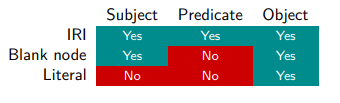
# REMEMBER THE . AFTER EACH SENTENCE (coglione)



| . (dot) | to end a sentence |
| --- | --- |
| ; (semicolon) | to refer to the last subject |
| , (comma) | to refer to the last subject and last predicate |
| + (after the predicate) | Transitive closure, for “*ex:alice ex:hasParent****+*** *?x”* you represent all the ancestors of Alice |
| \* (after the predicate) | Transitive closure, for “*ex:alice ex:hasParent****\**** *?x”* you represent all the ancestors of Alice but also including Alice |
| / (between two predicates) | *“ex:bob foaf:knows****/****foaf:name****?*** *?y”* . Is for People known by Bob, or the names of people known by Bob |
| SELECT \* | Will select all the variables that are in the WHERE |

(utto culo)

| **OPERATOR** | **WHAT IT DOES** | **EXAMPLE** | **COMMENTS** |
| --- | --- | --- | --- |
| @prefix | Creates a prefix, that is an alias (remember to specify the name used as prefix) | @prefix dbr: <<https://dbpedia.org/resource/>> . |  |
| rdf:type(utto culo) | Used to specify that something is a class, or that something is an instance of a class. (when you express that something is an instance of a class you need to have declared the class before) | ex:Person rdf:type rdfs:Class .  ex:bob rdf:type foaf:Person . | Built-in function. In the first sentence you specify that *Person* is a class, the second one tells that *bob* is an element of the class *Person.* |
| rdf:Property | Representing the class of RDF properties. |  | (pen.tito) |
| rdfs:subClassOf | States that one class is a subclass of another. You must specify both classes before. | *ex:Student rdfs:subClassOf ex:Person .* | *Every student is a person.* If we know that alice is a student we also know what she is a person. |
| rdfs:subPropertyOf | States that one property R is a subproperty of another property Q. | *ex:parentOf rdfs:subPropertyOf ex:relativeOf .* | If x is parent of y then we know that x is a relative of y. |
| rdfs:domain | Specifies the domain of a property. Remember to specify the class of the domain (ex:Person). | *ex:authorOf rdfs:domain ex:Person* | States that the subject of the predicate *ex:authorOf* must be an element of the class *ex:Person*.  If “*ex:alice ex:authorOf ex:abc*”, then “*ex:alice rdf:type ex:Person*”. |
| rdfs:range | Specifies the range of a property. Remember to specify the class of the range (ex:Book). | *ex:authorOf rdfs:range ex:Book* | If “*ex:alice ex:authorOf ex:abc*”, then  “*ex:abc rdf:type ex:Book*”. |
| owl:Class | Denotes the class of all classes (like rdfs:Class). |  |  |
| owl:Nothing | Denotes the empty set. | [*https://amzn.to/3uuqfnJ*](https://amzn.to/3uuqfnJ) |  |
| owl:Thing | Denotes the class that contains everything (in the domain of interest). |  |  |
| owl:equivalentClass | States that two classes have the same extension. | *:c owl:equivalentClass :d .* | This is just for classes if you want to specify that two individuals are the same you must use *owl:sameAs* . |
| owl:disjointWith | States that two classes are disjoint. | *:c owl:disjointWith :d .* |  |
| a | States that Alice is a (named) individual. | *:alice a owl:NamedIndividual .* |  |
| owl:sameAs | State that two individuals are the same. | *:x owl:sameAs :y .* | This is just for individuals if you want to specify that two classes are the same you must use *owl:equivalentClass .* |
| owl:differentFrom | States that two individuals are different. | *:x owl:differentFrom :y .* |  |

# 

# More tricky stuff

## UNION

Represent the union of two (or more) classes:

:GeoEntity owl:equivalentClass [ rdf:type owl:Class ;

owl:unionOf ( :Continent :Country :Region :City ) ]

In DL it would be *GeoEntity = Continent ⊔ Country ⊔ Region ⊔ City*

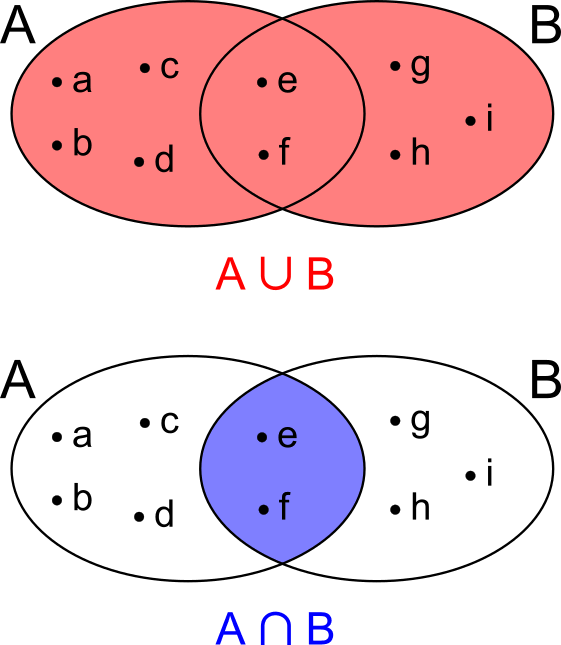
## COMPLEMENT

Represent the complement of two (or more) classes:

:DeadPerson owl:equivalentClass [ rdf:type owl:Class ;

owl:complementOf :LivingPerson ]

In DL DeadPerson = ¬LivingPerson



## NEGATION

Using the complements you can state that something is not an instance of a class.

*British is not a straight man*

:british rdf:type [ rdf:type owl:Class ;

owl:complementOf :straightMan ] .

## INVERSE

Express that a relation Q is the inverse of the relation R:

Q owl:inverseOf R

*ex:hasChild owl:inverseOf owl:hasParent*

which means that if *:bob :hasChild :alice*, then *:alice :hasParent :bob*.

## EXISTS (∃)

∃hasChild.Person is written as:

[ rdf:type owl:Restriction ;

owl:onProperty :hasChild ;

owl:someValuesFrom :Person ]

Parent = ∃hasChild.Person can be written as:

:Parent owl:equivalentClass [

rdf:type owl:Restriction ;

owl:onProperty :hasChild ;

owl:someValuesFrom :Person ]

## FORALL (∀)

Same structure of the exists (∃), but use instead of owl:someValuesFrom owl:allValuesFrom

# RELATIONS

| **NAME** | **DEFINITION** | **SYNTAX** |
| --- | --- | --- |
| Reflexive | if ∀x ∈ X : (x, x) ∈ R. | R rdf:type owl:ReflexiveProperty |
| Irreflexive | if ∀x ∈ X : (x, x) ∉ R. | R rdf:type owl:IrreflexiveProperty |
| Symmetric | if ∀x, y ∈ X : (x, y) ∈ R implies (y, x) ∈ R. | R rdf:type owl:SymmetricProperty |
| Asymmetric | if ∀x, y ∈ X : (x, y) ∈ R implies (y, x) ∉ R. | R rdf:type owl:AsymmetricProperty |
| Transitive | if ∀x, y, z ∈ X : (x, y) ∈ R and (y, z) ∈ R implies (x, z) ∈ R. | R rdf:type owl:TransitiveProperty |
| Functional | if ∀x, y, z ∈ X : (x, y) ∈ R and (x, z) ∈ R implies y = z | R rdf:type owl:FunctionalProperty |
| Inverse Functional | if ∀x, y, z ∈ X : (y, x) ∈ R and (z, x) ∈ R implies y = z | R rdf:type owl:InverseFunctionalProperty |

# MODIFIERS

| MODIFIERS | WHAT IT DOES | COMMENTS |
| --- | --- | --- |
| OPTIONAL { pattern } | If the optional part matches, it creates the corresponding bindings; otherwise, it creates no bindings but does not eliminate the solution. |  |
| LIMIT 10 : | Restricts the number of solutions to at most 10. |  |
| DISTINCT : | Ensures solutions are unique. |  |
| ORDER BY ?x: | Sorts the solutions with respect to the variable ?x, ascending. |  |
| ORDER BY DESC(?x): | Sorts the solutions with respect to the variable ?x, descending. |  |
| OFFSET 5: | Skips the first five solutions. |  |

SELECT DISTINCT ?x ?y ?z

WHERE { ?x ?y ?z . }

ORDER BY ?y

LIMIT 10

OFFSET 2

↑ will return at most 10 distinct solutions ordered by the variable ?y.

OFFSET 2 means that the first two solutions will be skipped.