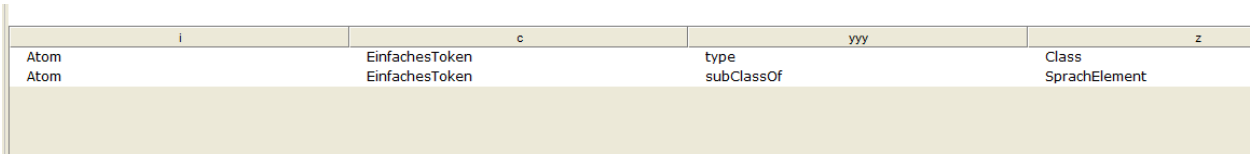
```
SELECT distinct * WHERE {
        ?i a owl:NamedIndividual.
        ?i a ?c.

        FILTER  (regex(str(?i),"atom","i") )
}
```

i	c
Atom	EinfachesToken
Atom	NamedIndividual

Feststellung: Bei den Atomen handelt es sich um einfache Tokens (und um eine *NamedIndividual*, was aber bereits im Vorfeld klar war).

Schritt 3: Beziehungen des einfachen Token herausfinden

```
SELECT distinct * WHERE {
        ?i a owl:NamedIndividual.
        ?i a ?c.
        ?c ?xx ?z.     # Gibt alle Beziehungen aus

        FILTER  (regex(str(?i),"atom","i") && ?c=pl:EinfachesToken)
}
```



i	c	yyy	z
Atom	EinfachesToken	type	Class
Atom	EinfachesToken	subClassOf	SprachElement

Feststellung: Wir sehen, dass einfache Tokens die Prädikate *type* und *subClassOf* mit den Subjekten *Class* bzw. *SprachElement* hat. Dies erlaubt nun die Einschränkung des Queries auf den Teil von Interesse, also *subClassOf*.

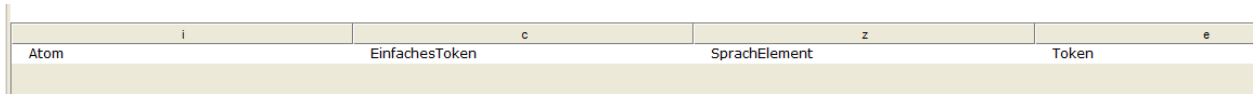
```
SELECT distinct * WHERE {
        ?i a owl:NamedIndividual.
        ?i a ?c.
        ?c rdfs:subClassOf ?z.

        FILTER  (regex(str(?i),"atom","i") && ?c=pl:EinfachesToken)
}
```

i	c	z
Atom	EinfachesToken	SprachElement

Feststellung: Einfache Tokens scheinen also Sprachelemente zu sein.

Schritt 4: Beziehungen der Klasse *SprachElement* herausfinden

```
SELECT distinct * WHERE {
        ?i a owl:NamedIndividual.
        ?i a ?c.

        ?c rdfs:subClassOf ?z.
        ?z rdfs:subClassOf ?e.

        FILTER  (regex(str(?i),"atom","i") && ?c=pl:EinfachesToken)
}
```



i	c	z	e
Atom	EinfachesToken	SprachElement	Token

Schritt 5: Beziehungen zu *Token*

Nachdem wir festgestellt haben, dass *Token* selbst keine weiteren Relationen mehr hat, wird in diesem Schritt die Anfrage umgedreht.

```
SELECT distinct * WHERE {
        ?i a owl:NamedIndividual.
        ?i a ?c.

        ?c rdfs:subClassOf ?z.
        ?z rdfs:subClassOf ?e.
        ?o ?w ?e.


        FILTER  (regex(str(?i),"atom","i") && ?c=pl:EinfachesToken)
}
```

i	c	z	e	o	w
Atom	EinfachesToken	SprachElement	Token	hatSyntaxElement some Token	someValuesFrom
Atom	EinfachesToken	SprachElement	Token	LogischesElement	subClassOf
Atom	EinfachesToken	SprachElement	Token	SprachElement	subClassOf
Atom	EinfachesToken	SprachElement	Token	hatSyntaxElement	range

Feststellung: *Token* wird von einer Klasse/einem Objekt mittels der Beziehung *hatSyntaxElement* als *range* verwendet.

Schritt 6: Objekt(e) mit Relation *hatSyntaxElement* herausfinden

Es soll herausgefunden werden, welche Objekte das Prädikat *hatSyntaxElement* als Eigenschaft und *Token* als Wert einer Einschränkung verwenden.

```
SELECT distinct * WHERE {
        ?i a owl:NamedIndividual.
        ?i a ?c.
        ?c rdfs:subClassOf ?z.
        ?z rdfs:subClassOf ?e.
        ?bez rdfs:range ?e.
        ?r a owl:Restriction;
                owl:onProperty ?bez;
                owl:someValuesFrom ?e.
        ?cl rdfs:subClassOf ?r.

        FILTER  (regex(str(?i),"atom","i") && ?c=pl:EinfachesToken)
}
```

i	c	z	e	bez	r	cl
Atom	EinfachesToken	SprachElement	Token	hatSyntaxElement	hatSyntaxElement some T	Prolog

Feststellung: Es handelt sich bei dem gefunden Objekt um Prolog!

Schritt 7: Beziehungen zu Prolog herausfinden

```
SELECT distinct * WHERE {
        ?i a owl:NamedIndividual.
        ?i a ?c.
        ?c rdfs:subClassOf ?z.
        ?z rdfs:subClassOf ?e.
        ?bez rdfs:range ?e.
        ?r a owl:Restriction;
               owl:onProperty ?bez;
               owl:someValuesFrom ?e.
        ?cl rdfs:subClassOf ?r.
        ?cl rdfs:subClassOf ?ccl.
        FILTER  (regex(str(?i),"atom","i") && ?c=pl:EinfachesToken)
}
```

i	c	z	e	bez	r	cl	ccl
Atom	EinfachesToken	SprachElement	Token	hatSyntaxElement	hatSyntaxElement son	Prolog	Programmiersprache
Atom	EinfachesToken	SprachElement	Token	hatSyntaxElement	hatSyntaxElement son	Prolog	hatSyntaxElement some Toke
Atom	EinfachesToken	SprachElement	Token	hatSyntaxElement	hatSyntaxElement son	Prolog	verwendet some Unifikation

Feststellung: *Prolog* hat als übergeordnetes Objekt die Klasse *Programmiersprache*, ist als Unterklasse von dieser. Weiter ist ersichtlich, dass Prolog aus *Token* Elementen besteht und *Unifikation* verwendet.

Zwischenfazit

Die gewonnen Erkenntnisse decken die Fragestellung grösstenteils ab, was aber nicht ersichtlich ist, ist, dass Atome über einen kleinen Anfangsbuchstaben verfügen.

Schritt 8: Kommentar(e) der Individuen ausgeben

Als Idee zur Lösung zum Zwischenfazit wird die Annotation *comment* von allen *NamedIndividual* Objekten ausgegeben.

```
SELECT distinct * WHERE {
        ?i a owl:NamedIndividual.
        ?i rdfs:comment ?x.
        FILTER  (regex(str(?i),"atom","i"))
}
```

i	x
Atom	"beginnt mit Kleinbuchstaben oder mit Apostroph"@

Analog kann dies natürlich in der Abfrage von *Schritt 7* verwendet werden.

Fazit

Die ursprüngliche Frage konnte also erfolgreich wie folgt beantwortet werden:

- *Atome sind Sprachelemente*
  Schritt 4

- *von Prolog*
  Schritt 6

- *welche mit einem Kleinbuchstaben oder einem Apostrophen beginnen.*
  Schritt 8

- *Atome sind einfache Tokens.*
  Schritt 3
-

30.10.2014

Sparql in Stardog

Alle Nachfahren von Doris

```
SELECT
        *
WHERE {
        ?subject :isAncestor ?object.
        #FILTER regex(?subject, "doris", "i")
}
```

Ist hans vorfahre von tea?

```
ASK WHERE {
        :hans :isAncestor :tea.
 }
```

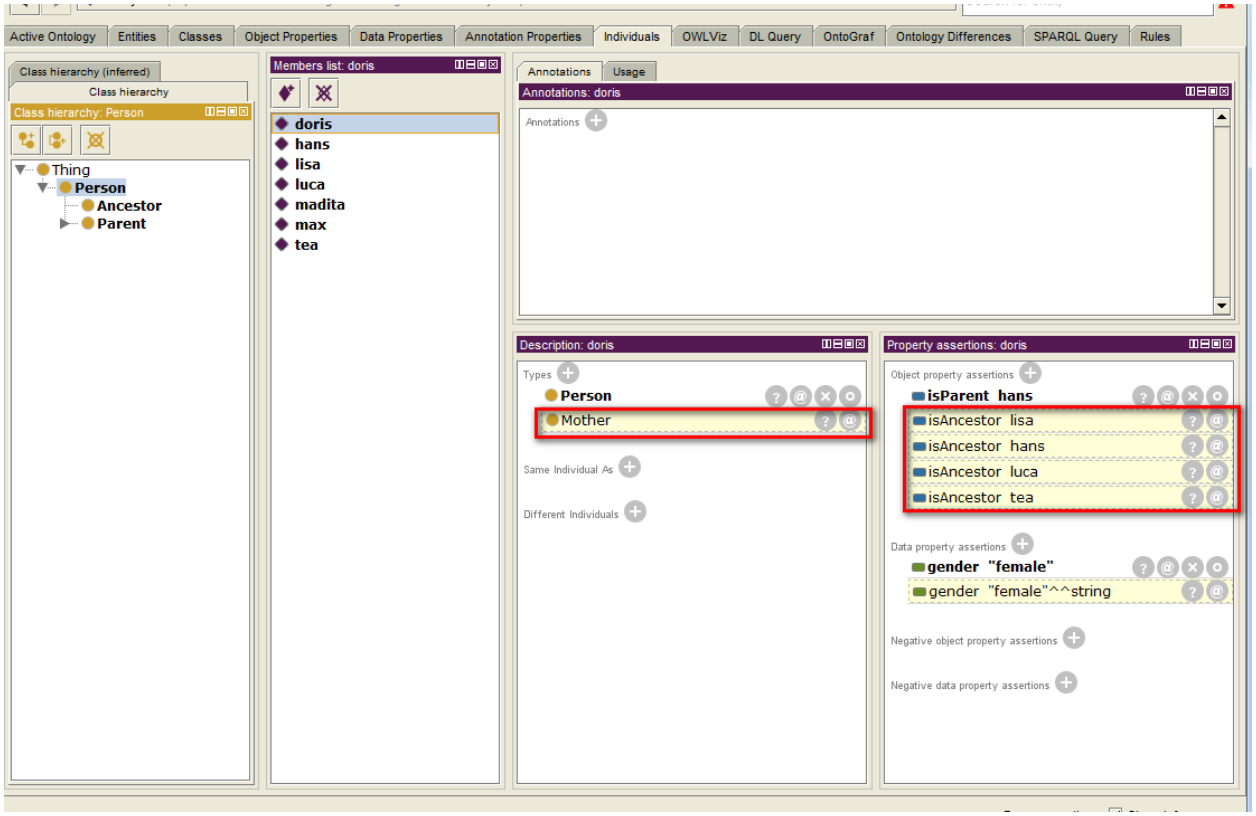
Familienbeispiel

Damit wir eine Bestätigung bekommen, dass unser Semantisches System die gleiche Mächtigkeit wie Prolog (also alle notwendigen Funktionalitäten) besitzt, haben wir ein FamilienProlog Beispiel abgebildet.

Die Situation in family.pl (unter Modell/family) soll abgebildet werden.

Fakten werden mit Klassen resp. Objekt oder Data Properties abgebildet. Die Argumente in Prolog werden als Individuen abgebildet. (siehe family.owl).
Die Regel können eins zu eins übernommen werden.

Die Beispielanfragen wurden genau gleich wie in Prolog beantwortet. Zusätzlich zu der Möglichkeit anfragen zu stellen zeigt der Reasoner in Protege seine Folgerungen direkt bei den Objekten an.



Der Reasoner konnte in diesem Beispiel dank unserer Regeln direkt erkennen das doris eine Mutter ist und Verfahre von lisa, hans, luca, tea.