

```
In [2]: # !pip install networkx
# !pip install matplotlib
# !pip install numpy
# !pip install -q requests -U --user
# Falls Sie die Notebooks lokal laufen und nicht unseren Docker Container verwenden - Graphviz (https://graphviz.org/download/)
# !pip install graphviz
# Stellt sicher dass die aktuelle Version von rdfify installiert ist.
!pip install -q jupyter-rdfify~1.3 --index-url https://git.rwth-aachen.de/api/v4/projects/49225/packages/pypi/simple
%reload_ext jupyter-rdfify
# %matplotlib notebook
```

Jupyter Notebook: RDF und Semantic Web - Vorlesung

Lernziele

Nach Abschluss dieses Notebooks sollen die Studierenden in der Lage sein:

- **Das Konzept und die Motivation hinter RDF zu erklären:**
 - Die grundlegende Idee des Semantic Web und des "Web of Data" zu verstehen.
 - Die Notwendigkeit eines standardisierten Datenmodells für die Datenintegration und -verknüpfung zu begründen.
 - Die Kernidee von RDF als graphbasiertes Datenmodell zu erfassen.
- **Die fundamentalen Bausteine von RDF zu identifizieren und zu beschreiben:**
 - Ein RDF-Tripel (Subjekt, Prädikat, Objekt) als grundlegende Aussageform zu erkennen und zu erläutern.
 - Die Rolle und Bedeutung von Internationalized Resource Identifiers (IRIs) für die eindeutige Benennung von Ressourcen und Eigenschaften zu verstehen.
 - Literale zur Darstellung von Datenwerten (z.B. Texte, Zahlen, Daten) zu kennen und ihre Verwendung mit Datentypen und Sprach-Tags zu verstehen.
 - Das Konzept und den Anwendungsfall von Blank Nodes (anonymen Ressourcen) zu erklären.
- **Einfache Informationen als RDF-Tripel zu modellieren:**
 - Gegebene Aussagen in die Tripel-Struktur zu übersetzen.
 - Geeignete IRIs und Literale für die Modellierung auszuwählen.
- **Die wesentlichen Vorteile und Anwendungsbereiche von RDF zu nennen:**
 - Die Flexibilität des RDF-Datenmodells zu begünden.
 - Potenzielle Einsatzszenarien für RDF (z.B. Wissensgraphen, Linked Data, Datenintegration) zu erkennen.

Einführung in RDF und das Semantische Web

Motivation: Die Evolution des Webs und die Notwendigkeit maschinenverständlicher Daten

Das World Wide Web, wie wir es kennen, hat eine bemerkenswerte Entwicklung durchlaufen. Um die Rolle und Notwendigkeit von Technologien wie RDF und dem Semantischen Web zu verstehen, werfen wir einen kurzen Blick auf diese Evolution:

- **Web 1.0: Das "Read-Only" Web (ca. 1990er - frühe 2000er)**
 - **Charakteristika:** Hauptsächlich statische Webseiten, die von Erstellern bereitgestellt und von Nutzern konsumiert wurden. Denken Sie an Online-Broschüren oder digitale Kataloge.
 - **Technologien:** Einfaches HTML, Hyperlinks. Interaktion war minimal (z.B. E-Mail-Formulare).
 - **Limitierung:** Informationen waren zwar digital verfügbar, aber primär für menschliche Leser aufbereitet. Maschinen konnten die Bedeutung der Inhalte kaum erfassen.
- **Web 2.0: Das "Read-Write-Participate" Web (ca. frühe 2000er - heute)**
 - **Charakteristika:** Das Web wurde zu einer partizipativen Plattform. Nutzer wurden zu aktiven Inhaltserstellern durch Blogs, Wikis, soziale Medien (Facebook, Twitter, YouTube) und Kommentarfunktionen.
 - **Technologien:** Dynamische Webanwendungen, AJAX, JavaScript-Frameworks, APIs (Schnittstellen zum Datenaustausch zwischen Anwendungen).
 - **Fortschritt:** Enorme Zunahme an nutzergenerierten Inhalten und Interaktivität. Daten wurden zwar geteilt, aber oft in anwendungsspezifischen "Silos" und immer noch primär für menschliche Interpretation. Die Bedeutung ("Semantik") der Daten war für Maschinen weiterhin schwer zugänglich.
- **Web 3.0: Das "Read-Write-Execute/Own" Web – Die Vision des Semantischen Webs**

- **Vision:** Ein intelligenteres, stärker vernetztes und potenziell dezentraleres Web. Hier kommt das **Semantische Web** ins Spiel.
- **Ziel des Semantischen Webs:** Daten so zu strukturieren und mit Bedeutung (Semantik) zu versehen, dass sie nicht nur von Menschen, sondern auch von Maschinen verstanden, verarbeitet und intelligent verknüpft werden können.
 - Tim Berners-Lee (Erfinder des WWW) beschrieb es als ein "Web von Daten", das von Maschinen direkt und indirekt verarbeitet werden kann.
- **Kernidee:** Wenn Maschinen die Bedeutung von Daten verstehen, können sie uns besser unterstützen – von intelligenteren Suchergebnissen über automatisierte Datenintegration bis hin zu komplexen Schlussfolgerungen und Assistenzsystemen.

Warum ist das für uns als Informatiker (insbesondere im Datenbankkontext) relevant?

1. **Datenintegration:** In der realen Welt existieren Daten oft in heterogenen Systemen und Formaten. Das Semantische Web bietet Standards und Technologien, um diese Daten auf einer bedeutungsvollen Ebene zu integrieren.
2. **Wissensrepräsentation:** Wie können wir Wissen so modellieren, dass es sowohl für Menschen verständlich als auch für Maschinen präzise interpretierbar ist? Ontologien spielen hier eine Schlüsselrolle.
3. **Flexible Datenmodelle für eine vernetzte Realität:** Die Welt ist selten tabellarisch. Informationen sind oft komplex vernetzt und entwickeln sich ständig weiter.
 - **Natürliche Abbildung:** RDF als Graphmodell erlaubt es, diese Vernetzungen intuitiv abzubilden – oft passender als starre relationale Schemata.
 - **Anpassungsfähig:** RDF ist schema-flexibel. Neue Arten von Informationen oder Beziehungen können einfach hinzugefügt werden, ohne bestehende Strukturen zu "zerbrechen". Das ist ideal für agile Projekte und wenn sich Anforderungen schnell ändern – Deine Datenmodelle wachsen mit Deinen Projekten!
4. **Intelligente Anwendungen:** Die Fähigkeit, Bedeutung zu verarbeiten und Schlussfolgerungen zu ziehen (Inferenz), ist die Grundlage für intelligentere Anwendungen und Systeme.

Um diese Ziele zu erreichen, benötigen wir geeignete Datenmodelle und Technologien.

Von Aussagen zu global identifizierbaren Daten

Um Informationen strukturiert und für Maschinen verarbeitbar darzustellen, benötigen wir ein grundlegendes Modell. Eine sehr intuitive Art, Wissen auszudrücken, ist durch einfache Aussagen, die eine Beziehung zwischen zwei Dingen oder einem Ding und einem Wert herstellen.

Natürlichsprachliche Sätze:

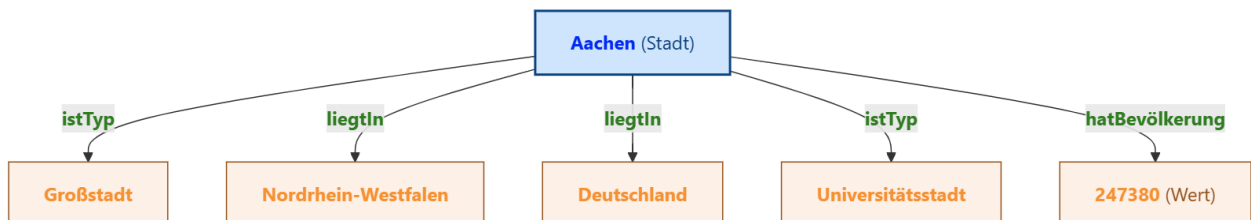
- Aachen ist eine Großstadt.
- Aachen liegt in Nordrhein-Westfalen.
- Aachen liegt in Deutschland.
- Aachen ist eine Universitätsstadt.
- Aachen hat eine Bevölkerung von 247380.

Strukturierung als Tripel (Subjekt ---Prädikat---> Objekt):

Diese Sätze können wir in eine strukturiertere Form bringen, die immer einem Muster folgt: ein Ding (das **Subjekt**), eine Beziehung oder Eigenschaft (das **Prädikat**) und ein weiteres Ding oder ein Wert (das **Objekt**).

- <Aachen> ---istTyp---> <Großstadt>
- <Aachen> ---liegtIn---> <Nordrhein-Westfalen>
- <Aachen> ---liegtIn---> <Deutschland>
- <Aachen> ---istTyp---> <Universitätsstadt>
- <Aachen> ---hatBevölkerung---> <"247380"> (Die Bevölkerungszahl ist hier ein Wert/Literal)

Diese Tripel bilden zusammen einen kleinen Wissensgraphen über Aachen.



Diese Subjekt-Prädikat-Objekt Struktur ist fundamental. Aber wie stellen wir sicher, dass <Aachen> hier und <Aachen> in einer anderen Datenquelle dasselbe meint? Und wie können wir diese Informationen über verschiedene Systeme hinweg austauschen und integrieren?

Globale Eindeutigkeit und Datenintegration: "Wieso Graphen (und

globale Identifier)?"

Die wahre Stärke einer graphenbasierten Wissensrepräsentation entfaltet sich, wenn wir Daten aus **unterschiedlichen Quellen** miteinander verbinden wollen. Stellen Sie sich vor:

1. Eine Quelle (z.B. **DBPedia.org**) enthält allgemeine Informationen über Aachen: den offiziellen Namen in verschiedenen Sprachen und die Bevölkerungszahl.
2. Eine andere Quelle (z.B. **OfeneDaten.Aachen.de**) enthält spezifische lokale Informationen: wer der Oberbürgermeister ist oder welche Ämter es in der Stadtverwaltung gibt.

Wenn beide Datenquellen das "Ding" Aachen mit einem **gemeinsamen, global eindeutigen Namen (Identifikator)** bezeichnen, können wir die Informationen aus beiden Quellen **automatisch zusammenführen**. Ein System kann dann erkennen, dass sich die Aussagen aus beiden Quellen auf dieselbe Entität "Aachen" beziehen.

Dadurch entsteht ein umfassenderer Wissensgraph, der Informationen aus beiden Quellen kombiniert. Ohne einen solchen gemeinsamen, globalen Identifikator wäre diese Verknüpfung nur schwer oder mit hohem manuellem Aufwand möglich. Man müsste komplexe Abgleiche durchführen und könnte sich nie ganz sicher sein, ob wirklich dasselbe gemeint ist.

Globale Identifier sind also die "Ankerpunkte", die es ermöglichen, ein verteiltes "Web der Daten" zu schaffen. Die folgende Codezelle enthält eine kleine Animation, die das verdeutlicht (die Datenquellen existieren, aber die Daten sind fiktiv). Bitte führen Sie den Python Code aus und drücken Sie einfach auf den Play-Knopf.

In [2]:

```
# Merging zweier Graphen
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML, display
import numpy as np
import matplotlib.patches as mpatches

def create_graph_merge_animation():
    """
    Creates an animation visualizing the merge of two knowledge graphs.
    Arrows connect to the edges of the node boxes.
    Arrow tips are smaller, and the final "merged" title is removed.
    """
    fig, ax = plt.subplots(figsize=(12, 8))

    G1 = nx.DiGraph() # DBPedia
    G2 = nx.DiGraph() # OffeneDaten.Aachen.de

    # Define graph G1 (DBPedia)
    G1.add_node("db:Aachen", pos=(0, 0), width=1.8, height=0.8)
    G1.add_node("\Oche\@ksh", pos=(-2.5, 2), width=1.6, height=0.8)
    G1.add_node("247380", pos=(0, 2), width=1.4, height=0.8)
    G1.add_node("\Aachen\@de", pos=(2.5, 2), width=1.8, height=0.8)
    G1.add_edge("db:Aachen", "\Oche\@ksh", label="rdfs:label")
    G1.add_edge("db:Aachen", "247380", label="wdf:Population")
    G1.add_edge("db:Aachen", "\Aachen\@de", label="rdfs:label")

    # Define graph G2 (OpenData.Aachen.de)
    G2.add_node("db:Aachen", pos=(0, -5), width=1.8, height=0.8)
    G2.add_node("Person:SibylleKeupen", pos=(-1.5, -7), width=2.2, height=0.8)
    G2.add_node("ac:Ordnungsamt", pos=(1.5, -7), width=2.0, height=0.8)
    G2.add_edge("db:Aachen", "Person:SibylleKeupen", label="ac:hasMajor")
    G2.add_edge("db:Aachen", "ac:Ordnungsamt", label="ac:hasDepartment")

    # Extract node attributes
    pos1 = nx.get_node_attributes(G1, 'pos')
    pos2 = nx.get_node_attributes(G2, 'pos')
    width1 = nx.get_node_attributes(G1, 'width')
    width2 = nx.get_node_attributes(G2, 'width')
    height1 = nx.get_node_attributes(G1, 'height')
    height2 = nx.get_node_attributes(G2, 'height')

    # Define merged positions and dimensions
    pos_merged = pos1.copy()
    width_merged = {**width1, **width2}
    height_merged = {**height1, **height2}
    for node in G2.nodes():
        if node != "db:Aachen":
            x, y = pos2[node]
            pos_merged[node] = (x, y + 5)

def get_intersection_point(cx, cy, w, h, target_x, target_y):
    """
    Calculates the intersection point of a line (from node center to target)
    with the node's rectangular boundary.
    """
    dx = target_x - cx
    dy = target_y - cy
    hw, hh = w / 2.0, h / 2.0

    if abs(dx) < 1e-9 and abs(dy) < 1e-9: return cx, cy

    if abs(dx) < 1e-9: # Vertical line
        return cx, cy + hh * np.sign(dy if dy != 0 else 1)
    if abs(dy) < 1e-9: # Horizontal line
        return cx + hw * np.sign(dx if dx != 0 else 1), cy

    t_to_vertical = hw / abs(dx)
    t_to_horizontal = hh / abs(dy)
    intersect_t = min(t_to_vertical, t_to_horizontal)
    intersect_x = cx + intersect_t * dx
    intersect_y = cy + intersect_t * dy
    return intersect_x, intersect_y

def draw_rectangle_node(x, y, width, height, color):
    """Draw a rectangle node on the given axes (ax)."""
    rectangle = mpatches.Rectangle(
        (x - width/2, y - height/2), width, height,
        facecolor=color, edgecolor='black', zorder=1
    )
    ax.add_patch(rectangle)
```

```

def draw_edge(pos_u_center, pos_v_center, label, w_u, h_u, w_v, h_v):
    """
    Draw a single directed edge (arc) with an arrow and label on ax.
    The arrow connects the boundaries of the source and target nodes.
    """
    start_point = get_intersection_point(
        pos_u_center[0], pos_u_center[1], w_u, h_u,
        pos_v_center[0], pos_v_center[1]
    )
    end_point = get_intersection_point(
        pos_v_center[0], pos_v_center[1], w_v, h_v,
        pos_u_center[0], pos_u_center[1]
    )

    arrow = mpatches.FancyArrowPatch(
        start_point, end_point,
        arrowstyle='->',
        connectionstyle='arc3,rad=0.1',
        mutation_scale=20, # <<<< REDUCED arrow tip size
        linewidth=2,
        color='black',
        zorder=2,
        shrinkA=0,
        shrinkB=0
    )
    ax.add_patch(arrow)

    mid_x = (start_point[0] + end_point[0]) / 2
    mid_y = (start_point[1] + end_point[1]) / 2
    dx_label = end_point[0] - start_point[0]
    dy_label = end_point[1] - start_point[1]
    length_label = np.sqrt(dx_label**2 + dy_label**2)
    label_offset_distance = 0.25
    if length_label > 0:
        offset_x = label_offset_distance * (-dy_label/length_label)
        offset_y = label_offset_distance * (dx_label/length_label)
    else:
        offset_x = offset_y = 0

    ax.text(
        mid_x + offset_x, mid_y + offset_y, label,
        fontsize=9, ha='center', va='center', zorder=3,
        bbox=dict(boxstyle="round,pad=0.3", fc="white", ec="none", alpha=0.8)
    )

def draw_graph(G, pos_dict, width_dict, height_dict, node_color):
    """Draw a complete graph with rectangular nodes, edges, and labels on ax."""
    for node_id in G.nodes():
        x, y = pos_dict[node_id]
        w = width_dict[node_id]
        h = height_dict[node_id]
        draw_rectangle_node(x, y, w, h, node_color)
        ax.text(
            x, y, node_id, ha='center', va='center', fontsize=10, zorder=4
        )
    for u, v, data in G.edges(data=True):
        draw_edge(
            pos_dict[u], pos_dict[v], data['label'],
            width_dict[u], height_dict[u],
            width_dict[v], height_dict[v]
        )

def animate(frame):
    ax.clear()
    ax.set_xlim([-5.5, 5.5])
    ax.set_ylim([-8, 4])
    ax.axis('off')
    ax.text(-5, 3.5, 'DBPedia.org', fontsize=16, color='navy', fontweight='bold')
    ax.text(-5, -8.5, 'OffeneDaten.Aachen.de', fontsize=16, color='navy', fontweight='bold')

    if frame <= 30: # Phase 1: Tracks disappear
        track_alpha = 1 - (frame / 30)
        if track_alpha > 0:
            track_y = -2.5
            ax.plot([-5, 5], [track_y, track_y], 'k-', alpha=track_alpha, linewidth=2)
            ax.plot([-5, 5], [track_y+0.3, track_y+0.3], 'k-', alpha=track_alpha, linewidth=2)
            for x_tie in np.linspace(-5, 5, 40):
                ax.plot([x_tie, x_tie], [track_y, track_y+0.3], 'k-', alpha=track_alpha, linewidth=1.5)
            draw_graph(G1, pos1, width1, height1, 'lightgray')
            draw_graph(G2, pos2, width2, height2, 'lightblue')

    elif frame <= 90: # Phase 2: Bottom graph moves up
        progress = (frame - 30) / 60
        current_pos2 = {}
        for node_id, (x_node, y_node) in pos2.items():

```

```

    for node_id, (x_node, y_node, w_node, h_node) in pos.items():
        target_y = 0 if node_id == "db:Aachen" else (y_node + 5)
        current_y = y_node + progress * (target_y - y_node)
        current_pos2[node_id] = (x_node, current_y)
        draw_graph(G1, pos1, width1, height1, 'lightgray')
        draw_graph(G2, current_pos2, width2, height2, 'lightblue')

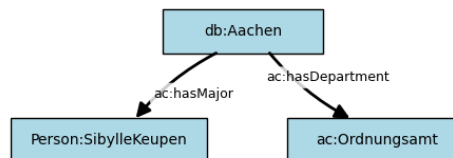
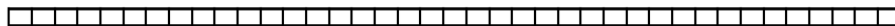
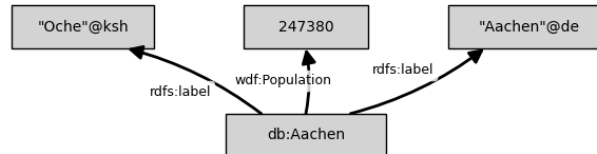
    else: # Phase 3: Merged state
        merged_node_ids = list(G1.nodes()) + [n for n in G2.nodes() if n != "db:Aachen"]
        for node_id in merged_node_ids:
            color = 'lightgreen' if node_id == "db:Aachen" else ('lightgray' if node_id in G1.nodes() else 'lightblue')
            x_node, y_node = pos_merged[node_id]
            w_node = width_merged[node_id]
            h_node = height_merged[node_id]
            draw_rectangle_node(x_node, y_node, w_node, h_node, color)
            ax.text(
                x_node, y_node, node_id,
                ha='center', va='center', fontsize=10, zorder=4
            )
        for G_orig in [G1, G2]:
            for u, v, data in G_orig.edges(data=True):
                draw_edge(
                    pos_merged[u], pos_merged[v], data['label'],
                    width_merged[u], height_merged[u],
                    width_merged[v], height_merged[v]
                )
        # ax.set_title("Knowledge Graphs Merged", fontsize=16, y=1.02) # <<<< REMOVED TITLE

    anim = animation.FuncAnimation(fig, animate, frames=100, interval=50, repeat=False)
    plt.close(fig)
    return HTML(anim.to_jshtml())

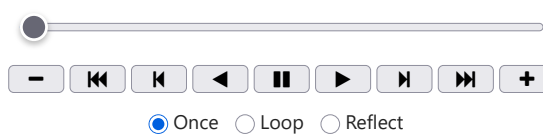
# To use in Jupyter notebook:
display(create_graph_merge_animation())

```

DBPedia.org



OffeneDaten.Aachen.de



Das Resource Description Framework (RDF) und Standards für das Semantische Web

Um die Vision eines Webs von Daten zu realisieren, das auf globalen Identifikatoren und einer graphenbasierten Struktur aufbaut, wurden Standards entwickelt. Der zentrale Standard für das Datenmodell ist das **Resource Description Framework (RDF)**.

- **RDF – Resource Description Framework**

- Es ist ein **Graph-basiertes Datenformat**: Informationen werden als ein Netzwerk von Knoten und Kanten dargestellt. Dies ermöglicht eine flexible Repräsentation von Beziehungen.
- **Objekte/Entitäten** werden durch **URIs** (Uniform Resource Identifiers) – oder genauer gesagt **IRIs** (Internationalized Resource Identifiers), die wir gleich genauer betrachten – eindeutig identifiziert. Dies ist entscheidend für die globale Eindeutigkeit.
- **Information wird verknüpft**: Der Schwerpunkt liegt auf den Beziehungen zwischen Datenpunkten, was die Entdeckung neuer Zusammenhänge ermöglicht.

Um diesen Daten Bedeutung zu verleihen und ein gemeinsames Verständnis zu ermöglichen, spielen **Vokabulare und Ontologien** eine zentrale Rolle:

- Sie ermöglichen das **gemeinsame Verständnis** für eine bestimmte Domäne (z.B. Medizin, Universitätswesen, Geographie).
- Sie erlauben das **Organisieren von Wissen** in einem maschinen-lesbaren Format.
- Sie geben Daten eine **auswertbare Bedeutung** (Semantik), die über die reine Datenstruktur hinausgeht. Standards hierfür sind **RDFS (RDF Schema)** und **OWL (Web Ontology Language)**.

Nun spezifischer zu RDF selbst:

- **RDF = Resource Description Framework**

- Es ist eine **W3C Empfehlung** (Standard) seit 1998. Das W3C ist die Standardisierungsorganisation für das World Wide Web. Der aktuelle Standard ist **RDF 1.1** (veröffentlicht 2014). Dokumente wie **RDF 1.1 Concepts and Abstract Syntax** definieren das Modell. (RDF 1.2 ist ein aktueller Entwurf für eine Weiterentwicklung). Die übergeordnete Seite für RDF-Standards beim W3C finden Sie unter www.w3.org/RDF/.
- RDF ist ein **Datenmodell**, keine spezifische Datenbanktechnologie oder ein Dateiformat (obwohl es verschiedene Serialisierungsformate gibt, um RDF-Daten zu speichern und auszutauschen).
- Ursprünglich wurde es oft für Metadaten auf Web-Seiten verwendet (z.B. um den Autor oder das Erstellungsdatum einer Seite zu beschreiben), aber sein Anwendungsbereich wurde schnell generalisiert, um beliebige Informationen zu repräsentieren.
- Es dient dazu, **strukturierte Informationen** auszudrücken, indem Aussagen über Ressourcen gemacht werden.
- Es ist ein **universales Format** für den automatisierten Datenaustausch zwischen verschiedenen Systemen und Anwendungen.
- Die zugrundeliegende Struktur ist immer ein **Graph** aus Knoten (die Ressourcen und Werte repräsentieren) und beschrifteten, gerichteten Kanten (die Beziehungen oder Eigenschaften repräsentieren).

Die formalen Komponenten des RDF-Datenmodells

Nachdem wir die grundlegende Idee von Tripeln und die Bedeutung von globalen Identifikatoren zur Datenintegration kennengelernt haben, betrachten wir nun die formalen Komponenten des **Resource Description Framework (RDF)**-Datenmodells, wie sie vom W3C standardisiert wurden.

Das RDF-Tripel

Die grundlegende Struktur in RDF ist das **RDF-Tripel**, bestehend aus:

1. **Subjekt (Subject)**: Eine IRI oder ein Blank Node (beides wird im folgenden erklärt)
2. **Prädikat (Predicate)**: Immer eine IRI.
3. **Objekt (Object)**: Eine IRI, ein Literal oder ein Blank Node.

In [9]:

```
# Ein RDF Triple
from graphviz import Digraph

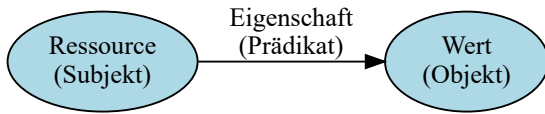
# Diagramm erstellen
g = Digraph('RDF_Triple', format='png')
g.attr(rankdir='LR') # Links nach rechts Layout

# Knoten hinzufügen mit ovaler Form
g.node('Ressource', 'Ressource\n(Subjekt)', shape='ellipse', style='filled', fillcolor='lightblue')
g.node('Wert', 'Wert\n(Objekt)', shape='ellipse', style='filled', fillcolor='lightblue')

# Kante mit Beschriftung hinzufügen
g.edge('Ressource', 'Wert', label='Eigenschaft\n(Prädikat)')

# Anzeigen im Notebook
g
```

Out[9]:



Ressourcen

- Im Kontext von RDF kann alles eine Ressource sein: physische Dinge (ein Student, ein Buch, eine Stadt wie Aachen), Dokumente (eine Webseite), abstrakte Konzepte (eine Vorlesung, eine Idee, der Status "Großstadt"), Zahlen, Zeichenketten etc.
- Der Begriff ist quasi synonym mit "Entität" oder "Ding". In RDF wird jede Ressource, über die wir sprechen wollen, typischerweise durch eine IRI identifiziert.

IRIs (Internationalized Resource Identifiers) 💡

Wie einleitend motiviert, ist das Ziel im Semantic Web, Ressourcen und ihre Beziehungen **eindeutig und global identifizierbar** zu machen. Genau hierfür dienen IRIs in RDF. Sie werden verwendet für:

- **Subjekte:** Die "Dinge", über die eine Aussage getroffen wird.
- **Prädikate:** Die Art der Eigenschaft oder Beziehung.
- **Objekte:** Den Wert einer Eigenschaft, falls dieser eine andere Ressource ist.

Kernidee: Jede Ressource (eine Person, ein Ort, ein abstraktes Konzept, eine Beziehung) erhält eine einzigartige IRI, ähnlich wie eine Webseite eine eindeutige URL hat.

Von URIs zu IRIs

- **URIs (Uniform Resource Identifiers):** Sie kennen URIs bereits in Form von URLs (Webadressen). Ihre Syntax ist im Standard [RFC 3986](#) der IETF (Internet Engineering Task Force) definiert. URIs verwenden einen begrenzten Zeichensatz (ASCII).
- **IRIs (Internationalized Resource Identifiers):** IRIs sind eine Erweiterung von URIs und im Standard [RFC 3987](#) definiert. Der Hauptunterschied ist, dass IRIs ein breiteres Spektrum an Unicode-Zeichen erlauben (z.B. Umlaute, asiatische Schriftzeichen), was sie für internationale Inhalte besser geeignet macht. **In RDF wird typischerweise von IRIs gesprochen**, auch wenn viele gängige Beispiele syntaktisch auch gültige URIs sind.

Aufbau einer IRI (basierend auf URI-Syntax)

Eine IRI hat eine generische Syntax, die oft wie folgt aussieht:

scheme ":" hier-part ["?" query] ["#" fragment]

- **scheme (Schema):** Definiert das verwendete Protokoll oder den Kontext. Für das Web und RDF sind `http` und `https` (für gesicherte Verbindungen) die mit Abstand wichtigsten Schemata.
 - *Beispiel:* `http`
- **hier-part (Hierarchischer Teil):** Enthält typischerweise Autoritätsinformationen (wie einen Domainnamen) und einen Pfad, der die Ressource innerhalb dieser Autorität spezifiziert.
 - *Beispiel (mit Scheme):* `http://de.wikipedia.org/wiki/Aachen`
- **query (Anfrageteil, optional):** Beginnt mit einem `?` und kann zusätzliche Parameter enthalten. Dies ist seltener Teil der Identität einer Ressource in RDF, kann aber bei Service-Endpunkten vorkommen.
 - *Beispiel (URI mit Query):* `http://example.org/suche?begriff=RDF`
- **fragment (Fragmentteil, optional):** Beginnt mit einem `#` und identifiziert einen Teil oder eine spezifische Ansicht einer Ressource. Dies wird sehr häufig in RDF verwendet, um mehrere Ressourcen innerhalb eines gemeinsamen "Basis-IRI" zu definieren.
 - *Beispiel (IRI mit Fragment):* `http://example.org/Vokabular#BegriffA`

Warum http(s) -IRIs in RDF so wichtig sind

Obwohl es auch andere Schemata wie `urn` (Uniform Resource Name) gibt (z.B. `urn:isbn:0451450523` für ein Buch), haben `http`-basierte IRIs einen entscheidenden Vorteil im Kontext des Semantic Web:

- **Dereferenzierbarkeit ("Look-up"):** Eine `http`-IRI *kann* als Webadresse verwendet werden. Man kann versuchen, sie in einem Browser aufzurufen. Idealerweise liefert der Server unter dieser Adresse weitere (RDF-)Daten über die identifizierte Ressource zurück. Dies ist ein Kernprinzip von Linked Data.
- **Bekanntheit und Infrastruktur:** Das HTTP-Protokoll und die dazugehörige Infrastruktur (DNS, Webserver) sind weltweit etabliert.

Anwendungsbeispiele für IRIs in RDF

Stellen wir uns vor, wir wollen ausdrücken: "Aachen ist eine Großstadt und liegt in Deutschland." Wir könnten dafür folgende (beispielhafte) DBpedia-IRIs verwenden:

- **Ressource Aachen (Subjekt):** `<http://dbpedia.org/resource/Aachen>`
- **Konzept Großstadt (Objekt, Typ der Ressource):** `<http://dbpedia.org/ontology/City>` (Hier ist `City` der spezifische Begriff innerhalb der DBpedia-Ontologie)
- **Eigenschaft/Beziehung "liegt in" (Prädikat):** `<http://dbpedia.org/ontology/isPartOf>`
- **Ressource Deutschland (Objekt):** `<http://dbpedia.org/resource/Germany>`
- Das folgende ist ebenfalls eine gültige IRI: `<urn:ISBN:3-8273-7019-1>`

Hinweis: Die spitzen Klammern `<...>` werden in manchen Serialisierungsformaten (wie Turtle, das wir später kennenlernen) verwendet, um IRIs im Datensatz zu kennzeichnen.

Ressourcen typisieren mit `rdf:type`

Bisher haben wir gesehen, wie wir Aussagen über Ressourcen machen können, indem wir ihnen Eigenschaften (Prädikate) und Werte (Objekte) zuweisen. Eine sehr häufige und wichtige Aussage ist die Angabe, **welcher Art oder welchen Typs** eine Ressource ist. Ist es eine Person, eine Stadt, ein Buch oder ein Film?

Für diese Art der Aussage gibt es im RDF-Standardvokabular ein spezielles Prädikat: `rdf:type`.

Ein Tripel mit `rdf:type` hat typischerweise folgenden Aufbau:

- **Subjekt:** Die IRI der Ressource, die wir typisieren möchten (z.B. `<http://dbpedia.org/resource/Aachen>`).
- **Prädikat:** Die IRI für "type", nämlich `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` (häufig mit dem Präfix `rdf:` als `rdf:type` abgekürzt). **Hinweis:** Die Schreibweise `rdf:type` ist eine gängige Abkürzung. Wie solche Präfixe (hier `rdf:`) definiert werden, um lange IRIs kompakt darzustellen, werden wir uns genauer ansehen, wenn wir Serialisierungsformate wie Turtle besprechen.
- **Objekt:** Eine IRI, die eine Klasse oder einen Typ repräsentiert (z.B. `<http://dbpedia.org/ontology/City>`). Wichtig ist: Diese Klasse ist selbst eine Ressource im RDF-Modell und wird durch ihre eigene IRI identifiziert.

Ein konkretes Beispiel-Tripel wäre also: (`<http://dbpedia.org/resource/Aachen>` , `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` , `<http://dbpedia.org/ontology/City>`)

Verständlicher ausgedrückt bedeutet dieses Tripel: "Die Ressource Aachen ist vom Typ Stadt".

Das Prädikat `rdf:type` ist fundamental. Es erlaubt uns, Ressourcen zu **kategorisieren** (z.B. alle Instanzen von 'Stadt' zu finden oder alle 'Personen' aufzulisten), spezifische Eigenschaften für bestimmte Typen zu erwarten und später komplexere Abfragen und Schlussfolgerungen (Inferenzen) durchzuführen. Das werden wir später noch genauer betrachten.

Literale (Literals)

Literale repräsentieren konkrete Datenwerte wie Zeichenketten, Zahlen oder Daten (z.B. ein Datumswert) und können in einem RDF-Tripel **ausschließlich als Objekt** auftreten. Im Aachen-Beispiel (angenommen, es geht um die Einwohnerzahl) wäre `"247380"` ein Literal.

Man unterscheidet hauptsächlich zwei Arten von Literalen:

- **Einfache (oder "plain") Literale:**
 - Bestehen aus einer reinen Zeichenkette, z.B. `"Xenokrates"`.
 - Können optional ein **Sprach-Tag (Language Tag)** nach [RFC 5646](#) (z.B. `de`, `en-GB`) enthalten, um die Sprache des Literals anzugeben. Dies wird mit einem `@` an die Zeichenkette angefügt.
 - Beispiel: `"Grundzüge"@de`, `"Logic"@en`.
 - Es gibt übrigens einen Sprachtag für Kölsch `ksh`, aber nicht für Öcher Platt.
- **Typisierte Literale (Typed Literals):**
 - Haben einen expliziten **Datentyp**, der angibt, wie die Zeichenkette zu interpretieren ist (z.B. als Zahl, Datum, etc.). Dieser Datentyp wird selbst durch eine IRI identifiziert.
 - **Syntax:** `"Datenwert"^^<Datentyp-IRI>`
 - Am gebräuchlichsten sind Datentypen aus dem **XML Schema Datatypes (XSD)** Namensraum (z.B. `xsd:integer`, `xsd:string`, `xsd:float`, `xsd:date`). **Hinweis:** Die Schreibweise `xsd:integer` ist eine gängige Abkürzung. Das Präfix `xsd:` verweist hier auf den XML Schema Datatypes Namensraum (`<http://www.w3.org/2001/XMLSchema#>`). Wie solche Präfixe allgemein definiert und verwendet werden, um lange IRIs kompakt darzustellen, werden wir uns genauer ansehen, wenn wir Serialisierungsformate wie Turtle besprechen. XML Schema erlaubt auf die Definition von **eigenen Datentypen**.
 - Beispiel für ein typisiertes Literal mit Präfix: `"247380"^^xsd:integer`

- Beispiel mit vollständiger Datentyp-IRI: "123"^^<http://www.w3.org/2001/XMLSchema#int>
- **Wichtig (Implizite Typisierung):**
 - Ein Literal ohne expliziten Datentyp UND ohne Sprach-Tag wird standardmäßig als `xsd:string` interpretiert.
 - Ein Literal mit einem Sprach-Tag (aber ohne expliziten Datentyp) ist vom impliziten Typ `rdf:langString`.
- **Semantische Interpretation durch Datentypen:** Datentypen sind entscheidend für die korrekte maschinelle Verarbeitung und Interpretation von Werten.
 - Ohne Datentyp würde ein Vergleich wie `"02" < "100"` lexikographisch (Zeichen für Zeichen) ausgewertet. Dies könnte zu unerwarteten Ergebnissen führen (z.B. `"02"` wäre kleiner als `"100"`, aber `"11"` wäre größer als `"100"` und kleiner als `"2"`).
 - Mit dem Datentyp `xsd:integer` wird der Vergleich korrekt numerisch durchgeführt (`2 < 100`, `11 < 100`, `2 < 11`).
- RDF selbst definiert nur zwei spezielle Datentypen vor: `rdf:HTML` (für HTML-Fragmente) und `rdf:XMLLiteral` (für XML-Inhalte). Alle anderen, wie die XSD-Typen, müssen explizit referenziert werden.

XML-Schema Datentypen-Hierarchie

Hier sind die eingebauten Datentypen von XML Schema kurz dargestellt. Die verschiedenen Typ-Kategorien sind durch Textfarben hervorgehoben.

Legende zur Text-Hierarchie und Farbkodierung

Die folgende hierarchische Liste verwendet Textfarben, um die verschiedenen Kategorien von Datentypen leichter unterscheidbar zu machen. Die Legende illustriert diese Farbkodierung:

- **Spezielle Typen** (z.B. ``anyType``, ``anySimpleType``)
 - Dies sind grundlegende Obertypen in der XML-Schema-Hierarchie.
- **Komplexe Typen** (z.B. all complex types)
 - Beschreibt Typen, die Elemente und Attribute enthalten können.
- **Primitive Typen** (z.B. ``string``, ``decimal``)
 - Grundlegende, nicht weiter zerlegbare Datentypen.
- **Andere eingebaute atomare Typen** (z.B. ``integer``, ``date``)
 - Von primitiven Typen abgeleitete atomare Typen.
- **Eingebaute Listen-Typen** (z.B. ``NMTOKENS``)
 - Typen, die eine Liste von atomaren Werten darstellen.
- **Ableitungen:** Einrückungen in der Liste zeigen an, dass ein Typ vom übergeordneten Typ abgeleitet ist. Verlinkungen führen zur jeweiligen W3C-Spezifikation.

Datentyp-Hierarchie

- **``anyType`` (Spezielle Typen)**
 - all complex types (Komplexe Typen)
 - **``anySimpleType`` (Spezielle Typen)**
 - **``anyAtomicType`` (Spezielle Typen)**
 - ``anyURI`` (Primitive Typen)
 - ``base64Binary`` (Primitive Typen)
 - ``boolean`` (Primitive Typen)
 - ``date`` (Primitive Typen)
 - ``dateTime`` (Primitive Typen)
 - ``dateTimeStamp`` (Andere eingebaute atomare Typen)
 - ``decimal`` (Primitive Typen)
 - ``integer`` (Andere eingebaute atomare Typen)
 - ``long`` (Andere eingebaute atomare Typen)
 - ``int`` (Andere eingebaute atomare Typen) ``short`` (Andere eingebaute atomare Typen) * ``byte`` (Andere eingebaute atomare Typen)
 - ``nonNegativeInteger`` (Andere eingebaute atomare Typen)
 - ``positiveInteger`` (Andere eingebaute atomare Typen) ``unsignedLong`` (Andere eingebaute atomare Typen)
 - ``unsignedInt`` (Andere eingebaute atomare Typen) ``unsignedShort`` (Andere eingebaute atomare Typen) * ``unsignedByte`` (Andere eingebaute atomare Typen)
 - ``nonPositiveInteger`` (Andere eingebaute atomare Typen)
 - * ``negativeInteger`` (Andere eingebaute atomare Typen)
 - ``double`` (Primitive Typen)
 - ``duration`` (Primitive Typen)
 - ``dayTimeDuration`` (Andere eingebaute atomare Typen)
 - ``yearMonthDuration`` (Andere eingebaute atomare Typen)

- ``float`` (Primitive Typen)
- ``gDay`` (Primitive Typen)
- ``gMonth`` (Primitive Typen)
- ``gMonthDay`` (Primitive Typen)
- ``gYear`` (Primitive Typen)
- ``gYearMonth`` (Primitive Typen)
- ``hexBinary`` (Primitive Typen)
- ``NOTATION`` (Primitive Typen)
- ``QName`` (Primitive Typen)
- ``string`` (Primitive Typen)
 - ``normalizedString`` (Andere eingebaute atomare Typen)
 - ``token`` (Andere eingebaute atomare Typen)
 - ``language`` (Andere eingebaute atomare Typen) ``Name`` (Andere eingebaute atomare Typen) ``NCName`` (Andere eingebaute atomare Typen) ``ENTITY`` (Andere eingebaute atomare Typen) ``ID`` (Andere eingebaute atomare Typen) ``IDREF`` (Andere eingebaute atomare Typen) * ``NMTOKEN`` (Andere eingebaute atomare Typen)
 - ``time`` (Primitive Typen)
- ``ENTITIES`` (Eingebaute Listen-Typen)
- ``IDREFS`` (Eingebaute Listen-Typen)
- ``NMTOKENS`` (Eingebaute Listen-Typen)

Blank Nodes (Leere / Anonyme Knoten)

Blank Nodes sind ein zentrales Konzept des Resource Description Framework (RDF). Sie repräsentieren **Ressourcen, denen bewusst keine globale IRI** (Internationalized Resource Identifier) zugewiesen wird. Dies geschieht typischerweise, weil eine solche eindeutige, globale Kennung für das jeweilige Szenario nicht erforderlich ist oder (noch) nicht existiert. Blank Nodes fungieren somit als Platzhalter für Entitäten, die zwar beschrieben werden müssen, aber keinen öffentlich stabilen und referenzierbaren Bezeichner benötigen.

Detaillierte Eigenschaften und Semantik

Unbenannte Ressourcen & Existenzielle Quantifizierung Logisch betrachtet drücken Blank Nodes aus, dass *etwas* mit bestimmten Eigenschaften existiert, ohne dieses „Etwas“ explizit global zu benennen. Sie agieren wie **existenzielle Quantifikatoren** ($\exists$) in der Logik. Man sagt damit aus: "Es gibt *etwas*, das diese und jene Eigenschaften hat."

- **Beispiel:** Um auszudrücken, dass "Hans Müller eine Adresse hat, die in der Melatener Straße 111 in Aachen liegt", könnte die Adresse selbst ein Blank Node sein, wenn sie keine eigene, global relevante IRI hat. Wir wissen, dass *eine solche Adresse existiert* und zu Hans Müller gehört, aber die Adresse selbst braucht vielleicht keine eigene, über diesen Kontext hinausgehende Identität.

In einem RDF-Graphen würde die Struktur dieser Information folgendermaßen aussehen:

- Es gäbe eine Ressource, die Hans Müller repräsentiert (z.B. identifiziert durch eine IRI wie `<http://example.org/person/HansMueller>`).
- Diese Hans-Müller-Ressource hätte eine Eigenschaft (z.B. `schema:address` von `schema.org`).
- Der Wert dieser `schema:address`-Eigenschaft wäre ein anonymer Knoten – unser Blank Node – der die spezifische Adresse von Hans Müller darstellt.
- Dieser Blank Node für die Adresse hätte dann wiederum eigene Eigenschaften mit konkreten Werten (Literalen):
 - Eine Eigenschaft `schema:streetAddress` mit dem Wert "Melatener Straße 111" (dies ist ein Literal).
 - Eine Eigenschaft `schema:addressLocality` mit dem Wert "Aachen" (ebenfalls ein Literal).
 - Eine Eigenschaft `schema:postalCode` mit dem Wert "52072" (auch ein Literal).

Der Blank Node bündelt also die zusammengehörigen Teile der Adresse (die als Literale dargestellt werden), ohne dass die Adresse selbst einen globalen Namen (IRI) benötigt. Die Literale "Melatener Straße 111", "Aachen" und "52072" sind hierbei die Werte der Eigenschaften des Blank Nodes, der die Adresse repräsentiert.

Lokale Identität Die Identität eines Blank Nodes ist **auf einen konkreten RDF-Graphen beschränkt**. Beim Parsen oder Serialisieren eines RDF-Graphen können interne Bezeichner für Blank Nodes (die in manchen Notationen z.B. mit `_:b0`, `_:b1` oder `_:irgendeinName` beginnen) variieren. Zwei Blank Nodes mit demselben Label in unterschiedlichen RDF-Dokumenten sind nicht notwendigerweise dieselbe Ressource.

Position im Tripel Blank Nodes können als **Subjekt** oder **Objekt** eines RDF-Tripels auftreten. Sie können jedoch **niemals als Prädikat** verwendet werden, da Prädikate immer durch IRIs identifiziert werden müssen, um die Bedeutung der Beziehung eindeutig zu definieren.

Logische Entsprechung (Skolemisierung) *Anmerkung:* In der formalen Logik entspricht die Rolle eines Blank Nodes dem Konzept einer **Skolemkonstante** (oder Skolemfunktion). Diese werden im Prozess der Skolemisierung eingeführt, um existenzielle Quantoren in logischen Formeln zu eliminieren. Für das grundlegende Verständnis von RDF reicht es aber meist, sie als "unbenannte Ressourcen" zu verstehen.

Blank Nodes in der Praxis: Interne ObjectIDs und Serialisierung

Viele RDF-Speicher und Graphdatenbanken nutzen intern eigene Mechanismen wie **ObjectIDs (OIDs)**, um alle Ressourcen – auch Blank Nodes – effizient zu verwalten und zu referenzieren. Es ist wichtig, diese internen IDs von der logischen Bedeutung und der externen Repräsentation von Blank Nodes zu unterscheiden.

Aspekt	Blank Node (RDF-Modell)	Interne ObjectID (Datenbank)
Sichtbarkeit	Extern potenziell sichtbar (z.B. durch spezielle Syntax in Serialisierungsformaten), aber ohne garantierte globale Bedeutung oder Stabilität der externen Kennung	Rein intern, für den RDF-Benutzer i.d.R. unsichtbar
Eindeutigkeit	Nur innerhalb eines spezifischen Graphen/Kontexts	Global eindeutig innerhalb des jeweiligen Speichers
Persistenz	Die externen Label (wie <code>_:b1</code>) können sich bei jeder Serialisierung/Parsing ändern	Bleibt über Transaktionen und Speicheroperationen hinweg meist stabil für die Lebenszeit des Knotens im Speicher

Auch wenn ein Blank Node intern durch eine persistente ObjectID repräsentiert wird, bleibt seine Natur als Blank Node im Sinne des RDF-Datenmodells davon unberührt: Er hat keine ihm eigene, stabile, global auflösbare IRI. Die konkrete Syntax zur Darstellung von Blank Nodes kann sich, je nach Serialisierungsformat (z.B. Turtle, RDF/XML), bei jeder Neu-Serialisierung eines Graphen ändern.

Der RDF-Graph

Eine Menge von RDF-Tripeln bildet einen **RDF-Graphen**. Dieser Graph kann als ein Netzwerk von Knoten und gerichteten, beschrifteten Kanten visualisiert werden:

- **Knoten** repräsentieren Ressourcen (identifiziert durch IRIs oder als Blank Nodes) und Werte (Literals).
- **Kanten** repräsentieren Prädikate (Eigenschaften oder Beziehungen), die immer durch IRIs identifiziert werden. Sie sind gerichtet und verlaufen vom Subjekt zum Objekt eines Tripels.

Die Fähigkeit, dass dieselbe IRI in einem Tripel Subjekt und in einem anderen Objekt sein kann, ermöglicht die Verknüpfung von Aussagen. Dies bildet die Grundlage für die Navigation durch die Daten und für das Ziehen von Schlussfolgerungen (Inferenzen).

In grafischen Darstellungen von RDF-Graphen werden die Hauptkomponenten oft durch unterschiedliche Formen gekennzeichnet, um die Lesbarkeit zu erhöhen:

Komponente	Symbolische Form	Beschreibung
Ressourcen-IRI	Ellipse (oder abgerundetes Rechteck)	Ist mit einer IRI beschriftet, oft abgekürzt (z.B. <code>ex:Aachen</code>).
Blank Node	Kreis (oder gestrichelte/leere Ellipse)	Hat keinen globalen Namen oder nur ein lokales Label (beginnt mit einem Unterstrich, z.B. <code>_:adresse123</code>).
Datenliteral	Rechteck	Enthält den konkreten Wert, z.B. <code>"Sibylle Keupen"</code> oder <code>"42"^^xsd:integer</code> .
Prädikat	Beschrifteter, gerichteter Pfeil	Zeigt vom Subjekt- zum Objekt-Knoten; <code>rdf:type</code> wird oft besonders hervorgehoben.

So entsteht ein intuitiv erfassbarer Graph: Die Knotenformen verraten auf einen Blick, ob es sich um eine benannte Ressource, einen Blank Node oder ein Literal handelt, während die Pfeile die semantischen Beziehungen zwischen ihnen abbilden.

Hinweis zu Abkürzungen: Um nicht immer die vollständigen, oft langen IRIs schreiben zu müssen, wird in RDF-Notationen üblicherweise ein Abkürzungsmechanismus mittels **Präfixen** (auch als Namespaces bezeichnet) verwendet. Diesen Mechanismus werden wir später genauer kennenlernen. In den folgenden Beispielen verwenden wir bereits solche abgekürzten Formen wie `:Aachen` (wobei `:` für ein Standardpräfix steht, das später definiert wird).

Dieser graphenbasierte Ansatz, kombiniert mit global eindeutigen Identifikatoren (IRIs), unterscheidet RDF fundamental von traditionellen relationalen Datenbankmodellen und bildet die Stärke von RDF für die Datenintegration und Wissensrepräsentation.

Beispiel eines RDF-Graphen

Der folgende Graph besteht aus insgesamt **6 Tripeln** mit den folgenden Komponenten:

1. **Ressource** `:Aachen`
 - Typ: `:Stadt`
 - Beziehung: `:hasMayor` → `:Q99688800`
2. **Ressource** `:Q99688800` (Bürgermeisterin)
 - Dateneigenschaft: `:name` `"Sibylle Keupen"`
 - Beziehung: `:hatAdresse` → `_:adr`
3. **Blank Node** `_:adr` (Adresse von Sibylle Keupen)

- :hatStraße "Markt"
- :inStadt "52062 Aachen"

Zusammengefasst als natürliche Sprache:

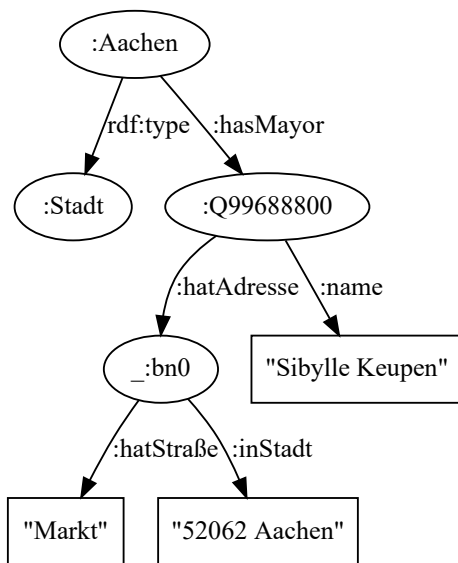
Die Stadt **Aachen** (eine Ressource) ist vom Typ **Stadt** und hat als Bürgermeisterin die Person (eine Ressource), die den Namen **"Sibylle Keupen"** (ein Literal) trägt. Diese Person hat eine Adresse (repräsentiert durch einen Blank Node), welche die Straße **"Markt"** (ein Literal) und die Postleitzahl und Stadt **"52062 Aachen"** (ein Literal) umfasst.

Visuelle Struktur

```
In [10]: %%rdf turtle
@prefix : <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

:Aachen rdf:type :Stadt ; :hasMayor :Q99688800 .

:Q99688800 :name "Sibylle Keupen" ; :hatAdresse _:bn0 .
_:bn0 :hatStraße "Markt" ;
      :inStadt "52062 Aachen" .
```



Verwendung von %%rdf turtle in Jupyter Notebooks

Die Zeile `%%rdf turtle`, die Sie in der obigen Zelle am Anfang der Code-Zelle sehen, ist eine sogenannte **"Magic Command"** (oder "Cell Magic"). Diese speziellen Befehle sind nicht Teil der Python-Sprache selbst, sondern eine Erweiterung, die von der IPython-Umgebung (dem Kern von Jupyter Notebooks für Python) bereitgestellt wird, oft durch zusätzliche Bibliotheken.

Was bewirkt %%rdf turtle ?

Wenn Sie eine Zelle mit `%%rdf turtle` beginnen, signalisieren Sie Jupyter, dass der gesamte Inhalt dieser Zelle als **RDF-Daten im Turtle-Format** interpretiert werden soll.

1. **Parser-Aktivierung:** Eine spezialisierte Bibliothek im Hintergrund (in Ihren Notebooks `jupyter-rdfify`, die wiederum `rdflib` nutzt) wird aktiviert, um den Turtle-Code zu parsen.
2. **RDF-Graph-Erstellung:** Der geparsete Turtle-Code wird in ein internes RDF-Graph-Datenmodell umgewandelt (z.B. ein `rdflib.Graph` - Objekt).
3. **Visualisierung/Ausgabe:** Die `jupyter-rdfify` -Bibliothek oder eine ähnliche Erweiterung sorgt dann dafür, dass dieser RDF-Graph direkt unter der Zelle visualisiert wird.

RDF im Kontext von Datenmodellen (Relationale DBs, NoSQL)

Nachdem wir das grundlegende RDF-Datenmodell kennengelernt haben, ist es hilfreich, RDF in den breiteren Kontext von Datenbanksystemen einzuordnen, insbesondere im Vergleich zum relationalen Modell, das Ihnen vertraut ist, und zu den später *entwickelten* NoSQL-Ansätzen.

Das relationale Modell zeichnet sich durch eine stark strukturierte, tabellarische Datenspeicherung mit vordefinierten Schemata und ACID-Eigenschaften aus. Die NoSQL-Bewegung ("Not Only SQL") entstand später, um bestimmte Limitierungen relationaler Datenbanken zu adressieren,

insbesondere im Hinblick auf Skalierbarkeit für sehr große Datenmengen (Big Data), Flexibilität bei Schemaänderungen und die Handhabung unstrukturierter oder semi-strukturierter Daten.

RDF und seine Beziehung zu NoSQL:

Interessanterweise hat RDF, das bereits 1999 vom W3C als *Empfehlung (Standard)* veröffentlicht wurde, einige der Kernideen, die später mit NoSQL assoziiert wurden, vorweggenommen – insbesondere die **Schema-Flexibilität** und den **graphenbasierten Ansatz**.

- **Schema-Flexibilität:** RDF erfordert nicht von vornherein ein starres Schema. Daten können hinzugefügt und erweitert werden, ohne dass ein zentrales Schema geändert werden muss. Semantische Schemata (Ontologien mit RDFS/OWL) können optional und schrittweise definiert werden, um die Bedeutung der Daten zu präzisieren ("Schema-later" oder "Schema-optional").
- **Graphenmodell:** RDF ist inhärent graphenbasiert und eignet sich natürlich für die Darstellung von komplexen Beziehungen und vernetzten Daten. Viele NoSQL-Datenbanken, insbesondere Graphdatenbanken (wie Neo4j), nutzen ebenfalls Graphenmodelle, oft aber mit unterschiedlichen Schwerpunkten und Abfragesprachen.

Obwohl es also Überlappungen in den technischen Ansätzen gibt (z.B. Flexibilität, Umgang mit nicht-tabellarischen Daten), ist der **primäre Fokus von RDF ein anderer:**

- **Wissensrepräsentation:** Es geht darum, Wissen maschinenverständlich zu machen.
- **Datenintegration im globalen Maßstab:** RDF zielt darauf ab, Daten aus heterogenen Quellen über das Web hinweg zu verknüpfen und integrierbar zu machen, basierend auf global eindeutigen IRIs.
- **Semantische Interoperabilität:** Durch die Verwendung gemeinsamer Vokabulare und Ontologien sollen Daten nicht nur syntaktisch, sondern auch semantisch interoperabel werden.
- **Logisches Schließen (Inferenz):** Die Möglichkeit, aus explizit vorhandenen Daten und ontologischen Regeln neues Wissen abzuleiten, ist ein Kernmerkmal.

Die folgende Tabelle stellt die Ansätze gegenüber:

Vergleichstabelle: RDF, Relational (SQL) und NoSQL

Feature	Relational (SQL)	NoSQL (Allgemein)	RDF (Graph-Datenmodell) & Triple Stores
Struktur	Tabellarisch (Zeilen, Spalten)	Variabel (Key-Value, Dokument, Graph, Spaltenfamilie)	Graph-basiert (Knoten, Kanten/Beziehungen)
Primäres Datenmodell	Relationen	<i>Abhängig vom Typ (z.B. Key-Value Paare, Dokumente, Spaltenfamilien, Property Graphen)</i>	Tripel (Subjekt-Prädikat-Objekt)
Schema	Vordefiniert, starr (Schema-on-Write)	Dynamisch, flexibel ("schemas", Schema-on-Read)	Flexibel ("Schema-optional" oder "Schema-later"); Semantisches Schema durch Ontologien (RDFS, OWL) definierbar
Skalierungsmodell	Primär vertikal; horizontal komplex	Primär horizontal	Variiert stark je nach Triple Store Implementierung (kann horizontal oder vertikal sein)
Datenintegrität/-konsistenz	ACID (Atomicity, Consistency, Isolation, Durability)	Oft BASE (Basically Available, Soft-state, Eventually Consistent)	Konsistenz und Validierung oft über Ontologien (RDFS, OWL) und Constraints (z.B. SHACL) definiert; Transaktionsunterstützung variiert je nach Triple Store
Abfragesprache	SQL	Diverse (spezifisch pro NoSQL-Typ, z.B. CQL, Cypher)	SPARQL
Fokus	Strukturierte Datenspeicherung, Transaktionen	Skalierbarkeit, Flexibilität, spezifische Anwendungsfälle	Datenintegration, semantische Verknüpfung, Inferenz, Wissensrepräsentation im globalen Kontext
Typische Workloads	Transaktionsorientiert, strukturierte Daten, komplexe Joins	Big Data, Echtzeit-Webanwendungen, unstrukturierte/semi-strukturierte Daten	Wissensmanagement, Linked Data, Datenintegration aus heterogenen Quellen, semantische Suche, Forschung, Aufbau globaler Informationssysteme
Beispielsysteme	MySQL, PostgreSQL, Oracle	Cassandra, Redis, Neo4j (Graph-DB basierend auf <i>Property Graphen</i> , nicht primär RDF), MongoDB	Apache Jena Fuseki, Virtuoso, GraphDB, Stardog, Amazon Neptune (unterstützt RDF)

Kernstärken und Einzigartigkeit von RDF

RDF ist mehr als "nur eine weitere Datenbank-Technologie". Es ist ein **semantisches Datenmodell**, das auf die Schaffung eines maschinenverständlichen "Web of Data" abzielt. Die Kombination aus:

- Globalem Namensraum durch IRIs
- Flexibler, graphbasierter Struktur
- Formaler Semantik durch Ontologien
- Inferenz-Fähigkeiten

Ausblick

Wir haben die Grundlagen des RDF Datenmodells kennengelernt. Um RDF-Graphen übertragen zu können, müssen sie in eine übertragbare Form gebracht werden. Als nächstes werden wir unterschiedliche Serialisierungsformate kennenlernen.

Weiterführende Ressourcen

- [Thom Heath & Chris Bizer: "Linked Data: Evolving the Web into a Global Data Space"](#) (Englisches Buch, grundlegend)
 - [RDF Primer \(W3C\)](#) (Englisch, W3C Einführung)
 - [RDF 1.1 Concepts and Abstract Syntax \(W3C\)](#) (Englisch, formale Spezifikation der RDF-Konzepte)
-

Übungsaufgaben zum RDF-Intro

Diese Aufgaben sollen Ihnen helfen, die Kernkonzepte von RDF zu festigen. Die Aufgaben und Lösungen sind nur für Sie bestimmt, sie brauchen nicht abgegeben zu werden. Sie können aber bei Bedarf in der Vorlesung besprochen werden.

Aufgabe 1: Tripel identifizieren (Verständnis)

Gegeben sei die folgende Aussage: "Die Stadt Aachen liegt im Bundesland Nordrhein-Westfalen und hat ungefähr 250.000 Einwohner."

1. Zerlegen Sie diese Information in einzelne Aussagen, die sich jeweils als ein RDF-Tripel (Subjekt, Prädikat, Objekt) formulieren lassen.
 2. Identifizieren Sie für jedes Tripel die Komponenten:
 - Was wäre das Subjekt (typischerweise eine IRI)?
 - Was wäre das Prädikat (eine IRI, die die Beziehung beschreibt)?
 - Was wäre das Objekt (eine IRI oder ein Literal)?
 3. Welche der Objekte sind Literale? Könnten Sie für diese Literale passende XSD-Datentypen vorschlagen (z.B. `xsd:string` , `xsd:integer`)?
-

Aufgabe 2: Eigene RDF-Aussagen formulieren (Anwendung)

Denken Sie sich drei eigene, zusammenhängende Aussagen über ein Thema Ihrer Wahl aus (z.B. über Ihre Universität, eine bekannte Persönlichkeit, ein aktuelles Ereignis).

1. Formulieren Sie jede Aussage als ein RDF-Tripel.
 2. Verwenden Sie für Ihre Subjekte, Prädikate und Objekte (wo passend) fiktive, aber plausible IRIs (z.B. `<http://mein-beispiel.org/thema/PersonX>` , `<http://mein-beispiel.org/vokabular/hatAlter>`). Kennzeichnen Sie Literale entsprechend (z.B. mit Anführungszeichen und optional einem Datentyp wie `"Text"^^xsd:string` oder `"25"^^xsd:integer`).
 3. Begründen Sie kurz Ihre Wahl für die einzelnen Komponenten (warum IRI, warum Literal).
-

Aufgabe 3: Einen kleinen RDF-Graphen konzipieren und beschreiben (Visualisierung & Struktur)

Stellen Sie sich vor, Sie möchten Informationen über Kurse an einer Universität modellieren. Betrachten Sie die folgenden Informationen:

- Der Kurs `:WebTech` (Web Technologien) wird von Professor `:Einstein` gehalten.
- Professor `:Einstein` gehört zum Institut `:Informatik` .
- Der Kurs `:WebTech` hat das Thema `"Semantic Web"` .
- Der Kurs `:WebTech` hat 3 ECTS-Punkte.

1. Formulieren Sie jede dieser Informationen als ein RDF-Tripel. Verwenden Sie dabei die bereits angedeuteten (fiktiven) IRIs und geeignete Literale.
 2. Zeichnen Sie (auf Papier oder mit einem einfachen Zeichentool) den RDF-Graphen, der diese Tripel darstellt. Verwenden Sie die im Notebook besprochenen Konventionen (Ellipsen für IRIs, Rechtecke für Literale, Pfeile für Prädikate).
 3. Listen Sie alle im Graphen vorkommenden IRIs, Literale und Prädikate auf.
-

Aufgabe 4: Blank Nodes – Anwendungsszenario (Konzepttransfer)

Betrachten Sie die folgende Situation: "Ein Forschungsprojekt mit dem Titel 'Zukunft der Mobilität' hat mehrere Projektpartner. Einer dieser Partner ist eine Arbeitsgruppe, die keinen eigenen offiziellen Namen oder eine eigene Webseite (und somit keine klare IRI) hat, aber sie besteht aus drei Forschern: Dr. Schmidt, Dr. Meier und Dr. Huber."

1. Wie könnten Sie die namenlose Arbeitsgruppe im RDF-Graphen des Forschungsprojekts darstellen? Begründen Sie Ihre Wahl.
 2. Formulieren Sie beispielhafte Tripel, um auszudrücken, dass das Projekt `proj:ZukunftMobilität` diese Arbeitsgruppe als Partner hat und dass Dr. Schmidt (identifiziert durch `pers:Schmidt`) Mitglied dieser Arbeitsgruppe ist. Verwenden Sie für die Arbeitsgruppe die von Ihnen in Teil 1 gewählte Darstellung. Die Eigenschaften könnten z.B. `proj:hatPartner` und `gruppe:hatMitglied` sein.
-

Aufgabe 5: RDF im Kontext – Stärken reflektieren (Reflexion)

Wir haben RDF mit relationalen Datenbanken und NoSQL-Ansätzen verglichen.

1. Erläutern Sie anhand eines selbst gewählten Beispiels (anders als im Notebook), wie die Schema-Flexibilität von RDF im Vergleich zu einem starren relationalen Schema vorteilhaft sein kann.
2. Das Notebook nennt "Datenintegration im globalen Maßstab" als einen Fokus von RDF. Was ist Ihrer Meinung nach die wichtigste Voraussetzung, die RDF hierfür mitbringt?

In []:

In []:

```
# !pip install -q requests -U --user
# !pip install -q jupyter-rdfify~1.3 --index-url https://git.rwth-aachen.de/api/v4/projects/49225/packages/pypi/simple
%reload_ext jupyter-rdfify
```

Turtle: Eine praktische Einführung in die RDF-Serialisierung

Dieses Jupyter Notebook bietet eine Einführung in die Turtle-Syntax (Terse RDF Triple Language) für RDF (Resource Description Framework). Es richtet sich an Personen mit Grundkenntnissen in RDF und vermittelt die Syntax von Turtle. *Hinweis: Die %%rdf turtle Syntax, die wir hier verwenden, um RDF-Graphen direkt im Notebook zu visualisieren, wird durch die Python-Bibliothek jupyter-rdfify ermöglicht, die am Lehrstuhl "Datenbanken und Informationssystem" der RWTH Aachen erstellt wurde. Sie parst den Turtle-Code, erstellt daraus einen rdflib - Graphen und stellt diesen dann grafisch sowie als Tripel-Liste dar. Sie ermöglicht es auch SPARQL Anfragen an eine Graphdatenbank zu stellen (kommt später)*

Einleitung

Das Resource Description Framework (RDF) dient der Modellierung von Informationen als Tripel, bestehend aus Subjekt, Prädikat und Objekt. Diese Tripel bilden einen Graphen, der komplexe Wissensnetze darstellen kann. Abstrakte RDF-Datenmodelle benötigen jedoch eine konkrete Syntax, um gespeichert, ausgetauscht und von Maschinen verarbeitet werden zu können. Diese Notwendigkeit ergibt sich aus dem Bedürfnis, heterogene Datenformate und Systeme auf einer semantisch reichen Basis zu integrieren und Interoperabilität zu gewährleisten. RDF-Serialisierungen sind somit entscheidend, um die maschinenlesbare Repräsentation von RDF-Graphen zu ermöglichen.

Überblick gängiger Serialisierungen

Es existieren verschiedene Serialisierungsformate für RDF, die sich in Lesbarkeit, Kompaktheit und Anwendungsfällen unterscheiden. Zu den gängigsten gehören RDF/XML, JSON-LD und Turtle.

Um die Syntaxunterschiede zu verdeutlichen, werden wir beispielhaft einfache Datensätze – hier Produktinformationen und Kontaktdaten – "rdf-ifizieren". Das bedeutet, wir überführen die Informationen in eine RDF-konforme Struktur (Subjekt-Prädikat-Objekt-Tripel), wobei wir für Entitäten (die Subjekte) und Eigenschaften (die Prädikate) IRIs verwenden und für konkrete Werte oft Literale. Anschließend stellen wir diese RDF-Struktur in den jeweiligen Formaten dar.

Beispieldaten: Produktinformationen

- Produkt (Subjekt-IRI): `http://example.org/product/item123`
- Eigenschaften und Werte:
 - Produktname (Prädikat `ex:name`): Super-Widget (de, Literal)
 - Marke (Prädikat `ex:brand`): MarkeX (Literal)
 - Preis (Prädikat `ex:price`): 29.99 (Dezimalzahl, Literal)

Beispieldaten: Kontaktdaten

- Person (Subjekt-IRI): `http://example.org/person#maxMustermann`
- Eigenschaften und Werte:
 - Typ (Prädikat `rdf:type`): Person (Klassen-IRI `foaf:Person`)
 - Personenname (Prädikat `foaf:name`): Max Mustermann (Literal)
 - E-Mail (Prädikat `foaf:mbox`): `mailto:max.mustermann@example.com` (IRI)
 - Telefon (Prädikat `foaf:phone`): `tel:+49-123-4567890` (IRI)

Hinweis: Sie müssen die folgenden Formate nicht im einzelnen Verstehen. Turtle wird dann im Detail erklärt, und Sie haben die anderen Formate dann schon mal gesehen.

1. RDF/XML

RDF/XML ist die älteste RDF-Serialisierung und nutzt die XML-Syntax.

• RDF/XML Beispiel (Produktinformationen):

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ex="http://example.org/product/">
  <rdf:Description rdf:about="http://example.org/product/item123">
    <ex:name xml:lang="de">Super-Widget</ex:name>
    <ex:brand>MarkeX</ex:brand>
    <ex:price rdf:datatype="http://www.w3.org/2001/XMLSchema#decimal">29.99</ex:price>
  </rdf:Description>
</rdf:RDF>
```

• RDF/XML Beispiel (Kontaktdaten):

```
<rdf:RDF
```

```

    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/">
<foaf:Person rdf:about="http://example.org/person#maxMustermann">
  <foaf:name>Max Mustermann</foaf:name>
  <foaf:mbox rdf:resource="mailto:max.mustermann@example.com"/>
  <foaf:phone rdf:resource="tel:+49-123-4567890"/>
</foaf:Person>
</rdf:RDF>

```

2. JSON-LD (JSON for Linking Data)

JSON-LD ist eine auf JSON basierende Serialisierung, die darauf abzielt, Linked Data mit dem weit verbreiteten JSON-Format auszudrücken.

• JSON-LD Beispiel (Produktinformationen):

```

{
  "@context": {
    "ex": "http://example.org/product/",
    "name": "ex:name",
    "brand": "ex:brand",
    "price": {
      "@id": "ex:price",
      "@type": "xsd:decimal"
    },
    "xsd": "http://www.w3.org/2001/XMLSchema#"
  },
  "@id": "http://example.org/product/item123",
  "name": { "@value": "Super-Widget", "@language": "de" },
  "brand": "MarkeX",
  "price": "29.99"
}

```

• JSON-LD Beispiel (Kontaktdaten):

```

{
  "@context": {
    "foaf": "http://xmlns.com/foaf/0.1/",
    "name": "foaf:name",
    "mbox": "foaf:mbox",
    "phone": "foaf:phone"
  },
  "@type": "foaf:Person",
  "@id": "http://example.org/person#maxMustermann",
  "name": "Max Mustermann",
  "mbox": "mailto:max.mustermann@example.com",
  "phone": "tel:+49-123-4567890"
}

```

3. Turtle (Terse RDF Triple Language)

Turtle ist eine textbasierte, kompakte und gut lesbare Syntax, die speziell für RDF entwickelt wurde.

• Turtle Beispiel (Produktinformationen):

```

@prefix ex: <http://example.org/product/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:item123
  ex:name "Super-Widget"@de ;
  ex:brand "MarkeX" ;
  ex:price "29.99"^^xsd:decimal .

```

• Turtle Beispiel (Kontaktdaten):

```

@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex: <http://example.org/person#> .

ex:maxMustermann a foaf:Person ;
  foaf:name "Max Mustermann" ;
  foaf:mbox <mailto:max.mustermann@example.com> ;
  foaf:phone <tel:+49-123-4567890> .

```

Vor- und Nachteile bzw. typische Anwendungsfälle

Format	Dateierweiterung	Syntax-Charakteristika	Vorteile	Nachteile	Typische Anwendungsfälle
RDF/XML	.rdf, .owl	XML-basiert, kann verschiedene Strukturen für gleiche Tripel haben	Gute Tool-Unterstützung im XML-Ökosystem, etabliert, unterstützt XML-Datentypen direkt.	Oft sehr ausführlich (verbose), schwer manuell zu lesen/schreiben, mehrere äquivalente Ausdrucksweisen können verwirren.	Legacy-Systeme, Integration mit XML-basierten Werkzeugen, wenn XML explizit gefordert ist, Ontologiedefinitionen (traditionell).

Format	Dateierweiterung	Syntax-Charakteristika	Vorteile	Nachteile	Typische Anwendungsfälle
JSON-LD	.jsonld	JSON-basiert, nutzt @context zur Verlinkung mit RDF-Vokabularen	Leicht für Web-Entwickler zu verstehen, gute Integration in Web-APIs und JavaScript-Anwendungen, von Google für SEO empfohlen (schema.org).	Parzen kann aufwendiger sein, um alle RDF-Nuancen abzubilden; die @context - Definition erfordert Einarbeitung.	Web-APIs, Einbettung strukturierter Daten in Webseiten (SEO), Datenaustausch in JSON-lastigen Ökosystemen, moderne Webanwendungen.
Turtle	.ttl	Textbasiert, kompakt, an N-Triples/SPARQL angelehnt	Sehr gut menschenlesbar und -schreibbar, prägnant durch Präfixe und Kurzschreibweisen, nah am abstrakten Tripel-Modell.	Geringfügig aufwendiger zu parzen als N-Triples (obwohl moderne Parser sehr effizient sind), weniger direkte Anbindung an XML/JSON-Toolchains.	Manuelle Erstellung/Pflege von RDF-Daten, Konfigurationsdateien, Beispiele, Tutorials, Datenaustausch, wenn Lesbarkeit und Prägnanz wichtig sind.

Der Fokus dieses Notebooks: Turtle

Dieses Tutorial konzentriert sich auf **Turtle**, da es eine optimale Balance zwischen menschlicher Lesbarkeit und maschineller Verarbeitbarkeit bietet und in der Semantic Web Community weit verbreitet ist. Turtle ermöglicht es, RDF-Daten kompakt und verständlich zu notieren, ohne dabei die Ausdruckskraft von RDF zu beeinträchtigen. Es ist oft die bevorzugte Syntax für das Erlernen von RDF und das Arbeiten mit RDF-Daten im Alltag.

Turtle-Syntax-Grundlagen

Turtle steht für "**Terse RDF Triple Language**". Es ist ein textbasiertes Format, das dazu dient, **RDF-Tripel als Zeichenketten (Strings) darzustellen**. Das Ziel ist es, RDF-Daten auf eine Weise zu schreiben, die sowohl für Menschen gut lesbar als auch für Maschinen einfach zu parzen ist. Die offizielle Spezifikation finden Sie beim W3C: <https://www.w3.org/TR/turtle/>.

In Turtle werden **IRIs (Internationalized Resource Identifiers) immer in spitzen Klammern (< >) eingeschlossen**. IRIs sind fundamental in RDF, da sie Ressourcen eindeutig identifizieren. Wie wir aus dem vorherigen Notebook wissen, werden alle Informationen in RDF als **Tripel** ausgedrückt. Ein Tripel besteht aus einem **Subjekt**, einem **Prädikat** und einem **Objekt**, gefolgt von einem **Punkt (.)**, der das Ende des Tripels signalisiert.

- **Subjekt:** Die Ressource, die beschrieben wird (immer eine IRI oder ein Blank Node).
- **Prädikat:** Eine Eigenschaft oder eine Beziehung, die das Subjekt betrifft (immer eine IRI).
- **Objekt:** Der Wert der Eigenschaft oder die Ressource, mit der das Subjekt in Beziehung steht (eine IRI, ein Literal oder ein Blank Node).

Die grundlegendste Form eines Tripels in Turtle besteht aus der direkten Aneinanderreihung der IRIs für Subjekt und Prädikat, gefolgt von der IRI oder dem Literal für das Objekt und einem abschließenden Punkt. Stellen wir uns vor, wir wollen ausdrücken, dass die Stadt Aachen den deutschen Namen "Aachen" hat. In Turtle könnten wir das wie folgt schreiben:

```
<http://dbpedia.org/resource/Aachen> <http://www.w3.org/2000/01/rdf-schema#label> "Aachen"@de .
```

Hier ist:

- `<http://dbpedia.org/resource/Aachen>` das **Subjekt** (eine URI für Aachen).
- `<http://www.w3.org/2000/01/rdf-schema#label>` das **Prädikat** (die IRI für die Eigenschaft "label" aus dem RDFS-Vokabular – RDFS lernen wir später noch genauer kennen).
- `"Aachen"@de` das **Objekt** (ein Literal mit Sprach-Tag für Deutsch).

Literale in Turtle

Literale repräsentieren konkrete Datenwerte wie Zeichenketten, Zahlen oder Daten (im Sinne von Kalenderdaten).

Einfache Literale (Strings) und Sprach-Tags

Einfache Zeichenketten-Literale werden in doppelten Anführungszeichen (" . . ") eingeschlossen. Wenn kein expliziter Datentyp angegeben wird, wird für das Literal standardmäßig der Datentyp `xsd:string` (XML Schema String) angenommen.

Man kann einem String-Literal auch ein **Sprach-Tag** hinzufügen, um die Sprache des Strings zu spezifizieren, z.B. `@de` für Deutsch. Dies geschieht durch Anhängen eines `@` gefolgt vom Sprachcode direkt nach den schließenden Anführungszeichen. Die Struktur dieser Sprach-Tags ist in [RFC 5646 \(Tags for Identifying Languages\)](#) der IETF spezifiziert. Die [IANA \(Internet Assigned Numbers Authority\)](#) pflegt die [Language Subtag Registry](#), die alle gültigen Subtags sammelt. (Und ja, es gibt dort auch einen Tag für Kölsch: `ksh`. Jeder kann einen Sprachtag bei der IANA vorschlagen. Siehe Bonusaufgabe am Ende diese Notebooks.)

Beispiel für ein Literal mit deutschem Sprach-Tag:

```
"Aachen"@de
```

Beispiel mit Kölsch:

```
"Oche"@ksh
```

Diese beiden Literale sind vom Datentyp `rdf:langString`.

Datentyp-Literale und Turtle-Kurzschreibweisen

Literale können auch **explizite Datentypen** haben. Dies wird durch Anhängen von `^^` gefolgt von der vollständigen Datentyp-IRI (z.B. `<http://www.w3.org/2001/XMLSchema#integer>`) oder einem Präfixnamen für den Datentyp (z.B. `xsd:integer` , wenn das Präfix `xsd:` definiert wurde) gekennzeichnet.

Für einige gebräuchliche XSD-Datentypen bietet Turtle **Kurzschreibweisen** für die Literale, wodurch die explizite Angabe `^^xsd:...` entfallen kann, da der Typ aus der Schreibweise des Werts erkannt wird:

- **Ganze Zahlen** (z.B. `123` , `-5`) werden automatisch als `xsd:integer` interpretiert.
- **Dezimalzahlen** mit Dezimalpunkt (z.B. `1.23` , `0.5` , `-2.0`) werden automatisch als `xsd:decimal` interpretiert.
- **Gleitkommazahlen** in wissenschaftlicher Notation (z.B. `1.0e6` , `2.5E-3` , `-4.2E0`) werden automatisch als `xsd:double` interpretiert.
- Die Schlüsselwörter **true** und **false** (ohne Anführungszeichen) werden automatisch als `xsd:boolean` interpretiert.
- **Zeichenketten in Anführungszeichen** ohne expliziten Datentyp und ohne Sprach-Tag (z.B. `"Hallo"`) werden, wie oben erwähnt, als `xsd:string` interpretiert.

Obwohl Turtle viele dieser Datentypen automatisch erkennt, ist es auch immer korrekt, den Datentyp explizit anzugeben (z.B. `"261472"^^xsd:integer` oder `261472` – beides resultiert im selben RDF-Literal). Die Kurzschreibweise ist oft prägnanter. Der Präfix `xsd:` steht konventionell für den XML Schema Datatypes Namespace `<http://www.w3.org/2001/XMLSchema#>` .

Beispiele für explizite vs. Kurzschreibweise:

Das Tripel:

```
<http://dbpedia.org/resource/Aache> <https://dbpedia.org/ontology/populationTotal> "261472"^^<http://www.w3.org/2001/XMLSchema#integer> .
```

ist äquivalent zu (und üblicherweise geschrieben als):

```
<http://dbpedia.org/resource/Aachen> <https://dbpedia.org/ontology/populationTotal> 261472 .
```

Dezimalzahlen:

```
<http://dbpedia.org/resource/Aachen> <https://schema.org/ratingValue> "4.7"^^<http://www.w3.org/2001/XMLSchema#decimal> .
```

ist äquivalent zu:

```
<http://dbpedia.org/resource/Aachen> <https://schema.org/ratingValue> 4.7 .
```

(Beachten Sie, dass `"4.7"` in Anführungszeichen ein `xsd:string` wäre, während `4.7` ohne Anführungszeichen ein `xsd:decimal` ist). Eine Zahl mit einem Dezimalpunkt wird automatisch als `xsd:decimal` erkannt.

Übersicht der automatisch erkannten Datentypen in Turtle:

(Hinweis: In den Beispielen der Tabelle verwenden wir bereits Präfixe wie `ex:` oder den leeren Präfix `:` , die später genauer erklärt werden. Hier geht es primär um die Schreibweise der Literal-Objekte.)

Literal-Schreibweise in Turtle	Interpretierter XSD Datentyp	Beispiel (mit fiktivem Präfix <code>ex:</code>)
<code>123</code>	<code>xsd:integer</code>	<code>ex:aachen ex:einwohner 249878 .</code>
<code>-5</code>	<code>xsd:integer</code>	<code>ex:sensor ex:temperaturOffset -5 .</code>
<code>4.5</code>	<code>xsd:decimal</code>	<code>ex:aachen ex:bewertung 4.5 .</code>
<code>0.0</code>	<code>xsd:decimal</code>	<code>ex:produkt ex:umweltbilanz 0.0 .</code>
<code>1.6E2</code> (oder <code>1.6e2</code>)	<code>xsd:double</code>	<code>ex:aachen ex:flaeche 1.6E2 .</code> (entspricht 160.0)
<code>"Ein Text"</code>	<code>xsd:string</code>	<code>ex:aachen ex:name "Aachen" .</code>
<code>true</code>	<code>xsd:boolean</code>	<code>ex:aachen ex:istStadt true .</code>
<code>false</code>	<code>xsd:boolean</code>	<code>ex:aachen ex:istHauptstadt false .</code>

Wichtige Hinweise:

- Alle **anderen** Datentypen (z.B. `xsd:date` , `xsd:dateTime` , spezifische anwenderdefinierte Typen) müssen immer explizit mit `^^` und der Datentyp-IRI gekennzeichnet werden.
- Die oben genannten automatisch erkannten Typen können ebenfalls explizit gekennzeichnet werden (z.B. `123^^xsd:integer`), dies macht den Turtle-Code aber meist weniger leserlich und ist nicht notwendig.

Whitespace (Leerzeichen)

Leerzeichen, Tabulatoren und Zeilenumbrüche werden außerhalb von IRIs (innerhalb `<...>`), Literalen (innerhalb `"..."`) und bestimmten Schlüsselwörtern (wie Präfixnamen) weitgehend ignoriert. Das bedeutet, Sie können Ihre Turtle-Dateien formatieren (z.B. durch Einrückungen und Leerzeilen), um sie lesbarer zu machen, ohne die Bedeutung der Daten zu ändern.

Das folgende Tripel:

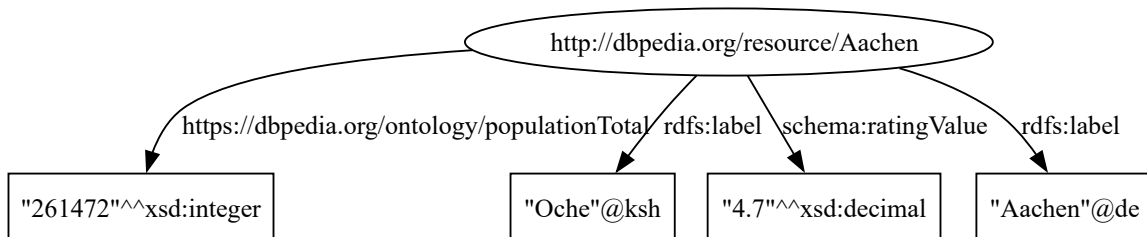
```
<http://dbpedia.org/resource/Aachen>
  <http://www.w3.org/2000/01/rdf-schema#label>
    "Aachen"@de .
```

ist äquivalent zu:

```
<http://dbpedia.org/resource/Aachen> <http://www.w3.org/2000/01/rdf-schema#label> "Aachen"@de .
```

In [2]:

```
%%rdf turtle
<http://dbpedia.org/resource/Aachen> <http://www.w3.org/2000/01/rdf-schema#label> "Aachen"@de .
<http://dbpedia.org/resource/Aachen> <http://www.w3.org/2000/01/rdf-schema#label> "Oche"@ksh .
<http://dbpedia.org/resource/Aachen> <https://dbpedia.org/ontology/populationTotal> 261472 .
<http://dbpedia.org/resource/Aachen> <https://schema.org/ratingValue> 4.7 .
```



Abkürzungen in Turtle

Namensräume in RDF

In RDF werden Ressourcen durch URIs (Uniform Resource Identifiers) – oder allgemeiner IRIs (Internationalized Resource Identifiers) – eindeutig identifiziert. Diese IRIs können sehr lang werden, insbesondere wenn sie aus verschiedenen Vokabularen stammen. Betrachten wir das folgende Beispiel mit Daten über Aachen, das wir bereits im Abschnitt zu Literalen gesehen haben, hier aber mit vollständig ausgeschriebenem Prädikat-IRIs:

```
<http://dbpedia.org/resource/Aachen> <http://www.w3.org/2000/01/rdf-schema#label> "Aachen"@de .
<http://dbpedia.org/resource/Aachen> <http://www.w3.org/2000/01/rdf-schema#label> "Oche"@ksh .
<http://dbpedia.org/resource/Aachen> <http://dbpedia.org/ontology/populationTotal> 261472 .
<http://dbpedia.org/resource/Aachen> <http://schema.org/ratingValue> 4.7 .
```

(Hinweis: In Turtle werden Zahlen wie 261472 als xsd:integer und 4.7 als xsd:decimal interpretiert, wie im vorherigen Abschnitt besprochen.)

Hier sehen wir verschiedene **Namensräume** (Namespaces), die als Basis für die verwendeten IRIs dienen:

- `http://dbpedia.org/resource/` (für DBpedia-Ressourcen wie `:Aachen`)
- `http://www.w3.org/2000/01/rdf-schema#` (für RDF Schema-Eigenschaften wie `:label`)
- `http://dbpedia.org/ontology/` (für DBpedia-Ontologie-Eigenschaften wie `:populationTotal`)
- `http://schema.org/` (für Schema.org-Eigenschaften wie `:ratingValue`)

Ein Namensraum ist im Grunde eine Sammlung von IRIs, die mit derselben Zeichenfolge beginnen (dem Basis-URI des Namensraums). Alle RDF Schema-Eigenschaften beginnen beispielsweise mit `http://www.w3.org/2000/01/rdf-schema#`.

Motivation der Prefix-Direktive

Die ständige Wiederholung langer Basis-URIs macht RDF-Daten schwer lesbar und fehleranfällig beim manuellen Schreiben. Die `@prefix`-Direktive in Turtle löst dieses Problem, indem sie kurze, benutzerdefinierte Abkürzungen (Präfix-Label) für die langen Namensraum-IRIs definiert:

```
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <http://schema.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> . # Oft benötigt für explizite Datentypen
```

```
dbr:Aachen rdfs:label "Aachen"@de .
dbr:Aachen rdfs:label "Oche"@ksh .
dbr:Aachen dbo:populationTotal 261472 .
dbr:Aachen schema:ratingValue 4.7 .
```

Das `@prefix`-Statement bindet also einen kurzen *Präfix-Namen* (z.B. `rdfs`) an eine *Namensraum-IRI* (z.B. `<http://www.w3.org/2000/01/rdf-schema#>`).

Die Vorteile sind offensichtlich:

- **Lesbarkeit:** `rdfs:label` ist deutlich verständlicher und kürzer als die vollständige IRI.
- **Kompaktheit:** Weniger redundante Zeichen führen zu kleineren Dateien.

- **Wartbarkeit:** Namensraum-IRIs werden zentral am Anfang des Dokuments definiert und können dort bei Bedarf leicht geändert werden.
- **Fehlerreduzierung:** Kürzere Bezeichner verringern die Wahrscheinlichkeit von Tippfehlern.

Präfixe funktionieren wie Textersetzungen vor der eigentlichen Interpretation: Der Turtle-Parser ersetzt z.B. `rdfs:label` intern durch die vollständige IRI `<http://www.w3.org/2000/01/rdf-schema#label>`, bevor die Tripel verarbeitet werden. Nachdem ein Präfix wie `@prefix dbr: <http://dbpedia.org/resource/> .` definiert wurde, sind Ausdrücke wie `dbr:Aachen` und `<http://dbpedia.org/resource/Aachen>` vollkommen äquivalent und austauschbar. Beide repräsentieren dieselbe eindeutige IRI. Die Präfix-Notation dient lediglich der besseren Lesbarkeit und Kompaktheit.

Bemerkung: Die hier gezeigte @prefix -Schreibweise ist die Standard-Schreibweise in Turtle. Seit Turtle 1.1 ist auch die aus SPARQL bekannte Schreibweise PREFIX rdfs: <...> (ohne @ und ohne Punkt am Ende der Direktive, wenn sie auf einer eigenen Zeile steht) erlaubt, wird aber in reinen Turtle-Dateien seltener verwendet. Im SPARQL-Teil dieses Kurses werden wir PREFIX verwenden.

Vereinfachung mit Semikolon-Syntax

Auch mit Präfixen ist im obigen Beispiel noch eine Redundanz sichtbar: Das Subjekt `dbr:Aachen` wird in jedem Tripel wiederholt. Turtle bietet hierfür eine elegante Lösung mit dem **Semikolon (;)**:

```
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <http://schema.org/> .
# xsd wird hier nicht explizit für Literale benötigt, da Turtle die Typen erkennt
```

```
dbr:Aachen
  rdfs:label "Aachen"@de ;
  rdfs:label "Oche"@ksh ;
  dbo:populationTotal 261472 ;
  schema:ratingValue 4.7 .
```

Das **Semikolon (;)** am Ende eines Prädikat-Objekt-Paares signalisiert, dass das nächste Prädikat-Objekt-Paar sich auf dasselbe Subjekt bezieht. Dies macht die Daten noch kompakter und zeigt deutlich, dass alle hier gruppierten Aussagen über dieselbe Ressource (`dbr:Aachen`) gemacht werden. Der abschließende Punkt (.) beendet die gesamte Aussageblock für dieses Subjekt.

Funktionsweise der Semikolon-Syntax:

- Das erste vollständige Tripel lautet: `dbr:Aachen rdfs:label "Aachen"@de .`
- Nach dem Semikolon (;) wird das Subjekt (`dbr:Aachen`) für das nächste Prädikat-Objekt-Paar implizit fortgeführt.
- Das zweite implizite Tripel ist also: `dbr:Aachen rdfs:label "Oche"@ksh .`
- Und so weiter für alle folgenden Zeilen, bis der Block mit einem Punkt (.) abgeschlossen wird.

Die Semikolon-Syntax ist besonders nützlich, wenn man mehrere Eigenschaften derselben Ressource beschreibt, was in RDF sehr häufig vorkommt.

Weitere Vereinfachung mit Komma-Syntax

Bei genauerer Betrachtung des letzten Beispiels fällt auf, dass wir zweimal dasselbe Prädikat `rdfs:label` für `dbr:Aachen` verwenden, nur mit unterschiedlichen Objekten. Auch hierfür hat Turtle eine Abkürzung – das **Komma (,)**:

```
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <http://schema.org/> .

dbr:Aachen
  rdfs:label "Aachen"@de, "Oche"@ksh ; # Komma für mehrere Objekte zum selben Prädikat
  dbo:populationTotal 261472 ;
  schema:ratingValue 4.7 .
```

Das **Komma (,)** am Ende eines Objekts signalisiert, dass das nächste Objekt sich auf dasselbe Subjekt **und** dasselbe Prädikat bezieht.

Funktionsweise der Komma-Syntax (in Kombination mit dem Semikolon):

- Das erste vollständige Tripel ist: `dbr:Aachen rdfs:label "Aachen"@de .`
- Nach dem Komma (,) werden Subjekt (`dbr:Aachen`) UND Prädikat (`rdfs:label`) für das nächste Objekt implizit fortgeführt.
- Das zweite implizite Tripel ist also: `dbr:Aachen rdfs:label "Oche"@ksh .`
- Das Semikolon (;) wechselt dann zu einem neuen Prädikat (`dbo:populationTotal`), bezieht sich aber weiterhin auf das ursprüngliche Subjekt (`dbr:Aachen`).

Zusammenfassung der Turtle-Abkürzungen für Tripel-Strukturen:

- **Punkt (.):** Beendet eine oder mehrere Aussagen über ein Subjekt (einen "Subjekt-Block").
- **Semikolon (;):** Trennt Prädikat-Objekt-Paare für dasselbe Subjekt. Das Subjekt bleibt gleich, Prädikat und Objekt sind neu.

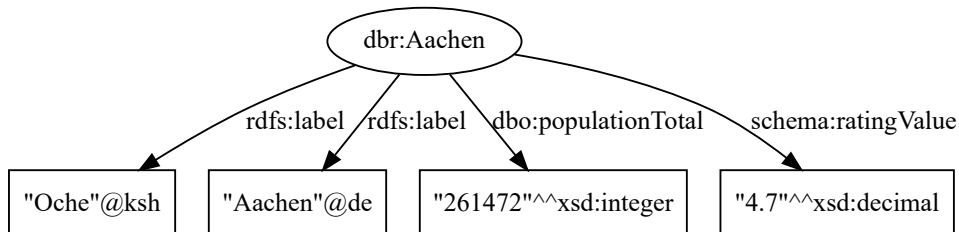
- **Komma (,)**: Trennt Objekte für dasselbe Subjekt und dasselbe Prädikat. Subjekt und Prädikat bleiben gleich, nur das Objekt ist neu.

Das folgende `%%rdf turtle`-Beispiel zeigt, dass die letzte, am meisten abgekürzte Version denselben RDF-Graphen ergibt wie die anfängliche, ausführliche Version mit vier einzelnen Tripeln:

In [3]:

```
%%rdf turtle
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <https://dbpedia.org/ontology/> .
@prefix schema: <https://schema.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

dbr:Aachen rdfs:label "Aachen"@de, "Oche"@ksh ;
           dbo:populationTotal 261472 ;
           schema:ratingValue 4.7 .
```



Weitere Konstrukte in Turtle

Kommentare (Comments)

Erklärung: Kommentare in Turtle beginnen mit dem Rautezeichen (#) und erstrecken sich bis zum Ende der Zeile. Sie dienen der menschlichen Lesbarkeit und werden vom RDF-Parser vollständig ignoriert, haben also keinen Einfluss auf den resultierenden Graphen.

Turtle-Beispiel:

```
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <https://schema.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> . # Für explizite Datentypen, falls benötigt

# Daten über Aachen
dbr:Aachen rdfs:label "Aachen"@de, "Oche"@ksh ; # Vielleicht sollten wir für Öcher Platt einen eigenen Tag bei
der IANA registrieren? ;)
           dbo:populationTotal 261472 ;          # Aktuelle Einwohnerzahl
           schema:ratingValue 4.7 .              # Eine fiktive Bewertung
```

Die @base -Direktive

Motivation: Angenommen, wir erweitern unsere Daten um weitere deutsche Städte, die alle aus dem DBpedia-Namensraum `http://dbpedia.org/resource/` stammen:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <https://schema.org/> .

<http://dbpedia.org/resource/Aachen> rdfs:label "Aachen"@de, "Oche"@ksh ;
                                     dbo:populationTotal 261472 ;
                                     schema:ratingValue 4.7 .
<http://dbpedia.org/resource/Cologne> rdfs:label "Köln"@de ;
                                     dbo:populationTotal 1073096 .
<http://dbpedia.org/resource/Düsseldorf> rdfs:label "Düsseldorf"@de ;
                                     dbo:populationTotal 619477 .
```

Hier wiederholt sich die Basis-IRI `http://dbpedia.org/resource/` bei jeder Ressourcen-IRI. Die **@base -Direktive** löst dieses Problem elegant, indem sie eine Basis-IRI für alle relativen IRIs im Dokument definiert:

```
@base <http://dbpedia.org/resource/> . # Basis-IRI für dieses Dokument

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <https://schema.org/> .

<Aachen> rdfs:label "Aachen"@de, "Oche"@ksh ; # <Aachen> wird zu <http://dbpedia.org/resource/Aachen>
          dbo:populationTotal 261472 ;
          schema:ratingValue 4.7 .
<Cologne> rdfs:label "Köln"@de ;
          dbo:populationTotal 1073096 .
```

```
<Düsseldorf> rdfs:label "Düsseldorf"@de ;
              dbo:populationTotal 619477 .
```

Funktionsweise der @base -Direktive:

- @base <IRI> definiert eine Basis-IRI für das aktuelle Turtle-Dokument. Es darf nur eine @base -Direktive pro Dokument geben (bzw. die letzte gilt).
- Relative IRIs, die in spitzen Klammern eingeschlossen sind (z.B. <Aachen> oder <unterordner/Ressource>), werden durch Voranstellen der @base -IRI zur vollständigen (absoluten) IRI aufgelöst.
- Wichtig: Die @base -Direktive beeinflusst **nur** relative IRIs in spitzen Klammern (< >), **nicht** die Auflösung von präfixierten Namen (wie dbr:Aachen). Diese werden immer über ihre jeweilige @prefix -Definition aufgelöst.

Vorteile der Base-Direktive:

- **Konsistenz:** Stellt eine einheitliche Basis-IRI für alle "lokalen" oder relativen Ressourcen des Dokuments sicher.
- **Flexibilität:** Erlaubt eine einfache Änderung der Basis-IRI an einer zentralen Stelle, falls sich z.B. die Domain ändert.
- **Klarheit:** Kann die Unterscheidung zwischen lokalen (relativen) und externen (absoluten oder anders präfixierten) Ressourcen verdeutlichen.
- **Portabilität:** RDF-Daten können leichter zwischen verschiedenen Umgebungen oder Servern verschoben werden, wenn relative IRIs verwendet werden, deren Basis dann einfach angepasst wird.

Die @base -Direktive ist besonders nützlich beim Erstellen eigener RDF-Datensets, bei denen viele Ressourcen denselben IRI-Stamm teilen.

Typisierung mit rdf:type und die Abkürzung a

Im vorherigen Notebook haben wir das Konzept von rdf:type kennengelernt, um anzugeben, welcher Klasse oder welchem Typ eine Ressource angehört. Schauen wir uns nun an, wie wir solche Aussagen in Turtle schreiben.

Zur Erinnerung: rdf:type ist ein fundamentales Prädikat im RDF-Vokabular. Es wird verwendet, um eine Ressource einer oder mehreren Klassen zuzuordnen. Erweitern wir unser Aachen-Beispiel um Typ-Informationen, wobei wir die zuvor definierte @base -Direktive nutzen:

```
@base <http://dbpedia.org/resource/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> . # Standardpräfix für RDF-Vokabular
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <https://schema.org/> .

<Aachen> rdf:type dbo:City ; # Aachen ist vom Typ dbo:City
          rdfs:label "Aachen"@de, "Oche"@ksh ;
          dbo:populationTotal 261472 ;
          schema:ratingValue 4.7 .
<Cologne> rdf:type dbo:City ;
           rdfs:label "Köln"@de ;
           dbo:populationTotal 1073096 .
<Düsseldorf> rdf:type dbo:City ;
             rdfs:label "Düsseldorf"@de ;
             dbo:populationTotal 619477 .
```

Da rdf:type so extrem häufig verwendet wird, bietet Turtle eine spezielle Abkürzung dafür: das **Schlüsselwort a** .

```
@base <http://dbpedia.org/resource/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <https://schema.org/> .
# Das Präfix rdf: ist für die Verwendung von 'a' nicht explizit nötig,
# da 'a' direkt für die IRI <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> steht.

<Aachen> a dbo:City ; # 'a' ist die Abkürzung für rdf:type
          rdfs:label "Aachen"@de, "Oche"@ksh ;
          dbo:populationTotal 261472 ;
          schema:ratingValue 4.7 .
<Cologne> a dbo:City ;
           rdfs:label "Köln"@de ;
           dbo:populationTotal 1073096 .
<Düsseldorf> a dbo:City ;
             rdfs:label "Düsseldorf"@de ;
             dbo:populationTotal 619477 .
```

Das Schlüsselwort a :

- a ist eine direkte Abkürzung für die IRI rdf:type (also <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>).
- <Aachen> a dbo:City bedeutet exakt dasselbe wie <Aachen> rdf:type dbo:City .
- Diese Abkürzung macht RDF-Daten in Turtle noch lesbarer und kompakter.

Warum ist rdf:type wichtig? Das Prädikat rdf:type ist fundamental, da es die semantische Struktur der Daten maßgeblich mitdefiniert. Es besagt, zu welcher Kategorie oder Klasse eine Ressource gehört. Dies ist die Grundlage für Schema-Definitionen und logisches Schließen (Inferenz). (Die Details zu Typen, Klassen und Schemasprachen wie RDFS und OWL werden wir in späteren Notebooks behandeln.)

Blank Nodes in Turtle

Wie wir bereits im vorherigen Notebook konzeptuell gelernt haben, sind Blank Nodes unbenannte Knoten in RDF-Graphen. Sie werden verwendet, um Ressourcen zu repräsentieren, für die keine globale IRI vergeben wird oder werden soll.

1. Benannte Blank Nodes (`_:identifizier`)

In Turtle kann ein Blank Node mit einem Label versehen werden, das mit `_:` beginnt (z.B. `_:meineAdresse` , `_:b1`). Erweitern wir unser Aachen-Beispiel um eine strukturierte Adressinformation, die wir über einen Blank Node modellieren:

```
@base <http://dbpedia.org/resource/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <https://schema.org/> . # Für Adress-Eigenschaften

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de, "Oche"@ksh ;
  dbo:populationTotal 261472 ;
  schema:ratingValue 4.7 ;
  schema:address _:address1 . # Verweis auf den Blank Node

_:address1 a schema:PostalAddress ; # Definition des Blank Nodes
  schema:streetAddress "Markt 39" ;
  schema:postalCode "52062" ;
  schema:addressLocality "Aachen" ;
  schema:addressCountry "Deutschland" .
```

Hier verwenden wir den Blank Node mit dem Label `_:address1` für die Adressinformation. Dieses Label (`address1`) hat **nur innerhalb des aktuellen Dokuments bzw. der aktuellen Turtle-Datei eine eindeutige Bedeutung**. Es dient primär der internen Referenzierung, wenn derselbe Blank Node an mehreren Stellen (z.B. als Subjekt weiterer Tripel oder als Objekt anderer Tripel im selben Dokument) verwendet werden soll. Es ist kein global eindeutiger Identifikator wie eine IRI.

2. Anonyme Blank Nodes mit eckigen Klammern (`[ ]`)

Turtle bietet eine noch kompaktere und oft elegantere Syntax für Blank Nodes, die **eckigen Klammern** `[ ]` . Diese Syntax ist besonders nützlich, wenn ein Blank Node als Objekt eines Tripels eingeführt wird und seine Eigenschaften direkt an dieser Stelle definiert werden, ohne dass der Blank Node an anderer Stelle über sein Label referenziert werden muss.

Das obige Beispiel lässt sich damit wie folgt schreiben:

```
@base <http://dbpedia.org/resource/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <https://schema.org/> .

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de, "Oche"@ksh ;
  dbo:populationTotal 261472 ;
  schema:ratingValue 4.7 ;
  schema:address [ # Beginn des anonymen Blank Nodes
    a schema:PostalAddress ;
    schema:streetAddress "Markt 39" ;
    schema:postalCode "52062" ;
    schema:addressLocality "Aachen" ;
    schema:addressCountry "Deutschland"
  ] . # Ende des anonymen Blank Nodes und des Aachen-Blocks
```

Funktionsweise der eckigen Klammern:

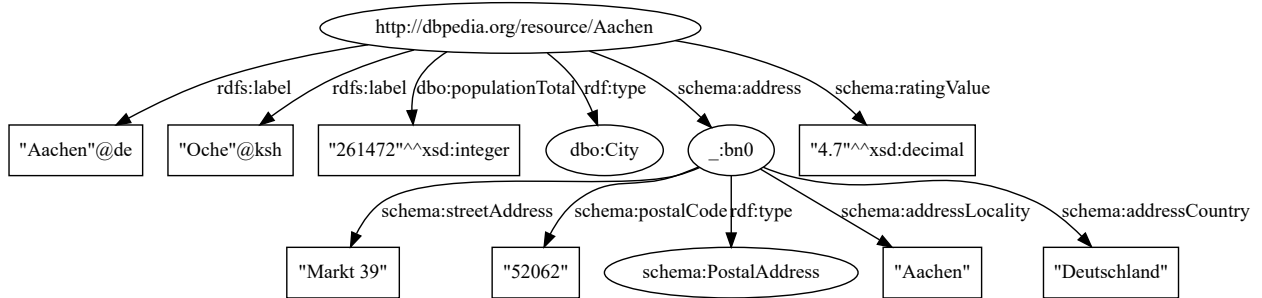
- Ein Paar eckiger Klammern `[ ]` erstellt automatisch einen neuen, anonymen (unbenannten) Blank Node.
- Alle Prädikat-Objekt-Paare, die innerhalb dieser Klammern (und durch Semikola getrennt) aufgeführt werden, werden diesem anonymen Blank Node als Eigenschaften zugeordnet. Das letzte Prädikat-Objekt-Paar darf **nicht** mit einem **Punkt (.)** beendet werden.
- Die gesamte `[ ... ]` -Struktur fungiert als Objekt des übergeordneten Tripels (hier als Objekt von `schema:address`).
- Diese Schreibweise entspricht exakt dem vorherigen Beispiel mit `_:address1` , ist aber oft übersichtlicher, wenn der Blank Node nicht an anderer Stelle über ein Label referenziert werden muss. Man verwendet `_:identifizier` also dann, wenn derselbe Blank Node an mehreren Stellen als Subjekt oder Objekt wiederverwendet werden soll.

Ein leeres Paar eckiger Klammern `[ ]` als Objekt erzeugt einen Blank Node ohne direkt zugewiesene Eigenschaften (obwohl ihm natürlich in anderen Tripeln, wo er dann Subjekt ist und über ein `_:label` referenziert wird, Eigenschaften zugewiesen werden könnten – aber dafür ist die `_:label` Syntax dann besser geeignet).

In [4]:

```
%%rdf turtle
@base <http://dbpedia.org/resource/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <https://dbpedia.org/ontology/> .
@prefix schema: <https://schema.org/> .

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de, "Oche"@ksh ;
  dbo:populationTotal 261472 ;
  schema:ratingValue 4.7 ;
  schema:address [ a schema:PostalAddress ;
    schema:streetAddress "Markt 39" ;
    schema:postalCode "52062" ;
    schema:addressLocality "Aachen" ;
    schema:addressCountry "Deutschland" ] .
```



Verschachtelte Blank Nodes

Oftmals sind die Informationen, die wir mit einem Blank Node strukturieren möchten, selbst wieder komplex und haben eigene Eigenschaften, die wiederum durch (weitere) Blank Nodes oder Literale beschrieben werden. Turtle erlaubt es, diese Strukturen sehr elegant und kompakt durch die **Verschachtelung von Blank Nodes** auszudrücken, insbesondere unter Verwendung der [...] -Syntax.

Blank Nodes können dabei prinzipiell beliebig tief verschachtelt werden. Erweitern wir unser vorheriges Beispiel für <Aachen> um einen detaillierten Kontaktpunkt, der eine Person mit Arbeitsort und dessen Adresse beinhaltet:

```
@base <http://dbpedia.org/resource/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <https://schema.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de, "Oche"@ksh ;
  dbo:populationTotal 261472 ;
  schema:ratingValue 4.7 ;
  schema:address [ a schema:PostalAddress ; # 1. Verschachtelungsebene (Adresse von Aachen)
    schema:streetAddress "Markt 39" ;
    schema:postalCode "52062" ;
    schema:addressLocality "Aachen" ;
    schema:addressCountry "Deutschland" ] ;
  schema:contactPoint [ a schema:ContactPoint ; # 1. Verschachtelungsebene (Kontaktpunkt von Aachen)
    schema:contactType "Tourismus" ;
    schema:telephone "+49 241 180-2960" ;
    foaf:person [ a foaf:Person ; # 2. Verschachtelungsebene (Person im
      Kontaktpunkt)
        foaf:name "Maria Hoffmann" ;
        foaf:title "Leiterin Tourismusmarketing" ;
        schema:workLocation [ a schema:Place ; # 3. Verschachtelungsebene
          (Arbeitsplatz der Person)
            schema:name "Aachen Tourist Service" ;
            schema:address [ a schema:PostalAddress ; #
              schema:streetAddress
                "Elisenbrunnen" ;
                schema:postalCode "52062"
              ]
            ]
          ]
        ] .
```

Verschachtelte Struktur und die verbindenden Eigenschaften:

Diese tief verschachtelte Struktur lässt sich wie folgt aufschlüsseln, wobei jede Ebene durch eine Eigenschaft mit der vorherigen verbunden ist:

- **Ebene 0:** <Aachen> (die benannte Ressource, unser Ausgangspunkt)
- **Ebene 1:**

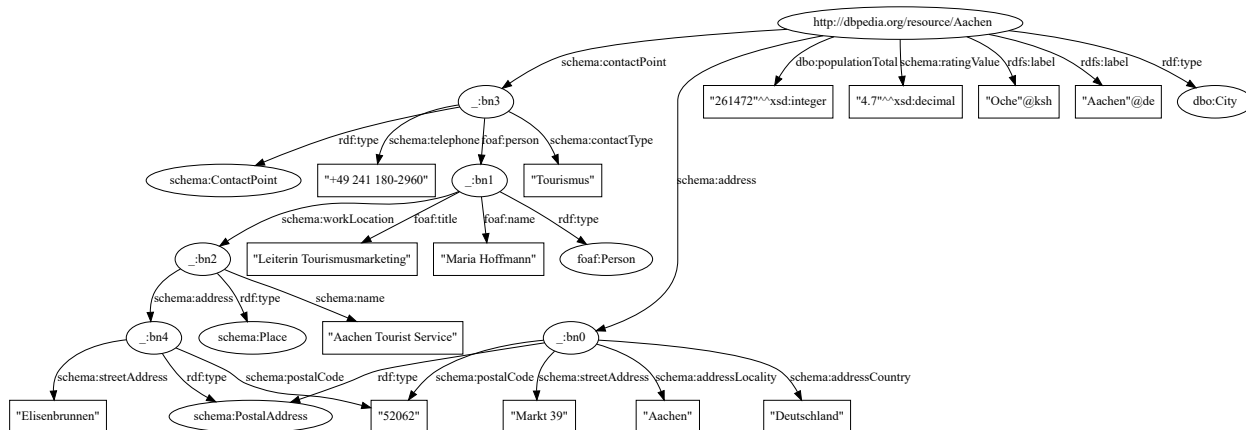
- Eine Adresse (Blank Node), verbunden via `schema:address` mit `<Aachen>` .
- Ein Kontaktpunkt (Blank Node), verbunden via `schema:contactPoint` mit `<Aachen>` .
- **Ebene 2:** Eine Person (Blank Node), verbunden via `foaf:person` mit dem Kontaktpunkt-Blank-Node.
- **Ebene 3:** Ein Arbeitsplatz (Blank Node), verbunden via `schema:workLocation` mit dem Personen-Blank-Node.
- **Ebene 4:** Eine Adresse (Blank Node), verbunden via `schema:address` mit dem Arbeitsplatz-Blank-Node.

Hinweis zur Äquivalenz: Diese tief verschachtelte Struktur mit der `[ ... ]` -Syntax ist äquivalent zu einer Darstellung, bei der für jede anonyme Ressource ein explizites Blank-Node-Label (z.B. `_:kontakt`, `_:person`, `_:arbeitsplatz`, `_:adresseArbeitsplatz`) vergeben und referenziert würde. Die hier gezeigte verschachtelte `[ ... ]` -Syntax ist jedoch deutlich kompakter und oft besser lesbar, insbesondere wenn diese anonymen Ressourcen nicht an anderer Stelle im Dokument erneut referenziert werden müssen. Sie ist ideal, um solche hierarchischen oder komponentenhaften Datenstrukturen direkt und übersichtlich darzustellen. Tatsächlich ist die graphische Darstellung nicht unbedingt übersichtlicher, wie die folgende Zelle zeigt.

In [5]:

```
%%rdf turtle
@base <http://dbpedia.org/resource/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix schema: <https://schema.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de, "Oche"@ksh ;
  dbo:populationTotal 261472 ;
  schema:ratingValue 4.7 ;
  schema:address [ a schema:PostalAddress ;      # 1. Verschachtelungsebene (Adresse von Aachen)
    schema:streetAddress "Markt 39" ;
    schema:postalCode "52062" ;
    schema:addressLocality "Aachen" ;
    schema:addressCountry "Deutschland" ] ;
  schema:contactPoint [ a schema:ContactPoint ; # 1. Verschachtelungsebene (Kontaktpunkt von Aachen)
    schema:contactType "Tourismus" ;
    schema:telephone "+49 241 180-2960" ;
    foaf:person [ a foaf:Person ; # 2. Verschachtelungsebene (Person im Kontaktpunkt)
      foaf:name "Maria Hoffmann" ;
      foaf:title "Leiterin Tourismusmarketing" ;
      schema:workLocation [ a schema:Place ; # 3. Verschachtelungsebene (Arbeitsplatz)
        schema:name "Aachen Tourist Service" ;
        schema:address [ a schema:PostalAddress ; # 4. Verschachtelungsebene (Adresse von Aachen Tourist Service)
          schema:streetAddress "Elisenbrunnen" ;
          schema:postalCode "52062"
        ]
      ]
    ]
  ] .
```



Vorteile der Verschachtelung von Blank Nodes:

- **Strukturierung:** Komplexe, hierarchische Datenstrukturen können klar und intuitiv abgebildet werden. Die Einrückung im Code spiegelt oft die Baumstruktur der Daten wider.
- **Kompaktheit:** Es reduziert die Notwendigkeit, für jede Eigenschaft eines Blank Nodes, der nur einmal als Objekt verwendet wird, ein explizites Label (wie `_:xyz`) zu vergeben und dieses Label dann wiederholt als Subjekt zu nutzen. Alle Eigenschaften können direkt an der "Entstehungsstelle" des Blank Nodes definiert werden.
- **Lesbarkeit:** Die Verschachtelung kann die logische Struktur der Daten oft besser widerspiegeln als eine "flache" Liste von Tripeln mit vielen explizit benannten Blank Nodes, besonders wenn es um klar abgegrenzte Unterstrukturen geht.
- **Kohäsion:** Eng zusammengehörige Informationen (z.B. alle Komponenten einer Adresse oder die Details eines Ansprechpartners) bleiben als eine logische Einheit im Code gruppiert.

Wann sollte man die Verschachtelung mit `[ ... ]` (anonyme Blank Nodes) verwenden?

Die [...] -Syntax für Blank Nodes ist besonders dann sinnvoll:

- Wenn ein Blank Node als **strukturiertes Objekt** für ein Prädikat dient und dieser Blank Node **nicht an anderer Stelle im selben RDF-Dokument erneut als Subjekt oder Objekt referenziert** werden muss. Er wird also quasi "inline" definiert und verbraucht.
- Zur Darstellung von **inhärent hierarchischen Informationen oder Komponenten**, die keine eigene, global eindeutige URI benötigen (z.B. die Adresse einer Person, die Details eines einzelnen Bestellpostens innerhalb einer Bestellung, Öffnungszeiten-Spezifikationen).
- Wenn die Verschachtelung die **Lesbarkeit und das unmittelbare Verständnis** der Datenstruktur im Vergleich zur Verwendung mehrerer explizit benannter und separat definierter Blank Nodes (`_:label1` , `_:label2` , ...) tatsächlich verbessert.

Ein wichtiger Hinweis zur Anwendung:

Die Verschachtelung von Blank Nodes sollte mit Bedacht eingesetzt werden. Obwohl sie für viele Situationen eine elegante und kompakte Lösung darstellt, kann eine **zu tiefe oder exzessive Verschachtelung** (z.B. mehr als 3-4 Ebenen) die Lesbarkeit des Turtle-Codes auch wieder **stark beeinträchtigen** und die Orientierung im Graphen erschweren. Es gilt hier, eine gute Balance zwischen Kompaktheit und Übersichtlichkeit zu finden. Im Zweifel ist eine flachere Struktur mit gut benannten Blank-Node-Labels (`_:einSprechenderName`) manchmal die verständlichere Wahl.

Listen (Collections) in Turtle

Oft müssen wir in RDF **geordnete Listen** von Elementen darstellen. Angenommen, wir wollen die Nachbarstädte von Aachen in einer bestimmten Reihenfolge auflisten (z.B. Herzogenrath, dann Würselen, dann Stolberg).

Die zugrundeliegende Struktur von RDF-Listen

RDF verwendet hierfür ein spezielles Listenkonstrukt, das auf einer verketteten Struktur von Blank Nodes basiert. Jeder Knoten in dieser Kette repräsentiert ein Element der Liste und verweist auf den Rest der Liste, bis das Listenende mit `rdf:nil` erreicht wird.

Schauen wir uns an, wie die Nachbarstädte von Aachen in dieser ausführlichen Form dargestellt würden:

```
@base <http://dbpedia.org/resource/> . # Gilt für <Aachen>, <Herzogenrath> etc.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de ;
  dbo:neighboringMunicipality _:listenkopf . # Verweis auf den Beginn der Liste

# Definition der Liste über verkettete Blank Nodes
_:listenkopf rdf:first <Herzogenrath> ; # Erster Blank Node: erstes Element und Rest
  rdf:rest _:restliste1 .
_:restliste1 rdf:first <Würselen> ; # Zweiter Blank Node: zweites Element und Rest
  rdf:rest _:restliste2 .
_:restliste2 rdf:first <Stolberg> ; # Dritter Blank Node: drittes Element und Ende
  rdf:rest rdf:nil . # rdf:nil signalisiert das Ende der Liste
```

Diese ausführliche Darstellung zeigt, wie RDF Listen intern repräsentiert:

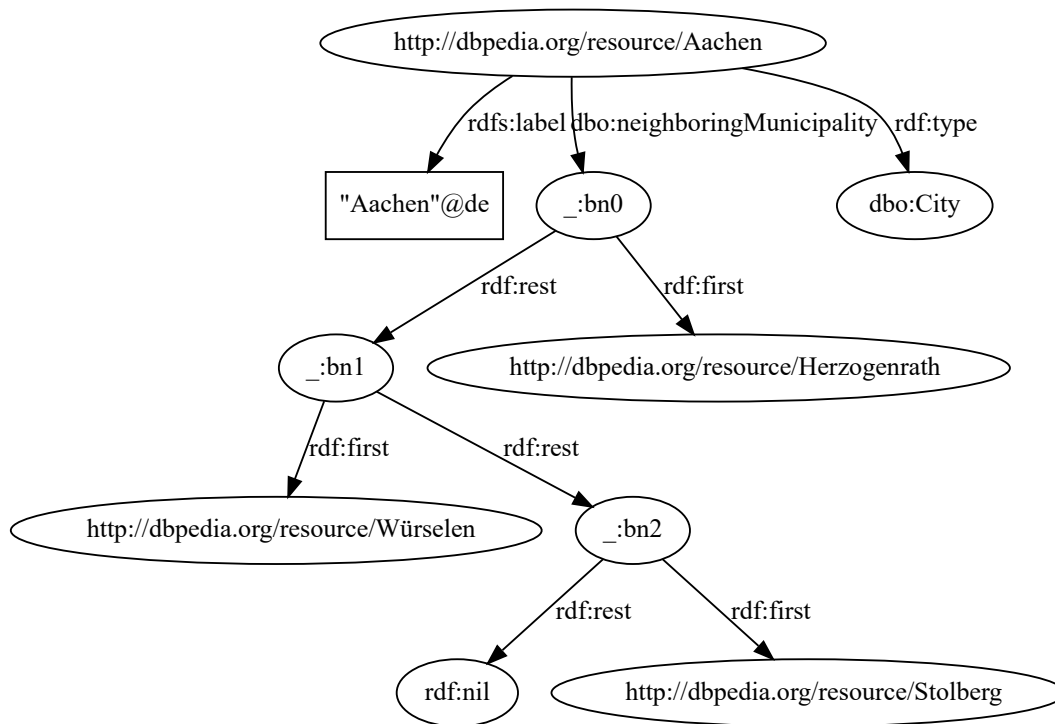
- Jeder Teil der Liste (außer dem Ende) ist typischerweise ein Blank Node (hier `_:listenkopf` , `_:restliste1` , `_:restliste2`).
- `rdf:first` zeigt auf das aktuelle Element dieses Listenknotens.
- `rdf:rest` zeigt auf den nächsten Listenknoten (den "Rest" der Liste).
- `rdf:nil` ist eine spezielle IRI, die das Ende einer Liste markiert (eine leere Liste).

Diese Schreibweise ist zwar exakt und zeigt die RDF-Struktur klar auf, aber für längere Listen wird sie schnell sehr umständlich und unübersichtlich. Glücklicherweise bietet Turtle hierfür eine sehr elegante und kompakte Abkürzung, die wir uns als Nächstes ansehen werden.

```
In [6]: %%rdf turtle
@base <http://dbpedia.org/resource/> . # Gilt für <Aachen>, <Herzogenrath> etc.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de ;
  dbo:neighboringMunicipality _:listenkopf . # Verweis auf den Beginn der Liste

# Definition der Liste über verkettete Blank Nodes
_:listenkopf rdf:first <Herzogenrath> ; # Erster Blank Node: erstes Element und Rest
  rdf:rest _:restliste1 .
_:restliste1 rdf:first <Würselen> ; # Zweiter Blank Node: zweites Element und Rest
  rdf:rest _:restliste2 .
_:restliste2 rdf:first <Stolberg> ; # Dritter Blank Node: drittes Element und Ende
  rdf:rest rdf:nil . # rdf:nil signalisiert das Ende der Liste
```



Collection-Syntax mit runden Klammern () – Die Turtle-Abkürzung für Listen

Nachdem wir die ausführliche, interne Struktur von RDF-Listen mit `rdf:first`, `rdf:rest` und `rdf:nil` kennengelernt haben, die oft Blank Nodes involviert, werden Sie die Turtle-Abkürzung dafür zu schätzen wissen. Turtle bietet eine deutlich kompaktere und intuitivere Syntax für Listen mit **runden Klammern** () :

```
@base <http://dbpedia.org/resource/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de ;
  dbo:neighboringMunicipality ( <Herzogenrath> <Würselen> <Stolberg> ) . # Liste mit IRIs als Elementen
```

Funktionsweise der Collection-Syntax () :

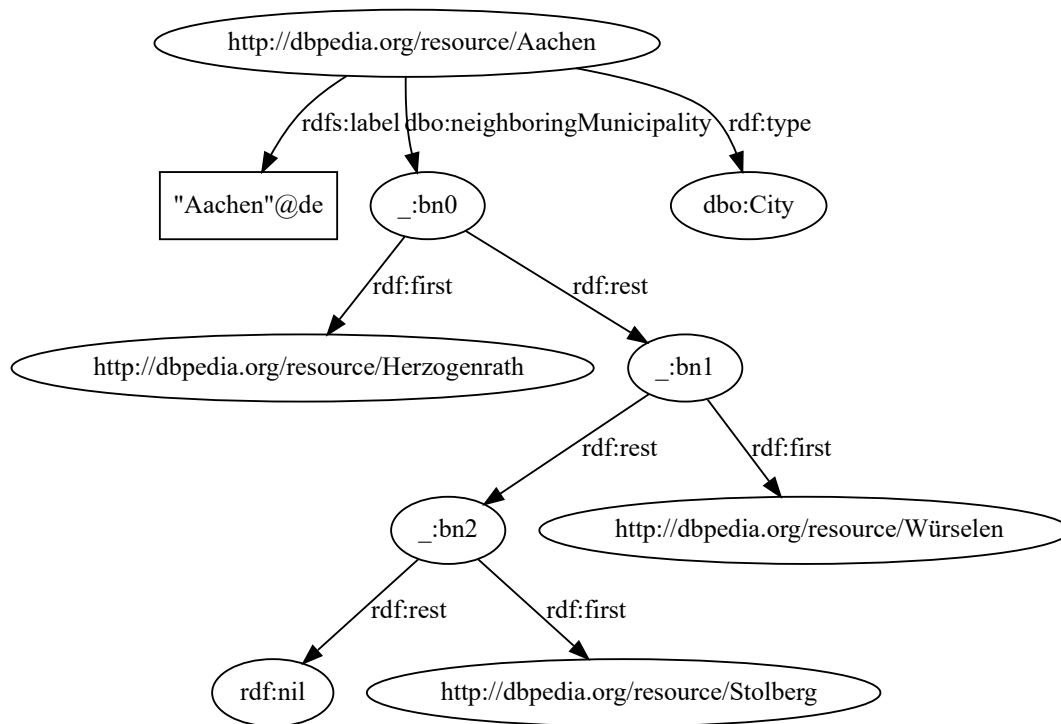
- Ein Ausdruck in runden Klammern (element1 element2 ... elementN) erstellt automatisch eine vollständige RDF-Liste (eine "RDF Collection").
- Die Elemente innerhalb der Klammern (getrennt durch Leerzeichen) können IRIs, Literale oder auch Blank Nodes (inklusive weiterer verschachtelter Listen oder [...] -Strukturen) sein.
- Die Reihenfolge der Elemente in den Klammern entspricht der Reihenfolge in der RDF-Liste.
- Diese () -Syntax wird vom Turtle-Parser intern exakt in die zuvor gezeigte ausführliche Darstellung mit einer Kette von (meist Blank) Nodes, `rdf:first` - und `rdf:rest` -Prädikaten und einem abschließenden `rdf:nil` übersetzt.
 - Eine leere Liste () wird direkt zur IRI `rdf:nil` aufgelöst.
 - Bei einer nicht-leeren Liste (e1 e2 ... en) wird eine Kette von Blank Nodes erzeugt, wobei der letzte Knoten dieser Kette mit `rdf:rest rdf:nil` abschließt.

Beobachtung im Notebook: Wenn Sie den obigen Turtle-Code in einer `%%rdf turtle` -Zelle ausführen, achten Sie genau auf die grafische Darstellung und die darunter ausgegebene Tripel-Liste. Sie werden sehen, wie die kompakte (<Herzogenrath> <Würselen> <Stolberg>) -Syntax in die detaillierte Struktur mit (intern benannten) Blank Nodes und den Prädikaten `rdf:first`, `rdf:rest` sowie dem terminierenden `rdf:nil` übersetzt wird. Dies verdeutlicht eindrucksvoll, dass () eine reine Syntax-Abkürzung für ein Standard-RDF-Konstrukt ist.

Diese Abkürzung macht die Arbeit mit geordneten Sammlungen in Turtle erheblich einfacher und lesbarer.

```
In [7]: %%rdf turtle
@base <http://dbpedia.org/resource/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de ;
  dbo:neighboringMunicipality ( <Herzogenrath> <Würselen> <Stolberg> ) . # Liste mit IRIs als Elementen
```



Verschiedene Collection-Beispiele und ihre Eigenschaften

Wie bereits angedeutet, sind RDF-Listen (Collections), die mit der `()`-Syntax in Turtle erstellt werden, sehr flexibel bezüglich der Typen ihrer Elemente. Die folgenden Beispiele demonstrieren diese Vielseitigkeit und zeigen auch eine leere Liste:

```
@base <http://dbpedia.org/resource/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix ex: <http://example.org/vokabular/> . # Beispiel-Präfix für eigene Begriffe
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> . # Nützlich für explizite Datentypen, falls benötigt

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de ;
  # Beispiel 1: Liste von Nachbarstädten (Elemente sind IRIs)
  dbo:neighboringMunicipality ( <Herzogenrath> <Würselen> <Stolberg> ) ;

  # Beispiel 2: Liste von Einwohnerzahlen der letzten Jahre (Elemente sind xsd:integer Literale)
  ex:populationHistory ( 260454 261007 261472 ) ; # Zahlen werden als xsd:integer interpretiert

  # Beispiel 3: Liste gemischter Datentypen
  ex:facts ( "Kaiserstadt" 1.234 true "UNESCO-Welterbe"@de <AltesRathaus> ) ;
  # Enthält: xsd:string, xsd:decimal, xsd:boolean, rdf:LangString, IRI

  # Beispiel 4: Leere Liste
  ex:emptyList ( ) . # Eine leere Liste wird zu rdf:nil
```

Beobachtung im Notebook: Führen Sie die Zelle unten aus. Achten Sie in der grafischen Darstellung und der Tripel-Ausgabe darauf, wie:

- die verschiedenen Listenelemente (IRIs, Zahlen, Strings, Wahrheitswerte) korrekt interpretiert werden.
- die leere Liste `()` als die spezielle IRI `rdf:nil` dargestellt wird.
- alle Listen intern die bekannte Struktur mit Blank Nodes, `rdf:first`, `rdf:rest` und `rdf:nil` verwenden.

Zusammenfassende Eigenschaften von RDF-Collections (Listen):

- **Geordnet:** Die Reihenfolge der Elemente, wie sie in den Klammern `()` notiert wird, bleibt bei der Interpretation als RDF-Graph erhalten.
- **Wiederholungen erlaubt:** Ein Element kann mehrfach in einer Liste vorkommen (z.B. `(<A> <B> <A>)`).
- **Typenvielfalt:** Listen können Elemente verschiedener RDF-Termtypen mischen (IRIs, Literale verschiedener Datentypen, Blank Nodes), wie im `ex:facts`-Beispiel gezeigt.
- **Verschachtelbar:** Listen können andere Listen als Elemente enthalten. Zum Beispiel: `ex:hauptstädteNachRegion ( "Europa" ( <Berlin> <Paris> <Rom> ) "Asien" ( <Peking> <Tokio> <Neu-Delhi> ) )`. Auch hier würde die innere Liste `( <Berlin> <Paris> <Rom> )` als eigenständige, verschachtelte RDF-Collection interpretiert.

Diese Flexibilität macht RDF-Listen zu einem nützlichen Werkzeug für die Darstellung geordneter Daten in Turtle.

In [8]:

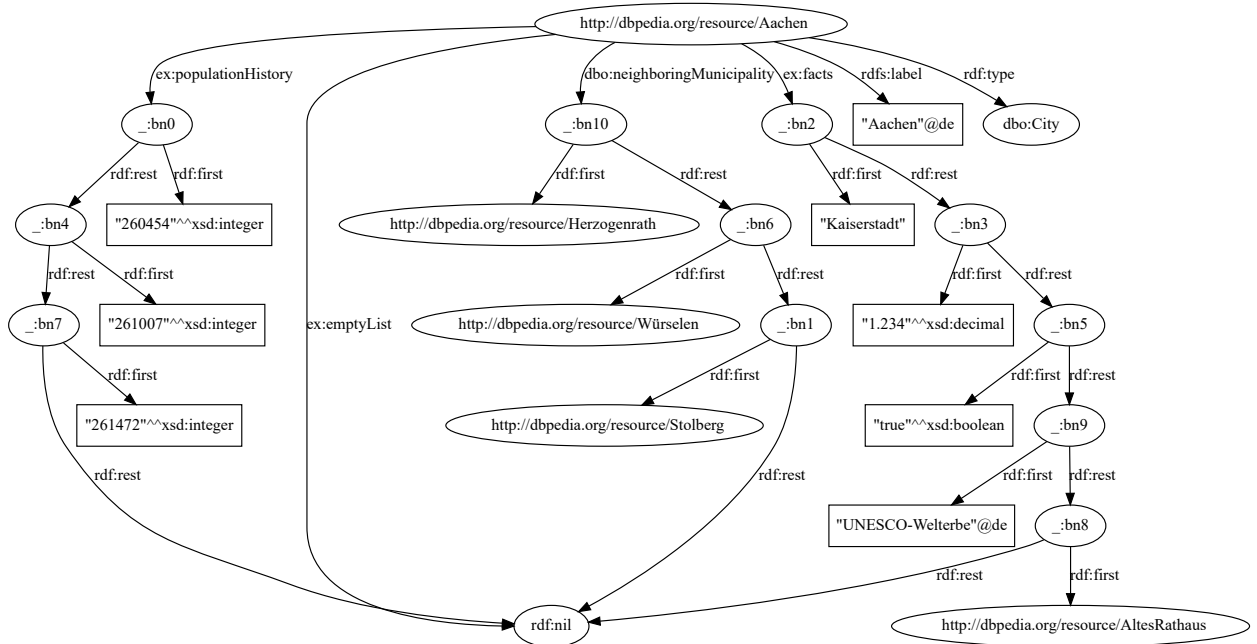
```
%%rdf turtle
@base <http://dbpedia.org/resource/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix ex: <http://example.org/vokabular/> . # Beispiel-Präfix für eigene Begriffe
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> . # Nützlich für explizite Datentypen, falls benötigt

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de ;
  # Beispiel 1: Liste von Nachbarstädten (Elemente sind IRIs)
  dbo:neighboringMunicipality ( <Herzogenrath> <Würselen> <Stolberg> ) ;

  # Beispiel 2: Liste von Einwohnerzahlen der letzten Jahre (Elemente sind xsd:integer Literale)
  ex:populationHistory ( 260454 261007 261472 ) ; # Zahlen werden als xsd:integer interpretiert

  # Beispiel 3: Liste gemischter Datentypen
  ex:facts ( "Kaiserstadt" 1.234 true "UNESCO-Welterbe"@de <AltesRathaus> ) ;
  # Enthält: xsd:string, xsd:decimal, xsd:boolean, rdf:langString, IRI

  # Beispiel 4: Leere Liste
  ex:emptyList ( ) . # Eine leere Liste wird zu rdf:nil
```



Verschachtelte Collections (Listen in Listen)

Wie bereits in den Eigenschaften von Collections erwähnt, können Listen auch andere Listen als Elemente enthalten. Dies ermöglicht die Darstellung von mehrdimensionalen oder gruppierten geordneten Daten.

Betrachten wir ein Beispiel, bei dem wir für Aachen eine Liste von Informationen zu seinen Stadtteilen speichern wollen, wobei jeder Eintrag für einen Stadtteil selbst wieder eine kleine Liste (hier: Name des Stadtteils und dessen ungefähre Einwohnerzahl) ist:

```
@base <http://dbpedia.org/resource/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix ex: <http://example.org/vokabular/> . # Beispiel-Präfix

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de ;
  ex:stadtteilInfo ( # Beginn der äußeren Liste
    ( "Aachen-Mitte" 75321 ) # 1. inneres Element: eine Liste aus Name und
                              # Einwohnerzahl
    ( "Brand" 18023 ) # 2. inneres Element: eine Liste
    ( "Eilendorf" 15710 ) # 3. inneres Element: eine Liste
    ( "Haaren" 12690 ) # 4. inneres Element: eine Liste
  ) . # Ende der äußeren Liste
```

Erläuterung des Beispiels:

- Die Eigenschaft `ex:stadtteilInfo` von `<Aachen>` hat als Objekt eine **äußere Liste**.
- Jedes Element dieser äußeren Liste ist selbst wieder eine **innere Liste**, die hier aus zwei Elementen besteht: dem Namen des Stadtteils (ein Literal vom Typ `xsd:string`) und dessen Einwohnerzahl (ein Literal vom Typ `xsd:integer`).

Wichtige Aspekte bei verschachtelten Listen:

1. **Zugrundeliegende RDF-Struktur:** Auch hier wird jede innere Liste (...) und die äußere Liste (...) vom Turtle-Parser in die bekannte RDF-Listenstruktur mit `rdf:first`, `rdf:rest` und `rdf:nil` übersetzt. Die Verschachtelung führt also zu einer tieferen Struktur von miteinander verbundenen (oft Blank) Nodes im resultierenden RDF-Graphen. Wenn Sie dieses Beispiel in einer `%%rdf turtle`-Zelle ausführen, wird die Visualisierung diese Komplexität zeigen.
2. **Semantik der inneren Listen:** In diesem Beispiel ist die innere Liste ("Aachen-Mitte" 75321) einfach eine geordnete Sammlung von zwei Literalen. Die *Bedeutung*, dass das erste Element der Name und das zweite die Einwohnerzahl ist, ergibt sich hier nur aus der **Konvention** oder dem Kontext, in dem diese Daten verwendet werden. RDF selbst schreibt diese Interpretation nicht vor.
3. **Alternative Modellierung für mehr Explizitheit:** Um die Bedeutung der Elemente innerhalb der inneren "Paare" expliziter zu machen, könnte man statt der inneren Listen auch Blank Nodes mit benannten Eigenschaften verwenden. Die äußere Liste würde dann eine Liste von Blank Nodes enthalten:

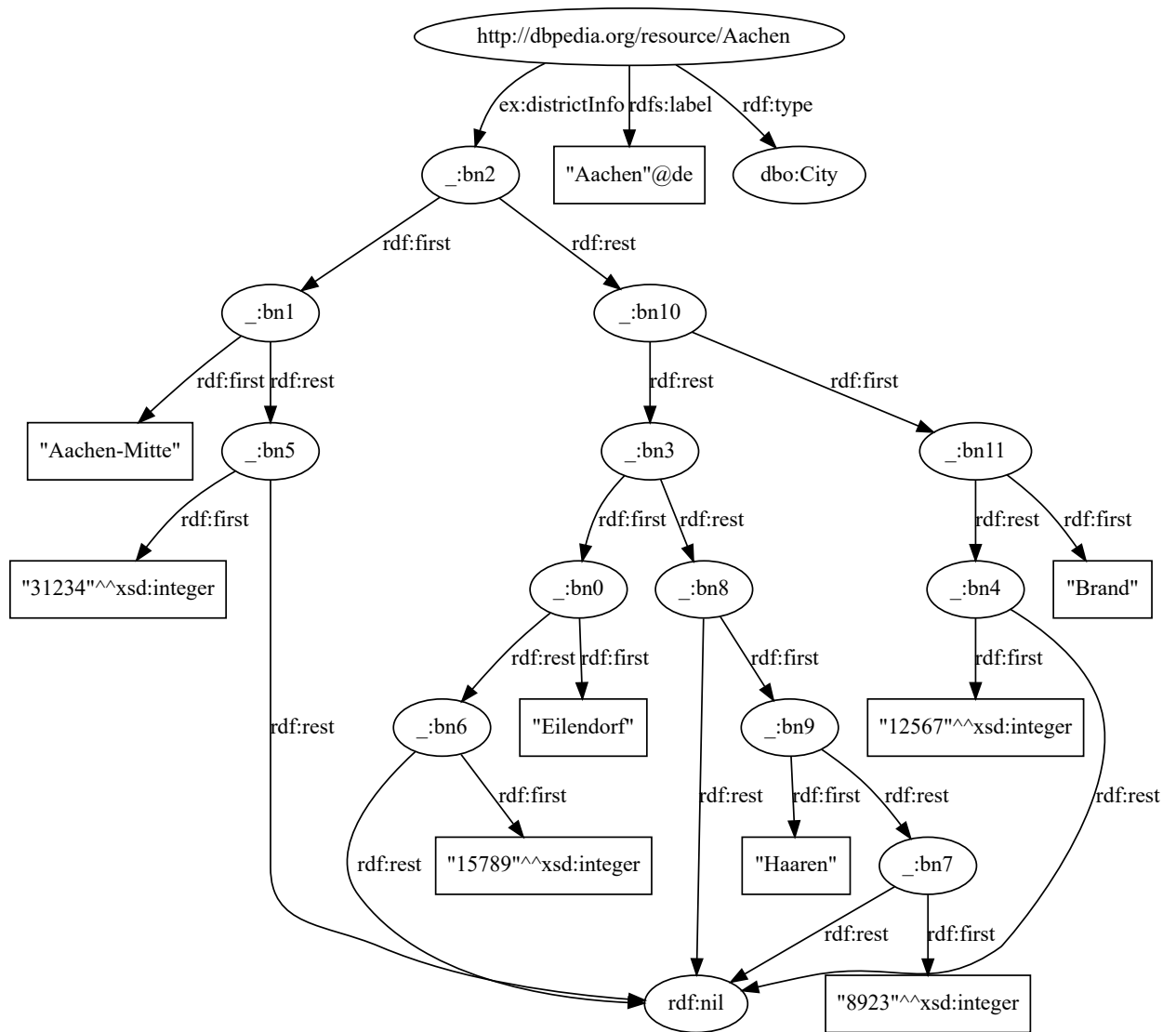
```
# Alternative mit Blank Nodes für mehr semantische Klarheit der inneren Struktur
<Aachen> ex:stadtteilInfoAlternative (
  [ ex:stadtteilName "Aachen-Mitte"; ex:einwohnerzahl 75321 ]
  [ ex:stadtteilName "Brand"; ex:einwohnerzahl 18023 ]
  [ ex:stadtteilName "Eilendorf"; ex:einwohnerzahl 15710 ]
  [ ex:stadtteilName "Haaren"; ex:einwohnerzahl 12690 ]
) .
```

Diese alternative Modellierung ist ausführlicher, aber die Rolle jedes Datums ist durch ein Prädikat klar definiert. Die Wahl hängt vom gewünschten Grad an Explizitheit und den Anforderungen der Anwendung ab. Die rein syntaktische Verschachtelung von Listen, wie im ersten Beispiel gezeigt, ist jedoch eine valide und oft sehr kompakte Möglichkeit, geordnete, gruppierte Daten darzustellen.

Die Fähigkeit, Listen zu verschachteln, erhöht die Ausdruckskraft von Turtle und ermöglicht die Repräsentation komplexer, geordneter Strukturen.

```
In [9]: %%rdf turtle
@base <http://dbpedia.org/resource/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dbo: <https://dbpedia.org/ontology/> .
@prefix ex: <http://example.org/> .

<Aachen> a dbo:City ;
  rdfs:label "Aachen"@de ;
  ex:districtInfo (
    ( "Aachen-Mitte" 31234 )
    ( "Brand" 12567 )
    ( "Eilendorf" 15789 )
    ( "Haaren" 8923 )
  ) .
```



In [10]:

```
%%rdf turtle -l aachen1
```

```
# 2. Modellierungsalternative
```

```
@base <http://dbpedia.org/resource/> .
```

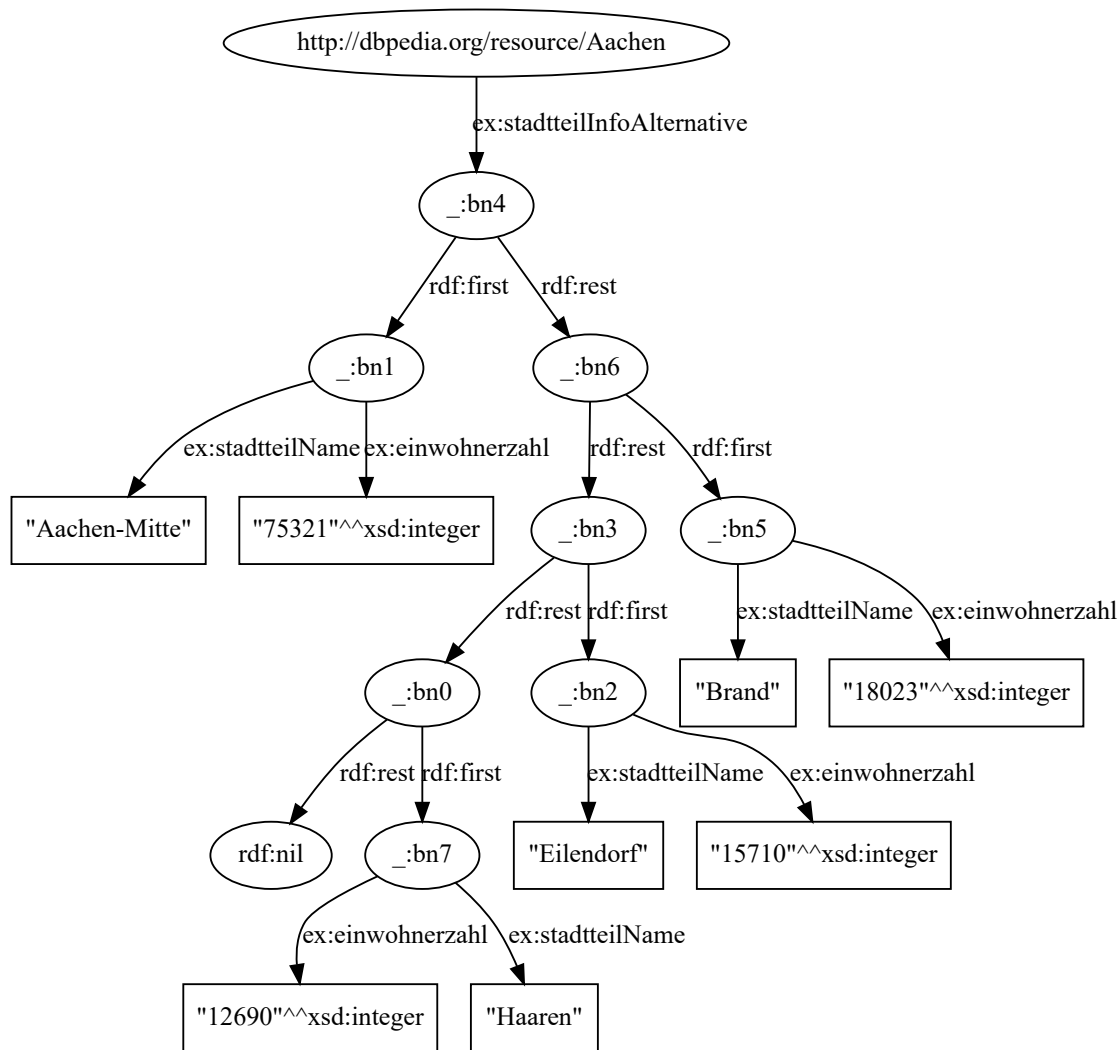
```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
```

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
```

```
@prefix dbo: <https://dbpedia.org/ontology/> .
```

```
@prefix ex: <http://example.org/> .
```

```
<Aachen> ex:stadtteilInfoAlternative (
  [ ex:stadtteilName "Aachen-Mitte"; ex:einwohnerzahl 75321 ]
  [ ex:stadtteilName "Brand"; ex:einwohnerzahl 18023 ]
  [ ex:stadtteilName "Eilendorf"; ex:einwohnerzahl 15710 ]
  [ ex:stadtteilName "Haaren"; ex:einwohnerzahl 12690 ]
) .
```



Verwendungszwecke für Collections:

- **Rangfolgen:** Geordnete Listen von Elementen
- **Koordinaten:** Geographische Punkte (Längen-/Breitengrad)
- **Zeitreihen:** Chronologische Datenfolgen
- **Strukturierte Daten:** Zusammengehörige Informationsgruppen

Wann Collections verwenden:

- Wenn die Reihenfolge wichtig ist
- Bei festen, zusammengehörigen Datengruppen
- Für numerische Sequenzen oder Koordinaten
- Wenn eine natürliche Listenstruktur vorliegt

Collections sind ein mächtiges Werkzeug in Turtle, um strukturierte, geordnete Daten kompakt und lesbar darzustellen.

Übungsaufgaben zu Turtle

Diese Aufgaben sollen Ihnen helfen, die verschiedenen Aspekte der Turtle-Syntax praktisch anzuwenden und zu verstehen. Nutzen Sie die `%rdf turtle` Zellen im Notebook, um Ihre Lösungen zu überprüfen und die resultierenden Graphen zu visualisieren!

Übung 1: Grundlagen – Tripel, IRIs und Literale

Gegeben sind die folgenden Informationen über eine fiktive Person und ein Buch:

- Person: Dr. Elara Vance
- Beruf der Person: Astrophysikerin
- Geburtsdatum der Person: 1985-07-22
- Buch-Titel (Englisch): "Cosmic Journeys"
- Buch-Titel (Deutsch): "Kosmische Reisen"
- Das Buch hat 350 Seiten.

- Das Buch ist spannend (Wahrheitswert).

Aufgaben:

1. Definieren Sie für Dr. Vance und das Buch "Cosmic Journeys" passende (fiktive) **vollständige IRIs** als Subjekte.
2. Wählen Sie passende (fiktive) **vollständige IRIs** für die Eigenschaften (Prädikate) "Beruf", "Geburtsdatum", "Titel", "AnzahlSeiten" und "istSpannend".
3. Formulieren Sie für jede oben genannte Information ein **einzelnes RDF-Tripel in Turtle-Syntax**.
 - Verwenden Sie für die Literale korrekte Schreibweisen:
 - Einfache Strings ("...")
 - Strings mit Sprach-Tags (...@de , ...@en)
 - Zahlen mit Turtle-Kurzschreibweisen (für Integer)
 - Wahrheitswerte (true / false)
 - Ein Datum im Format "YYYY-MM-DD"^^xsd:date (definieren Sie das xsd: Präfix oder schreiben Sie die Datentyp-IRI aus).

In [11]:

```
%%rdf turtle
```

Übung 2: Präfixe anwenden (@prefix und @base)

Gegeben seien die folgenden Tripel mit vollständigen IRIs:

```
<http://example.org/uni/veranstaltung/VL_Logik> <http://purl.org/dc/terms/title> "Logik für Informatiker"@de .
<http://example.org/uni/veranstaltung/VL_Logik> <http://example.org/uni/vokabular#hatECTS> "6"^^<http://
www.w3.org/2001/XMLSchema#integer> .
<http://example.org/uni/personal/Prof_Curie> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
xmlns.com/foaf/0.1/Person> .
<http://example.org/uni/personal/Prof_Curie> <http://xmlns.com/foaf/0.1/name> "Prof. Dr. Marie Curie" .
<http://example.org/uni/personal/Prof_Curie> <http://example.org/uni/vokabular#haeltVeranstaltung> <http://
example.org/uni/veranstaltung/VL_Logik> .
```

Aufgaben:

1. **Identifizieren:** Welche Teile der oben genannten IRIs wiederholen sich und könnten durch Namensraum-Präfixe abgekürzt werden? Notieren Sie die Namensraum-IRIs.
2. **Definieren:** Schreiben Sie für jeden identifizierten Namensraum eine passende @prefix -Anweisung. Wählen Sie kurze, aber aussagekräftige Präfix-Label (z.B. dcterms: für Dublin Core Terms, foaf: für Friend-of-a-Friend, uni_v: für Ihr Uni-Vokabular etc.). Definieren Sie auch ein Präfix für xsd: .
3. **Anwenden (Optional @base):** Überlegen Sie, ob die Verwendung einer @base -Direktive hier sinnvoll wäre, um die IRIs für Veranstaltungen und Personal weiter zu verkürzen. Definieren Sie gegebenenfalls eine @base -IRI.
4. **Umschreiben:** Schreiben Sie alle oben genannten RDF-Tripel in Turtle neu, unter maximaler Verwendung Ihrer definierten Präfixe und (falls Sie sich dafür entschieden haben) der @base -Direktive mit relativen IRIs.

Ziel: Der neue Turtle-Code soll denselben RDF-Graphen repräsentieren, aber deutlich kürzer und lesbarer sein. Überprüfen Sie Ihre Lösung idealerweise in einer %%rdf turtle -Zelle.

Lösung zu Übung 2: Präfixe anwenden (@prefix und @base)

In [12]:

```
%%rdf turtle
#
```

In [13]:

```
%%rdf turtle
#
```

Übung 3: Turtle-Abkürzungen (; und ,)

Gegeben ist der folgende (korrekte, aber etwas ausführliche) Turtle-Code:

```
@prefix schema: <http://schema.org/> .
@prefix ex: <http://example.org/entity/> .

ex:Produkt001 a schema:Product .
ex:Produkt001 schema:name "Super-Bohrmaschine"@de .
```

```
ex:Produkt001 schema:description "Leistungsstarke Bohrmaschine für Heimwerker"@de .
ex:Produkt001 schema:color "rot"@de .
ex:Produkt001 schema:color "schwarz"@de .
ex:Produkt001 schema:brand ex:MarkeXYZ .
ex:MarkeXYZ a schema:Brand .
ex:MarkeXYZ schema:name "Marke XYZ Tools" .
```

Aufgabe:

Schreiben Sie den obigen Turtle-Code so kompakt wie möglich neu, indem Sie die Abkürzungen Semikolon (;) und Komma (,) verwenden, wo immer es sinnvoll ist. Achten Sie darauf, dass der resultierende RDF-Graph identisch bleibt.

Ziel: Machen Sie sich mit der Anwendung der Semikolon- und Komma-Syntax vertraut, um Turtle-Code lesbarer und kürzer zu gestalten. Überprüfen Sie Ihre Lösung in einer `%%rdf turtle`-Zelle.

```
In [14]: %%rdf turtle
#
```

Übung 4: Blank Nodes (_: und [...])

Sie möchten Informationen über eine Veranstaltung und deren Organisator modellieren. Die Veranstaltung ist das "Sommerfest 2025". Der Organisator ist ein Komitee, das keinen eigenen offiziellen Namen oder eine eigene IRI hat. Dieses Komitee hat eine Kontakt-E-Mail-Adresse (`komitee@example.org`) und ein Telefon (`+49-123-987654`).

Aufgaben:

1. Modellierung mit benanntem Blank Node:

- Erstellen Sie RDF-Tripel in Turtle, um die Veranstaltung und das Organisationskomitee zu beschreiben.
- Verwenden Sie eine IRI für das Sommerfest (z.B. `event:Sommerfest25`).
- Stellen Sie das Komitee als **benannten Blank Node** dar (z.B. `_:orgaKomitee`).
- Weisen Sie dem Komitee die Eigenschaften E-Mail und Telefon zu. Verwenden Sie hierfür passende Prädikate (z.B. aus dem `schema.org` -Vokabular wie `schema:email` , `schema:telephone` oder definieren Sie eigene).

2. Modellierung mit anonymer Blank-Node-Syntax:

- Modellieren Sie dieselbe Information (Veranstaltung mit Organisator-Komitee, dessen E-Mail und Telefon) erneut.
- Verwenden Sie diesmal jedoch die **anonyme Blank-Node-Syntax** `[ ... ]` für das Komitee und dessen Eigenschaften, direkt als Objekt der Eigenschaft, die das Komitee mit der Veranstaltung verbindet.

3. Bonus: Verschachtelte Blank Nodes:

- Die Kontakt-E-Mail und das Telefon des Komitees gehören eigentlich zu einer bestimmten Kontaktperson innerhalb des Komitees, Frau Meier, die ebenfalls keine eigene IRI hat.
- Erweitern Sie Ihre Lösung aus Aufgabe 2. Der Blank Node, der das Komitee repräsentiert, soll nun eine Eigenschaft `schema:contactPoint` (oder ein ähnliches Prädikat) haben.
- Der Wert dieser `schema:contactPoint` -Eigenschaft soll ein *weiterer*, neuer (verschachtelter) Blank Node sein, der Frau Meier repräsentiert.
- Dieser innere Blank Node für Frau Meier soll dann die Eigenschaften für E-Mail und Telefon sowie einen Namen (z.B. `foaf:name` "Frau Meier") erhalten.

Ziel: Verstehen und Anwenden der beiden verschiedenen Syntax-Varianten für Blank Nodes in Turtle sowie deren Verschachtelung zur Strukturierung von Daten. Überprüfen Sie Ihre Lösungen in einer `%%rdf turtle`-Zelle.

```
In [15]: %%rdf turtle
#
```

```
In [16]: %%rdf turtle
#
```

```
In [17]: %%rdf turtle
#
```

Übung 5: Collections / Listen (...)

Ein Blog-Artikel (`blog:Artikel142`) hat die folgenden Schlagwörter (Tags) in dieser spezifischen Reihenfolge: "RDF", "Turtle", "Semantic Web", "Datenmodellierung".

Aufgaben:

1. **Einfache Liste erstellen:** Modellieren Sie die obige Information in Turtle. Verwenden Sie eine RDF-Liste, um die Schlagwörter darzustellen. Diese Liste soll das Objekt der Eigenschaft `blog:hatSchlagworte` (oder eines ähnlich benannten Prädikats Ihrer Wahl) für den Artikel `blog:Artikel142` sein. Die Schlagwörter selbst sollen einfache String-Literale sein. Denken Sie an die notwendigen `@prefix`-Definitionen.
2. **Verschachtelte Liste erstellen:** Fügen Sie dem Artikel `blog:Artikel142` eine weitere Eigenschaft `blog:autorenReihenfolge` hinzu. Der Wert dieser Eigenschaft soll eine Liste sein, die:
 - zuerst zwei Autoren-IRIs enthält (z.B. `pers:AutorA` und `pers:AutorB`),
 - und dann als drittes Element eine *verschachtelte Liste* mit zwei weiteren Co-Autoren-IRIs (z.B. `pers:CoAutorX` und `pers:CoAutorY`).
3. **Leere Liste darstellen:** Wie würden Sie in Turtle ausdrücken, dass der Artikel `blog:Artikel142` eine Eigenschaft `blog:kommentare` hat, für die es aber noch keine Kommentare gibt (d.h., der Wert ist eine leere Liste)?

Ziel: Die Syntax für RDF-Listen in Turtle verstehen und anwenden können, inklusive der Darstellung von Elementen verschiedener Typen, verschachtelten Listen und leeren Listen. Überprüfen Sie Ihre Lösungen in einer `%%rdf turtle`-Zelle und beobachten Sie, wie die Listenstruktur intern mit `rdf:first`, `rdf:rest` und `rdf:nil` aufgebaut wird.

```
In [18]: %%rdf turtle
#
```

```
In [19]: %%rdf turtle
#
```

```
In [20]: %%rdf turtle
#
```

```
In [21]: %%rdf turtle
#
```

Übung 6: Turtle lesen und interpretieren

Gegeben sei der folgende Turtle-Code:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ex: <http://example.org/stuff/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:Document1
  a foaf:Document ;
  dc:title "Einführung in RDF"@de ;
  dc:creator [
    a foaf:Person ;
    foaf:name "Dr. Eva Beispiel" ;
    foaf:mbox <mailto:eva@example.org>
  ] ;
  ex:revision "1.0" ;
  ex:keywords ( "RDF" "Semantic Web" "Linked Data" ) ;
  ex:relatedDocument ex:Document2 .

ex:Document2
  dc:title "Fortgeschrittene RDF-Techniken" .
```

Aufgaben:

1. **Präfixe verstehen:** Identifizieren Sie alle definierten Präfixe und geben Sie die vollständigen Namensraum-IRIs an, für die sie stehen.
2. **Tripel zählen:** Wie viele einzelne RDF-Tripel werden insgesamt durch diesen Code definiert? Versuchen Sie, diese gedanklich oder auf

Papier aufzuschlüsseln. (Tipp: Denken Sie daran, wie Blank Nodes mit Eigenschaften und Listen intern expandiert werden).

3. **Aussagen über ex:Document1** : Listen Sie alle direkten Eigenschaften (Prädikate) von ex:Document1 auf und geben Sie deren jeweilige Werte (Objekte) an.
4. **Blank Node analysieren:**
 - Welche Ressource wird als Objekt der Eigenschaft dc:creator von ex:Document1 verwendet?
 - Welche Aussagen werden über diese Ressource gemacht?
5. **Datentypen erkennen:** Welchen expliziten Datentyp hat der Wert der Eigenschaft ex:revision von ex:Document1 ?
6. **Listeninhalte identifizieren:** Was sind die einzelnen Elemente der Liste, die als Wert für die Eigenschaft ex:keywords von ex:Document1 dient?

Ziel: Die Fähigkeit trainieren, auch komplexere Turtle-Strukturen zu lesen, die verschiedenen Syntaxelemente korrekt zu interpretieren und die zugrundeliegenden RDF-Aussagen zu verstehen.

```
In [22]: %%rdf turtle
#
```

Übung 7: Eigene RDF-Daten in Turtle modellieren

Modellieren Sie die folgenden Informationen über die RWTH Aachen in Turtle. Verwenden Sie geeignete Präfixe (z.B. ex für Ihre eigenen Entitäten, dbo für DBpedia Ontology Begriffe wie dbo:University, foaf für Friend-of-a-Friend-Begriffe wie foaf:name oder foaf:homepage, und xsd für Datentypen) und Turtle-Abkürzungen, wo immer es sinnvoll ist:

- Die RWTH Aachen (z.B. ex:RWTHAachen) ist eine Universität (dbo:University).
- Ihr deutscher Name ist "Rheinisch-Westfälische Technische Hochschule Aachen" (Literal mit Sprachcode de).
- Ihr englischer Name ist "RWTH Aachen University" (Literal mit Sprachcode en).
- Sie hat eine Homepage unter der URL http://www.rwth-aachen.de (als IRI, verwenden Sie z.B. foaf:homepage).
- Sie befindet sich in der Stadt Aachen (z.B. ex:AachenCity).
- Die Stadt Aachen (ex:AachenCity) hat den Namen "Aachen" (Literal).
- Die Vorlesung "Datenbanken und Informationssysteme" (ex:DM_Lecture) wird an der ex:RWTHAachen gehalten und hat 6 ECTS-Punkte (xsd:integer).

Tragen Sie Ihre Lösung in die folgende Zelle ein und beobachten Sie den generierten Graphen und die Tripel.

```
In [23]: %%rdf turtle
#
```

Übung 8: Eigene RDF-Daten in Turtle modellieren

Bitte schauen Sie sich den folgenden RDF-Graphen an und beantworten Sie die folgenden Fragen: **Fragen:**

1. Wie lautet der Name des Kurses?
2. Wer hält den Kurs (Name und E-Mail des Dozenten/der Dozentin)? Wird für den Dozenten/die Dozentin eine eigene IRI verwendet oder ein Blank Node?
3. Wie viele Kreditpunkte (Credits) gibt der Kurs und welchen Datentyp hat dieser Wert?
4. Nennen Sie mindestens zwei Themen des Kurses.
5. In welcher Sprache ist die Beschreibung des Kurses verfasst?

```
In [24]: %%rdf turtle
@prefix ex: <http://example.org/courses#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix schema: <http://schema.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:AdvancedDataAnalytics
  a schema:Course ;
  foaf:name "Advanced Data Analytics" ;
  schema:description "Ein Kurs über fortgeschrittene Datenanalysemethoden."@de ;
  ex:taughtBy [
    a foaf:Person ;
    foaf:name "Dr. Eva Mustermann" ;
    foaf:mbox <mailto:eva.mustermann@example.com>
  ] ;
  ex:credits "7.5"^^xsd:decimal ;
  ex:topics ( "Maschinelles Lernen" "Big Data Technologien" "Datenvisualisierung" ) .
```


Beim Schreiben von Turtle können leicht kleine Fehler passieren. Achten Sie besonders auf:

- **Fehlender Punkt (.):** Jedes vollständige Tripel (oder eine Gruppe von Tripeln mit demselben Subjekt, abgeschlossen durch ; oder ,) muss mit einem Punkt enden.
- **Leerzeichen bei Präfixen:** `präfix:Name` ist korrekt. `präfix: Name` (mit Leerzeichen nach dem Doppelpunkt) ist falsch.
- **Klammerpaare:** Achten Sie auf korrespondierende spitze Klammern `<>`, runde Klammern `()` für Collections und eckige Klammern `[]` für Blank Nodes.
- **Anführungszeichen bei Literalen:** Literale müssen von doppelten Anführungszeichen umschlossen sein (z.B. `"Text"`). Mehrzeilige Literale verwenden `"""..."""`.
- **Korrekte Verwendung von ; und , :** Das Semikolon leitet ein neues Prädikat-Objekt-Paar für dasselbe Subjekt ein. Das Komma leitet ein neues Objekt für dasselbe Subjekt und dasselbe Prädikat ein.

Zusammenfassung und Ausblick

Dieses Tutorial hat eine Einführung in die Turtle-Syntax gegeben. Die wichtigsten Lerninhalte waren:

- IRIs, Literale (einfach, mit Sprach-Tag, mit Datentyp), Blank Nodes
- Präfixe (`@prefix` , `@base`) und der leere Präfix (`:`)
- Abkürzungen für Prädikat-Objekt-Paare (`;`) und Objekte (`,`)
- Die Abkürzung `a` für `rdf:type`
- Kommentare (`#`)
- Benannte Blank Nodes (`_:name`)
- Anonyme Blank Nodes mit direkter Eigenschaftszuweisung (`[]`)
- Sammlungen / Listen (`()`), auch mit komplexen Elementen oder verschachtelt

Diese Syntaxelemente machen Turtle zu einer ausdrucksstarken und lesbaren Alternative zu anderen RDF-Serialisierungen wie RDF/XML oder N-Triples, ohne dabei die semantische Bedeutung der zugrundeliegenden RDF-Triples zu verändern.

In []:

```
# Installiere benötigte Pakete
# !pip install rdflib matplotlib
# from IPython.display import display, HTML
# !pip install graphviz
# !pip install -q requests -U --user
# !pip install -q jupyter-rdfify~1.2 --index-url https://git.rwth-aachen.de/api/v4/projects/49225/packages/pypi/simple
%reload_ext jupyter-rdfify
```

RDF Schema (RDFS) - Eine Einführung in die Vokabularbeschreibung

Vorlesung: Semantic Web Technologies Zielgruppe: Bachelor Informatik, 4. Semester

Überblick und Lernziele

Aufbauend auf unserem Wissen über die Grundlagen von RDF (siehe Notebook 01) und dessen Serialisierung mittels Turtle (Notebook 02), führt dieses Notebook in **RDF Schema (RDFS)** ein. Wir werden untersuchen, wie RDFS es uns ermöglicht, explizite **Vokabulare** (Schemata) für unsere RDF-Daten zu definieren und ihnen dadurch mehr Struktur und semantische Bedeutung zu verleihen.

In dieser Lerneinheit werden Sie:

- verstehen, was RDF Schema ist und welche Rolle es im Semantic Web spielt.
- die wichtigsten RDFS-Konstrukte kennenlernen: **Klassen** (`rdfs:Class`), **Eigenschaften** (`rdf:Property`), sowie deren Gültigkeitsbereiche mittels **Domain** (`rdfs:domain`) und **Range** (`rdfs:range`).
- die Konzepte von **Klassen- und Eigenschafts-Hierarchien** (`rdfs:subClassOf` , `rdfs:subPropertyOf`) und die damit verbundene Vererbung nachvollziehen.
- den Unterschied zwischen der **terminologischen Ebene** (Schema/TBox, d.h. das Vokabular) und der **assertionalen Ebene** (Instanzdaten/ ABox, d.h. die konkreten Fakten) klarer verstehen.
- die Bedeutung von RDFS für **einfache Inferenzmechanismen** (automatisches Ableiten neuer Fakten) erkennen.
- diese Konzepte praktisch anwenden, indem wir unser bekanntes **Universitätsdatenbeispiel** mit RDFS erweitern und analysieren.

Voraussetzungen:

- Grundlegendes Verständnis von RDF-Konzepten (Tripel, IRIs, Literale, Blank Nodes).
- Kenntnisse der Turtle-Syntax zur Repräsentation von RDF-Daten.
- Grundkenntnisse in Datenmodellierung (z.B. aus ER-Modellierung oder relationalen Datenbanken) sind hilfreich für den Vergleich und das Verständnis von Schema-Konzepten.

Motivation: Warum RDF Schema?

Im vorherigen Notebook haben wir gelernt, wie RDF-Daten als eine Sammlung von Tripeln (Subjekt-Prädikat-Objekt) mithilfe der Turtle-Syntax repräsentiert werden. Diese Tripel bilden einen Graphen. Aber was bedeuten diese Daten eigentlich genau?

Das Problem mit "nacktem" RDF

Betrachten wir einige einfache RDF-Tripel, die Informationen über Studenten und Professoren darstellen könnten:

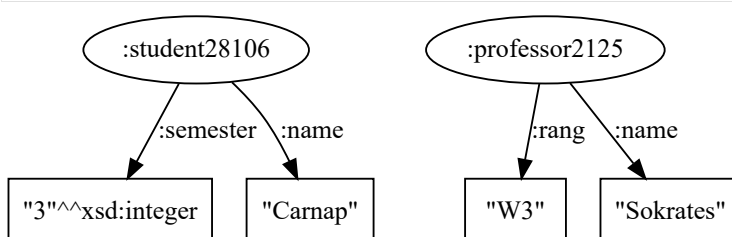
In [3]:

```
%rdf turtle
@base <http://data.rwth-aachen.de/resources/> .
@prefix : <> .

# RDF ohne Schema - nur reine Tripel

<student28106> :name "Carnap" .
<student28106> :semester 3 . # Deutlicherer Prädikatsname

<professor2125> :name "Sokrates" .
<professor2125> :rang "W3" .
```



Dieser RDF-Graph enthält zwar Daten, aber ohne ein zusätzliches Schema (ein Vokabular, das die Bedeutung und die erlaubte Verwendung der Terme definiert), stoßen wir schnell an Grenzen:

Probleme ohne ein explizites Schema:

1. Mangelnde Struktur und Typisierung von Ressourcen:

- Was ist `<student28106>` oder `<professor2125>`? Sind es Personen, IDs, Konzepte? Wir wissen es nicht sicher.
- Es gibt keine explizite Information darüber, zu welcher Art oder Klasse diese Ressourcen gehören.

2. Unklare Bedeutung und Verwendung von Eigenschaften (Properties):

- Was genau bedeuten die Eigenschaften `:name`, `:semester` oder `:rang`?
- Für welche Arten von Ressourcen sind diese Eigenschaften gedacht (z.B. ist `:semester` nur für Studenten relevant oder auch für andere Entitäten)?
- Welche Arten von Werten (z.B. Text, Zahl, eine andere Ressource) werden für diese Eigenschaften erwartet?

3. Eingeschränkte Möglichkeiten zur Datenvalidierung:

- Können wir überprüfen, ob die Daten konsistent und korrekt im Sinne unserer beabsichtigten Bedeutung sind?
- Darf ein `<student28106>` auch die Eigenschaft `voc:hatRang` haben? Darf `:semester` den Wert "drei" (als Text) annehmen statt einer Zahl? Ohne Schema fehlen uns die Regeln dafür.
- Hinweis: RDFS selbst bietet hier primär die Grundlage für Konsistenz durch Typinferenz; fortgeschrittenere Validierungsregeln werden oft mit SHACL oder OWL definiert.

4. Fehlende Grundlage für automatische Schlussfolgerungen (Inferenzen):

- Wenn wir wüssten, dass alle Studenten auch Personen sind und `<student28106>` ein Student ist, könnten wir ohne Schema nicht automatisch ableiten, dass `<student28106>` auch eine Person ist.
- Solche Ableitungen von implizitem Wissen aus explizit Gesagtem sind ein Kernziel des Semantic Web.

Die Lösung: RDF Schema (RDFS)

Um diese Unklarheiten zu beseitigen, die Daten besser zu strukturieren und eine Grundlage für reichhaltigere Anwendungen zu schaffen, benötigen wir eine Möglichkeit, die **Bedeutung unseres Vokabulars explizit zu machen**. Genau hier setzt **RDF Schema (RDFS)** an.

→ **RDF Schema bietet die grundlegenden Mechanismen, um Vokabulare für RDF-Daten zu definieren, ihnen Struktur zu verleihen und einfache Schlussfolgerungen zu ermöglichen!**

RDF Schema Grundlagen

Nachdem wir nun wissen, wie RDF-Daten mit Turtle serialisiert werden, wollen wir uns ansehen, wie wir diesen Daten eine explizite Struktur und Bedeutung geben können. Hier kommt RDF Schema (RDFS) ins Spiel.

Was ist RDF Schema (RDFS)?

RDF Schema (RDFS) ist eine **W3C-Empfehlung (Recommendation)**, die das grundlegende RDF-Vokabular erweitert, um die Definition von einfachen **Ontologien** oder **Schemata** für RDF-Daten zu ermöglichen. RDFS erlaubt es uns:

- ein **Vokabular** zur Beschreibung unserer RDF-Daten zu definieren (Welche Arten von Dingen und Beziehungen gibt es?).
- **Typisierung** für unsere Ressourcen einzuführen (Zu welcher Klasse gehört eine Ressource?).
- **Hierarchien** zwischen Klassen und Eigenschaften zu erstellen (Spezialisierung/Generalisierung).
- eine Basis für **automatische Schlussfolgerungen (Inferenzen)** zu legen (Aus vorhandenen Daten neues Wissen ableiten).

Im Wesentlichen hilft RDFS, die intendierte Semantik unserer Daten explizit zu machen und so die Interoperabilität und das Maschinenverständnis zu verbessern.

Kernkonzepte von RDFS im Überblick:

Die folgende Tabelle fasst die wichtigsten RDFS-Konstrukte zusammen und stellt ihnen bekannte Konzepte aus der ER-Modellierung und relationalen Datenbanken gegenüber. Beachten Sie, dass diese Vergleiche als erste Orientierung dienen; die genaue Semantik und Funktionsweise in RDFS unterscheidet sich oft im Detail, wie wir später sehen werden.

RDFS-Konstrukt	Zweck in RDFS	Ungefäher Vergleich mit ER / Relationalem Modell
<code>rdfs:Class</code>	Definiert eine Klasse (eine Gruppe oder Kategorie von Ressourcen).	Entitätstyp / Tabelle (Hinweis: In RDFS kann eine Ressource Instanz mehrerer Klassen sein)
<code>rdf:Property</code>	Definiert eine Eigenschaft (eine Beziehung zwischen Ressourcen oder von einer Ressource zu einem Literal).	Attribut / Spalte / Beziehungstyp (Hinweis: RDF Properties sind globaler definiert)
<code>rdfs:domain</code>	Legt fest, dass das Subjekt einer Eigenschaft Instanz einer bestimmten Klasse ist.	Attribut "gehört zu" Entitätstyp (Wichtig: In RDFS primär eine Inferenzregel, keine strikte Constraint)
<code>rdfs:range</code>	Legt fest, dass das Objekt (Wert) einer Eigenschaft Instanz einer bestimmten Klasse oder eines bestimmten Datentyps ist.	Datentyp einer Spalte / Ziel-Entitätstyp einer Beziehung (Wichtig: In RDFS primär eine Inferenzregel)
<code>rdfs:subClassOf</code>	Definiert eine Hierarchie zwischen Klassen (Spezialisierung/Generalisierung).	Spezialisierung/Generalisierung (z.B. IS-A Beziehung in ER-Modellen)

RDFS-Konstrukt	Zweck in RDFS	Ungefäher Vergleich mit ER / Relationalem Modell
<code>rdfs:subPropertyOf</code>	Definiert eine Hierarchie zwischen Eigenschaften.	<i>kein direktes Äquivalent</i>
<code>rdfs:label</code>	Stellt einen menschenlesbaren Namen für eine Ressource (Klasse, Eigenschaft etc.) bereit.	Spalten-/Tabellen-Alias, beschreibender Name
<code>rdfs:comment</code>	Stellt eine ausführlichere, menschenlesbare Beschreibung für eine Ressource bereit.	Kommentar/Beschreibung in DDL oder Datenmodell

Diese RDFS-Konstrukte sind selbst wieder Ressourcen, die durch IRI's aus dem RDFS-Namensraum (`http://www.w3.org/2000/01/rdf-schema#` , üblicherweise mit Präfix `rdfs:`) und dem RDF-Namensraum (`http://www.w3.org/1999/02/22-rdf-syntax-ns#` , üblicherweise mit Präfix `rdf:`) identifiziert werden. Im Folgenden werden wir uns diese Konzepte genauer ansehen.

Praktisches Beispiel: Ein RDFS-Schema für Universitätsdaten

Wir entwickeln nun schrittweise ein RDFS-Schema für unser bekanntes Universitätsbeispiel. Dabei werden wir die im vorherigen Abschnitt ("2. RDF Schema Grundlagen") vorgestellten RDFS-Kernkonzepte praktisch anwenden und erläutern.

Schritt 1: Klassen definieren mit `rdfs:Class`

Konzept: Eine Klasse (`rdfs:Class`) in RDFS beschreibt eine Gruppe oder Kategorie von Ressourcen, die gemeinsame Charakteristika oder Eigenschaften haben. Man kann sie als eine Art "Schablone" oder "Typ" für ähnliche Dinge verstehen. In RDFS ist eine Klasse selbst eine Ressource und wird durch eine IRI identifiziert.

Turtle-Code zum Definieren von Klassen: Beginnen wir damit, die grundlegenden Typen von Entitäten in unserem Universitätsbeispiel zu definieren: Studenten, Professoren und Vorlesungen.

```
@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <> . # Leerer Präfix für Lokale Ressourcen relativ zur @base IRI

# Definition von Klassen für unser Universitätsbeispiel
# Jede Klasse ist eine Ressource vom Typ rdfs:Class.

:Student a rdfs:Class ;
  rdfs:label "Student"@de , "Student"@en ; # Mehrsprachige Labels sind möglich
  rdfs:comment "Eine Person, die an einer Hochschule eingeschrieben ist und studiert."@de .

:Professor a rdfs:Class ;
  rdfs:label "Professor"@de , "Professor"@en ;
  rdfs:comment "Eine Lehrkraft mit akademischer Professur an einer Hochschule."@de .

:Vorlesung a rdfs:Class ;
  rdfs:label "Vorlesung"@de , "Lecture"@en ;
  rdfs:comment "Eine spezifische Lehrveranstaltung, die von Dozierenden gehalten wird."@de .
```

Erklärung der verwendeten RDFS-Konstrukte:

- `:Student rdf:type rdfs:Class`** : Diese Aussage deklariert `:Student` (was durch `@base` und den leeren Präfix `:` zur IRI `<http://data.rwth-aachen.de/resources#Student>` expandiert wird) als eine neue Klasse. Dasselbe gilt für `:Professor` und `:Vorlesung`. Beachten Sie, dass die Klasse selbst hier das Subjekt der Aussage ist – wir beschreiben die Klasse.
- `rdfs:label`** : Weist der Klasse einen oder mehrere menschenlesbare Namen (Labels) zu. Hier wurden beispielhaft deutsche und englische Labels hinzugefügt.
- `rdfs:comment`** : Fügt eine menschenlesbare Beschreibung oder Erläuterung für die Klasse hinzu.

Was sehen wir im Graphen? Wenn Sie die obige `%rdf turtle` -Zelle ausführen, visualisiert das Tool die drei von uns definierten Klassen (`:Student` , `:Professor` , `:Vorlesung`) als Knoten im Graphen. Von jedem dieser Klassenknoten gehen Kanten (Prädikate) wie `rdf:type` , `rdfs:label` und `rdfs:comment` zu ihren jeweiligen Objekten (`rdfs:Class` bzw. den Literalen für Label und Kommentar). Wichtig ist hier: Wir haben bisher nur die *Konzepte* oder *Typen* definiert, noch keine konkreten Instanzen (also noch keinen spezifischen Studenten wie "PeterPan" oder eine spezifische Vorlesung wie "SemanticWebLecture"). Das Anlegen von Instanzen und deren Zuweisung zu diesen Klassen folgt in einem späteren Schritt.

In [4]:

```
%%rdf turtle
@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <> . # Leerer Präfix für lokale Ressourcen relativ zur @base IRI

# Definition von Klassen für unser Universitätsbeispiel
# Jede Klasse ist eine Ressource vom Typ rdfs:Class.

:Student rdfs:type rdfs:Class ;
  rdfs:label "Student"@de , "Student"@en ; # Mehrsprachige Labels sind möglich
  rdfs:comment "Eine Person, die an einer Hochschule eingeschrieben ist und studiert."@de .

:Professor a rdfs:Class ;
  rdfs:label "Professor"@de , "Professor"@en ;
  rdfs:comment "Eine Lehrkraft mit akademischer Professur an einer Hochschule."@de .

:Vorlesung a rdfs:Class ;
  rdfs:label "Vorlesung"@de , "Lecture"@en ;
  rdfs:comment "Eine spezifische Lehrveranstaltung, die von Dozierenden gehalten wird."@de .
```



Schritt 2: Klassenhierarchien mit `rdfs:subClassOf`

Konzept: Klassen in RDFS sind nicht isoliert, sondern können in Hierarchien organisiert werden. Die Beziehung `rdfs:subClassOf` drückt aus, dass eine Klasse eine **Spezialisierung** einer anderen (allgemeineren) Klasse ist. Umgekehrt ist die allgemeinere Klasse eine **Generalisierung** der spezifischeren Klasse. Alle Instanzen einer Unterklasse sind automatisch auch Instanzen aller ihrer Oberklassen.

Turtle-Code zum Definieren von Klassenhierarchien: Wir erweitern unser Schema. Wir gehen davon aus, dass `:Person` und `:Student` bereits wie in Schritt 3.1 als Klassen definiert sind (oder definieren `:Person` hier neu, falls noch nicht geschehen). Nun führen wir die Klasse `:Angestellter` als Unterklasse von `:Person` ein. Anschließend definieren wir `:Professor` und die neue Klasse `:Assistent` als Unterklassen von `:Angestellter`.

(Annahme: Die `@base`-Direktive und die Präfixe `rdfs:`, `rdf:` und `:` aus dem vorherigen Code-Block 3.1 sind weiterhin gültig und werden hier implizit weiterverwendet.)

```
# Oberklasse :Person (falls nicht schon in 3.1 vollständig definiert)
:Person rdfs:type rdfs:Class ;
  rdfs:label "Person"@de , "Person"@en ;
  rdfs:comment "Ein menschliches Wesen."@de .

# Klasse :Student als Unterklasse von :Person (Definition aus 3.1, hier mit subClassOf ergänzt)
:Student rdfs:type rdfs:Class ;
  rdfs:label "Student"@de ;
  rdfs:comment "Eine Person, die an einer Hochschule eingeschrieben ist und studiert."@de ;
  rdfs:subClassOf :Person .

# Neue Oberklasse für Angestellte der Universität
:Angestellter rdfs:type rdfs:Class ;
  rdfs:label "Angestellter"@de , "Employee"@en ;
  rdfs:comment "Eine Person, die bei der Universität angestellt ist."@de ;
  rdfs:subClassOf :Person . # Ein Angestellter ist auch eine Person

# :Professor als Unterklasse von :Angestellter (Definition aus 3.1 angepasst)
:Professor rdfs:type rdfs:Class ;
  rdfs:label "Professor"@de ;
  rdfs:comment "Eine Lehrkraft mit akademischer Professur an einer Hochschule, angestellt bei der Universität."@de ;
  rdfs:subClassOf :Angestellter . # Ein Professor ist ein Angestellter

# Neue Klasse :Assistent als Unterklasse von :Angestellter
:Assistent rdfs:type rdfs:Class ;
  rdfs:label "Assistent"@de , "Assistant Staff"@en ;
  rdfs:comment "Ein wissenschaftlicher oder nicht-wissenschaftlicher Mitarbeiter, angestellt bei der Universität."@de ;
  rdfs:subClassOf :Angestellter . # Ein Assistent ist ein Angestellter
```

Bedeutung der Hierarchie und Inferenz mit `rdf:type`:

Die `rdfs:subClassOf`-Beziehung ist besonders mächtig in Kombination mit `rdf:type`-Aussagen über **Instanzen** (konkrete Individuen). Aus der Klassenhierarchie können RDFS-fähige Systeme (Reasoner) neues Wissen ableiten.

RDFS-Inferenzregel (Beispiel): Wenn eine Ressource `x` vom Typ `C1` ist (d.h. `x rdf:type C1 .`) UND die Klasse `C1` eine Unterklasse von `C2` ist (d.h. `C1 rdfs:subClassOf C2 .`), DANN kann logisch geschlossen werden, dass `x` auch vom Typ `C2` ist (d.h. `x rdf:type C2 .`). Diese Regel ist transitiv: Wenn `C2 rdfs:subClassOf C3 .`, dann ist `x` auch vom Typ `C3`.

Konkret für unser neues Beispiel:

Wenn wir später eine Instanz `:mayaMuster` als Assistentin deklarieren:

```
:mayaMuster a :Assistent .
```

Und da wir definiert haben:

```
:Assistent rdfs:subClassOf :Angestellter .
```

```
:Angestellter rdfs:subClassOf :Person .
```

Wird ein RDFS-Reasoner automatisch schlussfolgern (inferieren):

1. `:mayaMuster a :Angestellter .` (weil Assistent eine Unterklasse von Angestellter ist)
2. `:mayaMuster a :Person .` (weil Angestellter eine Unterklasse von Person ist, und somit auch Assistent via Transitivität eine Unterklasse von Person ist)

Obwohl wir nie explizit gesagt haben, dass Maya Muster eine Angestellte oder eine Person ist, wird dies aus der Klassenhierarchie und ihrer Zugehörigkeit zur Klasse `:Assistent` abgeleitet. Allerdings - um die Ableitungen durchzuführen brauchen wir eine Inferenzmaschine!
(Hinweis: dies geht über den Stoff der Einführungsveranstaltung hinaus)

Was sehen wir im Graphen?

- Die Visualisierung zeigt die Klassen `:Person`, `:Angestellter`, `:Professor`, `:Assistent` und `:Student`.
- Pfeile mit der Beschriftung `rdfs:subClassOf` zeigen von den Unterklassen zu ihren jeweiligen Oberklassen (z.B. von `:Assistent` zu `:Angestellter` und von `:Angestellter` zu `:Person`).
- Um die Graphen nicht zu gross werden zu lassen aber wir die anderen Information weggelassen.

```
In [5]: %%rdf turtle -l uni1
@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <> . # Leerer Präfix für lokale Ressourcen relativ zur @base IRI

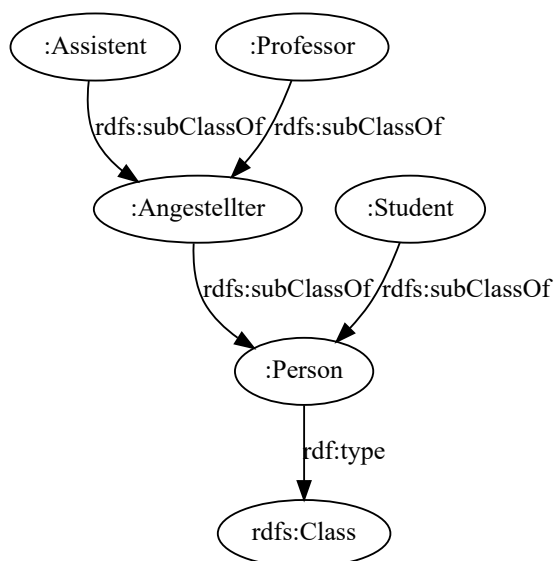
:Person rdf:type rdfs:Class.

# Klasse :Student als Unterklasse von :Person (Definition aus 3.1, hier mit subClassOf ergänzt)
:Student
    rdfs:subClassOf :Person .

# Neue Oberklasse für Angestellte der Universität
:Angestellter
    rdfs:subClassOf :Person . # Ein Angestellter ist auch eine Person

# :Professor als Unterklasse von :Angestellter (Definition aus 3.1 angepasst)
:Professor
    rdfs:subClassOf :Angestellter . # Ein Professor ist ein Angestellter

# Neue Klasse :Assistent als Unterklasse von :Angestellter
:Assistent
    rdfs:subClassOf :Angestellter . # Ein Assistent ist ein Angestellter
```



Schritt 3: Eigenschaften (Properties) definieren mit `rdf:Property`

Nachdem wir nun die grundlegenden Typen unserer Ressourcen (Klassen wie `:Student`, `:Professor`, `:Vorlesung`, `:Person`, `:Angestellter`, `:Assistent`) in unserem Schema definiert haben, benötigen wir als Nächstes die Mittel, um die **Beziehungen zwischen diesen Ressourcen** sowie die **Attribute von Ressourcen** zu beschreiben. Denken Sie an die Prädikate in unseren Tripeln – RDFS erlaubt uns, diese Prädikate explizit als Eigenschaften zu definieren und zu beschreiben.

Konzept: Eine **Eigenschaft** (Property) in RDF repräsentiert eine spezifische Art von Beziehung zwischen einem Subjekt und einem Objekt oder ein Attribut eines Subjekts. In RDFS wird jede Eigenschaft, die wir in unserem Vokabular definieren wollen, als eine Instanz der Klasse `rdf:Property` deklariert. Wie Klassen sind auch Eigenschaften Ressourcen und werden durch IRIs identifiziert. Sie sind die "Verben" (die Beziehungen ausdrücken) oder die "Beschreibungsmerkmale" (die Attribute festlegen) in unseren RDF-Aussagen.

Für jede neu definierte Eigenschaft können wir – analog zu den Klassen – beschreibende Informationen wie `rdfs:label` (ein menschenlesbarer Name) und `rdfs:comment` (eine ausführlichere Beschreibung) angeben. Noch wichtiger für die Strukturierung unseres Vokabulars ist, dass wir mit `rdfs:domain` (welche Art von Subjekt darf diese Eigenschaft haben?) und `rdfs:range` (welche Art von Objekt/Wert erwartet diese Eigenschaft?) festlegen können, wie die Eigenschaft typischerweise verwendet wird. Diese Konzepte `rdfs:domain` und `rdfs:range` werden wir in den folgenden Abschnitten (3.4 und 3.5) genauer betrachten.

Zuerst definieren wir nun einige grundlegende Eigenschaften, die wir für unser Universitätsbeispiel benötigen.

Turtle: (Annahme: Die `@base`-Direktive und die Präfixe `rdfs:`, `rdf:` aus den vorherigen Code-Blöcken sind weiterhin gültig. Wir verwenden hier `@prefix : <>`, was bedeutet, dass z.B. `:name` zu `<http://data.rwth-aachen.de/resources/name>` expandiert. Falls Sie für Ihr Vokabular eine #-Separation bevorzugen, z.B. `<.../resources#name>`, müssten Sie `@prefix : <#>` verwenden.)

```
@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <> . # Leerer Präfix für Lokale Ressourcen/Properties relativ zur @base IRI

# Einfache Eigenschaften (Attribute)
:name rdf:type rdf:Property ;
      rdfs:label "Name"@de , "Name"@en ;
      rdfs:comment "Der Name einer Person oder einer anderen Entität."@de .

:alter rdf:type rdf:Property ;
      rdfs:label "Alter"@de , "Age"@en ;
      rdfs:comment "Das Alter einer Person in Jahren."@de .

# Beziehungseigenschaften
:hört rdf:type rdf:Property ;
      rdfs:label "hört Vorlesung"@de , "attends Lecture"@en ;
      rdfs:comment "Beziehung zwischen einem Studenten und einer Vorlesung, die er/sie hört."@de .

:gelesenVon rdf:type rdf:Property ;
      rdfs:label "Vorlesung gelesen von"@de , "Lecture taught by"@en ;
      rdfs:comment "rd:comment \"Beziehung zwischen einer Vorlesung und einem Professor, von ihm/ihr gelehrt.\"@de .

:matrNr rdf:type rdf:Property ;
      rdfs:label "Matrikelnummer"@de ;
      rdfs:comment "Die eindeutige Identifikationsnummer eines Studenten."@de .
```

Erklärung der verwendeten RDFS-Konstrukte:

- `:name rdf:type rdf:Property` : Diese Aussage deklariert `:name` (was zu `<http://data.rwth-aachen.de/resources/name>` expandiert wird) als eine neue Eigenschaft. Dasselbe gilt für `:alter`, `:hört` usw. Wie die Klassen sind auch die Eigenschaften hier Subjekte von Aussagen, die sie beschreiben.
- `rdfs:label` : Weist der Eigenschaft einen oder mehrere menschenlesbare Namen zu.
- `rdfs:comment` : Fügt eine menschenlesbare Beschreibung oder Erläuterung für die Eigenschaft hinzu.
- Wir haben hier eine Unterscheidung zwischen "einfachen Eigenschaften" (die eher Attribute wie Name oder Alter darstellen) und "Beziehungseigenschaften" (die zwei Ressourcen miteinander verbinden, wie `:hört`) vorgenommen. Technisch gesehen sind in RDFS zunächst alle Instanzen von `rdf:Property`. Die genauere Spezifizierung ihres Verwendungszwecks erfolgt dann über `rdfs:domain` und `rdfs:range`.

Was sehen wir im Graphen? Wenn Sie die obige `%%rdf turtle`-Zelle ausführen, visualisiert das Tool die neu definierten Eigenschaften (z.B. `:name`, `:hört`) als Knoten im Graphen. Von jedem dieser Eigenschafts-Knoten gehen Kanten (Prädikate) wie `rdf:type`, `rdfs:label` und `rdfs:comment` zu ihren jeweiligen Objekten (`rdf:Property` bzw. den Literalen für Label und Kommentar).

Wichtig ist auch hier: Wir haben bisher nur die *Eigenschaften selbst* definiert – also die Arten von Beziehungen oder Attributen, die es in unserem Modell geben soll. Wir haben noch keine konkreten Tripel erstellt, die diese Eigenschaften verwenden, um z.B. einem Studenten einen Namen zuzuordnen (`:peterParker :name "Peter Parker"`) oder um auszudrücken, dass ein Student eine Vorlesung hört. Das Anwenden dieser Eigenschaften auf Instanzen und die Definition ihrer Gültigkeitsbereiche (`rdfs:domain` und `rdfs:range`) sind die nächsten Schritte. Der Graph unten wurde stark vereinfacht um ihn übersichtlicher zu machen.

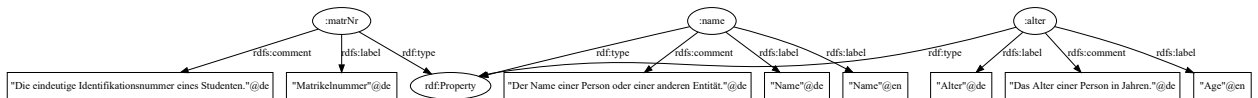
In [6]:

```
%%rdf turtle
@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <> . # Leerer Präfix für lokale Ressourcen/Properties relativ zur @base IRI

# Einfache Eigenschaften (Attribute)
:name rdf:type rdf:Property ;
  rdfs:label "Name"@de , "Name"@en ;
  rdfs:comment "Der Name einer Person oder einer anderen Entität."@de .

:alter rdf:type rdf:Property ;
  rdfs:label "Alter"@de , "Age"@en ;
  rdfs:comment "Das Alter einer Person in Jahren."@de .

:matrNr rdf:type rdf:Property ;
  rdfs:label "Matrikelnummer"@de ;
  rdfs:comment "Die eindeutige Identifikationsnummer eines Studenten."@de .
```



Schritt 4: Gültigkeitsbereiche für Eigenschaften mit `rdfs:domain` und `rdfs:range`

Nachdem wir Klassen (Typen von Ressourcen) und Eigenschaften (Arten von Beziehungen/Attributen) definiert haben, wollen wir nun festlegen, wie diese zueinander in Beziehung stehen. Dafür nutzen wir `rdfs:domain` und `rdfs:range`.

Konzept:

- **`rdfs:domain` (Definitionsbereich):** Gibt an, zu welcher Klasse (oder welchen Klassen) das **Subjekt** eines Tripels gehören muss, wenn diese Eigenschaft verwendet wird. Wichtig: In RDFS ist dies primär eine Regel für **Inferenzen**. Wenn eine Instanz `x` die Eigenschaft `p` hat (also `x p y` existiert) und für `p` ein `rdfs:domain C` definiert ist, dann kann ein RDFS-Reasoner schlussfolgern, dass `x rdf:type C` gilt, auch wenn dies nicht explizit im Graphen steht. Es ist *keine strikte Constraint-Regel*, die die Verwendung der Eigenschaft verhindert, falls der Typ nicht explizit deklariert wurde.
- **`rdfs:range` (Wertebereich):** Gibt an, von welchem Typ das **Objekt** (der Wert) eines Tripels sein muss, wenn diese Eigenschaft verwendet wird. Dies kann eine Klasse (für IRI- oder Blank-Node-Objekte) oder ein Datentyp (für Literale) sein. Auch hier gilt: Es ist primär eine Regel für **Inferenzen**. Wenn `x p y` existiert und für `p` ein `rdfs:range D` definiert ist, dann kann ein RDFS-Reasoner schlussfolgern, dass `y rdf:type D` (falls `D` eine Klasse ist) oder dass das Literal `y` den Datentyp `D` hat (falls `D` ein Datentyp ist).

Turtle-Code zum Definieren von Domain und Range für unsere Eigenschaften:

Wir erweitern nun die Definitionen unserer zuvor erstellten Eigenschaften (`:name`, `:alter`, `:hört`, `:gelesenVon`, `:matrNr`) um `rdfs:domain` und `rdfs:range`.

(Annahme: Die `@base`-Direktive und die Präfixe `rdfs:`, `rdf:`, `xsd:` und `:` aus den vorherigen Code-Blöcken sind weiterhin gültig.)

```
@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <#> . # Annahme: Wir verwenden # für Vokabularbegriffe innerhalb der Base-IRI

# Definitionen der Klassen (aus 3.1 und 3.2, hier für Kontext wiederholt)
:Person rdf:type rdfs:Class ; rdfs:label "Person"@de .
:Student rdf:type rdfs:Class ; rdfs:subClassOf :Person ; rdfs:label "Student"@de .
:Professor rdf:type rdfs:Class ; rdfs:subClassOf :Angestellter ; rdfs:label "Professor"@de . # Angenommen
:Angestellter ist auch definiert
:Angestellter rdf:type rdfs:Class ; rdfs:subClassOf :Person ; rdfs:label "Angestellter"@de .
:Vorlesung rdf:type rdfs:Class ; rdfs:label "Vorlesung"@de .

# Erweiterung/Definition der Eigenschaften mit Domain und Range

:name rdf:type rdf:Property ; # Gegebenenfalls Definition vervollständigen
  rdfs:label "Name"@de ;
  rdfs:comment "Der Name einer Entität."@de ;
  rdfs:domain :Person ; # Namen sind hier für Personen definiert
  rdfs:range xsd:string . # Der Wert eines Namens ist ein String

:alter rdf:type rdf:Property ; # Gegebenenfalls Definition vervollständigen
  rdfs:label "Alter"@de ;
  rdfs:comment "Das Alter einer Person in Jahren."@de ;
  rdfs:domain :Person ; # Alter ist für Personen relevant
  rdfs:range xsd:integer . # Alter ist eine Ganzzahl

:matrNr rdf:type rdf:Property ; # Gegebenenfalls Definition vervollständigen
```

```

    rdfs:label "Matrikelnummer"@de ;
    rdfs:comment "Die eindeutige Identifikationsnummer eines Studenten."@de ;
    rdfs:domain :Student ;          # Matrikelnummern haben Studenten
    rdfs:range xsd:string .         # Oft als String gespeichert (kann führende Nullen haben)

:rang rdf:type rdf:Property ; # Neue Eigenschaft für den Professorenrang
    rdfs:label "hat Rang"@de ;
    rdfs:comment "Der akademische oder administrative Rang eines Professors."@de ;
    rdfs:domain :Professor ;        # Professoren haben einen Rang
    rdfs:range xsd:string .         # z.B. "W3", "C4"

:hört rdf:type rdf:Property ; # Gegebenenfalls Definition vervollständigen
    rdfs:label "hört Vorlesung"@de ;
    rdfs:comment "Beziehung zwischen einem Studenten und einer Vorlesung, die er/sie hört."@de ;
    rdfs:domain :Student ;          # Studenten hören Vorlesungen
    rdfs:range :Vorlesung .         # Das Objekt ist eine Vorlesung

:gelesenVon rdf:type rdf:Property ; # Gegebenenfalls Definition vervollständigen
    rdfs:label "lehrt Vorlesung"@de ;
    rdfs:comment "Beziehung zwischen einem Professor und einer Vorlesung, die er/sie lehrt."@de ;
    rdfs:range :Professor ;         # Professoren Lehren Vorlesungen
    rdfs:domain :Vorlesung .        # Vorlesungen werden gelesen

```

Inferenz durch rdfs:domain und rdfs:range:

Fügen wir nun einige Instanzdaten hinzu und schauen, was daraus geschlossen werden kann, um die Auswirkungen von `rdfs:domain` und `rdfs:range` zu sehen:

```

@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <#> .

# Schema-Definitionen (gekürzt, nur die für das Inferenzbeispiel relevanten)
:Person rdf:type rdfs:Class .
:Student rdf:type rdfs:Class ; rdfs:subClassOf :Person .
:Professor rdf:type rdfs:Class ; rdfs:subClassOf :Person . # Angenommen Professor ist direkt SubClassOf Person
für dieses Bsp.
:Vorlesung rdf:type rdfs:Class .

:matrNr rdfs:domain :Student ; rdfs:range xsd:string .
:hört rdfs:domain :Student ; rdfs:range :Vorlesung .
:gelesenVon rdfs:domain :Vorlesung ; rdfs:range :Professor Vorlesung .

# Instanzdaten
:s12345 :matrNr "12345" .          # Student Peter
:s12345 :hört :v5678 .             # Peter hört Vorlesung Semantic Web

:v5678 :gelesenVon :p987 .         # Professor Müller lehrt Semantic Web
:v5678 rdfs:label "Semantic Web Vorlesung"@de .

```

Beobachtungen und abgeleitete Fakten (Inferenzen):

Eine Inferenzmaschine kann die folgenden Tripel ableiten:

- Aus `:s12345 :matrNr "12345" .` und `:matrNr rdfs:domain :Student .` wird inferiert:
 - `:s12345 rdf:type :Student .`
- Aus `:s12345 :hört :v5678 .` und `:hört rdfs:domain :Student .` wird ebenfalls (redundant, aber korrekt) inferiert:
 - `:s12345 rdf:type :Student .`
- Aus `:s12345 :hört :v5678 .` und `:hört rdfs:range :Vorlesung .` wird inferiert:
 - `:v5678 rdf:type :Vorlesung .`
- Aus `:v5678 :gelesenVon :p987 .` und `:gelesenVon rdfs:domain :Vorlesung .` wird inferiert:
 - `:v5678 rdf:type :Vorlesung .`
- Aus `:v5678 :gelesenVon :p987 .` und `:gelesenVon rdfs:range :Professor .` wird ebenfalls inferiert:
 - `:p987 rdf:type :Professor .`

Zusätzlich werden wegen `rdfs:subClassOf` (z.B. `:Student rdfs:subClassOf :Person .`) weitere Typisierungen abgeleitet:

- `:s12345 rdf:type :Person .`
- `:p987 rdf:type :Person .`

Diese Inferenzen zeigen, wie RDFS dazu beiträgt, implizites Wissen aus den definierten Schema-Regeln und den Instanzdaten explizit zu machen. Es validiert nicht strikt, sondern reichert die Daten an. Der Graph unten wurde stark vereinfacht um ihn übersichtlicher zu machen.

In [7]:

```
%%rdf turtle
@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <#> . # Annahme: Wir verwenden # für Vokabularbegriffe innerhalb der Base-IRI

# Definitionen der Klassen (aus 3.1 und 3.2, hier für Kontext wiederholt)

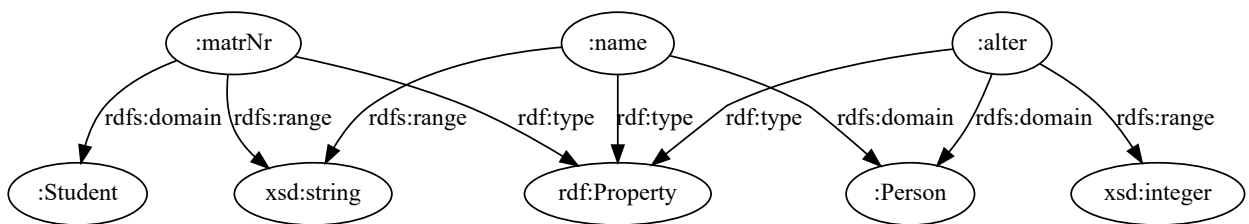
#:Student rdfs:subClassOf :Person .
#:Professor rdfs:subClassOf :Angestellter .
#:Angestellter rdfs:subClassOf :Person .

# Erweiterung/Definition der Eigenschaften mit Domain und Range

:name rdf:type rdf:Property ; # Gegebenenfalls Definition vervollständigen
    rdfs:domain :Person ; # Namen sind hier für Personen definiert
    rdfs:range xsd:string . # Der Wert eines Namens ist ein String

:alter rdf:type rdf:Property ; # Gegebenenfalls Definition vervollständigen
    rdfs:domain :Person ; # Alter ist für Personen relevant
    rdfs:range xsd:integer . # Alter ist eine Ganzzahl

:matrNr rdf:type rdf:Property ; # Gegebenenfalls Definition vervollständigen
    rdfs:domain :Student ; # Matrikelnummern haben Studenten
    rdfs:range xsd:string . # Oft als String gespeichert (kann führende Nullen haben)
```



Schritt 5: Eigenschafts-Hierarchien mit `rdfs:subPropertyOf`

Konzept: Analog zu Klassenhierarchien (`rdfs:subClassOf`) können auch Eigenschaften (Properties) hierarchisch organisiert werden. Die Beziehung `rdfs:subPropertyOf` bedeutet, dass eine Eigenschaft eine **speziellere Form** einer anderen, allgemeineren Eigenschaft ist. Wenn eine Eigenschaft `p1` eine Untereigenschaft von `p2` ist (`p1 rdfs:subPropertyOf p2`), dann impliziert jede Aussage `x p1 y` auch die Aussage `x p2 y` .

Dies ist nützlich, um feinere Unterscheidungen bei Beziehungen oder Attributen zu treffen, während gleichzeitig die allgemeinere Beziehung erhalten bleibt und für Abfragen oder Inferenzen genutzt werden kann.

Turtle-Code zum Definieren von Eigenschafts-Hierarchien:

Wir definieren zunächst eine allgemeine Eigenschaft `:hatKontaktinformation` und spezialisieren diese dann zu `:hatEmail` und `:hatTelefon` . Ebenso definieren wir eine allgemeine Verantwortlichkeit eines Professors für eine Lehrveranstaltung und spezialisieren `:gelesenVon` (aus 3.3) dazu.

(Annahme: Die `@base` -Direktive und die Präfixe `rdfs:` , `rdf:` , `xsd:` und `:` aus den vorherigen Code-Blöcken sind weiterhin gültig.)

```
@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <#> .

# Definition der Klassen (zur Erinnerung, relevant für Domain/Range)
:Person rdf:type rdfs:Class ; rdfs:label "Person"@de .
:Professor rdf:type rdfs:Class ; rdfs:subClassOf :Person ; rdfs:label "Professor"@de . # Angenommen
:Angestellter ist auch definiert
:Vorlesung rdf:type rdfs:Class ; rdfs:label "Vorlesung"@de .

# Allgemeine Kontakteigenschaft
:hatKontaktinformation rdf:type rdf:Property ;
    rdfs:label "hat Kontaktinformation"@de ;
    rdfs:comment "Eine allgemeine Eigenschaft, um eine Kontaktinformation für eine Person
anzugeben."@de ;
    rdfs:domain :Person . # Kontaktinformationen sind für Personen

# Spezialisierte Kontakteigenschaften
:hatEmail rdf:type rdf:Property ;
    rdfs:label "hat E-Mail-Adresse"@de ;
    rdfs:comment "Die E-Mail-Adresse einer Person."@de ;
    rdfs:subPropertyOf :hatKontaktinformation ; # :hatEmail ist eine spezielle Art von
:hatKontaktinformation
```

```

    rdfs:domain :Person ; # Domain wird oft von der Super-Property geerbt, kann aber auch spezifiziert
werden
    rdfs:range xsd:string . # E-Mail-Adressen sind typischerweise Strings

:hatTelefon rdf:type rdf:Property ;
    rdfs:label "hat Telefonnummer"@de ;
    rdfs:comment "Die Telefonnummer einer Person."@de ;
    rdfs:subPropertyOf :hatKontaktinformation ; # :hatTelefon ist auch eine spezielle Art von
:hatKontaktinformation
    rdfs:domain :Person ;
    rdfs:range xsd:string . # Telefonnummern werden oft als Strings gespeichert

# Allgemeine Verantwortlichkeit für Lehraktivitäten
:verantwortetVon rdf:type rdf:Property ;
    rdfs:label "ist verantwortlich für Lehrinhalt"@de ;
    rdfs:comment "Eine Professorin oder ein Professor ist für den Inhalt einer Vorlesung
verantwortlich."@de ;
    rdfs:domain :Vorlesung ;
    rdfs:range :Professor .

# :gelesenVon (aus 3.3) als Untereigenschaft
:gelesenVon rdf:type rdf:Property ; # Sicherstellen, dass es als Property definiert ist
    rdfs:label "lehrt Vorlesung"@de ; # Label aus 3.3
    rdfs:comment "Beziehung zwischen einer Vorlesung und einem Professor, von ihm/ihr gelehrt."@de ; #
Kommentar aus 3.3
    rdfs:domain :Vorlesung ; # Domain aus 3.4
    rdfs:range :Professor ; # Range aus 3.4
    rdfs:subPropertyOf :istVerantwortlichFuerLehrinhalt .
    # Die spezifische Tätigkeit des Lehrens impliziert die allgemeinere Verantwortung.

```

Bedeutung der Eigenschafts-Hierarchie und Inferenz:

Ähnlich wie bei `rdfs:subClassOf` ermöglicht `rdfs:subPropertyOf` Inferenzen:

RDFS-Inferenzregel (Beispiel): Wenn eine Eigenschaft `p1` eine Untereigenschaft von `p2` ist (d.h. `p1 rdfs:subPropertyOf p2 .`) UND ein Tripel `x p1 y .` existiert, DANN kann logisch geschlossen werden, dass auch das Tripel `x p2 y .` gilt.

Konkret für unser Beispiel: Wenn wir später eine Instanz definieren: `:annaSchlau :hatEmail "anna@example.com" .` Und da wir definiert haben: `:hatEmail rdfs:subPropertyOf :hatKontaktinformation .` Wird ein RDFS-Reasoner automatisch schlussfolgern (inferieren): `:annaSchlau :hatKontaktinformation "anna@example.com" .`

Obwohl wir nie explizit gesagt haben, dass Anna Schlau eine allgemeine Kontaktinformation "anna@example.com" hat, wird dies aus der Eigenschafts-Hierarchie und der spezifischen `hatEmail`-Aussage abgeleitet.

Was sehen wir im Graphen?

- Die Visualisierung zeigt die Eigenschaften und ihre `rdfs:subPropertyOf`-Beziehungen.

In [8]:

```
%%rdf turtle
@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <#> . # Annahme: Wir verwenden # für Vokabularbegriffe innerhalb der Base-IRI

# Allgemeine Kontakteigenschaft
:hatKontaktinformation rdf:type rdf:Property ;
    rdfs:domain :Person . # Kontaktinformationen sind für Personen

# Spezialisierte Kontakteigenschaften
:hatEmail rdf:type rdf:Property ;
    rdfs:subPropertyOf :hatKontaktinformation ; # :hatEmail ist eine spezielle Art von :hatKontaktinformation
    rdfs:domain :Person ; # Domain wird oft von der Super-Property geerbt, kann aber auch spezifiziert werden
    rdfs:range xsd:string . # E-Mail-Adressen sind typischerweise Strings

:hatTelefon rdf:type rdf:Property ;
    rdfs:subPropertyOf :hatKontaktinformation ; # :hatTelefon ist auch eine spezielle Art von :hatKontaktinformation
    rdfs:domain :Person ;
    rdfs:range xsd:string . # Telefonnummern werden oft als Strings gespeichert

# Allgemeine Verantwortlichkeit für Lehraktivitäten
:verantwortetVon rdf:type rdf:Property ;
    rdfs:label "ist verantwortlich für Lehrinhalt"@de ;
    rdfs:comment "Eine Professorin oder ein Professor ist für den Inhalt einer Vorlesung verantwortlich."@de ;
    rdfs:domain :Vorlesung ;
    rdfs:range :Professor .

# :gelesenVon (aus 3.3) als Untereigenschaft
:gelesenVon rdf:type rdf:Property ; # Sicherstellen, dass es als Property definiert ist
    rdfs:label "lehrt Vorlesung"@de ; # Label aus 3.3
    rdfs:comment "Beziehung zwischen einer Vorlesung und einem Professor, von ihm/ihr gelehrt."@de ;
    rdfs:domain :Vorlesung ; # Domain aus 3.4
    rdfs:range :Professor ; # Range aus 3.4
    rdfs:subPropertyOf :verantwortetVon .
    # Die spezifische Tätigkeit des Lehrens impliziert die allgemeinere Verantwortung.
```



Schritt 6: Das bisherige RDFS-Schema für die Universität im Überblick

In den vorherigen Schritten haben wir verschiedene RDFS-Konstrukte kennengelernt und auf unser Universitätsbeispiel angewendet:

- Wir haben Klassen wie `:Person`, `:Student`, `:Professor` und `:Vorlesung` definiert (Schritt 3.1).
- Wir haben Klassenhierarchien mit `rdfs:subClassOf` erstellt (Schritt 3.2).
- Wir haben Eigenschaften (Properties) wie `:name`, `:matrNr`, `:hört` und `:lehrt` mit `rdf:Property` definiert (Schritt 3.3).
- Wir haben die Gültigkeitsbereiche dieser Eigenschaften mit `rdfs:domain` und `rdfs:range` spezifiziert (Schritt 3.4).
- Wir haben Eigenschafts-Hierarchien mit `rdfs:subPropertyOf` kennengelernt (Schritt 3.5).*

Der folgende Turtle-Code fasst nun unser bisher entwickeltes RDFS-Schema für die Universitätsdaten zusammen. Dieses Schema dient als Vokabular, um später konkrete Instanzdaten (z.B. über einzelne Studenten, Professoren und deren Beziehungen) strukturiert und semantisch eindeutig beschreiben zu können.

Turtle-Code für das gesammelte RDFS-Schema: (Annahme: Die `@base`-Direktive und die Präfixe `rdfs:`, `rdf:`, `xsd:` aus den vorherigen Code-Blöcken sind weiterhin gültig. Wir verwenden hier `@prefix : <>` oder alternativ `@prefix : <#>` je nach gewählter Konvention für Ihr Vokabular.)

```
@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <#> . # Verwendung von '#' für Vokabularbegriffe innerhalb der Base-IRI

# === KLASSEDEFINITIONEN UND -HIERARCHIE ===
# (Kombination aus Schritt 3.1 und 3.2)

:Person rdf:type rdfs:Class ;
    rdfs:label "Person"@de, "Person"@en ;
    rdfs:comment "Ein menschliches Wesen."@de .

:Student rdf:type rdfs:Class ;
    rdfs:label "Student"@de, "Student"@en ;
    rdfs:comment "Eine Person, die an einer Hochschule eingeschrieben ist und studiert."@de ;
    rdfs:subClassOf :Person .
```

```

:Professor rdf:type rdfs:Class ;
  rdfs:label "Professor"@de, "Professor"@en ;
  rdfs:comment "Eine Lehrkraft mit akademischer Professur an einer Hochschule."@de ;
  rdfs:subClassOf :Person . # In diesem vereinfachten Schema direkt Unterklasse von Person
                             # (Alternativ: :Angestellter -> :Person, und :Professor -> :Angestellter)

:Vorlesung rdf:type rdfs:Class ;
  rdfs:label "Vorlesung"@de, "Lecture"@en ;
  rdfs:comment "Eine spezifische Lehrveranstaltung, die von Dozierenden gehalten wird."@de .

# === EIGENSCHAFTEN (PROPERTIES) MIT DOMAIN UND RANGE ===
# (Kombination aus Schritt 3.3 und 3.4)

:name rdf:type rdf:Property ;
  rdfs:label "Name"@de, "Name"@en ;
  rdfs:comment "Der Name einer Person oder einer anderen Entität."@de ;
  rdfs:domain :Person ;      # Gilt für alle Personen
  rdfs:range xsd:string .

:matrNr rdf:type rdf:Property ;
  rdfs:label "Matrikelnummer"@de ;
  rdfs:comment "Die eindeutige Identifikationsnummer eines Studenten."@de ;
  rdfs:domain :Student ;     # Spezifisch für Studenten
  rdfs:range xsd:string .   # Matrikelnummern sind oft alphanumerisch

:semester rdf:type rdf:Property ; # Umbenannt von :semester für Klarheit
  rdfs:label "Anzahl Semester"@de ;
  rdfs:comment "Die Anzahl der bisher studierten Fachsemester eines Studenten."@de ;
  rdfs:domain :Student ;
  rdfs:range xsd:integer .

:persNr rdf:type rdf:Property ;
  rdfs:label "Personalnummer"@de ;
  rdfs:comment "Die eindeutige Personalnummer eines Professors (oder Angestellten)."@de ;
  rdfs:domain :Professor ;   # Hier für Professoren, könnte allgemeiner sein für :Angestellter
  rdfs:range xsd:string .   # Personalnummern oft als String

:hatRang rdf:type rdf:Property ; # Umbenannt von :rang für Klarheit
  rdfs:label "hat Rang"@de ;
  rdfs:comment "Der akademische oder administrative Rang eines Professors (z.B. W3, C4)."@de ;
  rdfs:domain :Professor ;
  rdfs:range xsd:string .

:titel rdf:type rdf:Property ;
  rdfs:label "Titel der Vorlesung"@de ;
  rdfs:comment "Der offizielle Titel einer Lehrveranstaltung."@de ;
  rdfs:domain :Vorlesung ;
  rdfs:range xsd:string .

:sws rdf:type rdf:Property ;
  rdfs:label "SWS"@de, "Semesterwochenstunden"@de ;
  rdfs:comment "Die Anzahl der Semesterwochenstunden für eine Vorlesung."@de ;
  rdfs:domain :Vorlesung ;
  rdfs:range xsd:integer .

:hört rdf:type rdf:Property ; # Präziser Name
  rdfs:label "hört Vorlesung"@de ;
  rdfs:comment "Beziehung: Ein Student hört eine Vorlesung."@de ;
  rdfs:domain :Student ;
  rdfs:range :Vorlesung .

:gelesenVon rdf:type rdf:Property ; # Präziser Name
  rdfs:label "lehrt Vorlesung"@de ;
  rdfs:comment "Beziehung: Eine Vorlesung wird von einem Professor gelehrt."@de ;
  rdfs:range :Professor ;
  rdfs:domain :Vorlesung .

```

Erläuterung und Bedeutung des zusammengesetzten Schemas:

Dieses Turtle-Dokument definiert nun ein erstes, kohärentes Vokabular (Schema) für unser Universitätsbeispiel:

- Es legt fest, welche **Arten von Dingen** (Klassen wie `:Student`, `:Professor`, `:Vorlesung`) es in unserer Domäne gibt und wie diese in einer **Hierarchie** zueinander stehen (z.B. jeder `:Student` ist auch eine `:Person`).
- Es definiert, welche **Arten von Eigenschaften oder Beziehungen** (Properties wie `:name`, `:hört`) es gibt.
- Für jede Eigenschaft wird durch `rdfs:domain` und `rdfs:range` spezifiziert, **zwischen welchen Klassen bzw. Datentypen** diese typischerweise verwendet wird.
- Alle Klassen und Eigenschaften sind mit menschenlesbaren `rdfs:label` und `rdfs:comment` versehen, was die Dokumentation und das Verständnis des Schemas verbessert.

Wie geht es weiter? Mit diesem RDFS-Schema haben wir eine solide Grundlage geschaffen.

1. Wir können nun **Instanzdaten** erstellen, die dieses Vokabular verwenden (z.B. konkrete Studenten, Professoren und die Vorlesungen, die sie hören oder lehren).
2. Wenn wir diese Instanzdaten zusammen mit dem Schema in ein RDFS-fähiges System (Reasoner) laden, kann das System **neue Informationen ableiten** (Inferenzen ziehen), basierend auf den `rdfs:subClassOf` -, `rdfs:domain` - und `rdfs:range` -Axiomen. Zum Beispiel: Wenn `:maxMustermann` `:hört` `:webTechVL` . gilt, und `:hört` als `rdfs:domain` `:Student` hat, dann weiß das System, dass `:maxMustermann` ein `:Student` ist, auch wenn wir das nicht explizit gesagt haben.
3. Dieses Schema ist noch relativ einfach. Für komplexere Anforderungen (z.B. Kardinalitätsbeschränkungen, Disjunktheit von Klassen) bräuchten wir ausdrucksstärkere Sprachen wie OWL, die auf RDFS aufbauen.

Dieser Schritt des Zusammensetzens hilft, das Gesamtbild des Vokabulars zu sehen und zu überprüfen, ob die Definitionen zueinander passen.

In [9]:

```
%%rdf turtle
@base <http://data.rwth-aachen.de/resources/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <#> . # Verwendung von '#' für Vokabularbegriffe innerhalb der Base-IRI

# === KLASSENDEFINITIONEN UND -HIERARCHIE ===
# (Kombination aus Schritt 3.1 und 3.2)

:Person rdf:type rdfs:Class ;
  rdfs:label "Person"@de, "Person"@en ;
  rdfs:comment "Ein menschliches Wesen."@de .

:Student rdf:type rdfs:Class ;
  rdfs:label "Student"@de, "Student"@en ;
  rdfs:comment "Eine Person, die an einer Hochschule eingeschrieben ist und studiert."@de ;
  rdfs:subClassOf :Person .

:Professor rdf:type rdfs:Class ;
  rdfs:label "Professor"@de, "Professor"@en ;
  rdfs:comment "Eine Lehrkraft mit akademischer Professur an einer Hochschule."@de ;
  rdfs:subClassOf :Person . # In diesem vereinfachten Schema direkt Unterklasse von Person
                           # (Alternativ: :Angestellter -> :Person, und :Professor -> :Angestellter)

:Vorlesung rdf:type rdfs:Class ;
  rdfs:label "Vorlesung"@de, "Lecture"@en ;
  rdfs:comment "Eine spezifische Lehrveranstaltung, die von Dozierenden gehalten wird."@de .

# === EIGENSCHAFTEN (PROPERTIES) MIT DOMAIN UND RANGE ===
# (Kombination aus Schritt 3.3 und 3.4)

:name rdf:type rdf:Property ;
  rdfs:label "Name"@de, "Name"@en ;
  rdfs:comment "Der Name einer Person oder einer anderen Entität."@de ;
  rdfs:domain :Person ;      # Gilt für alle Personen
  rdfs:range xsd:string .

:matrNr rdf:type rdf:Property ;
  rdfs:label "Matrikelnummer"@de ;
  rdfs:comment "Die eindeutige Identifikationsnummer eines Studenten."@de ;
  rdfs:domain :Student ;    # Spezifisch für Studenten
  rdfs:range xsd:string .  # Matrikelnummern sind oft alphanumerisch

:semester rdf:type rdf:Property ; # Umbenannt von :semester für Klarheit
  rdfs:label "Anzahl Semester"@de ;
  rdfs:comment "Die Anzahl der bisher studierten Fachsemester eines Studenten."@de ;
  rdfs:domain :Student ;
  rdfs:range xsd:integer .

:persNr rdf:type rdf:Property ;
  rdfs:label "Personalnummer"@de ;
  rdfs:comment "Die eindeutige Personalnummer eines Professors (oder Angestellten)."@de ;
  rdfs:domain :Professor ;  # Hier für Professoren, könnte allgemeiner sein für :Angestellter
  rdfs:range xsd:string .  # Personalnummern oft als String

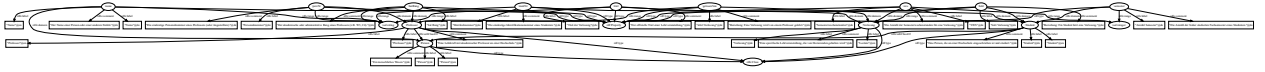
:hatRang rdf:type rdf:Property ; # Umbenannt von :rang für Klarheit
  rdfs:label "hat Rang"@de ;
  rdfs:comment "Der akademische oder administrative Rang eines Professors (z.B. W3, C4)."@de ;
  rdfs:domain :Professor ;
  rdfs:range xsd:string .

:titel rdf:type rdf:Property ;
  rdfs:label "Titel der Vorlesung"@de ;
  rdfs:comment "Der offizielle Titel einer Lehrveranstaltung."@de ;
  rdfs:domain :Vorlesung ;
  rdfs:range xsd:string .

:sws rdf:type rdf:Property ;
  rdfs:label "SWS"@de, "Semesterwochenstunden"@de ;
  rdfs:comment "Die Anzahl der Semesterwochenstunden für eine Vorlesung."@de ;
  rdfs:domain :Vorlesung ;
  rdfs:range xsd:integer .

:hört rdf:type rdf:Property ; # Präziser Name
  rdfs:label "hört Vorlesung"@de ;
  rdfs:comment "Beziehung: Ein Student hört eine Vorlesung."@de ;
  rdfs:domain :Student ;
  rdfs:range :Vorlesung .

:gelesenVon rdf:type rdf:Property ; # Präziser Name
  rdfs:label "lehrt Vorlesung"@de ;
  rdfs:comment "Beziehung: Eine Vorlesung wird von einem Professor gelehrt."@de ;
  rdfs:range :Professor ;
  rdfs:domain :Vorlesung .
```



Schritt 7: Dateninstanzen erstellen und Schema anwenden

Nachdem wir in den vorherigen Schritten unser RDFS-Schema (unser Vokabular mit Klassen, Eigenschaften und deren Beziehungen zueinander) definiert haben, wollen wir es nun anwenden. In diesem Schritt erstellen wir konkrete **Instanzdaten** (auch ABox oder "Assertional Box" genannt), die unser zuvor definiertes Schema (die TBox oder "Terminological Box") nutzen.

Wir werden Instanzen für unsere Klassen `:Student`, `:Professor` und `:Vorlesung` erstellen. Dabei verwenden wir `rdf:type`, um eine Instanz einer Klasse zuzuordnen, und die im Schema definierten Eigenschaften, um diese Instanzen zu beschreiben.

Turtle-Code mit Schema und Instanzdaten:

Die folgende Zelle enthält sowohl unser RDFS-Schema (aus Schritt 3.6) als auch die neu hinzugefügten Instanzdaten. Dies ermöglicht es RDF-Werkzeugen, die Daten im Kontext ihres Schemas zu interpretieren (z.B. um Labels anzuzeigen, auch wenn kein aktives Reasoning für Inferenzen stattfindet).

```
""turtle -I uni_mit_instanzen @base http://data.rwth-aachen.de/resources/ . @prefix rdfs: http://www.w3.org/2000/01/rdf-schema# . @prefix rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns# . @prefix xsd: http://www.w3.org/2001/XMLSchema# . @prefix : <#> . # Oder <> . je nach Ihrer Konvention aus 3.6
```

=== DATENINSTANZEN ===

Student

```
:student28106 rdf:type :Student ; # Explizite Typisierung gemäß Schema :name "Carnap" ; # Verwendet :name mit Domain :Person, Range xsd:string :matrNr 28106 ; # Verwendet :matrNr mit Domain :Student, Range xsd:integer :semester 3 . # Verwendet :semester mit Domain :Student, Range xsd:integer
```

Professor

```
:professor2125 rdf:type :Professor ; # Explizite Typisierung gemäß Schema :name "Sokrates" ; :persNr 2125 ; :rang "W3" .
```

Vorlesung

```
:vorlesung5041 rdf:type :Vorlesung ; # Explizite Typisierung gemäß Schema :titel "Ethik" ; :sWS 4 .
```

Beziehungen, die das Schema nutzen

```
:student28106 :hört :vorlesung5041 . # :hört hat Domain :Student, Range :Vorlesung :vorlesung5041 :gelesenVon :professor2125 . # :lehrt hat Domain :Professor, Range :Vorlesung ""
```

Was zeigt dieser Code?

1. **Schema-Anwendung:** Wir erstellen konkrete Individuen (Instanzen) wie `:student28106`, `:professor2125` und `:vorlesung5041`.
2. **Typisierung von Instanzen:** Mit `rdf:type` weisen wir jeder Instanz eine Klasse aus unserem definierten Schema zu (z.B. `:student28106` ist ein `:Student`).
3. **Verwendung definierter Eigenschaften:** Die Eigenschaften der Instanzen (z.B. `:name`, `:matrNr`, `:hört`) sind ebenfalls in unserem Schema definiert.
4. **Konsistenz mit Domain und Range (implizit):**
 - Wenn wir schreiben `:student28106 :matrNr 28106`, dann passt das zu den Schema-Definitionen `:matrNr rdfs:domain :Student` und `:matrNr rdfs:range xsd:integer`, da `:student28106` als `:Student` deklariert ist und `28106` eine Ganzzahl ist.
 - Ebenso passt `:student28106 :hört :vorlesung5041` zu `:hört rdfs:domain :Student` und `:hört rdfs:range :Vorlesung`, da die Typen der beteiligten Instanzen den Vorgaben entsprechen.

Bedeutung für die Datenqualität und das Verständnis:

Auch ohne die Betrachtung automatischer Inferenzen hilft ein RDFS-Schema dabei:

- **Daten zu strukturieren:** Es gibt einen klaren Rahmen vor, welche Arten von Dingen es gibt und welche Eigenschaften sie haben können.
- **Ein gemeinsames Verständnis zu schaffen:** Alle, die das Schema kennen, verstehen die intendierte Bedeutung der Daten.
- **Konsistenz zu fördern:** Man wird angeleitet, Daten so zu erstellen, dass sie den im Schema definierten Erwartungen (wie Domain und

Range) entsprechen. Obwohl RDFS selbst keine strikte Validierung erzwingt, dient es als wichtige Richtlinie.

Die explizite Typisierung und Verwendung der definierten Eigenschaften machen die Daten semantisch reicher und besser interpretierbar, sowohl für Menschen als auch für Maschinen (selbst wenn diese nur einfache Schema-Awareness haben und keine komplexen Inferenzen durchführen).

Was erreichen wir mit RDFS? Der Mehrwert des Schemas

Um den Nutzen von RDF Schema zu verdeutlichen, vergleichen wir noch einmal den Zustand unserer Daten ohne explizites Schema mit dem Zustand, nachdem wir ein RDFS-Vokabular definiert haben.

Vorher (Beispiel mit "nacktem" RDF aus Abschnitt 1):

```
@turtle
# Annahme: @base <http://data.rwth-aachen.de/resources/> . und @prefix : <> .
# oder alternativ @prefix : <http://data.rwth-aachen.de/resources/> .
# Die Prädikate sind hier ad-hoc gewählte IRIs ohne weitere Definition.
:student28106 :name "Carnap" .
:student28106 :semester 3 .
```

Ohne ein Schema sind dies einfach zwei isolierte Aussagen. Wir wissen nicht sicher:

- Was `:student28106` repräsentiert (eine Person, eine Matrikelnummer, etc.).
- Was `:name` oder `:semester` genau bedeuten, für welche Ressourcen sie gedacht sind oder welche Werte sie haben sollten.

Nachher (mit unserem RDFS-Schema aus Abschnitt 3.6 und beispielhaften Instanzdaten aus 3.7):

Angenommen, wir haben unser RDFS-Schema definiert und folgende Instanzdaten erstellt, die dieses Schema verwenden:

```
# Annahme: Das vollständige Schema aus 3.6 ist geladen/definiert.
# Hier relevante Instanzdaten:
:student28106 rdf:type :Student ;
              :name "Carnap" ;
              :semester 3 .
```

Durch die Kombination der Instanzdaten mit unserem RDFS-Schema gewinnen wir deutlich mehr Klarheit und können implizites Wissen ableiten:

- **Wir wissen explizit:**
 - `:student28106` ist ein `:Student` (dank `rdf:type :Student` .).
- **Aufgrund unseres Schemas wissen wir zusätzlich (semantische Anreicherung):**
 - Die Klasse `:Student` ist eine Unterklasse von `:Person` (definiert durch `:Student rdfs:subClassOf :Person` .).
 - Die Eigenschaft `:name` ist für Ressourcen vom Typ `:Person` vorgesehen und erwartet einen `xsd:string` als Wert (definiert durch `:name rdfs:domain :Person ; rdfs:range xsd:string` .).
 - Die Eigenschaft `:semester` ist für Ressourcen vom Typ `:Student` vorgesehen und erwartet einen `xsd:integer` als Wert (definiert durch `:semester rdfs:domain :Student ; rdfs:range xsd:integer` .).
- **Potenzial für Konsistenzprüfung und Datenqualität:**
 - Wir können nun besser beurteilen, ob unsere Daten den im Schema festgelegten Erwartungen entsprechen. Zum Beispiel passt die Verwendung von `:semester` für `:student28106` gut zu `rdfs:domain :Student` . Die Zuweisung eines nicht-numerischen Wertes für `:semester` würde der `rdfs:range xsd:integer` widersprechen.
 - **Wichtige Anmerkung zur "Validierung" in RDFS:** RDFS selbst erzwingt keine strikten Einschränkungen (Constraints) wie ein Datenbankschema, das falsche Eingaben abweisen würde. Stattdessen dient RDFS primär dazu, **Typen zu inferieren** (automatisch abzuleiten). Wenn z.B. Daten vorhanden sind, die einer `rdfs:domain` - oder `rdfs:range` -Definition widersprechen, führt dies in einem reinen RDFS-Kontext nicht unbedingt zu einem Fehler, sondern eher zu potenziell unerwünschten oder "überraschenden" Inferenzen. Echte Datenvalidierung im Sinne von "richtig/falsch" wird im Semantic Web eher durch Sprachen wie SHACL (Shapes Constraint Language) realisiert. RDFS legt aber die Grundlage, indem es die intendierte Struktur beschreibt.
- **Grundlage für automatische Schlussfolgerungen (Inferenzen):** Selbst wenn wir die Inferenzen hier nicht aktiv mit einem Tool beobachten, definiert unser RDFS-Schema die Regeln, nach denen ein RDFS-Reasoner neues, implizites Wissen ableiten *könnte*. Die wichtigsten RDFS-Inferenzregeln, die hier relevant wären:

1. Ableitung durch `rdfs:subClassOf` :

- Gegeben: `:student28106 rdf:type :Student` . (explizit in unseren Instanzdaten)
- Schema: `:Student rdfs:subClassOf :Person` .
- → Inferenz: `:student28106 rdf:type :Person` . (Jeder Student ist auch eine Person).

2. Ableitung durch `rdfs:domain` :

- Gegeben: `:student28106 :semester 3` . (explizit in unseren Instanzdaten)
- Schema: `:semester rdfs:domain :Student` .
- → Inferenz: `:student28106 rdf:type :Student` . (Wenn jemand ein Semester hat (gemäß unserem Schema), muss er ein Student sein).

3. Ableitung durch `rdfs:domain` (weiteres Beispiel):

- Gegeben: `:student28106 :name "Carnap" .` (explizit)
- Schema: `:name rdfs:domain :Person .`
- → Inferenz: `:student28106 rdf:type :Person .` (Wenn jemand einen Namen hat (gemäß unserem Schema), muss er eine Person sein). *(Diese Inferenz wäre redundant, wenn die erste Inferenz (`rdf:type :Person` via `subClassOf`) bereits gezogen wurde, aber sie ist ebenfalls eine gültige Ableitung basierend auf der Domain-Regel).*

Zusammenfassend lässt sich sagen: Mit RDFS können wir die Bedeutung und die beabsichtigte Struktur unserer RDF-Daten explizit machen. Dies verbessert nicht nur die Lesbarkeit und das gemeinsame Verständnis, sondern schafft auch die Grundlage für konsistentere Daten und die automatische Ableitung von neuem Wissen.

RDF Schema: Weiterführende Konzepte

Nachdem wir die Grundlagen von RDFS wie Klassen, Eigenschaften, Hierarchien, Domain und Range kennengelernt haben, betrachten wir nun einige weitere wichtige Konzepte und Konstrukte, die RDFS zur Verfügung stellt. Diese helfen uns, unsere Vokabulare noch präziser zu gestalten, Daten besser zu organisieren und die Semantik unserer RDF-Aussagen weiter zu verfeinern.

Die RDFS-Klassenhierarchie: `rdfs:Resource` als Wurzel

Konzept: RDFS definiert eine grundlegende Klassenhierarchie. An der Spitze dieser Hierarchie steht die Klasse `rdfs:Resource`. Alle Dinge, die in RDF beschrieben werden, gelten als Ressourcen. Somit sind alle Klassen (einschließlich `rdfs:Class` selbst) und auch alle Eigenschaften (`rdf:Property`) Unterklassen von `rdfs:Resource`. Auch Literale (`rdfs:Literal`) werden als spezielle Art von Ressource betrachtet.

```

urtle
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <http://data.rwth-aachen.de/resources/#> . # Beispiel-Namespace

# Die RDFS-Spezifikation definiert unter anderem:
rdfs:Resource rdf:type rdfs:Class . # rdfs:Resource ist der Typ aller Ressourcen

rdfs:Class rdf:type rdfs:Class ; # rdfs:Class ist selbst eine Klasse
           rdfs:subClassOf rdfs:Resource . # Jede Klasse ist eine Ressource

rdfs:Literal rdf:type rdfs:Class ; # rdfs:Literal ist die Klasse aller Literalwerte
           rdfs:subClassOf rdfs:Resource . # Literale sind auch Ressourcen

rdf:Property rdf:type rdfs:Class ; # rdf:Property ist die Klasse aller Eigenschaften
           rdfs:subClassOf rdfs:Resource . # Eigenschaften sind auch Ressourcen

# Für unsere selbstdefinierten Klassen (z.B. :Person) gilt daher implizit:
# :Person rdf:type rdfs:Class . (Unsere Definition)
# Da rdfs:Class eine Unterklasse von rdfs:Resource ist, und alle Instanzen
# einer Unterklasse auch Instanzen der Oberklasse sind, ist :Person
# (als Instanz von rdfs:Class) auch eine Instanz von rdfs:Resource.
# Zudem, da alle Instanzen von :Person auch Instanzen von rdfs:Resource sind,
# gilt auch :Person rdfs:subClassOf rdfs:Resource . (Dies wird oft inferiert)

```

Bedeutung: Alles, was wir in RDF beschreiben (Klassen, Eigenschaften, Individuen, Literale), ist letztendlich eine `rdfs:Resource`. Jede von uns selbst definierte Klasse (wie `:Person` oder `:Vorlesung`) ist implizit eine Unterklasse von `rdfs:Resource`. Dies unterstreicht die Universalität des Ressourcenkonzepts in RDF.

Dokumentations-Properties (`rdfs:label`, `rdfs:comment`, `rdfs:seeAlso`, `rdfs:isDefinedBy`)

Konzept: RDFS stellt Standardeigenschaften zur Verfügung, um Ressourcen (Klassen, Eigenschaften, Individuen) menschenlesbar zu dokumentieren und mit anderen Ressourcen in Beziehung zu setzen. Wir haben `rdfs:label` und `rdfs:comment` bereits verwendet.

- `rdfs:label` : Ein menschenlesbarer Name für die Ressource (kann mehrsprachig sein).
- `rdfs:comment` : Eine ausführlichere, menschenlesbare Beschreibung oder Erklärung (kann mehrsprachig sein).
- `rdfs:seeAlso` : Verweist auf eine andere Ressource, die zusätzliche Informationen über das Subjekt liefern könnte.
- `rdfs:isDefinedBy` : Verweist auf die Ressource (oft die Ontologie oder das Vokabular selbst als IRI), die das Subjekt definiert. Dies ist nützlich, um die Herkunft einer Definition anzugeben.

Turtle-Beispiel zur erweiterten Dokumentation unseres Schemas:

```

@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://data.rwth-aachen.de/resources/#> . # Unser Vokabular-Namespace
@prefix dbo: <http://dbpedia.org/ontology/> .

# Erweiterte Dokumentation für unsere Klassen und Eigenschaften
:Student rdfs:label "Student"@de , "Student"@en ;

```

```

    rdfs:comment "Eine Person, die an einer Hochschule eingeschrieben ist und studiert."@de ;
    rdfs:seeAlso <https://de.wikipedia.org/wiki/Student> ;
    rdfs:isDefinedBy :UnserUniVokabular . # Verweis auf die Ontologie/Vokabular-IRI

:Professor rdfs:label "Professor"@de , "Professor"@en ;
    rdfs:comment "Hochschullehrer mit Professur an einer Universität."@de ;
    rdfs:seeAlso <https://de.wikipedia.org/wiki/Professor> ;
    rdfs:isDefinedBy :UnserUniVokabular .

:hört rdfs:label "hört Vorlesung"@de , "attends lecture"@en ;
    rdfs:comment "Beziehung zwischen einem Studenten und einer Vorlesung, die er/sie aktiv besucht."@de
;

    rdfs:isDefinedBy :UnserUniVokabular .

# Die Definition unseres Vokabulars/Ontologie selbst (optional)
:UnserUniVokabular rdf:type rdfs:Resource ; # Oder owl:Ontology, wenn OWL verwendet wird
    rdfs:label "Universitäts Vokabular der RWTH (Beispiel)"@de .

```

Mehrfache rdfs:domain / rdfs:range - Die Schnittmengen-Semantik (Intersection)

Wichtiges Konzept: Wenn einer Eigenschaft mehrere rdfs:domain -Aussagen zugeordnet werden, bedeutet dies, dass das Subjekt dieser Eigenschaft Instanz **aller** angegebenen Domain-Klassen sein muss (logisches UND / Schnittmenge). Analoges gilt für mehrere rdfs:range -Aussagen: Das Objekt muss Instanz **aller** angegebenen Range-Klassen sein. Dies ist ein häufiges Missverständnis, da man intuitiv vielleicht eine ODER-Verknüpfung erwarten würde.

```

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://example.org/roles#> .
@prefix ex: <http://example.org/instances#> .

# Beispielklassen
:Professor rdf:type rdfs:Class ; rdfs:label "Professor"@de .
:PromoviertePerson rdf:type rdfs:Class ; rdfs:label "Promovierte Person"@de .
:Doktorand rdf:type rdfs:Class ; rdfs:label "Doktorand"@de . # Annahme: Doktorand ist auch eine Person

# Eigenschaft :betreutDoktorarbeit
:betreutDoktorarbeit rdf:type rdf:Property ;
    rdfs:label "betreut Doktorarbeit"@de ;
    rdfs:domain :Professor ; # Das Subjekt muss ein Professor sein
    rdfs:domain :PromoviertePerson ; # UND das Subjekt muss promoviert sein
    rdfs:range :Doktorand . # Das Objekt muss ein Doktorand sein

# Beispielhafte Instanzdaten
ex:ProfSchmidt rdf:type :Professor .
ex:ProfSchmidt rdf:type :PromoviertePerson . # Erfüllt beide Domain-Bedingungen
ex:MaxMustermann rdf:type :Doktorand .

ex:ProfSchmidt :betreutDoktorarbeit ex:MaxMustermann . # Korrekte Verwendung

# Inferenz:
# Aus ex:ProfMueller :betreutDoktorarbeit ex:LisaMeier .
# und den Domain-Angaben für :betreutDoktorarbeit würde ein RDFS-Reasoner folgern:
# ex:ProfMueller rdf:type :Professor .
# ex:ProfMueller rdf:type :PromoviertePerson .
# Und aus der Range-Angabe:
# ex:LisaMeier rdf:type :Doktorand .

```

Falsche Annahme (ODER-Semantik): Wenn man schreiben würde:

```

:hatKontaktinformation rdfs:domain :Student ;
    rdfs:domain :Professor .

```

Bedeutet dies **nicht**, dass das Subjekt von :hatKontaktinformation ein :Student **ODER** ein :Professor sein kann. Es bedeutet, dass es **sowohl** ein :Student **ALS AUCH** ein :Professor sein muss, was meist nicht die Intention ist. Für eine ODER-Semantik (Disjunktion) bräuchte man eine gemeinsame Oberklasse oder komplexere Konstrukte aus OWL.

RDFS Container für Sammlungen: rdfs:Bag, rdfs:Seq, rdfs:Alt

Neben den RDF Collections (Listen mit () -Syntax in Turtle, die auf rdf:List, rdf:first, rdf:rest und rdf:nil basieren und primär für **strikt geordnete Listen** gedacht sind), bietet RDFS auch sogenannte **Container-Klassen**. Diese dienen dazu, Gruppen von Ressourcen oder Literalen zu beschreiben, wobei die Art der Gruppierung variiert.

RDFS definiert drei Haupttypen von Containern:

1. **rdfs:Bag (Beutel/Sack):**

- Eine **ungeordnete** Sammlung von Ressourcen oder Literalen.
- Duplikate sind erlaubt.
- Die Reihenfolge der Elemente ist nicht signifikant.
- *Beispiel:* Eine Sammlung von Hobbys einer Person, bei der die Reihenfolge keine Rolle spielt.

2. **rdfs:Seq (Sequence/Sequenz):**

- Eine **geordnete** Sammlung von Ressourcen oder Literalen.
- Duplikate sind erlaubt.
- Die Reihenfolge der Elemente ist signifikant.
- *Beispiel:* Eine geordnete Liste von Kapiteln in einem Buch.

3. **rdfs:Alt (Alternative):**

- Eine Sammlung, die **Alternativen** für einen einzelnen Wert oder eine einzelne Ressource darstellt.
- Soll anzeigen, dass jedes Element der Sammlung eine mögliche Alternative ist.
- *Beispiel:* Alternative Sprachversionen eines Buchtitels oder verschiedene mögliche Prüfungstermine für ein Modul.

Elemente zu Containern hinzufügen

Es gibt zwei gebräuchliche Wege, um Elemente (Mitglieder) zu diesen Containern hinzuzufügen:

1. **Spezifische Membership-Eigenschaften (rdf:_1 , rdf:_2 , ...):**

- RDFS stellt Eigenschaften wie `rdf:_1` , `rdf:_2` , `rdf:_3` , usw. bereit. Dies sind Instanzen von `rdfs:ContainerMembershipProperty` .
- Diese werden oft verwendet, um Elemente hinzuzufügen, insbesondere wenn eine explizite Ordnung (wie bei `rdfs:Seq`) durch die Eigenschaft selbst ausgedrückt werden soll oder bei einer kleinen Anzahl von Elementen.
- Für `rdfs:Bag` drücken sie einfach nur Zugehörigkeit aus, nicht zwingend eine Ordnung.
- Für `rdfs:Alt` werden sie ebenfalls verwendet, um die verschiedenen Alternativen aufzulisten.

2. **Die allgemeine Eigenschaft rdfs:member :**

- `rdfs:member` ist eine allgemeinere Eigenschaft, um die Zugehörigkeit zu einem Container auszudrücken. Sie ist ebenfalls eine Instanz von `rdfs:ContainerMembershipProperty` .
- Sie kann für `rdfs:Bag` und `rdfs:Seq` verwendet werden.
- Bei `rdfs:Bag` ist ihre Verwendung oft intuitiver, da die Ordnung keine Rolle spielt.
- Bei `rdfs:Seq` kann `rdfs:member` verwendet werden, wenn die Ordnung implizit ist oder durch andere Mittel sichergestellt wird, obwohl `rdf:_1` , `rdf:_2` , ... hier oft klarer sind.

Wichtiger Unterschied zu RDF Collections ()

RDF Collections (oft "RDF Listen" genannt und in Turtle mit `(element1 element2 ...)` -Syntax dargestellt) sind für strikt geordnete Listen mit einer klar definierten, rekursiven Kopf-Rest-Struktur (`rdf:first` , `rdf:rest` , `rdf:nil`) vorgesehen. Sie sind in der Praxis für die Darstellung von Listen, bei denen die Ordnung und Vollständigkeit kritisch ist, weitaus gebräuchlicher und haben eine stärkere formale Semantik.

RDFS Container sind flexibler, aber ihre Semantik (insbesondere die genaue Bedeutung der Ordnung bei `rdfs:Seq` oder die Exklusivität/Auswahl bei `rdfs:Alt`) ist weniger stark definiert und wird von Reasonern oft nur begrenzt für automatische Schlussfolgerungen genutzt.

Turtle-Beispiele

```
@prefix rdf: [http://www.w3.org/1999/02/22-rdf-syntax-ns#](http://www.w3.org/1999/02/22-rdf-syntax-ns#) .
@prefix rdfs: [http://www.w3.org/2000/01/rdf-schema#](http://www.w3.org/2000/01/rdf-schema#) .
@prefix xsd: [http://www.w3.org/2001/XMLSchema#](http://www.w3.org/2001/XMLSchema#) . # Hinzugefügt für Datentypen
@prefix ex: [http://example.org/ontology/](http://example.org/ontology/) . # Beispiel-Namespace für Ontologiebegriffe
@prefix res: [http://example.org/resources/](http://example.org/resources/) . # Beispiel-Namespace für Ressourcen
@prefix uni: [http://example.org/uni/](http://example.org/uni/) . # Beispiel-Namespace für Uni-spezifische Dinge
@prefix voc: [http://example.org/vokabular/](http://example.org/vokabular/) . # Beispiel-Namespace für Vokabularbegriffe

# RDFS Container Klassen (Auszug, da sie Teil des RDFS-Vokabulars sind)
# rdfs:Bag rdf:type rdfs:Class ; rdfs:comment "Ungeordnete Sammlung, Duplikate erlaubt"@de .
# rdfs:Seq rdf:type rdfs:Class ; rdfs:comment "Geordnete Sammlung (Sequenz)"@de .
# rdfs:Alt rdf:type rdfs:Class ; rdfs:comment "Alternative Werte"@de .

# Beispiel 1: Ein Student hat eine Sammlung (Bag) von Hobbys mit rdf:_n
res:studentMaxMustermann ex:hatHobbys [
  rdf:type rdfs:Bag ;
  rdf:_1 "Lesen" ;
  rdf:_2 "Schwimmen" ;
  rdf:_3 "Programmieren"
] .
```

```
# Beispiel 2: Ein Modul hat alternative Prüfungstermine mit rdf:_n
ex:ModulWebTech ex:alternativePruefungstermine [
  rdf:type rdfs:Alt ;
  rdf:_1 "2025-07-20"^^xsd:date ;
  rdf:_2 "2025-09-15"^^xsd:date
] .

# Beispiel 3: Ein Bag von Vorlesungen für ein Curriculum mit rdfs:member
uni:curriculumInformatik rdf:type rdfs:Bag ;
  rdfs:label "Curriculum Informatik"@de ;
  rdfs:member uni:WebTechVorlesung ;
  rdfs:member uni:AlgoDatVorlesung ;
  rdfs:member uni:SWEVorlesung .

uni:WebTechVorlesung rdf:type voc:Vorlesung ; rdfs:label "Web Technologien"@de .
uni:AlgoDatVorlesung rdf:type voc:Vorlesung ; rdfs:label "Algorithmen und Datenstrukturen"@de .
uni:SWEVorlesung rdf:type voc:Vorlesung ; rdfs:label "Software Engineering"@de .

# Hinweis zu rdfs:ContainerMembershipProperty und rdf:_n
# Die Eigenschaften rdf:_1, rdf:_2, etc. sind Instanzen von rdfs:ContainerMembershipProperty.
# rdfs:member ist ebenfalls eine (Unter-)Eigenschaft davon und bietet eine allgemeinere Möglichkeit,
# die Zugehörigkeit auszudrücken, besonders wenn keine explizite Nummerierung gewünscht ist.
# rdf:_1 rdf:type rdfs:ContainerMembershipProperty . (usw. für rdf:_2, ...)
# rdfs:ContainerMembershipProperty rdf:type rdfs:Class ;
#                               rdfs:subClassOf rdf:Property .
```

Zusammenfassende Hinweise:

- Die expliziten Definitionen der Container-Klassen (`rdfs:Bag` etc.) und der Membership-Properties (`rdf:_1` , `rdfs:member` , `rdfs:ContainerMembershipProperty`) sind Teil des RDFS-Vokabulars und müssen normalerweise nicht selbst deklariert werden, es sei denn zu illustrativen Zwecken.
- Die Wahl zwischen `rdf:_n` -Eigenschaften und `rdfs:member` hängt vom spezifischen Anwendungsfall und der gewünschten Klarheit ab.
- Für Listen, bei denen die Ordnung und die genaue Struktur wichtig sind (z.B. für die Verarbeitung durch Programme), sind RDF Collections () oft die robustere Wahl. RDFS Container bieten mehr Flexibilität für losere Gruppierungen.

Datentypen in RDFS (`rdfs:Datatype`)

Konzept: RDFS führt die Klasse `rdfs:Datatype` ein, um Datentypen zu klassifizieren. `rdfs:Datatype` ist selbst eine Unterklasse von `rdfs:Class` . Die Standard-Datentypen aus XML Schema (wie `xsd:string` , `xsd:integer` , `xsd:date`) sind Instanzen von `rdfs:Datatype` .

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <http://example.org/ontology/> .
```

```
# RDFS Datentyp-Hierarchie (Auszug)
rdfs:Datatype rdf:type rdfs:Class ;
  rdfs:subClassOf rdfs:Class .
```

```
# XML Schema Datentypen sind Instanzen von rdfs:Datatype
xsd:string rdf:type rdfs:Datatype .
xsd:integer rdf:type rdfs:Datatype .
xsd:date rdf:type rdfs:Datatype .
# ... und so weiter für alle XSD-Typen.
```

```
# Beispiel: Verwendung eines Datentyps als Range
:hatGeburtsdatum rdf:type rdf:Property ;
  rdfs:label "hat Geburtsdatum"@de ;
  rdfs:domain :Person ;
  rdfs:range xsd:date . # Der Wertebereich ist der Datentyp xsd:date
```

```
# Definition eigener Datentypen (in RDFS eher konzeptuell)
# RDFS selbst bietet keine Mechanismen, um die Gültigkeit von Werten
# für selbstdefinierte Datentypen zu prüfen (z.B. dass eine Note im Bereich 1.0-5.0 liegt).
# Dies ist eher eine Domäne von OWL oder Datenvalidierungssprachen wie SHACL.
:DeutscheNotenskala rdf:type rdfs:Datatype ;
  rdfs:comment "Deutsche Notenskala von 1.0 bis 5.0."@de .
# rdfs:subClassOf xsd:decimal ; # Konzeptuelle Einordnung, aber ohne Validierungssemantik in
```

RDFS

```
:hatNote rdf:type rdf:Property ;
  rdfs:label "hat Note"@de ;
  rdfs:domain :Pruefungsleistung ; # Annahme: Klasse :Pruefungsleistung ist definiert
```

Beispiel eines erweiterten RDFS-Schemas (Zusammenfassung der RDFS-Konstrukte)

Dieser Abschnitt fasst viele der besprochenen RDFS-Konzepte in einem erweiterten Schema für unser Universitätsbeispiel zusammen. Er enthält Klassendefinitionen, Hierarchien, Eigenschaften mit Domain, Range und Dokumentation.

```
@base <http://data.rwth-aachen.de/ontology/> . # Geänderte Base IRI für das Vokabular selbst
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix : <#> . # Lokale Begriffe innerhalb dieses Vokabulars

# === ONTOLOGIE-METADATEN ===
<> rdf:type rdfs:Resource ; # Kann auch owl:Ontology sein, wenn OWL verwendet wird
    rdfs:label "Beispiel-Vokabular für Universitätsdaten"@de ;
    rdfs:comment "Ein einfaches RDFS-Schema zur Beschreibung von universitären Konzepten."@de ;
    rdfs:seeAlso <http://data.rwth-aachen.de/resources/> . # Bezug zur Instanzdaten-Basis-IRI

# === KLASSENHIERARCHIE ===
:Lebewesen rdf:type rdfs:Class ;
    rdfs:label "Lebewesen"@de .

:Person rdf:type rdfs:Class ;
    rdfs:subClassOf :Lebewesen ; # Person ist ein Lebewesen
    rdfs:label "Person"@de ;
    rdfs:comment "Ein menschliches Wesen."@de ;
    rdfs:isDefinedBy <> . # Definiert in dieser Ontologie

:Angestellter rdf:type rdfs:Class ;
    rdfs:subClassOf :Person ;
    rdfs:label "Angestellter"@de ;
    rdfs:comment "Eine Person, die an einer Organisation angestellt ist."@de ;
    rdfs:isDefinedBy <> .

:Student rdf:type rdfs:Class ;
    rdfs:subClassOf :Person ;
    rdfs:label "Student"@de ;
    rdfs:comment "Eine Person, die an einer Hochschule studiert."@de ;
    rdfs:isDefinedBy <> .

:Professor rdf:type rdfs:Class ;
    rdfs:subClassOf :Angestellter ; # Professor ist ein Angestellter
    rdfs:label "Professor"@de ;
    rdfs:comment "Eine Lehrkraft mit Professur."@de ;
    rdfs:isDefinedBy <> .

:Lehrveranstaltung rdf:type rdfs:Class ;
    rdfs:label "Lehrveranstaltung"@de ;
    rdfs:comment "Eine geplante Unterrichtseinheit."@de ;
    rdfs:isDefinedBy <> .

:Vorlesung rdf:type rdfs:Class ;
    rdfs:subClassOf :Lehrveranstaltung ; # Vorlesung ist eine Art Lehrveranstaltung
    rdfs:label "Vorlesung"@de ;
    rdfs:isDefinedBy <> .

:Seminar rdf:type rdfs:Class ;
    rdfs:subClassOf :Lehrveranstaltung ; # Seminar ist eine Art Lehrveranstaltung
    rdfs:label "Seminar"@de ;
    rdfs:isDefinedBy <> .

# === EIGENSCHAFTEN (PROPERTIES) MIT DOKUMENTATION, DOMAIN, RANGE UND HIERARCHIE ===
:hatName rdf:type rdf:Property ;
    rdfs:label "hat Namen"@de ;
    rdfs:comment "Gibt den Namen einer Entität an."@de ;
    rdfs:domain rdfs:Resource ; # Name kann für viele Dinge verwendet werden
    rdfs:range xsd:string ;
    rdfs:isDefinedBy <> .

:hatPersonalnummer rdf:type rdf:Property ;
    rdfs:label "hat Personalnummer"@de ;
    rdfs:domain :Angestellter ;
    rdfs:range xsd:string ; # Oft alphanumerisch
    rdfs:isDefinedBy <> .

:matrNr rdf:type rdf:Property ;
    rdfs:label "hat Matrikelnummer"@de ;
```

```

    rdfs:domain :Student ;
    rdfs:range xsd:string ; # Oft alphanumerisch
    rdfs:isDefinedBy <> .

:nimmtTeilAn rdf:type rdf:Property ;
    rdfs:label "nimmt teil an"@de ;
    rdfs:comment "Eine Person nimmt an einer Lehrveranstaltung teil."@de ;
    rdfs:domain :Person ;
    rdfs:range :Lehrveranstaltung ;
    rdfs:isDefinedBy <> .

:hört rdf:type rdf:Property ;
    rdfs:subPropertyOf :nimmtTeilAn ; # Speziellere Form von Teilnahme
    rdfs:label "hört Vorlesung"@de ;
    rdfs:domain :Student ; # Präzisierung der Domain
    rdfs:range :Vorlesung ; # Präzisierung der Range
    rdfs:isDefinedBy <> .

:wirdUnterrichtet rdf:type rdf:Property ;
    rdfs:label "wird unterrichtet"@de ;
    rdfs:comment "Ein Angestellter (z.B. Professor) unterrichtet eine Lehrveranstaltung."@de ;
    rdfs:range :Angestellter ;
    rdfs:domain :Lehrveranstaltung ;
    rdfs:isDefinedBy <> .

:gelesenVon rdf:type rdf:Property ;
    rdfs:subPropertyOf :unterrichtet ; # Speziellere Form von unterrichtet
    rdfs:label "lehrt Vorlesung"@de ;
    rdfs:range :Professor ; # Präzisierung der Range
    rdfs:domain :Vorlesung ; # Präzisierung der Domäne
    rdfs:isDefinedBy <> .

# Definition eines Studienplans, der eine Sammlung von Lehrveranstaltungen ist
:Studienplan rdf:type rdfs:Class ;
    rdfs:label "Studienplan"@de ;
    rdfs:comment "Eine geordnete Sammlung von Lehrveranstaltungen für einen Studiengang."@de ;
    rdfs:isDefinedBy <> .

:umfasstModul rdf:type rdf:Property ;
    rdfs:label "umfasst Modul"@de ;
    rdfs:domain :Studienplan ;
    rdfs:range rdf:List ; # Der Wert ist eine RDF Collection (geordnete Liste)
    rdfs:isDefinedBy <> .

```

Hinweis: In einem realen Szenario würden Instanzdaten (z.B. konkrete Studenten, Professoren, die das :hatCurriculum -Beispiel von früher nutzen) in einer separaten Datei oder einem separaten Graphen gehalten, der dieses Schema dann importiert oder referenziert.

RDF Reification - Aussagen über Aussagen

Konzept: Reification in RDF ist ein Mechanismus, der es erlaubt, **Aussagen über RDF-Tripel selbst** zu machen. Man kann also ein Tripel nehmen und es zu einer Ressource "verdinglichen" (engl. *to reify*), um dieser Ressource dann Eigenschaften zuzuweisen (z.B. wer hat die Aussage getroffen, wann, mit welcher Sicherheit, etc.). Dies sind Metadaten über Tripel.

Das RDF Reification Vokabular: RDF stellt hierfür vier spezielle Vokabularbegriffe zur Verfügung:

- `rdf:Statement` : Eine Klasse, deren Instanzen reifizierte Tripel sind.
- `rdf:subject` : Eine Eigenschaft, die von einer Instanz von `rdf:Statement` auf das Subjekt des ursprünglichen Tripels zeigt.
- `rdf:predicate` : Eine Eigenschaft, die von einer Instanz von `rdf:Statement` auf das Prädikat des ursprünglichen Tripels zeigt.
- `rdf:object` : Eine Eigenschaft, die von einer Instanz von `rdf:Statement` auf das Objekt des ursprünglichen Tripels zeigt.

Praktisches Beispiel: Prüfungsergebnis mit Metadaten

Angenommen, wir haben das ursprüngliche Tripel, das eine Note darstellt:

```

@prefix : <http://data.rwth-aachen.de/resources/#> .
@prefix voc: <http://data.rwth-aachen.de/vokabular/#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

# Ursprüngliches Tripel (Beispiel)
# :student28106 voc:hatNoteInPruefung [ voc:noteWert "1.7"^^xsd:decimal ; voc:geprueftIn :Mathe1Pruefung ] .
# Dies ist bereits eine gute Modellierung. Wenn wir aber die Aussage über die Note *selbst* kommentieren wollen:
:student28106 voc:hatNote "1.7"^^xsd:decimal . # Nehmen wir dieses einfachere Tripel zum Reifizieren

```

Wenn wir nun Metadaten zu genau dieser Aussage (z.B. wer hat die Prüfung abgenommen, wann) hinzufügen wollen, können wir das Tripel reifizieren:

```

@prefix : <http://data.rwth-aachen.de/resources/#> .
@prefix voc: <http://data.rwth-aachen.de/vokabular/#> .

```

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

Das ursprüngliche Tripel als Ressource (reifiziert)

```
_: AussageUeberNoteStudent28106 rdf:type rdf:Statement ; # Ein Blank Node oder eine IRI für das reifizierte Statement
```

```
    rdf:subject :student28106 ;
    rdf:predicate voc:hatNote ; # Bezieht sich auf die Eigenschaft, nicht das Literal
    rdf:object "1.7"^^xsd:decimal .
```

Metadaten über diese spezifische Noten-Aussage

```
_: AussageUeberNoteStudent28106 voc:geprueftVon :professor2125 ;
    voc:pruefungsdatum "2024-02-20"^^xsd:date ;
    voc:vertrauensgrad "0.95"^^xsd:float . # z.B. Konfidenz der Benotung
```

Nachteile und Alternativen:

- **Verbose:** Jedes ursprüngliche Tripel wird zu mindestens vier neuen Tripeln, um es zu reifizieren, plus weiteren Tripeln für die Metadaten. Das bläht RDF-Graphen stark auf.
- **Semantik:** Die reifizierte Aussage (_:AussageUeberNote...) ist *nicht logisch äquivalent* zum ursprünglichen Tripel (:student28106 voc:hatNote "1.7"^^xsd:decimal .). Das ursprüngliche Tripel muss möglicherweise immer noch separat assertiert werden, wenn es als direkte Tatsache im Graphen existieren soll.
- **Abfragekomplexität:** Das Abfragen von reifizierten Aussagen und ihren Metadaten kann umständlich werden.

Fazit zu Reification: Obwohl Reification im RDF-Standard definiert ist, wird es in der Praxis aufgrund seiner Umständlichkeit und der genannten Nachteile oft vermieden. Für viele Anwendungsfälle von "Aussagen über Aussagen" gibt es alternative Modellierungsmuster, die oft bevorzugt werden (z.B. Verwendung von benannten Graphen, speziellen Eigenschaften, die Kontext repräsentieren, oder fortgeschrittenere Ansätze wie RDF-Star/RDF/SPARQL – diese gehen aber über RDFS hinaus). Es ist jedoch gut, das Konzept der Reification zu kennen.

RDFS im Vergleich mit anderen Schema-Sprachen

Die folgende Tabelle stellt RDFS einigen anderen Ansätzen zur Schemadefinition und Datenmodellierung gegenüber, um die spezifischen Stärken und Schwächen von RDFS hervorzuheben:

Aspekt	RDFS	XML Schema (XSD)	JSON Schema	SQL DDL (Datenbank-Schema)
Primärer Fokus	Semantische Beschreibung, Vokabulare, einfache Ontologien	Validierung von XML-Dokumentstruktur & Datentypen	Validierung von JSON-Datenstruktur & Datentypen	Definition relationaler Tabellenstruktur, Integrität
Grundmodell	Graph (Tripel)	Baum (XML-Elemente/Attribute)	JSON-Objekte/-Arrays	Relationen (Tabellen)
Vererbung	✓ <code>rdfs:subClassOf</code> , <code>rdfs:subPropertyOf</code>	✓ (Typableitung durch <code>xs:extension</code> / <code>xs:restriction</code>)	Eingeschränkt (z.B. <code>allOf</code> , <code>oneOf</code>)	Begrenzt (Tabellenvererbung selten, nicht standardisiert)
Mehrfachvererbung	✓ (für Klassen und Eigenschaften)	X (für komplexe Typen)	✓ (via <code>allOf</code>)	X
Constraints/Validierung	Schwach (primär Inferenz über domain/range)	Stark (Datentypen, Kardinalität, Reihenfolge etc.)	Mittel bis Stark (Typen, Formate, Abhängigkeiten)	Stark (Keys, Constraints, Datentypen)
Flexibilität/Erweiterbarkeit	Sehr hoch (neue Tripel jederzeit hinzufügbar)	Mittel (Schema-Evolution kann komplex sein)	Hoch (JSON ist inhärent flexibel)	Niedrig bis Mittel (Schema-Änderungen sind oft aufwendig)
Inferenzfähigkeit	✓ (RDFS-Entailment-Regeln)	X	X	Begrenzt (Views, Trigger)
Annahme zur Welt	Offene-Welt-Annahme (OWA)	Geschlossene-Welt-Annahme (CWA) tendenziell	Geschlossene-Welt-Annahme (CWA) tendenziell	Geschlossene-Welt-Annahme (CWA)
Datentypen	Verwendet externe (z.B. XSD), <code>rdfs:Datatype</code>	XSD-Datentypen (reichhaltig)	JSON-Grundtypen, erweiterbar durch Formate	SQL-Datentypen

Grenzen von RDFS

Obwohl RDFS ein wichtiger Schritt zur Strukturierung von RDF-Daten ist, hat es auch klare Grenzen in seiner Ausdruckstärke. Es gibt viele Dinge, die man mit RDFS alleine **NICHT** ausdrücken kann, zum Beispiel:

- **Kardinalitätsbeschränkungen:**
 - Man kann nicht sagen, dass ein `:Professor` *genau einen* `:hatNamen` haben muss oder *mindestens drei* `:Vorlesungen` halten soll.
- **Disjunkte Klassen:**
 - Man kann nicht formal definieren, dass eine Instanz der Klasse `:Student` nicht gleichzeitig eine Instanz der Klasse `:Professor` sein kann (d.h. dass die Mengen der Studenten und Professoren disjunkt sind).
- **Äquivalenz von Klassen oder Eigenschaften:**
 - Man kann nicht ausdrücken, dass die Klasse `:Hochschullehrer` äquivalent zu `:Professor` ist, oder dass die Eigenschaft `:unterrichtetIn` die gleiche Bedeutung hat wie `:lehrtInSemester`.
- **Komplexere Klassenbeschreibungen und logische Beziehungen:**

- Man kann keine Klasse definieren als "alle Personen, die mindestens eine Vorlesung hören" (Existenzquantor) oder "alle Vorlesungen, die ausschließlich von W3-Professoren gehalten werden" (Allquantor, Wertbeschränkungen).
- **Einschränkung von Eigenschaftswerten auf bestimmte Individuen (Enumerationen/abgeschlossene Listen):**
 - Man kann nicht sagen, dass das Geschlecht einer Person nur die Werte `:maennlich`, `:weiblich` oder `:divers` annehmen darf.

Beispiel für fehlende Ausdruckskraft in RDFS (kann so NICHT direkt modelliert werden):

```
# Diese Aussagen sind mit reinem RDFS nicht oder nur sehr umständlich/indirekt formulierbar:
# "Ein Student muss genau eine Matrikelnummer haben." (Kardinalität)
# ":PersonIstMaennlich und :PersonIstWeiblich sind disjunkte Klassen von :Person." (Disjunktheit)
# ":Hochschuldozent ist äquivalent zu :Professor." (Äquivalenz)
# "Ein :FortgeschrittenenModul ist ein Modul, das von einem :Professor gehalten wird, der den Rang 'W3' hat."
(Komplexe Klasse)
```

Die Lösung für mehr Ausdruckskraft: Für diese und viele weitere, komplexere Modellierungsanforderungen wurde die **Web Ontology Language (OWL)** entwickelt. OWL baut auf RDFS auf und erweitert dessen Vokabular um Konstrukte, die eine deutlich reichhaltigere Semantik und stärkere Inferenzmöglichkeiten erlauben. Dies wird Thema eines späteren Notebooks sein.

Praktische Anwendungstipps für RDFS

1. **Dokumentation ist entscheidend:** Nutzen Sie konsequent `rdfs:label` (für verschiedene Sprachen) und `rdfs:comment`, um Ihr Schema verständlich und wartbar zu machen. `rdfs:seeAlso` und `rdfs:isDefinedBy` können helfen, Ihr Vokabular in einen größeren Kontext zu stellen.
2. **Domain und Range bewusst einsetzen:** Verstehen Sie die UND-Semantik bei mehrfachen `rdfs:domain` - oder `rdfs:range` -Angaben auf derselben Eigenschaft. Überlegen Sie, ob eine gemeinsame Oberklasse oder eine Aufteilung der Eigenschaft sinnvoller ist, falls Sie eine ODER-Logik benötigen. Denken Sie daran, dass sie primär der Inferenz dienen.
3. **RDFS Container (`rdfs:Bag`, `rdfs:Seq`, `rdfs:Alt`) sparsam verwenden:** Für geordnete Listen ist die Turtle-Syntax `()` (die zu `rdf:List` expandiert) oft intuitiver und verbreiteter. Container haben spezifische Anwendungsfälle, sind aber nicht immer die erste Wahl für Sammlungen.
4. **Klassenhierarchien sinnvoll gestalten:** Halten Sie Hierarchien tendenziell eher flach und stellen Sie sicher, dass jede `rdfs:subClassOf` -Beziehung semantisch klar motiviert ist ("Jede Instanz der Unterklasse ist auch eine Instanz der Oberklasse"). Zu tiefe oder unklare Hierarchien können unübersichtlich werden.
5. **Konsistente Namensgebung:** Verwenden Sie einheitliche Konventionen für die Benennung Ihrer Klassen (oft PascalCase, z.B. `:MeineKlasse`) und Eigenschaften (oft camelCase, z.B. `:meineEigenschaft`).
6. **Iterative Entwicklung:** Beginnen Sie mit einem einfachen Schema und erweitern Sie es schrittweise nach Bedarf. RDFS ist flexibel genug, um dies zu unterstützen.

Weiterführend: Für ausdrucksstärkere Schemata und komplexere logische Zusammenhänge, die über die Fähigkeiten von RDFS hinausgehen, ist **OWL (Web Ontology Language)** der nächste Schritt.

Übungsaufgaben zu RDF Schema (RDFS)

Diese Aufgaben sollen Ihnen helfen, die Konzepte von RDF Schema praktisch anzuwenden und die Auswirkungen von Schema-Definitionen auf RDF-Daten zu verstehen. Nutzen Sie `%%rdf turtle` Zellen um Ihre Schemata und Instanzdaten zu testen und zu visualisieren.

Übung 1: Eigene Klassenhierarchie erstellen

Sie möchten ein einfaches Vokabular für verschiedene Arten von Publikationen erstellen.

Aufgaben:

1. Definieren Sie die folgenden Klassen mit `rdfs:Class`, `rdfs:label` (Deutsch und Englisch) und `rdfs:comment` (Deutsch):
 - `:Publikation` (Eine allgemeine Publikation)
 - `:Buch` (Ein Buch)
 - `:Artikel` (Ein wissenschaftlicher oder journalistischer Artikel)
 - `:Konferenzbeitrag` (Ein Artikel, der auf einer Konferenz präsentiert wurde)
2. Erstellen Sie eine sinnvolle Klassenhierarchie mit `rdfs:subClassOf`. Zum Beispiel ist ein `:Buch` eine Art von `:Publikation`. Ein `:Konferenzbeitrag` ist eine spezielle Art von `:Artikel`.
3. Definieren Sie eine Instanz für jede der spezifischsten Klassen Ihrer Hierarchie (z.B. eine Instanz von `:Buch` und eine Instanz von `:Konferenzbeitrag`). Weisen Sie diesen Instanzen mit `rdf:type` ihre Klasse zu.
4. **Bonus (mit Reasoning):** Welche zusätzlichen `rdf:type` -Aussagen würden für Ihre Instanzen inferiert werden? Notieren Sie diese.

```
In [10]: %%rdf turtle
#
```

Übung 2: Eigenschaften mit Domain und Range definieren

Erweitern Sie Ihr Publikationsvokabular aus Übung 1.

Aufgaben:

- Definieren Sie die folgenden Eigenschaften mit `rdf:Property`, `rdfs:label`, `rdfs:comment`, `rdfs:domain` und `rdfs:range`:
 - `:hatTitel` (Soll für alle `:Publikation` en gelten und einen `xsd:string` als Wert haben)
 - `:hatAutor` (Soll für alle `:Publikation` en gelten und eine `:Person` als Wert haben – definieren Sie die Klasse `:Person` ebenfalls kurz mit Label/Comment)
 - `:erscheintInKonferenz` (Soll spezifisch für `:Konferenzbeitrag` gelten und eine Ressource vom Typ `:Konferenz` als Wert haben – definieren Sie `:Konferenz` ebenfalls kurz)
 - `:hatISBN` (Soll spezifisch für `:Buch` gelten und einen `xsd:string` als Wert haben)
- Erstellen Sie eine Beispielinstantz eines `:Buches` und eines `:Konferenzbeitrags` und weisen Sie ihnen einige der oben definierten Eigenschaften mit passenden Werten zu.
- Bonus (falls Reasoning verfügbar):** Wenn Sie einer Instanz `meinBuch :hatISBN "123-456"` . definieren, welche `rdf:type` - Aussage für `meinBuch` wird aufgrund der Domain-Definition von `:hatISBN` inferiert?

```
In [11]: %%rdf turtle
#
```

Übung 3: Eigenschafts-Hierarchie (`rdfs:subPropertyOf`)

Manchmal sind Eigenschaften Spezialisierungen anderer Eigenschaften.

Aufgaben:

- Definieren Sie eine allgemeine Eigenschaft `:hatIdentifizier` mit `rdfs:domain rdfs:Resource` und `rdfs:range xsd:string` .
- Definieren Sie die Eigenschaft `:hatISBN` (aus Übung 2) nun zusätzlich als `rdfs:subPropertyOf :hatIdentifizier` .
- Definieren Sie eine neue Eigenschaft `:hatDOI` (Digital Object Identifier) ebenfalls als `rdfs:subPropertyOf :hatIdentifizier` , mit `rdfs:domain :Artikel` und `rdfs:range xsd:string` .
- Erstellen Sie eine Instanz `:meinArtikel a :Artikel ; :hatDOI "10.1000/xyz"` . .
- Bonus (falls Reasoning verfügbar):** Welche Aussage kann für `:meinArtikel` aufgrund der `rdfs:subPropertyOf` -Beziehung und der Verwendung von `:hatDOI` zusätzlich inferiert werden?

```
In [12]: %%rdf turtle
#
```

Übung 4: Dokumentation und mehrfache Domains (Intersection)

- Dokumentation verbessern:** Wählen Sie eine Klasse und eine Eigenschaft aus den vorherigen Übungen. Fügen Sie diesen Definitionen `rdfs:seeAlso` (mit einem Link zu einer relevanten Webseite, z.B. Wikipedia) und `rdfs:isDefinedBy` (mit einer fiktiven IRI für Ihr Vokabular, z.B. `<http://example.org/meinPublikationsVokabular>`) hinzu.
- Mehrfache Domains verstehen:** Sie möchten eine Eigenschaft `:istHauptautorVon` definieren, die nur für Personen gilt, die sowohl `:Professor` als auch `:AktiverForscher` sind (definieren Sie diese beiden Klassen kurz). Das Objekt soll eine `:Publikation` sein.
 - Wie definieren Sie `rdfs:domain` für `:istHauptautorVon` , um diese UND-Bedingung auszudrücken?
 - Erstellen Sie eine Instanz `:profX` , die beide Elterntypen (`:Professor` , `:AktiverForscher`) explizit zugewiesen bekommt und die Eigenschaft `:istHauptautorVon` für eine `:Publikation` verwendet.
 - Was würde ein RDFS-Reasoner inferieren, wenn eine Instanz `:profY` nur den Typ `:Professor` hätte, aber ebenfalls `:istHauptautorVon` für eine Publikation verwenden würde?

```
In [13]: %%rdf turtle
#
```

```
In [14]: %%rdf turtle
#
```

Übung 5: RDFS Container vs. RDF Collection (())

Sie möchten die Mitglieder eines Projektteams und die geordneten Schritte eines Arbeitspakets modellieren.

Aufgaben:

- Projektteam (ungeordnet):** Modellieren Sie mit `rdfs:Bag` und `rdfs:member` (oder `rdf:_n`), dass das Projekt `:ProjektAlpha` die Teammitglieder `:PersonA` , `:PersonB` und `:PersonC` hat. Die Reihenfolge spielt keine Rolle.
- Arbeitspaket-Schritte (geordnet):** Modellieren Sie mit der Turtle-Listen-Syntax (()) , dass das Arbeitspaket `:AP1` die Schritte

"Planung", "Implementierung", "Test" und "Dokumentation" in dieser genauen Reihenfolge umfasst. Verwenden Sie eine Eigenschaft wie `:hatSchritte`.

3. Diskutieren Sie kurz, warum Sie für Aufgabe 1 einen RDFS Container und für Aufgabe 2 eine RDF Collection (`rdf:List`) gewählt haben. Was wären die Implikationen, wenn Sie die jeweils andere Struktur verwenden würden?

```
In [15]: %%rdf turtle
#
```

```
In [16]: %%rdf turtle
#
```

Übung 6 (optional): Grenzen von RDFS reflektieren

Wählen Sie eine der im Notebook unter "Grenzen von RDFS" genannten Einschränkungen (z.B. Kardinalitätsbeschränkungen, disjunkte Klassen).

1. Beschreiben Sie ein konkretes Szenario aus dem Universitätskontext, bei dem diese Einschränkung von RDFS relevant wird (d.h., wo Sie gerne etwas ausdrücken würden, was RDFS nicht oder nur unzureichend kann).
2. Wie könnte man diese Information versuchen, mit den Mitteln von RDFS anzudeuten oder zu dokumentieren (z.B. via `rdfs:comment`), auch wenn es nicht formal durchsetzbar ist?

Lösung Übung 6

Zusammenfassung und Ausblick

In diesem Notebook haben wir uns intensiv mit RDF Schema (RDFS) beschäftigt und gelernt, wie es uns ermöglicht, Vokabulare für unsere RDF-Daten zu definieren und diesen dadurch mehr Struktur und Bedeutung zu verleihen.

Was haben wir gelernt? (Kernkonzepte von RDFS)

1. **RDF Schema (RDFS)** ist eine Erweiterung von RDF, die grundlegende **Strukturierungs- und Vokabular-Definitionsmöglichkeiten** für RDF-Daten bereitstellt.
2. **Klassen (`rdfs:Class`) und Eigenschaften (`rdfs:Property`)** sind zentrale Bausteine, die eine explizite **Typisierung** von Ressourcen und die Definition von Beziehungen ermöglichen.
3. **Domain (`rdfs:domain`) und Range (`rdfs:range`)** spezifizieren den intendierten **Verwendungskontext** von Eigenschaften, indem sie typische Klassen/Datentypen für Subjekte bzw. Objekte festlegen (und Grundlage für Inferenzen sind).
4. **Hierarchien** mittels `rdfs:subClassOf` und `rdfs:subPropertyOf` erlauben die **Spezialisierung und Generalisierung** von Klassen und Eigenschaften und somit die Wiederverwendung und Organisation von Vokabularbestandteilen.
5. Einfache **Inferenzen (Schlussfolgerungen)**, basierend auf den RDFS-Konstrukten (insbesondere Hierarchien, Domain und Range), ermöglichen die automatische **Ableitung neuer, impliziter Fakten** aus den explizit vorhandenen Daten und dem Schema.
6. RDFS bietet eine hohe **Flexibilität** bei der schrittweisen Definition und Erweiterung von Schemata ("Schema-later" oder "Schema-optional" Ansätze sind möglich).

Nächste Schritte: Jenseits von RDFS

Obwohl RDFS mächtige Grundlagen legt, gibt es viele Modellierungsanforderungen, die über seine Ausdrucksstärke hinausgehen. Die logischen nächsten Schritte zur Vertiefung Ihres Wissens im Bereich Semantic Web und Ontologien sind:

- **OWL (Web Ontology Language)**: Für die Erstellung noch **ausdrucksstärkerer und formal präziserer Schemata (Ontologien)**. OWL ermöglicht komplexere Klassenbeschreibungen, Kardinalitätsbeschränkungen, Disjunktheit von Klassen, Äquivalenzbeziehungen und vieles mehr, was zu deutlich mächtigeren Inferenzmöglichkeiten führt.
- **SHACL (Shapes Constraint Language)**: Eine moderne Sprache zur **Validierung von RDF-Daten** anhand definierter "Shapes" (Datenstrukturen und Constraints). Während RDFS (und OWL) primär die Semantik beschreiben und Inferenzen ermöglichen, dient SHACL der expliziten Überprüfung der Datenkonformität.
- **Reasoning Engines und Triple Stores**: Die praktische Anwendung von RDFS und OWL entfaltet ihr volles Potenzial oft erst in Kombination mit **Reasoning Engines** (Schlussfolgerungswerkzeugen) und **Triple Stores** (spezialisierte Datenbanken für RDF-Daten), die Inferenzen automatisiert durchführen und SPARQL-Abfragen über abgeleitetes Wissen ermöglichen.

Empfohlene Literatur und Ressourcen

Für eine Vertiefung der hier behandelten und der weiterführenden Themen empfehlen sich:

- **Offizielle Spezifikationen des W3C:**
 - [RDF Schema 1.1 \(W3C Recommendation\)](#) (Die maßgebliche Quelle für RDFS)
 - [OWL 2 Web Ontology Language Primer \(W3C Recommendation\)](#) (Guter Einstieg in OWL)
 - [Shapes Constraint Language \(SHACL\) \(W3C Recommendation\)](#)

- **Fachbücher:**

- Allemang, Dean, Gandon, Fabien, & Hendler, James (2020). *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL (3rd ed.)*. ACM Books.

Dieses Notebook hat Ihnen hoffentlich eine solide Grundlage für das Verständnis und die Anwendung von RDF Schema vermittelt!

```
In [ ]: %reload_ext jupyter-rdfify
```

Die Beispieldatenbank aus der ER Modellierung

Das folgende Schema haben wir in der ER Modellierung schon entwickelt und kennengelernt:

Relationenschemata (ohne Wertebereiche):

- Student(MatrNr, Name, Semester)
- Professor(PersNr, Name, Rang, Raum)
- Vorlesung(VorlNr, Titel, SWS, gelesenVon)
- Assistent(PersNr, Name, Fachgebiet, Boss)
- Voraussetzen(Vorgänger, Nachfolger)
- Hört(MatrNr, VorlNr)
- Prüft(MatrNr, VorlNr, PersNr, Note)

Interrelationale Abhängigkeiten:

- Vorlesung[gelesenVon] $\subseteq$ Professor[PersNr]
- Assistent[Boss] $\subseteq$ Professor[PersNr]
- Voraussetzen[Vorgänger] $\subseteq$ Vorlesung[VorlNr]
- Voraussetzen[Nachfolger] $\subseteq$ Vorlesung[VorlNr]
- Hört[MatrNr] $\subseteq$ Student[MatrNr]
- Hört[VorlNr] $\subseteq$ Vorlesung[VorlNr]
- Prüft[MatrNr] $\subseteq$ Student[MatrNr]
- Prüft[VorlNr] $\subseteq$ Vorlesung[VorlNr]
- Prüft[PersNr] $\subseteq$ Professor[PersNr]
- Assistent[PersNr] $\cap$ Professor[PersNr] = $\emptyset$

Beispiel für eine Datenbank

Professor			
PersNr	Name	Rang	Raum
2125	Sokrates	W3	226
2126	Russel	W3	232
2127	Kopernikus	W2	310
2133	Popper	W2	52
2134	Augustinus	W2	309
2136	Curie	W3	36
2137	Kant	W3	7

Student		
MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Vorlesung			
VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

hört			
MatrNr	VorlNr		
26120	5001		
27550	5001		
27550	4052		
28106	5041		
28106	5052		
28106	5216		
28106	5259		
29120	5001		
29120	5041		
29120	5049		
29555	5022		
25403	5022		

voraussetzen	
Vorgänger	Nachfolger
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

Assistent			
PerslNr	Name	Fachgebiet	Boss
3002	Platon	Ideenlehre	2125
3003	Aristoteles	Syllogistik	2125
3004	Wittgenstein	Sprachtheorie	2126
3005	Rhetikus	Planetenbewegung	2127
3006	Newton	Keplersche Gesetze	2127
3007	Spinoza	Gott und Natur	2126

prüft			
MatrNr	VorlNr	PersNr	Note
28106	5001	2126	1,0
25403	5041	2125	2,0

Mit dem im vorherigen Notebook entwickelten RDF-Schema Konzepten können wir dies nach Turtle übertragen. Wir werden diese dann als Beispieldatenbank für die RDF Anfragesprache SPARQL verwenden. Einige Daten wurden modifiziert, um SPARQL Primitive besser demonstrieren zu können. Das ist bei den Turtle-Definitionen vermerkt.

In []:

```
%%rdf turtle -l uni --display none

@base <http://data.rwth-aachen.de/resources/> .
@prefix : <> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

# Klassen definieren
:Student a rdfs:Class ;
    rdfs:label "Student" .

:Professor a rdfs:Class ;
    rdfs:label "Professor" .

:Vorlesung a rdfs:Class ;
    rdfs:label "Vorlesung" .

:Assistent a rdfs:Class ;
    rdfs:label "Assistent" .

:Prüfung a rdfs:Class ;
    rdfs:label "Prüfung" .

# Properties definieren
:matrNr a rdf:Property ;
    rdfs:label "Matrikelnummer" ;
    rdfs:domain :Student ;
    rdfs:range xsd:integer .

:name a rdf:Property ;
    rdfs:label "Name" ;
    rdfs:range xsd:string .

:semester a rdf:Property ;
    rdfs:label "Semester" ;
    rdfs:domain :Student ;
    rdfs:range xsd:integer .

:persNr a rdf:Property ;
    rdfs:label "Personalnummer" ;
    rdfs:range xsd:integer .

:rang a rdf:Property ;
    rdfs:label "Rang" ;
    rdfs:domain :Professor ;
    rdfs:range xsd:string .

:raum a rdf:Property ;
    rdfs:label "Raum" ;
    rdfs:domain :Professor ;
    rdfs:range xsd:integer .

:vorlNr a rdf:Property ;
    rdfs:label "Vorlesungsnummer" ;
    rdfs:domain :Vorlesung ;
    rdfs:range xsd:integer .

:titel a rdf:Property ;
    rdfs:label "Titel" ;
    rdfs:domain :Vorlesung ;
    rdfs:range xsd:string .

:sws a rdf:Property ;
    rdfs:label "Semesterwochenstunden" ;
    rdfs:domain :Vorlesung ;
    rdfs:range xsd:integer .

:gelesenVon a rdf:Property ;
    rdfs:label "gelesen von" ;
    rdfs:domain :Vorlesung ;
    rdfs:range :Professor .

:fachgebiet a rdf:Property ;
    rdfs:label "Fachgebiet" ;
    rdfs:domain :Assistent ;
    rdfs:range xsd:string .

:boss a rdf:Property ;
    rdfs:label "Boss" ;
    rdfs:domain :Assistent ;
    rdfs:range :Professor .

:hatVoraussetzung a rdf:Property ;
    rdfs:label "hat Voraussetzung" ;
    rdfs:domain :Vorlesung ;
    rdfs:range :Vorlesung .

:hört a rdf:Property ;
```

```

    rdfs:label "hört" ;
    rdfs:domain :Student ;
    rdfs:range :Vorlesung .

:prüfer a rdf:Property ;
    rdfs:label "Prüfer" ;
    rdfs:domain :Prüfung ;
    rdfs:range :Professor .

:geprüfterStudent a rdf:Property ;
    rdfs:label "geprüfter Student" ;
    rdfs:domain :Prüfung ;
    rdfs:range :Student .

:geprüfteVorlesung a rdf:Property ;
    rdfs:label "geprüfte Vorlesung" ;
    rdfs:domain :Prüfung ;
    rdfs:range :Vorlesung .

:note a rdf:Property ;
    rdfs:label "Note" ;
    rdfs:domain :Prüfung ;
    rdfs:range xsd:string .

# Studenten
:student24002 a :Student ;
    :matrNr 24002 ;
    :name "Xenokrates" ;
    :semester 18 .

:student25403 a :Student ;
    :matrNr 25403 ;
    :name "Jonas" ;
    :semester 12 .

:student26120 a :Student ;
#    :matrNr 26120 ;
    :name "Fichte" ;
    :semester 10 .

:student26830 a :Student ;
    :matrNr 26830 ;
    :name "Aristoxenos" ;
    :semester 8 .

:student27550 a :Student ;
    :matrNr 27550 ;
    :name "Schopenhauer" ;
    :semester 6 .

:student28106 a :Student ;
    :matrNr 28106 ;
    :name "Carnap" ;
    :semester 3 .

:student29120 a :Student ;
    :matrNr 29120 ;
    :name "Theophrastos" ;
    :semester 2 .

:student29555 a :Student ;
    :matrNr 29555 ;
    :name "Feuerbach" ;
    :semester 2 .

# Professoren
:professor2125 a :Professor ;
    :persNr 2125 ;
    :name "Sokrates" ;
    :rang "W3" ;
    :raum 226 .

:professor2126 a :Professor ;
    :persNr 2126 ;
    :name "Russel" ;
    :rang "W3" ;
    :raum 232 .

:professor2127 a :Professor ;
    :persNr 2127 ;
    :name "Kopernikus" ;
    :rang "W2" ;
    :raum 310 .

:professor2133 a :Professor ;
    :persNr 2133 ;
    :name "Popper" ;
    :rang "W2" ;
    :raum 52 .

```

```

:professor2134 a :Professor ;
    :persNr 2134 ;
    :name "Augustinus" ;
    :rang "W2" ;
    :raum 309 .

:professor2136 a :Professor ;
    :persNr 2136 ;
    :name "Curie" ;
    :rang "W3" ;
    :raum 36 .

:professor2137 a :Professor ;
    :persNr 2137 ;
    :name "Kant" ;
    :rang "W3" ;
    :raum 7 .

# Vorlesungen
:vorlesung5001 a :Vorlesung ;
    :vorlNr 5001 ;
    :titel "Grundzüge" ;
    :sws 4 ;
    :gelesenVon :professor2137 .

:vorlesung5041 a :Vorlesung ;
    :vorlNr 5041 ;
    :titel "Ethik" ;
    :sws 4 ;
    :gelesenVon :professor2125 .

:vorlesung5043 a :Vorlesung ;
    :vorlNr 5043 ;
    :titel "Einführung in die Erkenntnistheorie" ;
    :sws 3 ;
    :gelesenVon :professor2126 .

:vorlesung5049 a :Vorlesung ;
    :vorlNr 5049 ;
    :titel "Mäeutik" ;
    :sws 2 ;
    :gelesenVon :professor2125 .

:vorlesung4052 a :Vorlesung ;
    :vorlNr 4052 ;
    :titel "Logik" ;
    :sws 4 ;
    :gelesenVon :professor2125 .

:vorlesung5052 a :Vorlesung ;
    :vorlNr 5052 ;
    :titel "Wissenschaftstheorie" ;
    :sws 3 ;
    :gelesenVon :professor2126 .

:vorlesung5216 a :Vorlesung ;
    :vorlNr 5216 ;
    :titel "Bioethik" ;
    :sws 2 ;
    :gelesenVon :professor2126 .

:vorlesung5259 a :Vorlesung ;
    :vorlNr 5259 ;
    :titel "Der Wiener Kreis" ;
    :sws 2 ;
    :gelesenVon :professor2133 .

:vorlesung5022 a :Vorlesung ;
    :vorlNr 5022 ;
    :titel "Glaube und Wissen" ;
    :sws 2 ;
    :gelesenVon :professor2134 .

:vorlesung4630 a :Vorlesung ;
    :vorlNr 4630 ;
    :titel "Die 3 Kritiken" ;
    :sws 4 ;
    :gelesenVon :professor2137 .

# Assistenten
:assistent3002 a :Assistent ;
    :persNr 3002 ;
    :name "Platon" ;
    :fachgebiet "Ideenlehre" ;
    :boss :professor2125 .

:assistent3003 a :Assistent ;
    :persNr 3003 ;
    :name "Aristoteles" ;
    :fachgebiet "Syllogistik" ;
    :boss :professor2125 .

```

```

:boss :professor2126 .

:assistant3004 a :Assistent ;
  :persNr 3004 ;
  :name "Wittgenstein" ;
  :fachgebiet "Sprachtheorie" ;
  :boss :professor2126 .

:assistant3005 a :Assistent ;
  :persNr 3005 ;
  :name "Rhetikus" ;
  :fachgebiet "Planetenbewegung" ;
  :boss :professor2127 .

:assistant3006 a :Assistent ;
  :persNr 3006 ;
  :name "Newton" ;
  :fachgebiet "Keplersche Gesetze" ;
  :boss :professor2127 .

:assistant3007 a :Assistent ;
  :persNr 3007 ;
  :name "Spinoza" ;
  :fachgebiet "Gott und Natur" ;
  :boss :professor2126 .

# Voraussetzungen
:vorlesung5041 :hatVoraussetzung :vorlesung5001 .
:vorlesung5043 :hatVoraussetzung :vorlesung5001 .
:vorlesung5049 :hatVoraussetzung :vorlesung5001 .
:vorlesung5216 :hatVoraussetzung :vorlesung5041 .
:vorlesung5052 :hatVoraussetzung :vorlesung5043, :vorlesung5041 .
:vorlesung5259 :hatVoraussetzung :vorlesung5052 .

:vorlesungXXXX a :Vorlesung ; :titel "Advanced Logik" ; :hatVoraussetzung :vorlesung5021 . # Für Property Path Beispiel
:vorlesung0042 a :Vorlesung ; # Dummy Vorlesung für Negation Beispiel
  :vorlNr 0042 ;
  :titel "Spezialthemen" ;
  :sws 2 ;
  :gelesenVon :professor2126 .

# Hört-Beziehungen
# :student26120 :hört :vorlesung5001 .
# Das student26120 hört keine Vorlesung - das Faktum wird auskommentiert. Wird später für ein Beispiel (Filte NOT EXISTS) ge

:student27550 :hört :vorlesung5001, :vorlesung4052 .
:student28106 :hört :vorlesung5041, :vorlesung5052, :vorlesung5216, :vorlesung5259 .
:student29120 :hört :vorlesung5001, :vorlesung5041, :vorlesung5049 .
:student29555 :hört :vorlesung5022 .
:student25403 :hört :vorlesung5022 .

# Prüfungen
:prüfung1 a :Prüfung ;
  :geprüfterStudent :student28106 ;
  :geprüfteVorlesung :vorlesung5001 ;
  :prüfer :professor2126 ;
  :note "1.0" .

:prüfung2 a :Prüfung ;
  :geprüfterStudent :student25403 ;
  :geprüfteVorlesung :vorlesung5041 ;
  :prüfer :professor2125 ;
  :note "2.0" .

```

Die Datenbank werden wir nun im folgenden verwenden. Bitte beachten Sie dass die verwendete RDF Bibliothek die Daten nicht persistent abgespeichert. Ums sie anzufragen, müssen wir sie lokal im Jupyter Notebook verwenden.

```
In [ ]: %reload_ext jupyter-rdfify
```

Die folgende Zelle unbedingt ausführen. Hier wird die Datenbank initialisiert!

In [3]:

```
%%rdf turtle -l uni --display none

@base <http://data.rwth-aachen.de/resources/> .
@prefix : <> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

# Klassen definieren
:Student a rdfs:Class ;
    rdfs:label "Student" .

:Professor a rdfs:Class ;
    rdfs:label "Professor" .

:Vorlesung a rdfs:Class ;
    rdfs:label "Vorlesung" .

:Assistent a rdfs:Class ;
    rdfs:label "Assistent" .

:Prüfung a rdfs:Class ;
    rdfs:label "Prüfung" .

# Properties definieren
:matrNr a rdf:Property ;
    rdfs:label "Matrikelnummer" ;
    rdfs:domain :Student ;
    rdfs:range xsd:integer .

:name a rdf:Property ;
    rdfs:label "Name" ;
    rdfs:range xsd:string .

:semester a rdf:Property ;
    rdfs:label "Semester" ;
    rdfs:domain :Student ;
    rdfs:range xsd:integer .

:persNr a rdf:Property ;
    rdfs:label "Personalnummer" ;
    rdfs:range xsd:integer .

:rang a rdf:Property ;
    rdfs:label "Rang" ;
    rdfs:domain :Professor ;
    rdfs:range xsd:string .

:raum a rdf:Property ;
    rdfs:label "Raum" ;
    rdfs:domain :Professor ;
    rdfs:range xsd:integer .

:vorlNr a rdf:Property ;
    rdfs:label "Vorlesungsnummer" ;
    rdfs:domain :Vorlesung ;
    rdfs:range xsd:integer .

:titel a rdf:Property ;
    rdfs:label "Titel" ;
    rdfs:domain :Vorlesung ;
    rdfs:range xsd:string .

:sws a rdf:Property ;
    rdfs:label "Semesterwochenstunden" ;
    rdfs:domain :Vorlesung ;
    rdfs:range xsd:integer .

:gelesenVon a rdf:Property ;
    rdfs:label "gelesen von" ;
    rdfs:domain :Vorlesung ;
    rdfs:range :Professor .

:fachgebiet a rdf:Property ;
    rdfs:label "Fachgebiet" ;
    rdfs:domain :Assistent ;
    rdfs:range xsd:string .

:boss a rdf:Property ;
    rdfs:label "Boss" ;
    rdfs:domain :Assistent ;
    rdfs:range :Professor .

:hatVoraussetzung a rdf:Property ;
    rdfs:label "hat Voraussetzung" ;
    rdfs:domain :Vorlesung ;
    rdfs:range :Vorlesung .

:hört a rdf:Property ;
```

```

    rdfs:label "hört" ;
    rdfs:domain :Student ;
    rdfs:range :Vorlesung .

:prüfer a rdf:Property ;
    rdfs:label "Prüfer" ;
    rdfs:domain :Prüfung ;
    rdfs:range :Professor .

:geprüfterStudent a rdf:Property ;
    rdfs:label "geprüfter Student" ;
    rdfs:domain :Prüfung ;
    rdfs:range :Student .

:geprüfteVorlesung a rdf:Property ;
    rdfs:label "geprüfte Vorlesung" ;
    rdfs:domain :Prüfung ;
    rdfs:range :Vorlesung .

:note a rdf:Property ;
    rdfs:label "Note" ;
    rdfs:domain :Prüfung ;
    rdfs:range xsd:string .

# Studenten
:student24002 a :Student ;
    :matrNr 24002 ;
    :name "Xenokrates" ;
    :semester 18 .

:student25403 a :Student ;
    :matrNr 25403 ;
    :name "Jonas" ;
    :semester 12 .

:student26120 a :Student ;
#    :matrNr 26120 ;
    :name "Fichte" ;
    :semester 10 .

:student26830 a :Student ;
    :matrNr 26830 ;
    :name "Aristoxenos" ;
    :semester 8 .

:student27550 a :Student ;
    :matrNr 27550 ;
    :name "Schopenhauer" ;
    :semester 6 .

:student28106 a :Student ;
    :matrNr 28106 ;
    :name "Carnap" ;
    :semester 3 .

:student29120 a :Student ;
    :matrNr 29120 ;
    :name "Theophrastos" ;
    :semester 2 .

:student29555 a :Student ;
    :matrNr 29555 ;
    :name "Feuerbach" ;
    :semester 2 .

# Professoren
:professor2125 a :Professor ;
    :persNr 2125 ;
    :name "Sokrates" ;
    :rang "W3" ;
    :raum 226 .

:professor2126 a :Professor ;
    :persNr 2126 ;
    :name "Russel" ;
    :rang "W3" ;
    :raum 232 .

:professor2127 a :Professor ;
    :persNr 2127 ;
    :name "Kopernikus" ;
    :rang "W2" ;
    :raum 310 .

:professor2133 a :Professor ;
    :persNr 2133 ;
    :name "Popper" ;
    :rang "W2" ;
    :raum 52 .

```

```

:professor2134 a :Professor ;
    :persNr 2134 ;
    :name "Augustinus" ;
    :rang "W2" ;
    :raum 309 .

:professor2136 a :Professor ;
    :persNr 2136 ;
    :name "Curie" ;
    :rang "W3" ;
    :raum 36 .

:professor2137 a :Professor ;
    :persNr 2137 ;
    :name "Kant" ;
    :rang "W3" ;
    :raum 7 .

# Vorlesungen
:vorlesung5001 a :Vorlesung ;
    :vorlNr 5001 ;
    :titel "Grundzüge" ;
    :sws 4 ;
    :gelesenVon :professor2137 .

:vorlesung5041 a :Vorlesung ;
    :vorlNr 5041 ;
    :titel "Ethik" ;
    :sws 4 ;
    :gelesenVon :professor2125 .

:vorlesung5043 a :Vorlesung ;
    :vorlNr 5043 ;
    :titel "Einführung in die Erkenntnistheorie" ;
    :sws 3 ;
    :gelesenVon :professor2126 .

:vorlesung5049 a :Vorlesung ;
    :vorlNr 5049 ;
    :titel "Mäeutik" ;
    :sws 2 ;
    :gelesenVon :professor2125 .

:vorlesung4052 a :Vorlesung ;
    :vorlNr 4052 ;
    :titel "Logik" ;
    :sws 4 ;
    :gelesenVon :professor2125 .

:vorlesung5052 a :Vorlesung ;
    :vorlNr 5052 ;
    :titel "Wissenschaftstheorie" ;
    :sws 3 ;
    :gelesenVon :professor2126 .

:vorlesung5216 a :Vorlesung ;
    :vorlNr 5216 ;
    :titel "Bioethik" ;
    :sws 2 ;
    :gelesenVon :professor2126 .

:vorlesung5259 a :Vorlesung ;
    :vorlNr 5259 ;
    :titel "Der Wiener Kreis" ;
    :sws 2 ;
    :gelesenVon :professor2133 .

:vorlesung5022 a :Vorlesung ;
    :vorlNr 5022 ;
    :titel "Glaube und Wissen" ;
    :sws 2 ;
    :gelesenVon :professor2134 .

:vorlesung4630 a :Vorlesung ;
    :vorlNr 4630 ;
    :titel "Die 3 Kritiken" ;
    :sws 4 ;
    :gelesenVon :professor2137 .

# Assistenten
:assistent3002 a :Assistent ;
    :persNr 3002 ;
    :name "Platon" ;
    :fachgebiet "Ideenlehre" ;
    :boss :professor2125 .

:assistent3003 a :Assistent ;
    :persNr 3003 ;
    :name "Aristoteles" ;
    :fachgebiet "Syllogistik" ;
    :boss :professor2125 .

```

```

:boss :professor2126 .

:assistant3004 a :Assistent ;
  :persNr 3004 ;
  :name "Wittgenstein" ;
  :fachgebiet "Sprachtheorie" ;
  :boss :professor2126 .

:assistant3005 a :Assistent ;
  :persNr 3005 ;
  :name "Rhetikus" ;
  :fachgebiet "Planetenbewegung" ;
  :boss :professor2127 .

:assistant3006 a :Assistent ;
  :persNr 3006 ;
  :name "Newton" ;
  :fachgebiet "Keplersche Gesetze" ;
  :boss :professor2127 .

:assistant3007 a :Assistent ;
  :persNr 3007 ;
  :name "Spinoza" ;
  :fachgebiet "Gott und Natur" ;
  :boss :professor2126 .

# Voraussetzungen
:vorlesung5041 :hatVoraussetzung :vorlesung5001 .
:vorlesung5043 :hatVoraussetzung :vorlesung5001 .
:vorlesung5049 :hatVoraussetzung :vorlesung5001 .
:vorlesung5216 :hatVoraussetzung :vorlesung5041 .
:vorlesung5052 :hatVoraussetzung :vorlesung5043, :vorlesung5041 .
:vorlesung5259 :hatVoraussetzung :vorlesung5052 .

:vorlesungXXXX a :Vorlesung ; :titel "Advanced Logik"@en ; :hatVoraussetzung :vorlesung5021 . # Für Property Path Beispiel
:vorlesung0042 a :Vorlesung ; # Dummy Vorlesung für Negation Beispiel
  :vorlNr 0042 ;
  :titel "Spezialthemen" ;
  :sws 2 ;
  :gelesenVon :professor2126 .

# Hört-Beziehungen
# :student26120 :hört :vorlesung5001 .
# Das student26120 hört keine Vorlesung - das Faktum wird auskommentiert. Wird später für ein Beispiel (Filte NOT EXISTS) ge

:student27550 :hört :vorlesung5001, :vorlesung4052 .
:student28106 :hört :vorlesung5041, :vorlesung5052, :vorlesung5216, :vorlesung5259 .
:student29120 :hört :vorlesung5001, :vorlesung5041, :vorlesung5049 .
:student29555 :hört :vorlesung5022 .
:student25403 :hört :vorlesung5022 .

# Prüfungen
:prüfung1 a :Prüfung ;
  :geprüfterStudent :student28106 ;
  :geprüfteVorlesung :vorlesung5001 ;
  :prüfer :professor2126 ;
  :note "1.0" .

:prüfung2 a :Prüfung ;
  :geprüfterStudent :student25403 ;
  :geprüfteVorlesung :vorlesung5041 ;
  :prüfer :professor2125 ;
  :note "2.0" .

```

SPARQL Tutorial: Eine praktische Einführung in die RDF-Abfragesprache

Dieses Jupyter Notebook bietet eine Einführung in SPARQL (SPARQL Protocol and RDF Query Language - in der Tradition von rekursiven Namen wie [GNU](#)). Es richtet sich an Studierende mit Grundkenntnissen in RDF, RDFS (siehe Notebook 01, 03) und der Turtle-Syntax (Notebook 02). Das Tutorial vermittelt die wichtigsten SPARQL-Konzepte anhand eines Beispieldatensatzes zum Thema Universität und zeigt praktische Beispiele sowohl für lokale Daten als auch für öffentliche SPARQL-Endpunkte. Der erste Teil des Tutorials - dieser hier - vermittelt grundlegende Kenntnisse. Der zweite - optionale - Teil geht weiter und vermittelt fortgeschrittene Konzepte wie PropertyPath, föderale Anfragen und mehr. Der dritte - kurze - Teil fasst nochmal alles zusammen und gibt auch weitere Ressourcen an.

Vorbereitung: Ausführen von SPARQL-Abfragen im Notebook

Um die folgenden SPARQL-Abfragen direkt in diesem Jupyter Notebook auszuführen, verwenden wir die Jupyter-Erweiterung `jupyter-rdfify` (die Sie schon kennen und die intern z.B. die Python-Bibliothek `rdflib` nutzt).

Wir nehmen an, dass unser **Beispieldatensatz uni** (basierend auf den Inhalten von `04-RDF-Example.ipynb`) bereits initialisiert und in das Notebook geladen wurde. **Wichtig:** Falls Sie das Notebook neu starten, stellen Sie sicher, dass die Setup-Zellen, die den `uni`-Graphen laden (am Anfang des Notebooks), ausgeführt wurden, damit dieser lokale Graph für Abfragen verfügbar ist. Außerdem muss die `%rdf`-Magic verfügbar sein.

Das `%rdf sparql` Magic Command, das durch `jupyter-rdfify` bereitgestellt wird (und für das wir oft einen Alias wie `%rdfsparql` im Setup definieren), ermöglicht das Abfragen von:

1. **Externen SPARQL-Endpunkten** (öffentliche Web-Services, die SPARQL-Anfragen entgegennehmen).
2. **Lokal im Notebook geladenen RDF-Graphen.**

1. Abfragen externer SPARQL-Endpunkte

Um einen existierenden SPARQL-Endpunkt abzufragen, wird in unserer Notebook-Konfiguration der Parameter `-e` (oder `--endpoint`) verwendet.

Beispiel: Abfrage des Wikidata-Endpunkts Die folgende Abfrage sucht nach bis zu 10 Instanzen von "Katze" (`wd:Q146`) auf Wikidata und gibt deren Item-IRI sowie das Label zurück. Der `SERVICE`-Block sorgt dafür, dass die Labels in der bevorzugten Sprache (oder Englisch) angezeigt werden.

```
%rdf sparql -e https://query.wikidata.org/sparql
# Viele öffentliche Endpunkte wie Wikidata haben gängige Präfixe vordefiniert.
# Für eigene Abfragen werden wir Präfixe meist explizit deklarieren.
SELECT ?item ?itemLabel WHERE {
  ?item wdt:P31 wd:Q146. # wd:Q146 ist das Wikidata-Item für "Katze"
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
} LIMIT 10
```

2. Abfragen lokal geladener Graphen

`jupyter-rdfify` erlaubt auch das Abfragen von Graphen, die zuvor im Notebook geladen und mit einem **Label** (Namen) versehen wurden. In unserer Notebook-Konfiguration verwenden wir hierfür den Parameter `-l` (oder `--local`).

Beispiel: Abfrage unseres lokalen uni-Graphen Angenommen, der Graph mit unseren Universitätsdaten wurde mit dem Label `uni` geladen (z.B. durch eine Zelle wie `%rdf turtle -l uni --data uni_data_sparql_ttl`, wobei `uni_data_sparql_ttl` die Turtle-Daten als Python-String enthält).

Die folgende einfache Abfrage sucht im Graphen `uni` nach Ressourcen, die vom Typ `:Student` sind, sowie deren Namen (`:name`), und gibt diese paarweise zurück (limitiert auf 10 Ergebnisse).

```
%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/> # Namespace für unseren uni-Graphen

SELECT ?person ?name
WHERE {
  ?person a :Student ; :name ?name . # 'a' ist die Abkürzung für rdf:type
}
LIMIT 10
```

Fokus dieses Notebooks: Im Folgenden werden die meisten SPARQL-Beispiele in diesem Notebook den lokalen Graphen `uni` abfragen. Gelegentlich (im optionalen, fortgeschrittenen Teil) werden wir auch externe Endpunkte wie DBpedia oder Wikidata für Vergleiche oder spezifische Beispiele heranziehen. Probieren Sie doch die beiden Anfragen in den nächsten Zellen gleich einmal aus!

In [4]:

```
%rdf sparql -e https://query.wikidata.org/sparql
# Viele öffentliche Endpunkte wie Wikidata haben gängige Präfixe vordefiniert.
# Für eigene Abfragen werden wir Präfixe meist explizit deklarieren.
SELECT ?item ?itemLabel WHERE {
  ?item wdt:P31 wd:Q146. # wd:Q146 ist das Wikidata-Item für "Katze"
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
} LIMIT 10
```

item	itemLabel
<http://www.wikidata.org/entity/Q378619>	CC@en
<http://www.wikidata.org/entity/Q498787>	Muezza@en
<http://www.wikidata.org/entity/Q677525>	Orangey@en
<http://www.wikidata.org/entity/Q893453>	Unsinkable Sam@en
<http://www.wikidata.org/entity/Q1050083>	Catmando@en
<http://www.wikidata.org/entity/Q1185550>	Oscar@en
<http://www.wikidata.org/entity/Q1201902>	Tama@en
<http://www.wikidata.org/entity/Q1207136>	Dewey Readmore Books@en

<http://www.wikidata.org/entity/Q1371145>

Socks@en

<http://www.wikidata.org/entity/Q1386318>

F. D. C. Willard@en

In [5]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/> # Namespace für unseren uni-Graphen

SELECT ?person ?name
WHERE {
  ?person a :Student ; :name ?name . # 'a' ist die Abkürzung für rdf:type
}
LIMIT 10
```

	?person	?name
	http://data.rwth-aachen.de/resources/student24002	Xenokrates
	http://data.rwth-aachen.de/resources/student25403	Jonas
	http://data.rwth-aachen.de/resources/student26120	Fichte
	http://data.rwth-aachen.de/resources/student26830	Aristoxenos
	http://data.rwth-aachen.de/resources/student27550	Schopenhauer
	http://data.rwth-aachen.de/resources/student28106	Carnap
	http://data.rwth-aachen.de/resources/student29120	Theophrastos
	http://data.rwth-aachen.de/resources/student29555	Feuerbach

Was ist SPARQL und warum ist es nützlich?

SPARQL ist die **standardisierte Abfragesprache für RDF-Daten**. Ähnlich wie SQL für relationale Datenbanken ermöglicht SPARQL das Abfragen, Filtern und Manipulieren von RDF-Graphen. SPARQL wurde vom W3C entwickelt und ist seit 2008 ein offizieller W3C-Standard ([SPARQL 1.0 Query Language](#)). Die heute maßgebliche und erweiterte Version ist **SPARQL 1.1** ([Übersicht zu SPARQL 1.1 von 2013](#)). Eine neue Version 1.2 ist in Vorbereitung [SPARQL 1.2](#). Für uns ist aber die Version 1.1 massgeblich.

Warum SPARQL? Die Vorteile auf einen Blick

SPARQL wurde speziell für die Abfrage von Daten entwickelt, die als RDF-Graph modelliert sind, und bietet daher einige Vorteile:

- **Semantische Abfragen:** SPARQL-Anfragen basieren auf der Graphenstruktur und den in RDF und RDFS definierten Bedeutungen (Semantik) der Daten.
- **Flexible Datenintegration:** Es kann Daten aus verschiedenen, heterogenen RDF-Quellen kombinieren und abfragen, als wären sie Teil eines einzigen großen Graphen.
- **Standardisiert:** Als W3C-Standard bietet es eine einheitliche Syntax und einheitliches Verhalten über verschiedene RDF-Datenbanken (Triple Stores) und Werkzeuge hinweg.
- **Ausdrucksstark und Mächtig:** Unterstützt komplexe Graphmuster, optionale Teile, Filter, Aggregationen und kann über Daten inklusive abgeleiteter (inferierter) Aussagen (z.B. aus RDFS oder OWL) ausgeführt werden.

SPARQL vs. SQL - Ein konzeptioneller Vergleich

Für Studierende mit Vorkenntnissen in SQL kann ein Vergleich helfen, die Besonderheiten von SPARQL einzuordnen:

Aspekt	SQL (Relationale Datenbanken)	SPARQL (RDF-Graphen)
Primäres Datenmodell	Tabellen (Relationen) mit Zeilen & Spalten	Graph mit Knoten (IRIs, Blank Nodes, Literale) & Kanten (Prädikate als IRIs)
Schema	Fest vordefiniert (Schema-on-Write)	Flexibel; Schema optional durch RDFS/OWL (Schema-on-Read / Schema-later)
Abfrageeinheit	Zeilen/Tupel aus Tabellen	Tripel-Muster / Subgraphen
Verknüpfungen (Joins)	Explizite JOIN -Syntax	Implizit durch gemeinsame Variablen in Graphmustern
Fehlende Werte	NULL -Werte und deren spezielle Behandlung	OPTIONAL -Klausel für optionale Graphmuster
Standardisierung	ISO/ANSI SQL	W3C SPARQL

Grundlagen: Die ersten SPARQL-Abfragen

Nachdem wir die konzeptionellen Grundlagen und die Motivation für SPARQL betrachtet haben, steigen wir nun in die praktische Anwendung ein.

Die grundlegende SELECT -Abfrage und die FROM -Klausel

Die häufigste Art von SPARQL-Anfrage ist die SELECT -Anfrage. Sie dient dazu, Daten aus einem RDF-Graphen abzurufen und in tabellarischer Form zurückzugeben. Die erweiterte Grundstruktur sieht oft so aus:

```

SELECT ?variable1 ?variable2 ...
FROM <graphIRI>
WHERE {
    # Graphmuster, das im RDF-Graph gefunden werden soll`
    # (eine oder mehrere Tripel-Vorlagen)`
}
# Optionale Modifikatoren (z.B. LIMIT, ORDER BY)`

```

- Die **SELECT -Klausel** listet die Variablen auf (beginnend mit `?` oder `$`), deren gebundene Werte wir als Ergebnis sehen möchten. `SELECT *` gibt alle Variablen zurück, die im `WHERE`-Teil verwendet werden.
- Die **FROM <graphIRI> -Klausel** (optional bei nur einem Default-Graph im System) spezifiziert, welcher **Named Graph** als Default-Graph für die `WHERE`-Klausel verwendet werden soll. In vielen Systemen gibt es einen globalen Default-Graph; `FROM` erlaubt es, diesen explizit zu setzen oder aus einem oder mehreren Named Graphs zusammenzusetzen.
- Die **WHERE -Klausel** enthält ein oder mehrere **Tripel-Muster** (auch Graph-Muster genannt). Diese Muster werden mit den Tripeln im RDF-Graphen (definiert durch die `FROM`-Klausel oder den systemweiten Default-Graphen) abgeglichen.
- Variablen in diesen Mustern werden vom SPARQL-Prozessor an passende IRIs, Literale oder Blank Nodes im Graphen **gebunden**. Jede erfolgreiche Bindung aller Variablen in der `WHERE`-Klausel für ein gegebenes Muster-Set bildet eine **Lösung** (solution). Die `SELECT`-Anfrage gibt dann eine Menge dieser Lösungen zurück.

Hinweis zur FROM -Klausel in diesem Notebook: Wenn wir `%%rdf sparql -l uni` verwenden, legt der Parameter `-l uni` bereits fest, dass unser lokal geladener Graph `uni` als Default-Graph für die Anfrage dient. Eine explizite `FROM`-Klausel, die auf diesen spezifischen lokalen Graphen verweist, ist daher innerhalb des SPARQL-Codes in der Zelle technisch oft nicht zwingend erforderlich, da der Kontext schon gesetzt ist. Um einen *lokalen* Graphen wie `uni` mit `FROM` zu adressieren, bräuchte man eine IRI, die diesen Graphen *als Graphen* benennt, z.B. `<urn:graph:uni>`. In der Praxis mit `-l uni` ist dies aber meist nicht nötig.

Beispiel: Alle Studenten und ihre Namen finden

Wir wollen alle Ressourcen finden, die vom Typ `:Student` sind, und zusätzlich deren Namen ermitteln.

```

%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?student ?name
# FROM <urn:graph:uni> # Konzeptuell: Wir fragen den Graphen 'uni' ab - hier der Default graph.
# Bei %%rdf sparql -l uni ist dies implizit und die Zeile oft unnötig.

WHERE {
    ?student rdf:type :Student ; # Findet alle ?student, die vom Typ :Student sind
    :name ?name . # und für diese ?student den Wert von :name in ?name bindet
    # 'a' kann ebenfalls als Abkürzung für rdf:type verwendet werden
}
LIMIT 10 # Begrenzt die Ausgabe auf die ersten 10 Ergebnisse

```

Was passiert hier?

1. `PREFIX : <...>` definiert einen Namespace. In diesem Fall verwenden wir `:` für Begriffe aus unserem Universitätsvokabular.
2. `SELECT ?student ?name` legt fest, dass wir die Werte für die Variablen `?student` und `?name` in unserer Ergebnistabelle sehen wollen.
3. Die (hier auskommentierte) `FROM <urn:graph:uni>`-Klausel würde in einem allgemeinen SPARQL-Kontext den Graphen spezifizieren, der abgefragt wird. Da wir `-l uni` verwenden, ist der Kontext bereits auf unseren lokalen `uni`-Graphen gesetzt.
4. Die `WHERE`-Klausel enthält zwei Tripel-Muster, die über das gemeinsame Subjekt `?student` implizit verknüpft sind:
 - `?student rdf:type :Student .` : Findet alle Subjekte (`?student`), die die Eigenschaft `rdf:type` mit dem Objekt `:Student` haben.
 - `?student :name ?name .` : Für jedes so gefundene `?student` wird versucht, eine `:name`-Eigenschaft zu finden und deren Wert an die Variable `?name` zu binden.
5. `LIMIT 10` beschränkt die Anzahl der zurückgegebenen Lösungen.

Das Ergebnis dieser Anfrage ist eine Tabelle mit den Spalten `student` und `name`, die die IRIs der Studenten und ihre zugehörigen Namen auflistet (maximal 10).

In [6]:

```

%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?student ?name
# FROM <urn:graph:uni> # Konzeptuell: Wir fragen den Graphen 'uni' ab - hier der Default graph.
# Bei %%rdf sparql -l uni ist dies implizit und die Zeile oft unnötig.

WHERE {
    ?student rdf:type :Student ; # Findet alle ?student, die vom Typ :Student sind
    :name ?name . # und für diese ?student den Wert von :name in ?name bindet
    # 'a' kann ebenfalls als Abkürzung für rdf:type verwendet werden
}
LIMIT 10 # Begrenzt die Ausgabe auf die ersten 10 Ergebnisse

```

student	name
---------	------

http://data.rwth-aachen.de/resources/student24002	Xenokrates
http://data.rwth-aachen.de/resources/student25403	Jonas
http://data.rwth-aachen.de/resources/student26120	Fichte
http://data.rwth-aachen.de/resources/student26830	Aristoxenos
http://data.rwth-aachen.de/resources/student27550	Schopenhauer
http://data.rwth-aachen.de/resources/student28106	Carnap
http://data.rwth-aachen.de/resources/student29120	Theophrastos
http://data.rwth-aachen.de/resources/student29555	Feuerbach

Übung 1: Basis-Abfragen

Aufgabe 1: Schreiben Sie eine SPARQL-Abfrage, die alle Professoren und ihre Namen zurückgibt. (Hinweis - orientieren Sie sich an die Abfrage in der Zelle obendrüber!)

In []: `%%rdf sparql -l uni`

Kombinieren von Tripel-Mustern: Basic Graph Patterns (BGPs)

Die `WHERE`-Klausel einer SPARQL-Anfrage enthält ein sogenanntes **Graphmuster**. Die einfachste Form eines Graphmusters ist ein einzelnes Tripel-Muster, wie wir es im ersten Beispiel gesehen haben (obwohl unser Beispiel dort technisch schon zwei Tripel-Muster für dasselbe Subjekt kombinierte).

In den meisten Fällen besteht das Graphmuster in der `WHERE`-Klausel aus einer **Menge von Tripel-Mustern**. Diese Menge wird als **Basic Graph Pattern (BGP)** bezeichnet.

Wichtige Eigenschaften von BGPs:

- Implizites UND:** Alle Tripel-Muster innerhalb eines BGPs müssen im RDF-Graphen übereinstimmende Tripel finden, damit eine Gesamtlösung für das BGP entsteht. Die Muster sind also implizit mit einem logischen UND verknüpft.
- Implizite Joins durch gemeinsame Variablen:** Wenn dieselbe Variable in mehreren Tripel-Mustern eines BGPs vorkommt, muss diese Variable in allen diesen Mustern an *dasselbe* RDF-Term (dieselbe IRI, dasselbe Literal oder denselben Blank Node) gebunden werden. Dies ist die Art und Weise, wie in SPARQL implizite "Joins" oder Verknüpfungen zwischen verschiedenen Teilen des Graphen hergestellt werden.

Syntax-Abkürzungen in Graphmustern (aus Turtle bekannt)

Um SPARQL-Anfragen (insbesondere die `WHERE`-Klausel) kompakter und lesbarer zu gestalten, übernimmt SPARQL einige nützliche Abkürzungssyntaxen direkt aus Turtle:

- Semikolon (;):** Dient dazu, mehrere Prädikat-Objekt-Paare für dasselbe Subjekt aufzulisten. Das Subjekt muss dann nicht für jedes Prädikat-Objekt-Paar wiederholt werden. Es schließt die vorherige Prädikat-Objekt-Gruppe ab und leitet eine neue für dasselbe Subjekt ein.
 - Statt:


```
?student rdf:type :Student .
?student :name ?studentName .
```
 - Kann man schreiben:


```
?student rdf:type :Student ;
:name ?studentName .
```
- Komma (,):** Dient dazu, mehrere Objekte für dasselbe Subjekt UND dasselbe Prädikat aufzulisten. Subjekt und Prädikat müssen dann nicht für jedes Objekt wiederholt werden.
 - Statt:


```
dbr:Aachen rdfs:label "Aachen"@de .
dbr:Aachen rdfs:label "Oche"@ksh .
```
 - Kann man schreiben:


```
dbr:Aachen rdfs:label "Aachen"@de, "Oche"@ksh .
```
- a als Abkürzung für rdf:type:** Wie in Turtle ist auch in SPARQL das Schlüsselwort `a` eine Abkürzung für das Prädikat `rdf:type`.
 - `?student a :Student .` ist äquivalent zu `?student rdf:type :Student .`

Diese Abkürzungen werden wir in den folgenden Beispielen häufig verwenden, da sie Standard in SPARQL sind und die Lesbarkeit deutlich erhöhen.

Beispiel: Studenten und die Titel der von ihnen gehörten Vorlesungen

Wir möchten eine Liste von Studentennamen und den Titeln der Vorlesungen, die diese Studenten hören. Dafür müssen wir Informationen über Studenten, ihre Beziehung zu Vorlesungen und die Titel dieser Vorlesungen verknüpfen.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName ?vorlesungTitel
WHERE {
    ?student rdf:type :Student ;           # Finde eine Instanz ?student vom Typ :Student
            :name ?studentName ;          # Binde den Namen dieses Studenten an ?studentName
            :hört ?vorlesung .             # Finde die Vorlesung ?vorlesung, die dieser Student hört

    ?vorlesung :titel ?vorlesungTitel .    # Für diese ?vorlesung, finde den Titel und binde ihn an ?vorlesungTitel
}
LIMIT 20
```

Was passiert in dieser Abfrage?

1. **SELECT ?studentName ?vorlesungTitel** : Wir wollen die Namen der Studenten und die Titel der Vorlesungen sehen.
2. **WHERE { ... }** : Die Klausel enthält ein BGP, das aus vier Tripel-Mustern besteht:
 - Die ersten drei Muster beziehen sich auf die Variable **?student** :
 - **?student rdf:type :Student .**
 - **?student :name ?studentName .**
 - **?student :hört ?vorlesung .** Diese drei Muster müssen *alle gemeinsam* für eine bestimmte Ressource (die an **?student** gebunden wird) im Graphen matchen. Dabei wird **?studentName** an den Namen und **?vorlesung** an die IRI der gehörten Vorlesung gebunden.
 - Das vierte Tripel-Muster ist:
 - **?vorlesung :titel ?vorlesungTitel .** Dieses Muster ist über die **gemeinsame Variable ?vorlesung** mit dem vorherigen Muster **?student :hört ?vorlesung .** verbunden. Der SPARQL-Prozessor wird also nur solche Lösungen finden, bei denen die im dritten Muster an **?vorlesung** gebundene Vorlesungs-IRI auch die Subjekt-IRI im vierten Muster ist. Der Titel dieser spezifischen Vorlesung wird dann an **?vorlesungTitel** gebunden.
3. **LIMIT 20** : Begrenzt die Ausgabe.

Das Ergebnis ist eine Tabelle, die Paare von Studentennamen und den Titeln der von ihnen besuchten Vorlesungen anzeigt.

In [8]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName ?vorlesungTitel
WHERE {
    ?student rdf:type :Student ;           # Finde eine Instanz ?student vom Typ :Student
            :name ?studentName ;          # Binde den Namen dieses Studenten an ?studentName
            :hört ?vorlesung .             # Finde die Vorlesung ?vorlesung, die dieser Student hört

    ?vorlesung :titel ?vorlesungTitel .    # Für diese ?vorlesung, finde den Titel und binde ihn an ?vorlesungTitel
}
LIMIT 20
```

?studentName	?vorlesungTitel
Jonas	Glaube und Wissen
Schopenhauer	Grundzüge
Schopenhauer	Logik
Carnap	Ethik
Carnap	Wissenschaftstheorie
Carnap	Bioethik
Carnap	Der Wiener Kreis
Theophrastos	Grundzüge
Theophrastos	Ethik
Theophrastos	Mäeutik
Feuerbach	Glaube und Wissen

Übung 2: Erweiterung der obigen Abfrage

Aufgabe 1: Erweitern Sie die obige SPARQL-Abfrage über Studenten und Vorlesungstitel, sodass auch die Namen der die Vorlesung lesenden Professoren aufgelistet werden.

```
In [ ]: %%rdf sparql -l uni
```

Einschränken von Ergebnissen mit FILTER

Bisher haben unsere Anfragen alle Lösungen zurückgegeben, die auf die Graphmuster in der `WHERE`-Klausel passen. Oft möchten wir diese Ergebnisse jedoch weiter einschränken, basierend auf bestimmten Bedingungen, die die Werte der Variablen erfüllen müssen. Hierfür dient die `FILTER`-Klausel.

Ein `FILTER` Ausdruck wertet eine Bedingung aus. Wenn die Bedingung für eine potentielle Lösung `true` (wahr) ergibt, wird die Lösung beibehalten. Ergibt sie `false` (falsch) oder einen Fehler, wird die Lösung verworfen.

`FILTER` werden innerhalb der `WHERE`-Klausel platziert, typischerweise nach den Graphmustern, auf deren Variablen sie sich beziehen.

Filtern mit Vergleichsoperatoren

Wir können numerische Werte, Strings oder auch andere Datentypen mit gängigen Operatoren vergleichen:

- Numerische Vergleiche: `=`, `!=`, `<`, `>`, `<=`, `>=`
- String-Vergleiche (lexikographisch): `=`, `!=`, `<`, `>`, `<=`, `>=` (beachten Sie, dass String-Vergleiche Groß-/Kleinschreibung berücksichtigen, sofern nicht anders behandelt)

Beispiel: Studenten in höheren Semestern finden

Wir suchen alle Studenten, die in einem Semester größer als 2 sind, und geben ihren Namen und ihr Semester aus.

```
%%rdf sparql -l uni
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> # Für Datentypen bei Vergleichen wichtig

SELECT ?studentName ?semesterAnzahl
WHERE {
  ?student a :Student ;
           :name ?studentName ;
           :semester ?semesterAnzahl .

  FILTER (?semesterAnzahl > 2) # Die Bedingung für das Semester
}
ORDER BY DESC(?semesterAnzahl) # Optional: Sortieren wir nach Semester absteigend
```

Erläuterung:

- Die `WHERE`-Klausel findet zunächst alle Studenten und deren Semesteranzahl.
- Der `FILTER (?semesterAnzahl > 2)`-Ausdruck wird dann auf jede dieser potentiellen Lösungen angewendet. Nur wenn der Wert der Variablen `?semesterAnzahl` (die als `xsd:integer` interpretiert wird) größer als 2 ist, wird die Lösung (also das Paar `?studentName`, `?semesterAnzahl`) in das Endergebnis aufgenommen.
- `ORDER BY DESC(?semesterAnzahl)` sortiert die Ausgabe zusätzlich.

```
In [10]: %%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> # Für Datentypen bei Vergleichen wichtig

SELECT ?studentName ?semesterAnzahl
WHERE {
  ?student a :Student ;
           :name ?studentName ;
           :semester ?semesterAnzahl .

  FILTER (?semesterAnzahl > 2) # Die Bedingung für das Semester
}
ORDER BY DESC(?semesterAnzahl) # Optional: Sortieren wir nach Semester absteigend
```

?studentName ?semesterAnzahl

Xenokrates	18
Jonas	12
Fichte	10
Aristoxenos	8
Schopenhauer	6
Carnap	3

Filtern mit String-Funktionen: REGEX und STRSTARTS / STRENDS

SPARQL bietet Funktionen, um Strings zu untersuchen. Besonders nützlich ist `REGEX` für reguläre Ausdrücke.

- `REGEX(?variable, "muster", "flags")` : Prüft, ob der String-Wert von `?variable` auf das "muster" (ein regulärer Ausdruck) passt. "flags" können z.B. "i" für case-insensitivity sein.
- `STRSTARTS(?variable, "präfix")` : Prüft, ob der String-Wert von `?variable` mit "präfix" beginnt.
- `STRENDS(?variable, "suffix")` : Prüft, ob der String-Wert von `?variable` mit "suffix" endet.

Beispiel 4: Professoren finden, deren Name mit "S" beginnt

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?profName
WHERE {
  ?prof a :Professor ;
        :name ?profName .

  FILTER (STRSTARTS(LCASE(?profName), "s")) # Name beginnt mit 's' (case-insensitive)
  # Alternative mit REGEX für case-insensitivity:
  # FILTER REGEX(?profName, "^S", "i")
}
```

Erläuterung:

- `LCASE(?profName)` wandelt den Namen zuerst in Kleinbuchstaben um, damit der Vergleich mit "s" unabhängig von der Groß-/Kleinschreibung des ersten Buchstabens im Namen funktioniert. `STRSTARTS` selbst ist case-sensitiv.
- Die auskommentierte `REGEX`-Alternative zeigt, wie man dasselbe mit einem regulären Ausdruck erreichen kann, der auf "S" am Anfang des Strings prüft, wobei der Flag "i" für case-insensitivity sorgt.

In [11]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?profName
WHERE {
  ?prof a :Professor ;
        :name ?profName .

  FILTER (STRSTARTS(LCASE(?profName), "s")) # Name beginnt mit 's' (case-insensitive)
  # Alternative mit REGEX für case-insensitivity:
  # FILTER REGEX(?profName, "^S", "i")
}
```

?profName

Sokrates

Filtern basierend auf Sprache: LANGMATCHES

Wenn Literale Sprach-Tags haben, können wir mit `LANGMATCHES(LANG(?variable), "sprachtag")` prüfen, ob ein Literal einen bestimmten Sprach-Tag hat oder einen Sub-Tag davon.

- `LANG(?variable)` : Gibt den Sprach-Tag eines Literals zurück.
- `LANGMATCHES(sprachtagLiteral, "sprachenRange")` : `sprachenRange` kann ein exakter Tag (z.B. "de-DE") oder ein Prefix (z.B. "de", um alle deutschen Varianten wie de-DE, de-AT zu matchen) sein.

Beispiel: Vorlesungstitel auf Englisch finden

Angenommen, Vorlesungstitel können mehrsprachig sein: ``sparql %%rdf sparql -l uni PREFIX : <http://data.rwth-aachen.de/resources/> PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

```
SELECT ?vorlesung ?titel WHERE { ?vorlesung a :Vorlesung ; :titel ?titel . # Annahme: :titel kann mehrsprachige Objekte haben
```

```
FILTER (LANGMATCHES(LANG(?titel), "en")) } ``
```

In [12]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?vorlesung ?titel
WHERE {
    ?vorlesung a :Vorlesung ;
              :titel ?titel . # Annahme: :titel kann mehrsprachige Objekte haben

    FILTER (LANGMATCHES(LANG(?titel), "en"))
}
```

?vorlesung	?titel
------------	--------

http://data.rwth-aachen.de/resources/vorlesungXXXX	Advanced Logik
--	----------------

Existenz von Bindungen prüfen: BOUND und !BOUND

Manchmal möchten wir prüfen, ob eine Variable in einem optionalen Teil einer Abfrage (siehe später OPTIONAL) gebunden wurde oder nicht.

- **BOUND(?variable)** : Ist true, wenn ?variable in der aktuellen Lösung gebunden ist.
- **!BOUND(?variable)** : Ist true, wenn ?variable nicht gebunden ist.

Dieses Konzept wird seine volle Stärke im Zusammenhang mit OPTIONAL zeigen, aber wir führen es hier schon ein. Wir zeigen später ein Beispiel.

Filtern nach Datentyp: DATATYPE

Mit DATATYPE(?variable) können wir die IRI des Datentyps eines Literals abrufen und vergleichen.

Beispiel: Ressourcen finden, deren Name explizit als xsd:string getypt ist

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?resource ?name
WHERE {
    ?resource :name ?name .
    FILTER (DATATYPE(?name) = xsd:string)
}
LIMIT 10
```

(Hinweis: unsere Beispieldaten - nachdem sie aus einer relationalen Datenbank stammen - sind sehr homogen. Ein realer Graph wäre sehr viel heterogener)

In [13]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?resource ?name
WHERE {
    ?resource :name ?name .
    FILTER (DATATYPE(?name) = xsd:string)
}
LIMIT 10
```

?resource	?name
-----------	-------

http://data.rwth-aachen.de/resources/student24002	Xenokrates
http://data.rwth-aachen.de/resources/student25403	Jonas
http://data.rwth-aachen.de/resources/student26120	Fichte
http://data.rwth-aachen.de/resources/student26830	Aristoxenos
http://data.rwth-aachen.de/resources/student27550	Schopenhauer
http://data.rwth-aachen.de/resources/student28106	Carnap
http://data.rwth-aachen.de/resources/student29120	Theophrastos
http://data.rwth-aachen.de/resources/student29555	Feuerbach
http://data.rwth-aachen.de/resources/professor2125	Sokrates
http://data.rwth-aachen.de/resources/professor2126	Russel

Logische Verknüpfungen in Filtern: && (UND), || (ODER), ! (NICHT)

Mehrere Bedingungen können in einem `FILTER` kombiniert werden:

Beispiel: Studenten im 3. oder 4. Semester, deren Name "A" enthält

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName ?semesterAnzahl
WHERE {
  ?student a :Student ;
           :name ?studentName ;
           :semester ?semesterAnzahl .

  FILTER ( (?semesterAnzahl = 3 || ?semesterAnzahl = 4) && CONTAINS(LCASE(?studentName), "a") )
  # CONTAINS ist eine weitere nützliche String-Funktion (case-sensitive, daher LCASE)
}
```

`FILTER` ist ein sehr mächtiges Werkzeug, um die Ergebnismengen präzise auf die relevanten Daten zu reduzieren.

In [14]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName ?semesterAnzahl
WHERE {
  ?student a :Student ;
           :name ?studentName ;
           :semester ?semesterAnzahl .

  FILTER ( (?semesterAnzahl = 3 || ?semesterAnzahl = 4) && CONTAINS(LCASE(?studentName), "a") )
  # CONTAINS ist eine weitere nützliche String-Funktion (case-sensitive, daher LCASE)
}
```

`?studentName` `?semesterAnzahl`

Carnap 3

Filtern basierend auf der Existenz von Graphmustern: `EXISTS` und `NOT EXISTS`

Manchmal möchten wir Lösungen nicht nur basierend auf den Werten einzelner Variablen filtern, sondern auch darauf, ob bestimmte **zusätzliche Graphmuster** für eine gegebene Lösung existieren oder nicht existieren. Hierfür gibt es `EXISTS { ... }` und `NOT EXISTS { ... }`.

- **`EXISTS { graphPattern }`**: Dieser Filter ist `true`, wenn das angegebene `graphPattern` (ein oder mehrere Tripel-Muster) im Graphen gefunden werden kann, unter Berücksichtigung der Variablenbindungen aus dem umgebenden Teil der `WHERE`-Klausel. Variablen, die außerhalb des `EXISTS`-Blocks gebunden wurden, können innerhalb des Musters verwendet werden. Variablen, die *nur* innerhalb des `EXISTS`-Musters gebunden werden, sind außerhalb nicht sichtbar.
- **`NOT EXISTS { graphPattern }`**: Dieser Filter ist `true`, wenn das angegebene `graphPattern` im Graphen **nicht** gefunden werden kann, wiederum unter Berücksichtigung der äußeren Variablenbindungen.

Beispiel: Studenten finden, die mindestens eine Vorlesung hören

Wir möchten alle Studenten und ihre Namen auflisten, aber nur diejenigen, für die auch tatsächlich eine Aussage existiert, dass sie eine Vorlesung hören. (Dies könnte man auch mit einem direkten Join erreichen, aber `EXISTS` ist nützlich für komplexere Existenzprüfungen, bei denen man die Werte aus dem Existenzmuster nicht im `SELECT` benötigt).

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName
WHERE {
  ?student a :Student ;
           :name ?studentName .

  FILTER EXISTS { ?student :hört ?irgendeineVorlesung . }
  # Das ?irgendeineVorlesung muss nicht im SELECT auftauchen, es geht nur um die Existenz der Beziehung.
}
```

Erläuterung zum Beispiel:

- Die äußere `WHERE`-Klausel (`?student a :Student ; :name ?studentName .`) findet zunächst alle Entitäten, die vom Typ `:Student` sind, und bindet deren IRI an `?student` sowie deren Namen an `?studentName`.
- Für jeden so gefundenen `?student` (mit seinem `?studentName`) wird der `FILTER EXISTS` ausgewertet:
 - Das Muster `{ ?student :hört ?irgendeineVorlesung . }` versucht, eine beliebige Vorlesung (`?irgendeineVorlesung`) zu finden, die der aktuell gebundene `?student` über die Eigenschaft `:hört` besucht.

- Wenn mindestens eine solche Vorlesung für den aktuellen `?student` gefunden wird, ergibt `EXISTS { ... }` den Wert `true`. Die Bedingung des `FILTER` ist erfüllt, und der `?studentName` wird für das Endergebnis beibehalten.
- Wenn für den aktuellen `?student` keine solche `:hört`-Beziehung gefunden wird, ergibt `EXISTS { ... }` den Wert `false`. Die Bedingung des `FILTER` ist nicht erfüllt, und der `?studentName` wird verworfen.

In [15]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName
WHERE {
  ?student a :Student ;
          :name ?studentName .

  FILTER EXISTS { ?student :hört ?irgendeineVorlesung . }
  # Das ?irgendeineVorlesung muss nicht im SELECT auftauchen, es geht nur um die Existenz der Beziehung.
}
```

?studentName

```
Jonas
Schopenhauer
Carnap
Theophrastos
Feuerbach
```

Manchmal möchte man Ergebnisse ausschließen, die einem bestimmten Muster entsprechen. `FILTER NOT EXISTS`: Prüft, ob ein Graph-Pattern nicht erfüllt werden kann. Die folgende SPARQL-Anfrage zielt darauf ab, die Namen aller Studenten aus dem lokalen Graphen `uni` zu extrahieren, die nachweislich **keine** Vorlesung hören.

In [16]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName
WHERE {
  ?student a :Student ;
          :name ?studentName .

  FILTER NOT EXISTS { ?student :hört ?irgendeineVorlesung . }
  # Das ?irgendeineVorlesung muss nicht im SELECT auftauchen, es geht nur um die Existenz der Beziehung.
}
```

?studentName

```
Xenokrates
Fichte
Aristoxenos
```

Übung 3: Finden Sie alle Studenten, die *nicht* "Logik" hören und deren Name nicht mit "F" anfangen.

In []:

```
%%rdf sparql -l uni
```

Umgang mit optionalen Informationen: Die `OPTIONAL`-Klausel

In realen RDF-Datensätzen haben nicht immer alle Ressourcen dieselben Eigenschaften. Ein Student hat vielleicht eine E-Mail-Adresse hinterlegt, ein anderer nicht. Wenn wir nun alle Studenten und ihre E-Mail-Adressen abfragen würden und eine E-Mail-Adresse als zwingend erforderlich im Muster definieren, würden Studenten ohne E-Mail-Adresse gar nicht im Ergebnis erscheinen.

Um solche Fälle zu behandeln, in denen bestimmte Informationen vorhanden sein *können*, aber nicht *müssen*, gibt es in SPARQL die `OPTIONAL`-Klausel.

Konzept und Funktionsweise:

Die `OPTIONAL`-Klausel umschließt ein Graphmuster. Der SPARQL-Prozessor versucht, dieses optionale Muster an die bereits gefundenen Lösungen (aus dem nicht-optionalen Teil der `WHERE`-Klausel) zu binden.

- **Wenn das optionale Muster matcht:** Die Variablen aus dem optionalen Muster werden an die entsprechenden Werte gebunden und zur aktuellen Lösung hinzugefügt.
- **Wenn das optionale Muster nicht matcht:** Die Lösung aus dem nicht-optionalen Teil bleibt trotzdem erhalten. Die Variablen, die *nur* im optionalen Muster vorkommen, bleiben für diese Lösung *ungebunden* (unbound).

Dadurch gehen keine Informationen aus dem obligatorischen Teil der Abfrage verloren, nur weil ein optionaler Teil fehlt.

Syntax: OPTIONAL { graphPattern }

Die OPTIONAL -Klausel wird innerhalb der WHERE -Klausel platziert.

Beispiel: Alle Studenten und optional ihre Matrikelnummer

Wir möchten die Namen aller Studenten und, falls vorhanden, ihre Matrikelnummer (:matrNr) ausgeben.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?studentName ?matrikelnummer
WHERE {
    ?student a :Student ;
            :name ?studentName .

    OPTIONAL {
        ?student :matrNr ?matrikelnummer .
    }
}
ORDER BY ?studentName
```

Erläuterung:

- Der erste Teil der WHERE -Klausel (?student a :Student ; :name ?studentName .) findet alle Studenten und ihre Namen. Dies ist der **obligatorische Teil**.
- Der OPTIONAL { ?student :matrNr ?matrikelnummer . } -Block versucht dann, für jeden gefundenen ?student eine Matrikelnummer zu finden.
 - Hat ein Student eine Matrikelnummer, wird ?matrikelnummer gebunden.
 - Hat ein Student keine Matrikelnummer, bleibt ?matrikelnummer für diesen Studenten ungebunden, aber der Student und sein Name erscheinen trotzdem im Ergebnis. In der Ergebnistabelle wird für ungebundene Variablen oft ein leerer Wert oder ein spezieller Marker angezeigt.

In [18]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?studentName ?matrikelnummer
WHERE {
    ?student a :Student ;
            :name ?studentName .

    OPTIONAL {
        ?student :matrNr ?matrikelnummer .
    }
}
ORDER BY ?studentName
```

?studentName ?matrikelnummer

Aristoxenos	26830
Carnap	28106
Feuerbach	29555
Fichte	None
Jonas	25403
Schopenhauer	27550
Theophrastos	29120
Xenokrates	24002

Kombination von OPTIONAL mit FILTER und BOUND()

Die Funktion BOUND(?variable) ist besonders nützlich in Kombination mit OPTIONAL . BOUND(?variable) gibt true zurück, wenn die Variable ?variable in der aktuellen Lösung gebunden ist, andernfalls false .

Beispiel: Studenten, die definitiv KEINE Matrikelnummer haben

Wir können OPTIONAL und FILTER(!BOUND(...)) verwenden, um Ressourcen zu finden, für die eine bestimmte optionale Information explizit fehlt.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName

WHERE {

 ?student a :Student ;
 :name ?studentName .

OPTIONAL { ?student :matrNr ?matrNrValue . } # Versuche, eine Matrikelnummer zu finden

FILTER (!BOUND(?matrNrValue)) # Behalte nur die, bei denen ?matrNrValue NICHT gebunden wurde

}

ORDER BY ?studentName

Erläuterung: Diese Abfrage listet die Namen aller Studenten auf, für die im **OPTIONAL** -Block *keine* Matrikelnummer (?matrNrValue) gefunden und gebunden werden konnte.

In [19]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName
WHERE {
    ?student a :Student ;
            :name ?studentName .

    OPTIONAL { ?student :matrNr ?matrNrValue . } # Versuche, eine Matrikelnummer zu finden
    FILTER (!BOUND(?matrNrValue)) # Behalte nur die, bei denen ?matrNrValue NICHT gebunden wurde
}
ORDER BY ?studentName
```

?studentName

Fichte

Beispiel: Studenten und eine Angabe, ob eine Matrikelnummer vorhanden ist

Manchmal möchte man nicht filtern, sondern explizit anzeigen, ob ein optionaler Wert vorhanden war. Dafür kann man **BIND** mit **BOUND** verwenden:

%%rdf sparql -l uni

PREFIX : <http://data.rwth-aachen.de/resources/>

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName ?matrikelnummer (BOUND(?matrikelnummer) **AS** ?hatMatrikelnummer)

WHERE {

 ?student a :Student ;
 :name ?studentName .

OPTIONAL {

 ?student :matrNr ?matrikelnummer .

 }

}

ORDER BY ?studentName

Erläuterung:

- (BOUND(?matrikelnummer) AS ?hatMatrikelnummer) : Diese Konstruktion in der **SELECT** -Klausel erstellt eine neue Variable ?hatMatrikelnummer .
- Ihr Wert ist **true** , wenn ?matrikelnummer für die jeweilige Lösung gebunden wurde (d.h. eine Matrikelnummer vorhanden ist), und **false** , wenn nicht.
- In der Ergebnistabelle sehen Sie also neben dem Namen und der (potenziell leeren) Matrikelnummer eine zusätzliche Spalte, die **true** oder **false** anzeigt.

In [20]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName ?matrikelnummer (BOUND(?matrikelnummer) AS ?hatMatrikelnummer)
WHERE {
    ?student a :Student ;
            :name ?studentName .

    OPTIONAL {
        ?student :matrNr ?matrikelnummer .
    }
}
ORDER BY ?studentName
```

?studentName ?matrikelnummer ?hatMatrikelnummer

Aristoxenos	26830	true
Carnap	28106	true

Feuerbach	29555	true
Fichte	None	false
Jonas	25403	true
Schopenhauer	27550	true
Theophrastos	29120	true
Xenokrates	24002	true

Mehrere OPTIONAL -Blöcke: Man kann auch mehrere OPTIONAL -Blöcke in einer WHERE -Klausel verwenden. Jeder OPTIONAL -Block wird unabhängig vom anderen gegen die bis dahin gefundenen Lösungen des nicht-optionalen Teils der Anfrage geprüft.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX schema: <http://schema.org/> # Annahme für schema:email

SELECT ?studentName ?matrikelnummer ?email
WHERE {
    ?student a :Student ;
             :name ?studentName .

    OPTIONAL { ?student :matrNr ?matrikelnummer . }
    OPTIONAL { ?student schema:email ?email . } # Annahme: Studenten könnten eine schema:email haben
}
ORDER BY ?studentName
```

In diesem Fall wird versucht, sowohl eine Matrikelnummer als auch eine E-Mail-Adresse für jeden Studenten zu finden. Das Fehlen des einen beeinflusst nicht das Finden des anderen.

Die OPTIONAL -Klausel ist ein fundamentaler Bestandteil von SPARQL, um der Natur von oft unvollständigen oder heterogenen RDF-Daten gerecht zu werden.

In [21]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX schema: <http://schema.org/> # Annahme für schema:email

SELECT ?studentName ?matrikelnummer ?email
WHERE {
    ?student a :Student ;
             :name ?studentName .

    OPTIONAL { ?student :matrNr ?matrikelnummer . }
    OPTIONAL { ?student schema:email ?email . } # Annahme: Studenten könnten eine schema:email haben
}
ORDER BY ?studentName
```

?studentName	?matrikelnummer	?email
Aristoxenos	26830	None
Carnap	28106	None
Feuerbach	29555	None
Fichte	None	None
Jonas	25403	None
Schopenhauer	27550	None
Theophrastos	29120	None
Xenokrates	24002	None

Kombinieren von Ergebnissen verschiedener Muster mit UNION

Bisher haben wir gelernt, wie wir mit Basic Graph Patterns (BGP) nach Strukturen suchen, bei denen *alle* angegebenen Tripel-Muster zutreffen müssen (implizites UND), wie wir mit FILTER Bedingungen setzen und wie wir mit OPTIONAL nach möglicherweise vorhandenen Daten suchen.

Manchmal möchten wir jedoch Ergebnisse finden, die *entweder* einem Satz von Mustern *oder* einem anderen Satz von Mustern entsprechen. Für solche Fälle bietet SPARQL die UNION -Klausel.

Konzept und Funktionsweise:

Die UNION -Klausel kombiniert die Lösungen von zwei oder mehr Graphmustern. Jedes Graphmuster, das durch UNION getrennt ist, wird unabhängig voneinander ausgewertet. Die Ergebnismengen dieser unabhängigen Auswertungen werden dann zu einer einzigen Ergebnismenge zusammengeführt.

Syntax: { graphPattern1 } UNION { graphPattern2 } UNION { ... }

Die UNION -Klausel wird innerhalb der WHERE -Klausel verwendet, um alternative Graphmuster zu definieren.

Beispiel: Alle Studenten UND alle Professoren mit ihren Namen auflisten

Wir möchten eine Liste aller Personen, die entweder Studenten oder Professoren sind, zusammen mit ihren Namen.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?person ?name
WHERE {
  {
    ?person a :Student ;           # Erster Block: Finde Studenten
          :name ?name .
  }
  UNION
  {
    ?person a :Professor ;        # Zweiter Block: Finde Professoren
          :name ?name .
  }
}
ORDER BY ?name
LIMIT 20
```

Erläuterung:

- Der SPARQL-Prozessor wertet zuerst das erste Graphmuster { ?person a :Student ; :name ?name . } aus und findet alle Studenten und ihre Namen.
- Dann wertet er unabhängig davon das zweite Graphmuster { ?person a :Professor ; :name ?name . } aus und findet alle Professoren und ihre Namen.
- Die UNION -Klausel fügt die Ergebnismengen beider Blöcke zusammen.
- Das Ergebnis ist eine Tabelle mit den Spalten ?person und ?name , die sowohl Studenten als auch Professoren enthält.
- ORDER BY ?name sortiert die kombinierte Liste.

In [22]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?person ?name
WHERE {
  {
    ?person a :Student ;           # Erster Block: Finde Studenten
          :name ?name .
  }
  UNION
  {
    ?person a :Professor ;        # Zweiter Block: Finde Professoren
          :name ?name .
  }
}
ORDER BY ?name
LIMIT 20
```

?person	?name
http://data.rwth-aachen.de/resources/student26830	Aristoxenos
http://data.rwth-aachen.de/resources/professor2134	Augustinus
http://data.rwth-aachen.de/resources/student28106	Carnap
http://data.rwth-aachen.de/resources/professor2136	Curie
http://data.rwth-aachen.de/resources/student29555	Feuerbach
http://data.rwth-aachen.de/resources/student26120	Fichte
http://data.rwth-aachen.de/resources/student25403	Jonas
http://data.rwth-aachen.de/resources/professor2137	Kant
http://data.rwth-aachen.de/resources/professor2127	Kopernikus
http://data.rwth-aachen.de/resources/professor2133	Popper
http://data.rwth-aachen.de/resources/professor2126	Russel
http://data.rwth-aachen.de/resources/student27550	Schopenhauer
http://data.rwth-aachen.de/resources/professor2125	Sokrates
http://data.rwth-aachen.de/resources/student29120	Theophrastos
http://data.rwth-aachen.de/resources/student24002	Xenokrates

Wichtige Aspekte bei UNION :

1. **Unabhängige Auswertung:** Die Muster in den durch UNION getrennten Blöcken werden separat ausgewertet. Eine Variable, die in einem Block gebunden wird, beeinflusst nicht direkt die Bindung derselben Variable in einem anderen UNION -Block (außer dass sie Teil derselben SELECT -Zeile wird, wenn sie im SELECT angefordert wird).
2. **Variablenbindung:** Wenn eine Variable, die in der SELECT -Klausel aufgeführt ist, in einem der UNION -Blöcke nicht gebunden wird (weil sie in diesem spezifischen Muster nicht vorkommt), ist sie für die Lösungen aus diesem Block ungebunden.
3. **Duplikate:** UNION behält standardmäßig Duplikate bei, wenn eine Lösung durch mehrere Zweige der UNION erzeugt werden kann (obwohl dies seltener vorkommt als bei SQL UNION ALL , da RDF-Graphen Mengen von Tripeln sind). Wenn man Duplikate in der Ausgabe vermeiden will, verwendet man SELECT DISTINCT .

Beispiel: Entitäten mit verschiedenen Titel-Eigenschaften finden

Angenommen, in unserem uni -Graphen haben Vorlesungen eine Eigenschaft :titel und Assistenten haben eine Eigenschaft :fachgebiet . Wir wollen alle Entitäten und ihre jeweiligen Titel bzw. Fachgebiet auflisten, egal ob Vorlesung oder Assistent.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

# Annahme: Es gibt Instanzen von :Vorlesung mit :titel
# und Instanzen von :Assistent mit:fachgebiet in unserem uni-Graphen.

SELECT ?entitaet ?titel
WHERE {
  {
    ?entitaet a :Vorlesung ;
              :titel ?titel .
  }
  UNION
  {
    ?entitaet a :Assistent ; # Annahme: Die Klasse :Projekt existiert im Schema
                          :fachgebiet ?titel .
  }
}
LIMIT 20
```

Erläuterung:

- Diese Anfrage sucht entweder nach Vorlesungen und ihren :titel oder nach Assistenten und ihren :fachgebiet .
- Die Variable ?titel wird in beiden Blöcken verwendet und in der SELECT -Klausel ausgegeben. Sie enthält dann entweder den Vorlesungstitel oder das Fachgebiet, je nachdem, welcher Teil der UNION eine Lösung gefunden hat.

UNION ist ein mächtiges Konstrukt, um alternative Pfade oder Bedingungen in einem Graphen abzufragen und deren Ergebnisse zu vereinen.

In [23]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

# Annahme: Es gibt Instanzen von :Vorlesung mit :titel
# und Instanzen von :Assistent mit:fachgebiet in unserem uni-Graphen.

SELECT ?entitaet ?titel
WHERE {
  {
    ?entitaet a :Vorlesung ;
              :titel ?titel .
  }
  UNION
  {
    ?entitaet a :Assistent ; # Annahme: Die Klasse :Projekt existiert im Schema
                          :fachgebiet ?titel .
  }
}
LIMIT 20
```

?entitaet	?titel
http://data.rwth-aachen.de/resources/vorlesung5001	Grundzüge
http://data.rwth-aachen.de/resources/vorlesung5041	Ethik
http://data.rwth-aachen.de/resources/vorlesung5043	Einführung in die Erkenntnistheorie
http://data.rwth-aachen.de/resources/vorlesung5049	Mäeutik
http://data.rwth-aachen.de/resources/vorlesung4052	Logik
http://data.rwth-aachen.de/resources/vorlesung5052	Wissenschaftstheorie
http://data.rwth-aachen.de/resources/vorlesung5216	Bioethik

http://data.rwth-aachen.de/resources/vorlesung5259	Der Wiener Kreis
http://data.rwth-aachen.de/resources/vorlesung5022	Glaube und Wissen
http://data.rwth-aachen.de/resources/vorlesung4630	Die 3 Kritiken
http://data.rwth-aachen.de/resources/vorlesungXXXX	Advanced Logik
http://data.rwth-aachen.de/resources/vorlesung0042	Spezialthemen
http://data.rwth-aachen.de/resources/assistent3002	Ideenlehre
http://data.rwth-aachen.de/resources/assistent3003	Syllogistik
http://data.rwth-aachen.de/resources/assistent3004	Sprachtheorie
http://data.rwth-aachen.de/resources/assistent3005	Planetenbewegung
http://data.rwth-aachen.de/resources/assistent3006	Keplersche Gesetze
http://data.rwth-aachen.de/resources/assistent3007	Gott und Natur

Ausschließen von Mustern mit MINUS

Nachdem wir gelernt haben, wie wir optionale Informationen mit `OPTIONAL` einbeziehen und Ergebnisse verschiedener Muster mit `UNION` kombinieren können, wollen wir uns nun ansehen, wie wir gezielt Lösungen **ausschließen** können, die einem bestimmten unerwünschten Muster entsprechen. Hierfür dient die `MINUS`-Klausel.

Konzept und Funktionsweise:

Die `MINUS`-Klausel entfernt Lösungen aus der Ergebnismenge der vorangehenden Graphmuster, die auch dem in der `MINUS`-Klausel spezifizierten Graphmuster entsprechen.

1. Zuerst werden Lösungen für das Graphmuster *vor* der `MINUS`-Klausel ermittelt.
2. Dann wird das Graphmuster *innerhalb* der `MINUS`-Klausel ausgewertet.
3. Alle Lösungen aus dem ersten Schritt, die auch Lösungen für das `MINUS`-Muster sind (basierend auf den gemeinsamen Variablen), werden aus der endgültigen Ergebnismenge entfernt.

Wichtig: Variablen, die nur im `MINUS`-Muster vorkommen und nicht im vorangehenden Muster, beeinflussen den Filterprozess nicht direkt (d.h., sie müssen nicht an dieselben Werte binden wie in Lösungen außerhalb des `MINUS`-Blocks, um eine Lösung zu entfernen). Entscheidend ist, ob die *gesamte Lösung* des vorangehenden Musters (hinsichtlich der gemeinsamen Variablen) eine Entsprechung im `MINUS`-Muster findet.

Syntax: `{ graphPattern1 } MINUS { graphPatternToRemove }`

Die `MINUS`-Klausel wird innerhalb der `WHERE`-Klausel platziert.

Beispiel: Studenten finden, die eine bestimmte Vorlesung NICHT hören

Wir möchten alle Studenten und ihre Namen finden, die *nicht* die Vorlesung `:vorlesung4052` hören. (Annahme: `:vorlesung4052` ist die IRI einer spezifischen Vorlesung in unserem `uni`-Graphen).

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName
WHERE {
  # Schritt 1: Finde alle Studenten und ihre Namen
  ?student a :Student ;
           :name ?studentName .

  # Schritt 2: Schließe diejenigen aus, die die Logik-Vorlesung hören
  MINUS {
    ?student :hört :vorlesung4052 .
  }
}
```

ORDER BY ?studentName

Erläuterung:

1. Der erste Teil der `WHERE`-Klausel (`?student a :Student ; :name ?studentName .`) ermittelt alle Studenten und ihre Namen.
2. Die `MINUS`-Klausel wird dann auf diese Menge angewendet. Das Muster `{ ?student :hört :vorlesung4052 . }` sucht für jeden Studenten aus der ersten Menge, ob er die Vorlesung `:vorlesung4052` hört.
3. Wenn ein Student die Vorlesung `:vorlesung4052` hört, wird diese Lösung (also der Student und sein Name) aus dem Endergebnis entfernt.
4. Übrig bleiben nur die Studenten, für die das `MINUS`-Muster *nicht* zutrifft.

In [24]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName
WHERE {
  # Schritt 1: Finde alle Studenten und ihre Namen
  ?student a :Student ;
           :name ?studentName .

  # Schritt 2: Schließe diejenigen aus, die die Logik-Vorlesung hören
  MINUS {
    ?student :hört :vorlesung4052 .
  }
}
ORDER BY ?studentName
```

?studentName

Aristoxenos
Carnap
Feuerbach
Fichte
Jonas
Theophrastos
Xenokrates

MINUS vs. FILTER NOT EXISTS

Obwohl beide Konstrukte dazu dienen, negative Bedingungen auszudrücken, gibt es wichtige Unterschiede:

- **FILTER NOT EXISTS { P } :**
 - Prüft für jede gefundene Lösung des Hauptmusters, ob das Pattern *P* (unter Verwendung der bereits gebundenen Variablen) *nicht* existiert.
 - Variablen, die *nur innerhalb* des **NOT EXISTS** -Blocks gebunden werden, sind außerhalb nicht sichtbar und beeinflussen nicht die Bindungen der äußeren Abfrage.
 - Es werden keine Lösungen entfernt, sondern nur Lösungen *gefiltert*, die die Bedingung erfüllen.
- **A MINUS { P } :**
 - Berechnet die Lösungsmenge für *A* und die Lösungsmenge für *P*.
 - Entfernt dann alle Lösungen aus *A*, die auch in *P* vorkommen, basierend auf den *gemeinsamen Variablen* zwischen *A* und *P*. Wenn *P* Variablen enthält, die nicht in *A* vorkommen, können diese innerhalb von *P* beliebig gebunden werden, um eine Übereinstimmung zu finden.
 - **MINUS** operiert auf den gesamten Lösungsmengen.

Wann was verwenden?

- **FILTER NOT EXISTS** ist oft nützlicher, wenn Sie eine Bedingung prüfen wollen, die sich eng auf die aktuell betrachtete Lösung bezieht und keine Variablen außerhalb des Filters binden soll. Es ist typischerweise performanter für einfache Nicht-Existenz-Prüfungen.
- **MINUS** ist mächtiger (und manchmal notwendig), wenn der Ausschluss auf komplexeren Mustern basiert oder wenn die Logik einer echten Mengendifferenz entspricht. Es kann aber auch zu unerwarteten Ergebnissen führen, wenn die Variablenbindung nicht sorgfältig bedacht wird, insbesondere wenn im **MINUS** -Pattern Variablen vorkommen, die im Hauptpattern nicht vorkommen (diese agieren dann existenziell im **MINUS** -Pattern).

Beispiel: Professoren, die keine Vorlesung mit "Einführung" im Titel lehren

Wir suchen Professoren, die keine Vorlesung lehren, deren Titel das Wort "Einführung" enthält.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?profName
WHERE {
  ?prof a :Professor ;
       :name ?profName .

  MINUS {
    ?vorlesung :gelesenVon ?prof .
    ?vorlesung :titel ?titel .
    FILTER(CONTAINS(LCASE(?titel), "einführung"))
  }
}
```

```
}  
ORDER BY ?profName
```

Erläuterung zu Beispiel:

- Zuerst werden alle Professoren und ihre Namen gefunden.
- Dann wird die MINUS -Klausel angewendet:
 - Sie versucht, für jeden Professor (?prof) eine von ihm gelehrte Vorlesung (?vorlesung) zu finden, deren Titel (?titel) das Wort "einführung" (case-insensitive) enthält.
 - Wenn ein Professor eine solche Vorlesung lehrt, wird dieser Professor aus der Ergebnismenge entfernt.
- Es bleiben die Professoren übrig, für die das gesamte Muster innerhalb von MINUS nicht zutrifft.

Die MINUS -Klausel bietet also eine weitere Möglichkeit, die Ergebnismenge basierend auf negativen Bedingungen zu formen, die über einfache Filter hinausgehen.

```
In [25]: %%rdf sparql -l uni  
PREFIX : <http://data.rwth-aachen.de/resources/>  
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  
  
SELECT ?profName  
WHERE {  
  ?prof a :Professor ;  
        :name ?profName .  
  
  MINUS {  
    ?vorlesung :gelesenVon ?prof .  
    ?vorlesung :titel ?titel .  
    FILTER(CONTAINS(LCASE(?titel), "einführung"))  
  }  
}  
ORDER BY ?profName
```

?profName

Augustinus
Curie
Kant
Kopernikus
Popper
Sokrates

Übung 4: Studenten, die bereit für fortgeschrittene Kurse sind (einstufige Voraussetzungen)

Problemstellung:

Sie sollen eine SPARQL-Abfrage entwickeln, die Studenten identifiziert, welche für bestimmte Vorlesungen qualifiziert sind. Ein Student ist "bereit" für eine Vorlesung, wenn er:

1. Alle **direkten** Voraussetzungen dieser Vorlesung erfüllt hat und die Vorlesung Voraussetzungen hat.
2. Die Vorlesung selbst noch nicht belegt

Ihre Aufgabe:

Schreiben Sie eine SPARQL-Abfrage, die folgende Informationen zurückgibt:

- Name des Studenten
- Titel der Vorlesung, für die er qualifiziert ist

Anforderungen:

1. **Verwenden Sie FILTER NOT EXISTS** um zu prüfen, dass alle direkten Voraussetzungen erfüllt sind
2. **Verwenden Sie MINUS** um bereits belegte Kurse auszuschließen
3. Berücksichtigen Sie nur die direkten Voraussetzungen (:hatVoraussetzung)

Hinweis:

Schopenhauer und Theophrastos sind in ihren Studien fortgeschritten.

```
In [ ]: %%rdf sparql -l uni
```

Modifizieren der Ergebnismenge: ORDER BY, LIMIT, OFFSET und DISTINCT

Diese Modifikatoren kennen wir schon von SQL - wir haben sie teilweise schon verwendet - daher werden wir diese hier nur kurz darstellen.

Bisher haben wir uns darauf konzentriert, *welche* Daten wir durch unsere Graphmuster in der `WHERE`-Klausel (ggf. mit `FILTER`, `OPTIONAL`, `UNION`, `MINUS`) selektieren. SPARQL bietet aber auch Klauseln, um die **Darstellung und Auswahl der Ergebnismenge** zu beeinflussen, nachdem die Lösungen gefunden wurden. Diese nennt man Lösungsmodifikatoren.

Ergebnisse sortieren mit `ORDER BY`

Mit `ORDER BY` können die gefundenen Lösungen nach dem Wert einer oder mehrerer Variablen sortiert werden.

Syntax: `ORDER BY ?variable` (aufsteigende Sortierung, Standard) `ORDER BY ASC(?variable)` (explizit aufsteigend) `ORDER BY DESC(?variable)` (absteigende Sortierung) `ORDER BY ?var1 DESC(?var2)` (Sortierung nach mehreren Kriterien)

Die `ORDER BY`-Klausel steht nach der `WHERE`-Klausel und vor eventuellen `LIMIT`- oder `OFFSET`-Klauseln. **Erläuterung:**

- Die Sortierung erfolgt gemäß den Regeln für die Datentypen der Variablen (z.B. alphabetisch für Strings, numerisch für Zahlen).
- Ungebundene Werte oder Werte, die nicht zum erwarteten Typ passen, werden typischerweise ans Ende oder den Anfang der Sortierung gestellt, je nach Implementierung.

Duplikate entfernen mit `DISTINCT`

Manchmal kann eine SPARQL-Anfrage Lösungen generieren, die in Bezug auf die ausgewählten Variablen identisch sind. `SELECT DISTINCT` stellt sicher, dass jede Kombination von Werten für die ausgewählten Variablen nur einmal im Ergebnis vorkommt.

Syntax: `SELECT DISTINCT ?variable1 ?variable2 ...`

Beispiel 20: Alle unterschiedlichen Vorlesungstitel, die von irgendjemandem gehört werden

Wenn mehrere Studenten dieselbe Vorlesung hören, würde eine einfache Abfrage den Titel der Vorlesung mehrfach liefern. Mit `DISTINCT` erhalten wir jeden Titel nur einmal.

Ergebnismenge beschränken mit `LIMIT` und `OFFSET`

Diese beiden Klauseln werden oft zusammen für Paginierung (seitenweise Anzeige von Ergebnissen) verwendet.

- **`LIMIT n`** : Beschränkt die Ausgabe auf maximal `n` Lösungen.
- **`OFFSET m`** : Überspringt die ersten `m` Lösungen.

Syntax: Stehen typischerweise am Ende der SPARQL-Anfrage. Wenn `ORDER BY` verwendet wird, wird erst sortiert, dann `OFFSET` angewendet und schließlich `LIMIT`.

Diese Modifikatoren sind sehr wichtig, um die Ausgabe von SPARQL-Anfragen zu kontrollieren und benutzerfreundlich aufzubereiten.

Aggregieren von Ergebnissen: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX` mit `GROUP BY` und `HAVING`

Die Aggregationsfunktionen kennen wir von SQL - daher werden wir diese hier nur kurz streifen. Experimentieren Sie gerne damit! Bisher haben unsere SPARQL-Anfragen individuelle Lösungen (Variablenbindungen) zurückgegeben, die den Mustern in der `WHERE`-Klausel entsprechen. Oft sind wir jedoch nicht an den einzelnen Datenpunkten interessiert, sondern an **zusammengefassten oder aggregierten Informationen** über diese Daten. Zum Beispiel:

- Wie viele Studenten sind in unserem Datensatz?
- Was ist die durchschnittliche Semesteranzahl der Studenten?
- Welcher Professor hält die meisten Vorlesungen?

Für solche Aufgaben bietet SPARQL **Aggregatfunktionen**. Diese Funktionen berechnen einen einzelnen Wert aus einer Menge von Werten.

Gängige Aggregatfunktionen

Die wichtigsten Aggregatfunktionen in SPARQL sind:

- **`COUNT`** : Zählt Lösungen oder Werte.
 - `COUNT(?variable)` : Zählt, wie oft `?variable` in den Lösungen gebunden ist (ungebundene Variablen werden nicht mitgezählt).
 - `COUNT(DISTINCT ?variable)` : Zählt die Anzahl der *unterschiedlichen* gebundenen Werte für `?variable`.
 - `COUNT(*)` : Zählt die Anzahl der Lösungen (Zeilen), die vom `WHERE`-Pattern generiert werden.
- **`SUM(?variable)`** : Summiert die numerischen Werte von `?variable`.
- **`AVG(?variable)`** : Berechnet den Durchschnitt der numerischen Werte von `?variable`.
- **`MIN(?variable)`** : Findet den kleinsten Wert von `?variable` (numerisch, alphabetisch, etc., je nach Datentyp).
- **`MAX(?variable)`** : Findet den größten Wert von `?variable`.

Aggregatfunktionen werden typischerweise in der `SELECT`-Klausel verwendet, oft in Verbindung mit `AS`, um der Ergebnisspalte einen

Namen zu geben: `SELECT (COUNT(?student) AS ?anzahlStudenten) ...`

Einfache Aggregation über alle Ergebnisse

Wenn keine `GROUP BY`-Klausel verwendet wird (siehe nächster Abschnitt), operieren Aggregatfunktionen auf der gesamten Ergebnismenge, die von der `WHERE`-Klausel erzeugt wird.

Beispiel: Gesamtzahl aller bekannten Studenten

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT (COUNT(?student) AS ?anzahlAallerStudenten)
WHERE {
  ?student a :Student . # Finde alle Entitäten vom Typ :Student
}
```

Erläuterung:

- `COUNT(?student)` zählt jede Lösung, bei der die Variable `?student` an eine IRI (oder einen Blank Node) gebunden ist, die/der vom Typ `:Student` ist.
- `AS ?anzahlAallerStudenten` gibt der Ergebnisspalte, die diesen Zählwert enthält, den Namen `anzahlAallerStudenten`.
- Das Ergebnis ist eine einzelne Zeile mit einer einzelnen Spalte.

Gruppieren von Ergebnissen mit `GROUP BY`

Oft möchten wir Aggregate nicht über die gesamte Ergebnismenge, sondern für **Gruppen von Lösungen** berechnen. Hierfür verwenden wir die `GROUP BY`-Klausel. `GROUP BY` gruppiert Lösungen, die in den angegebenen Variablen oder Ausdrücken denselben Wert haben. Die Aggregatfunktionen werden dann auf jede dieser Gruppen separat angewendet.

Wichtige Regel: Wenn `GROUP BY` verwendet wird, dürfen in der `SELECT`-Klausel nur noch Variablen erscheinen, die auch in der `GROUP BY`-Klausel stehen, oder Variablen, die innerhalb einer Aggregatfunktion verwendet werden.

Beispiel: Anzahl der Studenten pro Fachsemester

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?semesterAnzahl (COUNT(?student) AS ?anzahlInSemester)
WHERE {
  ?student a :Student ;
           :semester ?semesterAnzahl .
}
GROUP BY ?semesterAnzahl # Gruppiere nach dem Wert von ?semesterAnzahl
ORDER BY ASC(?semesterAnzahl)
```

Erläuterung:

- Die `WHERE`-Klausel findet alle Studenten und deren Semesteranzahl.
- `GROUP BY ?semesterAnzahl` bildet Gruppen von Studenten, die im selben Fachsemester sind.
- `SELECT ?semesterAnzahl (COUNT(?student) AS ?anzahlInSemester) :`
 - `?semesterAnzahl` darf hier stehen, da es in `GROUP BY` vorkommt.
 - `COUNT(?student)` wird nun für jede dieser Gruppen (jedes Semester) separat ausgeführt und zählt die Studenten in dieser Gruppe.

Filtern von Gruppen mit `HAVING`

Nachdem die Daten mit `GROUP BY` gruppiert und Aggregate berechnet wurden, möchten wir manchmal diese **Gruppen filtern**, basierend auf dem Ergebnis der Aggregatfunktion. Hierfür dient die `HAVING`-Klausel. `HAVING` funktioniert ähnlich wie `FILTER`, wird aber *nach* `GROUP BY` und den Aggregatberechnungen angewendet.

Syntax: `HAVING` (BedingungMitAggregat)

Beispiel : Fachsemester anzeigen, in denen mehr als 5 Studenten sind

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?semesterAnzahl (COUNT(?student) AS ?anzahlInSemester)
WHERE {
  ?student a :Student ;
           :semester ?semesterAnzahl .
}
GROUP BY ?semesterAnzahl
```

```
HAVING (COUNT(?student) > 1) # Behalte nur Gruppen, in denen das Aggregat > 1 ist
ORDER BY ASC(?semesterAnzahl)
```

Erläuterung:

- Zuerst werden Studenten nach Semestern gruppiert und die Anzahl pro Semester gezählt (wie in Beispiel 23).
- HAVING (COUNT(?student) > 5) filtert dann diese Gruppen und behält nur diejenigen Semester, für die der zuvor berechnete COUNT(?student) -Wert größer als 5 ist.

Weitere Aggregatfunktionen: SUM, AVG, MIN, MAX

Neben COUNT gibt es weitere nützliche Aggregatfunktionen.

Beispiel: Durchschnittliche, minimale und maximale Semesterwochenstunden (SWS) aller Vorlesungen

```
%%rdf sparql -1 uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT
  (AVG(?sws) AS ?durchschnittSWS)
  (MIN(?sws) AS ?minSWS)
  (MAX(?sws) AS ?maxSWS)
  (SUM(?sws) AS ?gesamtSWS) # Gesamtzahl aller SWS als Bonus
WHERE {
  ?vorlesung a :Vorlesung ;
             :sws ?sws . # Annahme: :sws hat numerische Werte (z.B. xsd:integer)
}
```

Aggregatfunktionen, insbesondere in Kombination mit GROUP BY, sind extrem mächtig, um quantitative Einblicke und Zusammenfassungen aus Ihren RDF-Daten zu gewinnen.

Übung 5: Durchschnittliche SWS pro Professor

Problemstellung:

Sie sollen eine SPARQL-Abfrage entwickeln, die für jeden Professor die durchschnittliche Anzahl der Semesterwochenstunden (SWS) seiner Vorlesungen berechnet.

Ihre Aufgabe:

Schreiben Sie eine SPARQL-Abfrage, die folgende Informationen zurückgibt:

- Name des Professors
- Durchschnittliche SWS seiner Vorlesungen
- Anzahl der Vorlesungen, die er hält

Anforderungen:

1. **Verwenden Sie GROUP BY** um die Ergebnisse nach Professor zu gruppieren
2. **Verwenden Sie AVG()** um den Durchschnitt der SWS zu berechnen
3. **Verwenden Sie COUNT()** um die Anzahl der Vorlesungen zu ermitteln
4. Sortieren Sie das Ergebnis absteigend nach der durchschnittlichen SWS

Erwartetes Ergebnis:

Die Abfrage sollte eine Liste aller Professoren mit ihrer durchschnittlichen Lehrbelastung ausgeben, z.B.:

- Professor X: 4.5 SWS im Durchschnitt, 3 Vorlesungen
- Professor Y: 3.0 SWS im Durchschnitt, 2 Vorlesungen

Hinweis:

Achten Sie darauf, dass Sie sowohl die Professor-Entität als auch die Vorlesungs-Entität in Ihrem Pattern verwenden und diese über die entsprechende Property verknüpfen.

In []:

```
%%rdf sparql -1 uni
```

Optionaler Teil: Fortgeschrittenes SPARQL

Der nun folgende Teil (bis - aber nicht einschliesslich Finale Betrachtungen) ist für Sie optional. Wir empfehlen es durchzuarbeiten - und es ist hoffentlich auch interessant, ist aber für die Vorlesung nicht nötig.

Weitere SPARQL-Anfrageformen: ASK, CONSTRUCT und DESCRIBE

Bisher haben wir uns intensiv mit `SELECT`-Anfragen beschäftigt, die uns tabellarische Ergebnisse (Variablenbindungen) aus dem RDF-Graphen liefern. SPARQL bietet jedoch weitere wichtige Anfrageformen, die für unterschiedliche Zwecke optimiert sind und unterschiedliche Arten von Ergebnissen zurückgeben:

- **ASK** : Überprüft die Existenz eines Musters und gibt einen Wahrheitswert (`true` oder `false`) zurück.
- **CONSTRUCT** : Erzeugt einen neuen RDF-Graphen basierend auf einem Template und den Ergebnissen eines Graphmusters.
- **DESCRIBE** : Liefert eine RDF-Beschreibung (einen Subgraphen) von einer oder mehreren Ressourcen.

Wir werden uns diese nun genauer ansehen.

Existenz überprüfen mit ASK

Manchmal möchten wir nicht wissen, *welche* Lösungen es für ein Muster gibt, sondern nur, *ob* es überhaupt mindestens eine Lösung gibt. Hierfür ist die `ASK`-Anfrage gedacht.

Konzept und Funktionsweise: Eine `ASK`-Anfrage gibt entweder `true` zurück, wenn das in der `WHERE`-Klausel angegebene Graphmuster mindestens einmal im abgefragten Graphen gefunden werden kann, oder `false`, wenn das Muster keine Übereinstimmung findet.

Syntax: `ASK WHERE { graphPattern }` (Die `WHERE`-Klausel ist optional, aber meist vorhanden. Ohne `WHERE`-Klausel (`ASK {}`) prüft es, ob der Default-Graph leer ist – was selten nützlich ist).

Beispiel 27: Gibt es Studenten im 6. Semester oder höher?

Wir wollen wissen, ob es in unserem `uni`-Graphen mindestens einen Studenten gibt, dessen Semesteranzahl 6 oder mehr beträgt.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
```

```
ASK
WHERE {
  ?student a :Student ;
           :semester ?semester .
  FILTER(?semester >= 6)
}
```

Erläuterung und erwartete Ausgabe:

- Die `WHERE`-Klausel sucht nach Studenten und filtert nach denen im 6. Semester oder höher.
- Wenn mindestens ein solcher Student gefunden wird, gibt die gesamte `ASK`-Anfrage `True` zurück.
- Wenn kein solcher Student gefunden wird, gibt die Anfrage `False` zurück. Die Ausgabe der `%%rdf sparql`-Zelle wird also einfach `True` oder `False` sein.

`ASK`-Anfragen sind sehr nützlich für Ja/Nein-Fragen oder um Bedingungen in Anwendungen zu überprüfen.

In [29]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ASK
WHERE {
  ?student a :Student ;
           :semester ?semester .
  FILTER(?semester >= 6)
}
```

True

RDF-Graphen erzeugen mit CONSTRUCT

Im Gegensatz zu `SELECT` (tabellarische Ergebnisse) oder `ASK` (Wahrheitswert) gibt eine `CONSTRUCT`-Anfrage einen **neuen RDF-Graphen** zurück. Dieser neue Graph wird basierend auf einem **Graph-Template** erstellt, das in der `CONSTRUCT`-Klausel definiert wird. Die Variablen in diesem Template werden durch die Lösungen gefüllt, die von der `WHERE`-Klausel der Anfrage gefunden werden.

Konzept und Funktionsweise:

1. Die `WHERE`-Klausel wird ausgewertet und erzeugt eine Menge von Lösungen (Variablenbindungen), genau wie bei einer `SELECT`-Anfrage.

2. Für jede dieser Lösungen wird das **Template** in der `CONSTRUCT` -Klausel genommen und die Variablen im Template durch die entsprechenden Bindungen aus der Lösung ersetzt.
3. Alle so erzeugten Tripel bilden zusammen den Ergebnisgraphen.

Syntax: `CONSTRUCT { triplePatternTemplate1 . triplePatternTemplate2 ... } WHERE { graphPatternToFindSolutions }`
 # Optionale Modifikatoren wie `ORDER BY`, `LIMIT` etc. können auch hier verwendet werden, # um die Lösungen aus der `WHERE`-Klausel zu beeinflussen, bevor sie ins Template eingesetzt werden.

Beispiel 28: Einen vereinfachten Graphen nur mit Namen von Studenten erstellen

Wir wollen einen neuen Graphen erzeugen, der für jeden Studenten nur ein Tripel enthält, das den Studenten mit seinem Namen verbindet, wobei wir ein eigenes, vereinfachtes Prädikat `ex:studentName` verwenden.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ex: <http://example.org/simplifiedVocabulary/> # Ein neues Vokabular für den Output

CONSTRUCT {
  ?student ex:studentName ?name . # Dies ist das Template für die neuen Tripel
}
WHERE {
  ?student a :Student ;
          :name ?name .          # Finde Studenten und ihre ursprünglichen Namen
}
LIMIT 5 # Begrenzt die Anzahl der Studenten, für die neue Tripel generiert werden
```

Erläuterung und erwartete Ausgabe:

- Die `WHERE` -Klausel findet alle Studenten (`?student`) und ihre Namen (`?name`) aus dem `uni` -Graphen.
- Für jeden gefundenen Studenten wird das `CONSTRUCT` -Template `?student ex:studentName ?name .` genommen und die Variablen `?student` und `?name` mit den Werten aus der jeweiligen Lösung ersetzt.
- Das Ergebnis ist ein neuer RDF-Graph, der nur Tripel der Form `<studentIRI> ex:studentName "Studentenname" .` enthält.
- Die `%%rdf sparql` -Zelle wird diesen neuen Graphen ausgeben (oft in Turtle-Syntax oder als Tripel-Liste und ggf. auch visualisiert).

`CONSTRUCT` -Anfragen sind extrem nützlich für Datenumwandlungen, das Extrahieren spezifischer Sichten auf Daten oder das Erzeugen von RDF-Daten, die einem bestimmten Zielschema entsprechen.

In [30]:

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ex: <http://example.org/simplifiedVocabulary/> # Ein neues Vokabular für den Output

CONSTRUCT {
  ?student ex:studentName ?name . # Dies ist das Template für die neuen Tripel
}
WHERE {
  ?student a :Student ;
          :name ?name .          # Finde Studenten und ihre ursprünglichen Namen
}
LIMIT 5 # Begrenzt die Anzahl der Studenten, für die neue Tripel generiert werden
```



Ressourcen beschreiben mit `DESCRIBE`

Die `DESCRIBE` -Anfrage liefert einen RDF-Graphen zurück, der eine oder mehrere angegebene Ressourcen beschreibt. Was "eine Beschreibung" genau bedeutet – also welche Tripel zurückgegeben werden – ist **nicht strikt in der SPARQL-Spezifikation festgelegt** und hängt von der Implementierung des SPARQL-Prozessors ab.

Konzept und Funktionsweise: Man gibt eine oder mehrere IRIs oder Variablen an, deren Ressourcen beschrieben werden sollen. Der SPARQL-Prozessor entscheidet dann, welche Tripel relevant sind, um diese Ressourcen zu beschreiben. Typischerweise sind das:

- Alle Tripel, bei denen die Ressource Subjekt ist.
- Manchmal auch Tripel, bei denen die Ressource Objekt ist (inverse Eigenschaften).
- Manchmal auch Beschreibungen von Blank Nodes, die direkt von der Ressource erreicht werden.

Syntax: `DESCRIBE <IRI> DESCRIBE ?variable` # Oft in Kombination mit einer `WHERE`-Klausel, um die zu beschreibenden Ressourcen dynamisch zu finden: `DESCRIBE ?resource WHERE { ... ?resource ... }`

Beispiel: Beschreibung eines spezifischen Studenten

Wir wollen eine RDF-Beschreibung des Studenten `:student28106` .

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
# Weitere Präfixe je nach Daten im Graphen für den Studenten
```

```
DESCRIBE :student28106
```

Beispiel: Beschreibung aller Vorlesungen

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
# Weitere Präfixe je nach Daten

DESCRIBE ?vorlesung
WHERE {
  ?vorlesung a :Vorlesung .
}
LIMIT 2 # Beschreibe nur die ersten 2 gefundenen Vorlesungen
```

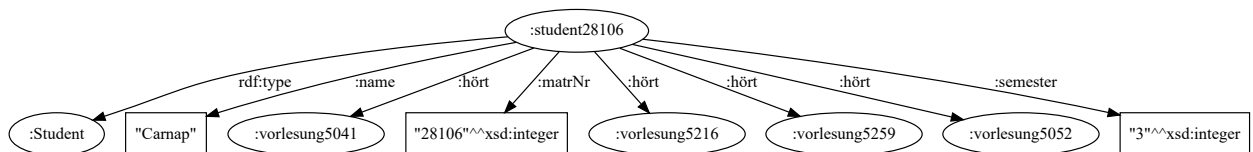
Erläuterung und erwartete Ausgabe:

- Das Ergebnis einer DESCRIBE -Anfrage ist ein RDF-Graph.
- Die %%rdf sparql -Zelle wird diesen Graphen ausgeben (als Turtle, Tripel-Liste und/oder Visualisierung).
- **Wichtig:** Experimentieren Sie mit DESCRIBE auf Ihrem System! Die genauen Ergebnisse können variieren. Für :student28106 könnten Sie z.B. Tripel zu seinem Namen, Semester, und den von ihm gehörten Vorlesungen erhalten - was genau die RDF Datenbank zurückliefert ist nicht im SPARQL-Standard festgelegt.

DESCRIBE ist nützlich, wenn man schnell einen Überblick über eine Ressource und ihre direkten Zusammenhänge bekommen möchte, ohne genau spezifizieren zu müssen, welche Eigenschaften man sehen will. Es ist oft ein Werkzeug für explorative Datenanalyse.

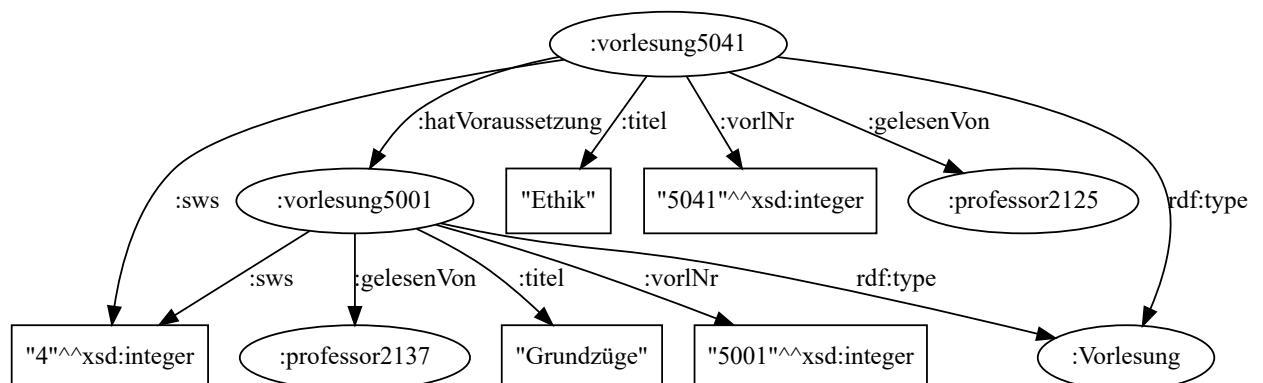
```
In [31]: %%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>

DESCRIBE :student28106
```



```
In [32]: %%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

DESCRIBE ?vorlesung
WHERE {
  ?vorlesung a :Vorlesung .
}
LIMIT 2 # Beschreibe nur die ersten 2 gefundenen Vorlesungen
```



```
In [33]: %%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName
WHERE {
  :vorlesung5041 ^:hört ?student . # Finde ?student, sodass ?student :hört :vorlesung5041 gilt
  ?student :name ?studentName .
}
```

?studentName

Arbeiten mit verschiedenen SPARQL-Endpoints

Bisher haben wir unsere SPARQL-Anfragen primär gegen unseren lokalen uni-Graphen ausgeführt. Eine der großen Stärken von SPARQL und RDF ist jedoch die Möglichkeit, Daten direkt aus dem Web von verschiedenen Datenquellen, sogenannten SPARQL-Endpoints, abzufragen und sogar zu kombinieren. Ein SPARQL-Endpoint ist im Grunde eine Web-Schnittstelle, die SPARQL-Anfragen entgegennimmt, sie gegen einen oder mehrere RDF-Graphen ausführt und die Ergebnisse in einem standardisierten Format zurückgibt.

DBpedia: Wikipedia als RDF-Graph abfragen

DBpedia ist ein bekanntes Community-Projekt, das strukturierte Informationen aus Wikipedia extrahiert und als RDF-Daten über einen öffentlichen SPARQL-Endpoint zugänglich macht. Dies ermöglicht es uns, das riesige Wissen von Wikipedia mit SPARQL zu durchsuchen.

Wichtige Hinweise zu DBpedia:

- **Sprachversionen:** DBpedia existiert für viele Sprachversionen von Wikipedia. Der "Haupt"-Endpoint (<https://dbpedia.org/sparql>), den wir hier verwenden, aggregiert oft Daten aus der englischen Wikipedia als Hauptquelle, enthält aber auch Links und Daten zu anderen Sprachversionen. Für spezifisch deutschsprachige Entitäten und Labels ist oft der direkte deutsche Endpoint (<http://de.dbpedia.org/sparql>) eine Option, oder man filtert explizit nach deutschen Sprach-Tags.
- **Vokabulare:** DBpedia verwendet eigene Ontologien (`dbo:` für <http://dbpedia.org/ontology/>) und Properties (`dbp:` für <http://dbpedia.org/property/>), aber auch Standardvokabulare wie `rdfs:label`.
- **Datenmenge:** DBpedia ist sehr umfangreich. Präzise Anfragen und die Verwendung von `LIMIT` sind oft empfehlenswert.

Beispiel: Die 10 größten deutschen Entitäten (Bundesländer nach Einwohnerzahl) von DBpedia abrufen

Wir möchten von DBpedia eine Liste der 10 deutschen Bundesländer mit mehr als 1 Million Einwohnern erhalten, sortiert nach Einwohnerzahl absteigend, und dabei die deutsche Bezeichnung des Bundesländer anzeigen.

```
%%rdf sparql --endpoint http://dbpedia.org/sparql
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?entity ?label ?population
WHERE {
  # Finde Entitäten , die ...
  ?entity dbo:country dbr:Germany ;           # ... im Land Deutschland liegen
          dbo:populationTotal ?population ;    # ... eine Gesamtbevölkerung haben
          rdfs:label ?label .                 # ... und ein Label besitzen.

  # Filtere die Ergebnisse:
  FILTER(LANG(?label) = "de" && ?population > 1000000)
  # Das Label muss deutsch sein UND die Population muss größer als 1 Million sein.
}
ORDER BY DESC(?population) # Sortiere nach Bevölkerung absteigend
LIMIT 10                   # Gib nur die Top 10 Ergebnisse aus
```

Erläuterung der Anfrage:

1. `%%rdf sparql --endpoint http://dbpedia.org/sparql :`
 - Weist unser Jupyter Notebook an, die folgende SPARQL-Anfrage an den öffentlichen DBpedia-Endpoint zu senden.
2. `PREFIX ... :`
 - Wir definieren gängige Präfixe für DBpedia (`dbo:` , `dbr:`) und RDFS (`rdfs:`), um die Anfrage lesbarer zu machen.
3. `SELECT ?city ?label ?population :`
 - Wir möchten die IRI der Entitäten (`?city`), ihr deutsches Label (`?label`) und ihre Einwohnerzahl (`?population`) als Ergebnis erhalten.
4. `WHERE { ... } :`
 - `?entity dbo:country dbr:Germany ; :` Findet alle Ressourcen, die über die Eigenschaft `dbo:country` mit der Ressource `dbr:Germany` (Deutschland) verbunden sind.
 - `dbo:populationTotal ?population ; :` Für diese Entitäten wird die Eigenschaft `dbo:populationTotal` gesucht und ihr Wert an `?population` gebunden.
 - `rdfs:label ?label . :` Ebenso wird das `rdfs:label` der Entität an `?label` gebunden.
 - `FILTER(LANG(?label) = "de" && ?population > 1000000) :` Dieser Filter schränkt die bis dahin gefundenen Lösungen ein:
 - `LANG(?label) = "de" :` Nur Lösungen, bei denen der Sprach-Tag des Labels `?label` deutsch ist, werden beibehalten.
 - `?population > 1000000 :` Nur Lösungen, bei denen die Einwohnerzahl `?population` größer als 1.000.000 ist, werden beibehalten.

- && : Beide Bedingungen müssen erfüllt sein.

5. ORDER BY DESC(?population) :

- Die Ergebnismenge wird nach der Einwohnerzahl (?population) in **absteigender** Reihenfolge sortiert (die Stadt mit den meisten Einwohnern zuerst).

6. LIMIT 10 :

- Es werden nur die ersten 10 Zeilen der sortierten Ergebnismenge zurückgegeben.

In []:

```
%%rdf sparql --endpoint http://dbpedia.org/sparql
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?entity ?label ?population
WHERE {
  # Finde Entitäten , die ...
  ?entity dbo:country dbr:Germany ;      # ... im Land Deutschland liegen
        dbo:populationTotal ?population ; # ... eine Gesamtbevölkerung haben
        rdfs:label ?label .              # ... und ein Label besitzen.

  # Filtere die Ergebnisse:
  FILTER(LANG(?label) = "de" && ?population > 1000000)
  # Das Label muss deutsch sein UND die Population muss größer als 1 Million sein.
}
ORDER BY DESC(?population) # Sortiere nach Bevölkerung absteigend
LIMIT 10                   # Gib nur die Top 10 Ergebnisse aus
```

Wikidata: Eine kollaborative Wissensdatenbank abfragen

Wikidata ist ein weiteres prominentes Projekt, das eine riesige, frei zugängliche und kollaborativ bearbeitete Wissensdatenbank darstellt. Sie dient als zentrale Datenquelle für viele Wikimedia-Projekte, einschließlich Wikipedia, und bietet ebenfalls einen leistungsstarken **SPARQL-Endpoint** unter <https://query.wikidata.org/sparql> .

Wikidata verwendet ein eigenes System von Entitäten (Items, z.B. wd:Q146 für Katze) und Eigenschaften (Properties, z.B. wdt:P31 für "instance of").

Besonderheit: Der wikibase:label Service Eine sehr nützliche Besonderheit bei Wikidata-SPARQL-Abfragen ist die SERVICE wikibase:label -Klausel. Sie ermöglicht es, automatisch die menschenlesbaren Labels für die IRIs von Entitäten (Items) und Eigenschaften (Properties) in der vom Nutzer bevorzugten Sprache (oder einer Fallback-Sprache) abzurufen, ohne dass man explizit rdfs:label -Tripel für jede Sprache anfordern muss.

Beispiel: Häufigkeit von Augenfarben bei Menschen auf Wikidata

Die folgende Anfrage ermittelt verschiedene Augenfarben (wdt:P1340), die für Menschen (wd:Q5) in Wikidata erfasst sind, zählt die Anzahl der Menschen pro Augenfarbe und versucht, den RGB-Farbcode (wdt:P465) für die Augenfarbe zu finden.

```
%%rdf sparql --endpoint https://query.wikidata.org/sparql
PREFIX wd: <http://www.wikidata.org/entity/>      # Wikidata Items (z.B. Q5 für Mensch, Q107209اى für Augenfarbe)
PREFIX wdt: <http://www.wikidata.org/prop/direct/> # Wikidata Properties (z.B. P31 für "instance of", P1340 für "eye color")
PREFIX wikibase: <http://wikiba.se/ontology#>    # Wikidata Vokabular, u.a. für den Label Service
PREFIX bd: <http://www.bigdata.com/rdf#>          # Vokabular des Blazegraph-Backends von Wikidata, u.a. für den Label Service

SELECT ?eyeColor ?eyeColorLabel ?rgb (COUNT(?human) AS ?count)
WHERE {
  # Finde alle Entitäten ?human, die eine Instanz von Mensch sind (wd:Q5 ist "human")
  ?human wd:P31 wd:Q5.
  # Finde die Augenfarbe ?eyeColor dieser Menschen (wd:P1340 ist "eye color")
  ?human wd:P1340 ?eyeColor.

  # Optional: Versuche, den RGB-Farbcode für die Augenfarbe zu finden (wd:P465 ist "color space")
  # (Nicht jede Augenfarbe-Entität wird einen direkten RGB-Code haben)
  OPTIONAL { ?eyeColor wdt:P465 ?rgb. }

  # Nutze den Wikidata Label Service, um menschenlesbare Labels für ?eyeColor zu bekommen.
  # ?eyeColorLabel wird automatisch mit dem Label von ?eyeColor befüllt.
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],mul,en". }
  # "[AUTO_LANGUAGE]" versucht die Browsersprache, "mul" für mehrsprachige Labels, "en" als Fallback.
  # Man könnte auch explizit "de,en" für Deutsch dann Englisch angeben.
}
GROUP BY ?eyeColor ?eyeColorLabel ?rgb # Gruppiere nach Augenfarbe, deren Label und RGB-Code
ORDER BY DESC(?count)                 # Sortiere nach Häufigkeit absteigend
LIMIT 15                               # Zeige die Top 15
```

Erläuterung der Anfrage:

1. `%%rdf sparql --endpoint https://query.wikidata.org/sparql` :

- Sendet die Anfrage an den öffentlichen Wikidata SPARQL-Endpoint.

2. `PREFIX ...` :

- Definiert die für Wikidata typischen Präfixe.

3. `SELECT ?eyeColor ?eyeColorLabel ?rgb (COUNT(?human) AS ?count)` :

- Wir möchten die IRI der Augenfarbe, ihr Label, den RGB-Code (falls vorhanden) und die Anzahl der Menschen mit dieser Augenfarbe (`?count`) sehen.

4. `WHERE { ... }` :

- `?human wdt:P31 wd:Q5` . : Wählt alle Instanzen (`?human`) aus, die Menschen (`wd:Q5`) sind.
- `?human wdt:P1340 ?eyeColor` . : Findet für diese Menschen die Eigenschaft Augenfarbe (`wdt:P1340`) und bindet den Wert (eine weitere Entität, z.B. `wd:Q171227` für "brown eye color") an `?eyeColor` .
- `OPTIONAL { ?eyeColor wdt:P465 ?rgb. }` : Versucht optional, für die gefundene Augenfarbe-Entität `?eyeColor` einen RGB-Farbcode (`wdt:P465` ist "sRGB color hex triplet") zu finden. Wenn keiner vorhanden ist, bleibt `?rgb` ungebunden.
- `SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],mul,en". }` : Dies ist der spezielle Aufruf an den Wikidata Label Service. Er sorgt dafür, dass für alle Variablen, die im Scope sind und IRIs von Wikidata-Entitäten oder -Eigenschaften darstellen (hier insbesondere `?eyeColor`), automatisch eine zusätzliche Variable mit dem Suffix `Label` (also `?eyeColorLabel`) generiert und mit dem passenden menschenlesbaren Label in der angegebenen Sprache (hier automatisch erkannt, multilingual oder Englisch als Fallback) befüllt wird.

5. `GROUP BY ?eyeColor ?eyeColorLabel ?rgb` :

- Gruppirt die Ergebnisse, sodass wir pro einzigartiger Kombination von Augenfarb-IRI, Augenfarb-Label und RGB-Code zählen können.

6. `ORDER BY DESC(?count)` :

- Sortiert die Augenfarben nach der Anzahl der Menschen, die sie haben, absteigend.

7. `LIMIT 15` :

- Beschränkt die Ausgabe auf die 15 häufigsten erfassten Augenfarbenkombinationen.

Was lernen wir daraus (spezifisch für Wikidata)?

- Wikidata ist eine extrem reichhaltige Datenquelle für alle Arten von Entitäten und deren Beziehungen.
- Die Properties (`P` -Nummern) und Items (`Q` -Nummern) sind der Schlüssel zum Navigieren. Es ist oft hilfreich, die Webseite von Wikidata parallel zu nutzen, um die richtigen `P`- und `Q`-Nummern für die gewünschten Konzepte zu finden.
- Der `SERVICE wikibase:label` -Mechanismus ist sehr komfortabel, um schnell an lesbare Bezeichnungen zu gelangen, ohne komplexe `OPTIONAL` -Muster für `rdfs:label` mit Sprachfiltern schreiben zu müssen.

Dieser Abschnitt zeigt, dass das Grundprinzip von SPARQL-Abfragen über verschiedene Endpunkte hinweg gleichbleibt, aber spezifische Endpunkte wie Wikidata zusätzliche, nützliche Funktionalitäten oder Datenmodelleigenschaften haben können, die man kennen sollte.

In [35]:

```
%%rdf sparql --endpoint https://query.wikidata.org/sparql
PREFIX wd: <http://www.wikidata.org/entity/>      # Wikidata Items (z.B. Q5 für Mensch, Q107209 für Augenfarbe)
PREFIX wdt: <http://www.wikidata.org/prop/direct/> # Wikidata Properties (z.B. P31 für "instance of", P1340 für "eye color")
PREFIX wikibase: <http://wikiba.se/ontology#>    # Wikidata Vokabular, u.a. für den Label Service
PREFIX bd: <http://www.bigdata.com/rdf#>         # Vokabular des Blazegraph-Backends von Wikidata, u.a. für den Label Service

SELECT ?eyeColor ?eyeColorLabel ?rgb (COUNT(?human) AS ?count)
WHERE
{
  # Finde alle Entitäten ?human, die eine Instanz von Mensch sind (wd:Q5 ist "human")
  ?human wdt:P31 wd:Q5.
  # Finde die Augenfarbe ?eyeColor dieser Menschen (wdt:P1340 ist "eye color")
  ?human wdt:P1340 ?eyeColor.

  # Optional: Versuche, den RGB-Farbcode für die Augenfarbe zu finden (wdt:P465 ist "color space")
  # (Nicht jede Augenfarbe-Entität wird einen direkten RGB-Code haben)
  OPTIONAL { ?eyeColor wdt:P465 ?rgb. }

  # Nutze den Wikidata Label Service, um menschenlesbare Labels für ?eyeColor zu bekommen.
  # ?eyeColorLabel wird automatisch mit dem Label von ?eyeColor befüllt.
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],mul,en". }
  # "[AUTO_LANGUAGE]" versucht die Browsersprache, "mul" für mehrsprachige Labels, "en" als Fallback.
  # Man könnte auch explizit "de,en" für Deutsch dann Englisch angeben.
}
GROUP BY ?eyeColor ?eyeColorLabel ?rgb # Gruppieren nach Augenfarbe, deren Label und RGB-Code
ORDER BY DESC(?count)                 # Sortiere nach Häufigkeit absteigend
LIMIT 15                               # Zeige die Top 15
```

eyeColor	eyeColorLabel	rgb	count
<http://www.wikidata.org/entity/Q17122705>	brown@en	800000	4887^http://www.w3.org/2001/XMLSchema#integer

<http://www.wikidata.org/entity/ Q17122834>	blue@en	608090	2213^^http://www.w3.org/2001/ XMLSchema#integer
<http://www.wikidata.org/entity/ Q17244894>	dark brown@en	600000	1256^^http://www.w3.org/2001/ XMLSchema#integer
<http://www.wikidata.org/entity/ Q17122854>	green@en	707020	987^^http://www.w3.org/2001/ XMLSchema#integer
<http://www.wikidata.org/entity/ Q17244465>	black@en	201010	734^^http://www.w3.org/2001/ XMLSchema#integer
<http://www.wikidata.org/entity/ Q17122740>	hazel@en	5A391C	582^^http://www.w3.org/2001/ XMLSchema#integer
<http://www.wikidata.org/entity/ Q17245659>	gray@en	707070	238^^http://www.w3.org/2001/ XMLSchema#integer
<http://www.wikidata.org/entity/ Q3375649>	blue-green@en	608070	144^^http://www.w3.org/2001/ XMLSchema#integer
<http://www.wikidata.org/entity/ Q79399164>	light brown@en	704030	81^^http://www.w3.org/2001/ XMLSchema#integer
<http://www.wikidata.org/entity/ Q42845936>	blue-gray@en	708090	62^^http://www.w3.org/2001/ XMLSchema#integer
<http://www.wikidata.org/entity/ Q61600018>	light blue@en	6888A0	50^^http://www.w3.org/2001/ XMLSchema#integer
<http://www.wikidata.org/entity/ Q60172366>	dark blue@en	20^^http://www.w3.org/2001/ XMLSchema#integer	
<http://www.wikidata.org/entity/ Q17291407>	amber@en	19^^http://www.w3.org/2001/ XMLSchema#integer	
<http://www.wikidata.org/entity/ Q2932937>	coffee@en	6F4E37	14^^http://www.w3.org/2001/ XMLSchema#integer
<http://www.wikidata.org/entity/ Q78374938>	light green@en	11^^http://www.w3.org/2001/ XMLSchema#integer	

Fortgeschrittene Graphmuster: Property Paths (Eigenschaftspfade)

Bisher haben wir in unseren `WHERE`-Klauseln explizit jede einzelne Beziehung (jedes Prädikat) in unseren Tripel-Mustern angegeben. Wenn wir jedoch nach Ressourcen suchen wollen, die über eine *Kette* von Beziehungen miteinander verbunden sind, oder wenn wir alternative Beziehungen zulassen wollen, wird dies mit einfachen Tripel-Mustern schnell umständlich und erfordert viele benannte Zwischenvariablen.

SPARQL bietet hierfür eine sehr mächtige und kompakte Lösung: **Property Paths (Eigenschaftspfade)**. Mit Property Paths können wir komplexe Pfade durch den RDF-Graphen definieren, denen der SPARQL-Prozessor folgen soll, um Subjekte mit Objekten zu verbinden. Property Paths werden an der Stelle des Prädikats in einem Tripel-Muster verwendet.

Sequenzpfade (`property1 / property2`)

Ein Sequenzpfad erlaubt es, eine Kette von Eigenschaften anzugeben. Das Muster `?x p1/p2 ?y` matcht, wenn es eine Ressource `?z` gibt, sodass `?x p1 ?z` . und `?z p2 ?y` . gilt. Die Zwischenressource `?z` muss hier nicht explizit benannt werden.

Beispiel: Namen der Studenten und die Titel der Vorlesungen, die von den Professoren dieser Studenten gelesen werden

Wir suchen Studenten, die Vorlesungen hören. Diese Vorlesungen werden von Professoren gelesen. Wir wollen den Namen des Studenten und den Namen des Professors, der die vom Studenten gehörte Vorlesung liest.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName ?profName ?vorlesungTitel
WHERE {
  ?student a :Student ;
           :name ?studentName ;
           :hört/:gelesenVon ?professor . # Sequenzpfad: Student hört Vorlesung, Vorlesung gelesenVon Professor
  ?professor :name ?profName .
  ?student :hört ?vorlesung . # Wir brauchen die Vorlesung für ihren Titel
  ?vorlesung :titel ?vorlesungTitel .
}
LIMIT 10
```

Erläuterung:

- `:hört/:gelesenVon` bedeutet: Finde einen Studenten `?student` , der eine Vorlesung hört (Zwischenknoten, implizit), und dieser Zwischenknoten (die Vorlesung) wird von einem Professor `?professor` gelesen.
- Anschließend holen wir uns die Namen und den Vorlesungstitel.

Alternative Pfade (property1 | property2)

Ein Alternativpfad erlaubt es, anzugeben, dass entweder die eine *oder* die andere Eigenschaft (oder ein anderer komplexerer Pfad) zwischen Subjekt und Objekt bestehen soll. Das Muster `?x p1|p2 ?y` matcht, wenn entweder `?x p1 ?y` . oder `?x p2 ?y` . gilt.

Beispiel: Professoren, die eine Rolle als Chef oder Prüfer innehaben

Wir möchten die Namen aller Professoren finden, die entweder als `:boss` eines Assistenten fungieren ODER als `:prüfer` für eine Prüfung eingetragen sind.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?profName
WHERE {
  # Finde eine Entität (?einheit), die
  # entweder einen Professor als :boss hat (Eigenschaft von ?einheit)
  # ODER einen Professor als :prüfer hat (Eigenschaft von ?einheit).
  ?einheit (:boss | :prüfer) ?professor .

  # Stelle sicher, dass ?professor auch wirklich ein Professor ist und wir den Namen bekommen.
  ?professor a :Professor ;
             :name ?profName .
}
ORDER BY ?profName
LIMIT 20
```

Erläuterung des Beispiels:

- **SELECT DISTINCT ?profName** : Wir wollen die eindeutigen Namen der Professoren.
- **WHERE { ... }** :
 - **?einheit (:boss | :prüfer) ?professor .** : Dies ist der Kern des Alternativpfads.
 - Der Property Path `(:boss | :prüfer)` steht an der **Prädikatstelle**.
 - Er besagt: Finde eine beliebige Entität `?einheit`, die entweder eine `:boss`-Beziehung zu einer Ressource `?professor` hat, ODER eine `:prüfer`-Beziehung zu einer Ressource `?professor` hat.
 - Wenn eine `:boss`-Beziehung matcht, wird `?einheit` an einen `:Assistent` gebunden und `?professor` an dessen Chef.
 - Wenn eine `:prüfer`-Beziehung matcht, wird `?einheit` an eine `:Prüfung` gebunden und `?professor` an den Prüfer.
 - **?professor a :Professor ; :name ?profName .** : Dieses Muster stellt sicher, dass die über den Alternativpfad als Objekt (`?professor`) gefundene Ressource tatsächlich vom Typ `:Professor` ist (was durch die `rdfs:range` Definitionen Ihrer Eigenschaften `:boss` und `:prüfer` ohnehin der Fall sein sollte) und bindet deren Namen an `?profName`.

Das Ergebnis ist eine Liste von Namen von Professoren, von denen jeder entweder mindestens einen Assistenten als `:boss` hat oder mindestens einmal als `:prüfer` in einer Prüfung fungiert.

Pfade mit Wiederholungen (*, +, ?)

Diese Operatoren erlauben es, Eigenschaften null-mal, einmal oder beliebig oft auf einem Pfad zu durchlaufen:

- **p* (Zero-or-more / Stern-Operator)**: Die Eigenschaft `p` kommt null oder mehrmals vor. `?x p* ?y` matcht, wenn `?y` von `?x` über einen Pfad aus null oder mehr `p`-Kanten erreichbar ist. (Bei null Vorkommen ist `?x` gleich `?y`).
- **p+ (One-or-more / Plus-Operator)**: Die Eigenschaft `p` kommt einmal oder mehrmals vor. `?x p+ ?y` matcht, wenn `?y` von `?x` über einen Pfad aus mindestens einer `p`-Kante erreichbar ist.
- **p? (Zero-or-one / Fragezeichen-Operator)**: Die Eigenschaft `p` kommt null oder einmal vor. `?x p? ?y` matcht, wenn entweder `?x p ?y` . gilt oder `?x` gleich `?y` ist (Pfadlänge null).

Beispiel: Alle direkten und indirekten Voraussetzungen für die Vorlesung "Wissenschaftstheorie" (:vorlesung5052)

Eine Vorlesung kann Voraussetzungen haben, die wiederum Voraussetzungen haben. Wir wollen alle auflisten.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?voraussetzung ?titel
WHERE {
  :vorlesung5052 :hatVoraussetzung+ ?voraussetzung . # Pfad der Länge 1 oder Länger
  ?voraussetzung :titel ?titel .
}

```

Erläuterung:

- `:vorlesung5052 :hatVoraussetzung+ ?voraussetzung .` findet alle Vorlesungen `?voraussetzung`, die direkt oder indirekt (über eine Kette von `:hatVoraussetzung`-Beziehungen) Voraussetzung für `:vorlesung5052` sind.

- Wenn Sie auch `:vorlesung5052` selbst in der Liste hätten wollen (Pfadlänge 0), würden Sie `:hatVoraussetzung*` verwenden.

Inverse Pfade (`^property`)

Manchmal möchte man eine Beziehung in umgekehrter Richtung abfragen (vom Objekt zum Subjekt). Hierfür dient der Invers-Operator `^`.

Syntax: `^p` bedeutet "die inverse Eigenschaft von p". `?x ^p ?y` ist äquivalent zu `?y p ?x`.

Beispiel: Alle Studenten, die die Vorlesung "Ethik" (`:vorlesung5041`) hören

Wir starten bei der Vorlesung "Ethik" und suchen "rückwärts" nach den Studenten.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?studentName
WHERE {
  :vorlesung5041 ^:hört ?student . # Finde ?student, sodass ?student :hört :vorlesung5041 gilt
  ?student :name ?studentName .
}
```

Weitere Aspekte von Property Paths

- **Kombinationen:** Alle diese Operatoren können komplex kombiniert und mit Klammern (`()`) gruppiert werden, um die Präzedenz zu steuern (z.B. `(p1/p2)|p3` oder `p1/(p2|p3)*`).
- **Negated Property Sets:** Es gibt auch Konstrukte wie `!p` (jede Eigenschaft außer p) oder `!(p1|p2)` (jede Eigenschaft außer p1 oder p2), die aber seltener verwendet werden und deren Unterstützung variieren kann. Sie gehen über den Rahmen dieser Einführung hinaus.

Property Paths sind ein sehr ausdrucksstarkes Mittel in SPARQL, um komplexe Beziehungen in RDF-Graphen aufzuspüren, ohne jede Zwischenstufe explizit im Query benennen zu müssen. Sie machen viele Abfragen deutlich kürzer und intuitiver.

Bereitstellen von Inline-Daten mit `VALUES`

Manchmal möchten wir eine SPARQL-Anfrage mit einer festen Menge von Werten parametrisieren, ohne diese Werte direkt in komplexe `FILTER`-Bedingungen (z.B. viele `||` (ODER) Verknüpfungen) einbauen zu müssen. Die `VALUES`-Klausel bietet hierfür eine saubere und strukturierte Möglichkeit, Variablen direkt innerhalb der Anfrage an eine oder mehrere vordefinierte Wertekombinationen zu binden.

Konzept und Funktionsweise:

Mit `VALUES` können Sie eine Art "Inline-Tabelle" von Daten erstellen. Jede "Zeile" dieser Tabelle stellt eine mögliche Bindung für die angegebenen Variablen dar. Wenn die `WHERE`-Klausel dann ausgewertet wird, werden nur Lösungen berücksichtigt, bei denen die Variablen die in der `VALUES`-Klausel vorgegebenen Werte annehmen.

Syntax:

Es gibt zwei Hauptformen:

1. Für eine einzelne Variable: `VALUES ?variable { wert1 wert2 wert3 ... }`
2. Für mehrere Variablen (als Tupel): `VALUES (?variable1 ?variable2 ...) { (wert1a wert1b ...) (wert2a wert2b ...) ... }` Hierbei muss die Anzahl der Werte in jedem Tupel der Anzahl der Variablen entsprechen.

Die `VALUES`-Klausel wird üblicherweise innerhalb der `WHERE`-Klausel platziert, oft am Anfang, um die möglichen Werte für Variablen frühzeitig einzuschränken.

`VALUES` mit einer einzelnen Variable

Beispiel: Informationen zu spezifischen Studenten abrufen

Wir möchten die Namen und Semesteranzahlen von einer festen Liste von Studenten (identifiziert durch ihre IRIs) abrufen.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?student ?studentName ?semesterAnzahl
WHERE {
  VALUES ?student { :student28106 :student25403 } # Binde ?student an diese beiden IRIs

  ?student a :Student ;
    :name ?studentName ;
    :semester ?semesterAnzahl .
}
```

Erläuterung:

- VALUES ?student { :student28106 :student25403 } sorgt dafür, dass das nachfolgende Graphmuster nur für die Studenten :student28106 und :student25403 ausgewertet wird.
- Für jede dieser beiden IRIs werden dann Name und Semesteranzahl gesucht.
- Dies ist oft lesbarer und manchmal performanter als ein langer FILTER (?student = :student28106 || ?student = :student25403) .

VALUES mit mehreren Variablen

Diese Form ist nützlich, um Kombinationen von Werten vorzugeben, die als Grundlage für die weitere Mustererkennung dienen oder um "Parameter" in die Anfrage einzuspeisen.

Beispiel: Informationen zu spezifischen Professoren abrufen und deren Rang mit einem Soll-Rang vergleichen

Wir möchten Informationen (Name, tatsächlicher Rang) zu Professoren "Sokrates" und "Kopernikus" finden. Zusätzlich haben wir für diese Professoren "Soll-Ränge" (hier für beide "W3"), die wir direkt in der Anfrage definieren und mit ausgeben wollen, um sie eventuell mit dem tatsächlichen Rang zu vergleichen.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/> # Basis-Namespace für unsere lokalen Begriffe (Klassen,
Eigenschaften, Instanzen)
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
# xsd: wird hier nicht explizit benötigt, da Turtle die Literaltypen (String, Integer) erkennt

SELECT ?professor ?profNameSoll ?profRangSoll ?profRang
WHERE {
  VALUES (?profNameSoll ?profRangSoll) { # Definiere Tupel von "Soll-Werten" oder "Suchparametern"
    ("Sokrates" "W3")
    ("Kopernikus" "W3") # Kopernikus hat laut Beispieldaten Rang W2, der Soll-Wert hier ist W3
  }

  ?professor a :Professor ;          # :Professor aus dem Default-Namespaces
            :name ?profNameSoll ;    # :name aus dem Default-Namespaces. Findet Profs mit diesem Namen.
            :rang ?profRang .        # :rang aus dem Default-Namespaces. Gibt den *tatsächlichen* Rang dieses
Professors zurück.
}
```

Erläuterung:

- VALUES (?profNameSoll ?profRangSoll) { ("Sokrates" "W3") ("Kopernikus" "W3") } : Diese Klausel definiert zwei Paare von Werten. In jeder Lösung des WHERE -Blocks wird ?profNameSoll entweder an "Sokrates" oder "Kopernikus" gebunden sein, und ?profRangSoll wird (basierend auf dem jeweiligen Tupel aus der VALUES -Klausel) an "W3" gebunden sein.
- ?professor a :Professor ; :name ?profNameSoll ; :rang ?profRang . :
 - Die Anfrage sucht nach Ressourcen ?professor , die vom Typ :Professor sind.
 - Zusätzlich muss der Wert der Eigenschaft :name dieses Professors mit dem Wert von ?profNameSoll aus der aktuellen VALUES -Zeile übereinstimmen (also entweder "Sokrates" oder "Kopernikus").
 - Für den so gefundenen Professor wird dessen *tatsächlicher* Rang (der Wert der Eigenschaft :rang) aus dem Graphen gelesen und an die Variable ?profRang gebunden. *Wichtig:* Der Wert von ?profRangSoll aus der VALUES -Klausel wird hier **nicht** verwendet, um die Professoren anhand ihres Ranges zu *filtern*. Er wird nur als Wert bereitgestellt. Die Query findet also Sokrates (dessen Rang in den Daten "W3" ist) und Kopernikus (dessen Rang in den Daten "W2" ist).
- SELECT ?professor ?profNameSoll ?profRangSoll ?profRang : Die SELECT -Klausel gibt aus:
 - ?professor : Die IRI des gefundenen Professors.
 - ?profNameSoll : Den Namen, der aus der VALUES -Klausel für die Übereinstimmung verwendet wurde.
 - ?profRangSoll : Den "Soll-Rang", der in der VALUES -Klausel neben dem Namen stand.
 - ?profRang : Den *tatsächlichen* Rang des Professors, wie er im RDF-Graphen gespeichert ist.

Diese Art der Anfrage ist nützlich, um vordefinierte Parameter (hier Name und ein erwarteter/Soll-Rang) mit den tatsächlichen Daten im Graphen abzugleichen und beides gemeinsam auszugeben.

Wichtige Anwendungsfälle für VALUES :

- Parametrisierung von Anfragen:** Übergabe einer Liste von Entitäten oder Werten, für die Informationen abgerufen werden sollen.
- Vermeidung langer FILTER IN (...) oder || -Ketten:** Macht die Anfrage übersichtlicher.
- Erstellung von "Mapping-Tabellen"** direkt in der Anfrage, um z.B. Kürzel auf Langformen abzubilden, die dann weiterverwendet werden.

Die VALUES -Klausel ist ein flexibles Werkzeug, um die Menge der zu prüfenden Daten in einer SPARQL-Anfrage von vornherein oder an einer bestimmten Stelle im Query gezielt einzuschränken.

Variablen binden und Werte dynamisch erzeugen mit BIND

Bisher haben wir Variablen in SPARQL primär durch das Matchen von Graphmustern in der `WHERE`-Klausel an Werte aus dem RDF-Graphen gebunden. Manchmal möchten wir jedoch:

- Neue Werte basierend auf vorhandenen Variablen berechnen (z.B. arithmetische Operationen).
- Komplexe Ausdrücke zur besseren Lesbarkeit oder Wiederverwendung einer neuen Variable zuweisen.
- Werte für die spätere Verwendung in Filtern, Gruppierungen oder der `SELECT`-Klausel vorbereiten.

Hierfür stellt SPARQL die **BIND**-Klausel zur Verfügung.

Konzept und Funktionsweise:

Die `BIND`-Klausel weist das Ergebnis eines Ausdrucks einer Variablen zu. Diese Zuweisung wird Teil der aktuellen Lösung(en).

Syntax: `BIND (Ausdruck AS ?neueVariable)`

- **Ausdruck** : Kann ein konstanter Wert, eine andere Variable oder eine Funktion (z.B. arithmetisch, String-Manipulation, Datumsfunktionen) sein, die auf Variablen oder Konstanten operiert.
- **AS ?neueVariable** : Der Variablenname, an den das Ergebnis des Ausdrucks gebunden wird. Wenn die Variable bereits existiert und in diesem Lösungsschritt gebunden ist, kann ihr Wert (je nach SPARQL-Prozessor und Kontext) überschrieben werden, üblicherweise wird `BIND` aber verwendet, um neue Variablen zu erzeugen oder Variablen zu binden, die in optionalen Zweigen noch nicht gebunden wurden.

Die `BIND`-Klausel wird innerhalb der `WHERE`-Klausel platziert. Die neu gebundene Variable ist dann für nachfolgende Muster, `FILTER`-, `GROUP BY`-, `HAVING`-, `ORDER BY`-Klauseln und natürlich in der `SELECT`-Klausel im selben Geltungsbereich verfügbar.

Berechnungen mit BIND

Eine häufige Anwendung ist die Durchführung von Berechnungen.

Beispiel: Restsemester bis zur Regelstudienzeit berechnen (illustrativ)

Wir möchten für jeden Studenten anzeigen, wie viele Semester ihm noch bis zu einer angenommenen Regelstudienzeit von 6 Semestern "fehlen".

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> # Wichtig für numerische Operationen

SELECT ?studentName ?aktuellesSemester ?restSemesterBisRegelstudienzeit
WHERE {
    ?student a :Student ;
             :name ?studentName ;
             :semester ?aktuellesSemester . # Ihre Eigenschaft für die Semesteranzahl

    # Annahme: Regelstudienzeit ist 6 Semester
    BIND ((6 - ?aktuellesSemester) AS ?restSemesterBisRegelstudienzeit)
}
ORDER BY ?studentName
LIMIT 10
```

Erläuterung:

- Für jeden Studenten wird das aktuelle Semester (`?aktuellesSemester`) ermittelt.
- `BIND ((6 - ?aktuellesSemester) AS ?restSemesterBisRegelstudienzeit)` berechnet die Differenz zur angenommenen Regelstudienzeit von 6 Semestern. Das Ergebnis wird an die neue Variable `?restSemesterBisRegelstudienzeit` gebunden.
- Das Ergebnis zeigt den Studentennamen, sein aktuelles Semester und die berechneten "Restsemester".
- Beachten Sie, dass arithmetische Operationen auf Literalen mit passenden numerischen Datentypen (hier `xsd:integer` für `:semester`) am besten funktionieren.

String-Manipulation mit BIND

`BIND` kann auch verwendet werden, um Strings zu verketteten oder zu modifizieren.

Beispiel: Vollständiger Name und Titel für Professoren generieren

Wir wollen für jeden Professor eine einzelne Zeichenkette ausgeben, die "Prof. Dr. [Name]" oder ähnlich lautet (vereinfacht hier, da nicht alle Professoren einen Dr.-Titel haben müssen oder wollen).

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?professor ?formellerName
WHERE {
```

```

?professor a :Professor ;
          :name ?name .

BIND (CONCAT("Prof. ", ?name) AS ?formellerName) # CONCAT für String-Verkettung
}
LIMIT 10

```

Erläuterung:

- `CONCAT("Prof. ", ?name)` nimmt den String "Prof. " und den Wert der Variablen `?name` und verbindet sie zu einem neuen String, der an `?formellerName` gebunden wird. SPARQL bietet diverse String-Funktionen wie `STRLEN`, `SUBSTR`, `UCASE`, `LCASE`, `REPLACE` etc.

Extrahieren von Teilen aus IRIs oder Literalen

Manchmal möchte man Teile von IRIs oder komplexeren Literalen extrahieren.

Beispiel: Lokalen Namen aus einer IRI extrahieren (wenn keine Präfixe im Ergebnis gewünscht sind)

Angenommen, wir wollen nur den lokalen Teil der Studenten-IRI (also den Teil nach dem letzten `/` oder `#`, je nach Ihrer IRI-Struktur).

```

%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/> # Ihre Basis-IRI
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?student ?name ?lokalerTeilStudentIRI
WHERE {
  ?student a :Student ;
          :name ?name .

  # Annahme: Ihre Studenten-IRIs sind z.B. <http://data.rwth-aachen.de/resources/studentXXXXX>
  BIND (STRAFTER(STR(?student), "http://data.rwth-aachen.de/resources/") AS ?lokalerTeilStudentIRI_temp)
  # Falls Ihre IRIs ein # verwenden, z.B. <.../resources#studentXXXXX>, dann:
  # BIND (STRAFTER(STR(?student), "#") AS ?lokalerTeilStudentIRI_temp)

  # Optionale Verbesserung: Nur binden, wenn STRAFTER erfolgreich war und nicht Leer ist.
  # Ansonsten den vollen IRI-String nehmen.
  BIND (IF(BOUND(?lokalerTeilStudentIRI_temp) && STRLEN(?lokalerTeilStudentIRI_temp) > 0, ?
lokalerTeilStudentIRI_temp, STR(?student)) AS ?lokalerTeilStudentIRI)
}
LIMIT 5

```

Erläuterung:

- `STR(?student)` wandelt die IRI `?student` in einen String um.
- `STRAFTER(string, trenner)` gibt den Teil des Strings *nach* dem ersten Vorkommen des `trenner`-Strings zurück. Hier muss der `trenner` exakt dem Basisteil Ihrer IRIs entsprechen, den Sie entfernen möchten.
- Die zweite `BIND`-Anweisung mit `IF` ist eine robustere Variante, die prüft, ob `STRAFTER` einen sinnvollen Wert geliefert hat, bevor sie ihn zuweist.

Anwendungsbereiche für `BIND` :

- Durchführung von Berechnungen für die Ausgabe oder für Filterbedingungen.
- Normalisierung oder Umformatierung von Daten (z.B. Strings, Datumsangaben).
- Erstellung von komplexen Werten, die in `GROUP BY` oder `ORDER BY` verwendet werden sollen.
- Verbesserung der Lesbarkeit von Anfragen, indem komplexe Ausdrücke einer benannten Variable zugewiesen werden.

Die `BIND`-Klausel ist ein flexibles Werkzeug, um die Aussagekraft und Anpassungsfähigkeit Ihrer SPARQL-Anfragen deutlich zu erhöhen.

Strukturierung komplexer Anfragen: Subqueries (Unteranfragen)

Mit zunehmender Komplexität unserer Abfragen kann es hilfreich werden, Teile der Logik in separate, in sich geschlossene Blöcke zu zerlegen. SPARQL ermöglicht dies durch **Subqueries** (auch Unteranfragen oder Sub-SELECTs genannt). Eine Subquery ist im Wesentlichen eine `SELECT`-Anfrage, die innerhalb der `WHERE`-Klausel einer äußeren Anfrage eingebettet ist.

Konzept und Funktionsweise:

1. **Auswertung der Subquery:** Die Subquery wird zuerst (oder zumindest konzeptuell zuerst) ausgewertet. Ihr Ergebnis ist eine Tabelle von Variablenbindungen (eine Lösungssequenz), genau wie bei einer normalen `SELECT`-Anfrage.
2. **Verwendung der Ergebnisse:** Die von der Subquery erzeugten Variablenbindungen werden dann als Eingabe für die Muster der äußeren Anfrage verwendet. Das bedeutet, die äußere Anfrage operiert auf den Ergebnissen der inneren Anfrage.
3. **Variablen-Scope:** Nur die Variablen, die in der `SELECT`-Klausel der Subquery aufgeführt sind, sind für die äußere Anfrage sichtbar und können dort weiterverwendet werden. Variablen, die nur innerhalb der `WHERE`-Klausel der Subquery verwendet werden, sind für die äußere Anfrage nicht zugänglich.

Syntax: Die Subquery wird in geschweifte Klammern { ... } eingeschlossen, genau wie ein normales Graphmuster.

```
SELECT ... WHERE { # Äußere Graphmuster { # Beginn der Subquery SELECT ?var1 ?var2 ... WHERE { ... } #  
Innere Graphmuster der Subquery # Modifikatoren für die Subquery (ORDER BY, LIMIT etc.) } # Ende der Subquery  
# Weitere äußere Graphmuster, die ?var1, ?var2 etc. verwenden können } # Modifikatoren für die äußere Query
```

Warum Subqueries verwenden?

- **Anwendung von Lösungsmodifikatoren auf Teilergebnisse:** LIMIT , OFFSET , ORDER BY , DISTINCT können auf die Ergebnisse einer Subquery angewendet werden, bevor diese in der äußeren Query weiterverarbeitet werden.
- **Vorbereitung oder Vorfilterung von Daten:** Komplexe Filter oder Berechnungen können in einer Subquery gekapselt werden.
- **Aggregationen als Zwischenschritt:** Ergebnisse von Aggregatfunktionen aus einer Subquery können in der äußeren Query verwendet werden.
- **Verbesserung der Lesbarkeit und Modularität:** Sehr komplexe Anfragen können in logische Blöcke unterteilt werden.

Subquery zur Vorfilterung mit LIMIT und ORDER BY

Ein häufiger Anwendungsfall ist, zuerst eine sortierte und limitierte Menge von Ressourcen zu finden und dann weitere Informationen zu diesen spezifischen Ressourcen abzurufen.

Beispiel: Die Namen und Vorlesungen der zwei Studenten mit den höchsten Semesterzahlen

Wir wollen zuerst die zwei Studenten finden, die am längsten studieren, und dann für diese beiden Studenten ihre Namen und die von ihnen gehörten Vorlesungen (Titel) ausgeben.

```
%%rdf sparql -l uni  
PREFIX : <http://data.rwth-aachen.de/resources/>  
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  
  
SELECT ?studentName ?vorlesungTitel  
WHERE {  
  { # Subquery beginnt: Finde die Top 2 Studenten nach Semesteranzahl  
    SELECT ?student  
    WHERE {  
      ?student a :Student ;  
               :semester ?semesterAnzahl . # Annahme: :semester für Semesteranzahl  
    }  
    ORDER BY DESC(?semesterAnzahl) # Sortiere absteigend nach Semester  
    LIMIT 2 # Nimm nur die ersten beiden  
  } # Subquery endet  
  
  # Hauptquery verwendet die ?student Ergebnisse (IRIs) aus der Subquery  
  ?student :name ?studentName ;  
           :hört ?vorlesung . # Annahme: :hoert für Student hört Vorlesung  
  ?vorlesung :titel ?vorlesungTitel . # Annahme: :titel für Vorlesungstitel  
}
```

Erläuterung:

1. **Subquery:**
 - Wählt die Variable ?student aus.
 - Findet alle Studenten und ihre Semesteranzahl.
 - Sortiert diese absteigend nach der Semesteranzahl.
 - LIMIT 2 beschränkt das Ergebnis der Subquery auf die IRIs der zwei Studenten mit den höchsten Semesterzahlen.
2. **Äußere Query:**
 - Die WHERE -Klausel der äußeren Query erhält die zwei ?student -IRIs aus der Subquery.
 - Für jede dieser beiden Studenten-IRIs werden dann der :name und die :hoert -Beziehungen zu Vorlesungen (?vorlesung) gesucht.
 - Schließlich wird der :titel der jeweiligen Vorlesung ermittelt.

Ohne Subquery wäre es schwierig, zuerst die Top-N-Studenten zu bestimmen und *dann* deren weitere Details abzufragen, da LIMIT global am Ende der Query wirkt.

Subquery mit Aggregation zur Vorselektion

Subqueries sind auch sehr nützlich, um Daten vorzuaggregieren und dann mit diesen aggregierten Werten in der äußeren Query weiterzuarbeiten.

Beispiel: Namen der Professoren, die mehr als eine Vorlesung lehren (und die Anzahl)

Wir wollen die Namen der Professoren und die Anzahl der von ihnen gehaltenen Vorlesungen, aber nur für diejenigen Professoren, die mehr als eine Vorlesung halten.

```
%%rdf sparql -l uni
```

```

PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?profName ?anzahlVorlesungen
WHERE {
  { # Subquery beginnt: Finde Professoren und zähle ihre Vorlesungen, filtere nach Anzahl > 1
    SELECT ?professor (COUNT(?vorlesung) AS ?anzahlVorlesungen)
    WHERE {
      ?professor a :Professor .
      ?vorlesung :gelesenVon ?professor .
    }
    GROUP BY ?professor
    HAVING (COUNT(?vorlesung) > 1) # Behalte nur Profs mit mehr als einer Vorlesung
  } # Subquery endet

  # Hauptquery: Hole den Namen für die qualifizierten Professoren
  ?professor :name ?profName .
}
ORDER BY DESC(?anzahlVorlesungen) ?profName

```

Erläuterung:

1. Subquery:

- Wählt ?professor und die berechnete ?anzahlVorlesungen.
- Findet alle Professoren und die von ihnen gelehrten Vorlesungen.
- GROUP BY ?professor gruppiert die Ergebnisse pro Professor.
- COUNT(?vorlesung) zählt die Vorlesungen für jeden Professor.
- HAVING (COUNT(?vorlesung) > 1) filtert diese Gruppen und behält nur Professoren, die mehr als eine Vorlesung halten.
- Das Ergebnis dieser Subquery ist eine Tabelle mit zwei Spalten: professor (IRI) und anzahlVorlesungen.

2. Äußere Query:

- Die äußere Query nimmt die ?professor -IRIs und die zugehörige ?anzahlVorlesungen aus der Subquery.
- Mit ?professor :name ?profName . wird dann der Name zu jedem dieser Professoren geholt.
- SELECT ?profName ?anzahlVorlesungen gibt den Namen und die bereits in der Subquery berechnete Anzahl aus.

Subqueries können die Struktur komplexer Anfragen erheblich verbessern und Operationen ermöglichen, die sonst nur schwer oder umständlich auszudrücken wären. Sie sind ein wichtiges Werkzeug für fortgeschrittene SPARQL-Nutzer.

Daten aus verteilten Quellen integrieren: Föderierte Anfragen mit SERVICE

Eines der Kernprinzipien des Semantic Web und von Linked Data ist die Idee, dass Daten nicht an einem einzigen Ort gespeichert sein müssen, sondern über zahlreiche verschiedene Server und SPARQL-Endpunkte im Web verteilt sein können. Die SERVICE -Klausel in SPARQL ermöglicht es, Teile einer Anfrage an einen **anderen (entfernten) SPARQL-Endpunkt** zu delegieren und dessen Ergebnisse in die lokale Anfrage zu integrieren.

Konzept und Funktionsweise:

Mit SERVICE <endpointIRI> { graphPattern } wird das angegebene graphPattern an den SPARQL-Endpunkt unter <endpointIRI> gesendet und dort ausgeführt.

- **Variablenübergabe:** Variablen, die im äußeren Teil der Anfrage (vor dem SERVICE -Block) bereits gebunden wurden, können innerhalb des graphPattern im SERVICE -Block verwendet werden. Der SPARQL-Prozessor sendet diese Bindungen typischerweise als Teil der Anfrage an den entfernten Endpunkt (oft durch Umschreiben der Anfrage mit einer VALUES -Klausel für die gebundenen Variablen oder durch direkte Substitution).
- **Ergebnisintegration:** Die Ergebnisse (Variablenbindungen), die vom entfernten Endpunkt zurückgegeben werden, stehen dann dem restlichen Teil der äußeren Anfrage zur Verfügung, als wären sie lokal erzeugt worden.

Syntax: SERVICE <IRI_des_Endpunkts> { # Graphmuster, das am entfernten Endpunkt ausgeführt wird }

Die SERVICE -Klausel ist Teil der WHERE -Klausel.

Anwendungsfälle:

- **Datenanreicherung:** Lokale Daten mit zusätzlichen Informationen aus großen öffentlichen Wissensgraphen (z.B. DBpedia, Wikidata) anreichern.
- **Datenintegration:** Informationen aus verschiedenen, spezialisierten SPARQL-Endpunkten (z.B. von unterschiedlichen Institutionen) in einer einzigen Anfrage kombinieren.

Einfache Föderierte Anfrage: Lokale Daten mit DBpedia anreichern

Beispiel: Deutsche Beschreibung für unseren lokalen Professor "Sokrates" von DBpedia holen

Angenommen, wir haben in unserem lokalen uni -Graphen den Professor :professor2125 mit dem Namen "Sokrates". Wir möchten nun die deutsche Zusammenfassung (Abstract) für die Entität "Sokrates" von DBpedia abrufen.

```

%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>      # Für unsere Lokalen Begriffe (ggf. anpassen, z.B. auf #
endend)
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> # Für rdfs:label
PREFIX schema: <http://schema.org/>                # Für schema:description
# Wikidata-spezifische Präfixe
PREFIX wd: <http://www.wikidata.org/entity/>         # Wikidata Entitäten
PREFIX wdt: <http://www.wikidata.org/prop/direct/>    # Wikidata direkte Eigenschaften

SELECT ?localName ?wikidataBeschreibung
WHERE {
  # 1. Alle Professoren und ihre Namen aus dem Lokalen Graphen holen
  ?person a :Professor .
  ?person :name ?localName . # Annahme: :name ist die Eigenschaft für Namen in Ihrem Vokabular

  # 2. Erzeuge ein deutsch-getaggttes Literal aus dem Lokalen Namen für den Wikidata-Abgleich
  BIND(STRLANG(STR(?localName), "de") AS ?germanLabelToSearch)

  # 3. Föderierte Anfrage an Wikidata
  SERVICE <https://query.wikidata.org/sparql> {
    # Suche nach einem Wikidata-Item (?wikidataItem), das:
    # a) als Beschäftigung "Philosoph" hat (wdt:P106 ist "occupation", wd:Q4964182 ist "philosopher")
    # b) dessen deutsches rdfs:label dem ?germanLabelToSearch entspricht
    # c) und hole dessen deutsche schema:description
    ?wikidataItem wdt:P106 wd:Q4964182 ;          # Ist ein Philosoph
                  rdfs:label ?germanLabelToSearch ; # Hat den gesuchten deutschen Namen
                  schema:description ?wikidataBeschreibung .
    FILTER(LANG(?wikidataBeschreibung) = "de")    # Stelle sicher, dass die Beschreibung deutsch ist
  }
}

```

Erläuterung dieser Anfrage:

1. Lokale Datenauswahl:

- `?person a :Professor . ?person :name ?localName .`
- Zuerst werden alle Ressourcen in Ihrem lokalen `uni` -Graphen gefunden, die vom Typ `:Professor` sind, und deren `:name` wird an die Variable `?localName` gebunden.

2. Vorbereitung für den Wikidata-Abgleich:

- `BIND(STRLANG(STR(?localName), "de") AS ?germanLabelToSearch)`
- Für jeden lokalen Professorennamen `?localName` wird hier ein neues Literal `?germanLabelToSearch` erzeugt, das den gleichen String-Wert wie `?localName` hat, aber explizit mit dem deutschen Sprach-Tag (`@de`) versehen ist. `STR(?localName)` stellt sicher, dass wir den reinen String-Wert bekommen, falls `?localName` selbst schon einen (anderen) Sprach-Tag hätte. Dies ist wichtig, da `rdfs:label` auf Wikidata stark sprachgetaggt ist.

3. SERVICE <https://query.wikidata.org/sparql> { ... } :

- Dieser Block sendet für jede Bindung von `?localName` (bzw. `?germanLabelToSearch`) eine Anfrage an den Wikidata SPARQL-Endpunkt.
- `?wikidataItem wdt:P106 wd:Q4964182 ;` : Sucht auf Wikidata nach Entitäten (`?wikidataItem`), die die Beschäftigung (`wdt:P106`) Philosoph (`wd:Q4964182`) haben.
- `rdfs:label ?germanLabelToSearch ;` : Von diesen Philosophen werden diejenigen ausgewählt, deren `rdfs:label` exakt mit dem zuvor erstellten `?germanLabelToSearch` (z.B. "Sokrates"@de) übereinstimmt.
- `schema:description ?wikidataBeschreibung .` : Für die gefundenen Wikidata-Items wird deren `schema:description` an `?wikidataBeschreibung` gebunden.
- `FILTER(LANG(?wikidataBeschreibung) = "de")` : Es wird sichergestellt, dass nur die deutschsprachige Beschreibung ausgewählt wird.

4. SELECT ?localName ?wikidataBeschreibung :

- Gibt schließlich Paare von lokalen Professorennamen und den (falls gefundenen) zugehörigen deutschen Wikidata-Beschreibungen zurück. Wenn für einen lokalen Professor kein passender Eintrag auf Wikidata gefunden wird oder dieser keine deutsche Beschreibung hat, erscheint dieser Professor nicht im Ergebnis.

Diese Art von föderierter Anfrage ist sehr typisch, um lokale Datensätze mit den umfangreichen Informationen aus globalen Wissensgraphen wie Wikidata anzureichern.

SERVICE SILENT für robustere Anfragen

Externe Endpunkte sind nicht immer verfügbar oder können Fehler zurückliefern. Wenn ein `SERVICE` -Aufruf fehlschlägt, schlägt standardmäßig die gesamte SPARQL-Anfrage fehl. Um dies zu verhindern, kann `SILENT` verwendet werden:

Syntax: `SERVICE SILENT <IRI_des_Endpunkts> { graphPattern }`

Wenn der `SERVICE SILENT` -Aufruf fehlschlägt (z.B. Timeout, Fehler vom Endpunkt), werden für diesen Teil der Anfrage einfach keine

Lösungen/Bindungen erzeugt, aber die äußere Anfrage läuft weiter.

Beispiel: Optionale Anreicherung mit SERVICE SILENT

Wir holen die Namen aller unserer lokalen Professoren und versuchen *optional und fehlertolerant*, deren Geburtsdatum von DBpedia zu bekommen.

```
%%rdf sparql -l uni
PREFIX : <http://data.rwth-aachen.de/resources/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX schema: <http://schema.org/>

SELECT ?localProfName ?birthDate
WHERE {
  ?profLocal a :Professor ;
             :name ?localProfName .

  OPTIONAL { # Wir wollen alle Professoren, auch wenn DBpedia nichts liefert
    SERVICE SILENT <https://dbpedia.org/sparql> {
      ?dbpediaPerson rdfs:label ?localProfName ; # Suche nach dem Namen
                    a dbo:Person ;              # Sollte eine Person sein
                    dbo:birthDate ?birthDate . # dbo:birthDate oder schema:birthDate
    }
  }
}
LIMIT 10
```

Erläuterung:

- Die äußere WHERE -Klausel findet alle lokalen Professoren und ihre Namen.
- Der OPTIONAL -Block stellt sicher, dass wir auch dann Ergebnisse für lokale Professoren bekommen, wenn der SERVICE -Aufruf keine passenden Daten liefert oder fehlschlägt.
- SERVICE SILENT innerhalb des OPTIONAL -Blocks versucht, das Geburtsdatum von DBpedia zu holen. Schlägt dies fehl oder gibt es kein Ergebnis, bleibt ?birthDate für diesen Professor ungebunden, aber der Professor erscheint trotzdem in der Ergebnisliste.

Wichtige Überlegungen bei Föderierten Anfragen:

- **Performance:** Können langsam sein, abhängig von der Geschwindigkeit und Auslastung der externen Endpunkte und der Komplexität der Sub-Anfrage.
- **Zuverlässigkeit:** Die Verfügbarkeit externer Endpunkte kann nicht garantiert werden (SILENT kann hier helfen).
- **Datenqualität und -mapping:** Die Verknüpfung von lokalen Daten mit externen Daten erfordert oft sorgfältiges Mapping (z.B. über owl:sameAs -Links oder Übereinstimmung von Namen/Labels, was fehleranfällig sein kann).

Föderierte Anfragen mit SERVICE sind ein Kernstück des Linked Data Prinzips und ermöglichen es, das "Web of Data" direkt durch SPARQL-Anfragen zu nutzen und zu verbinden.

Finale Betrachtungen

Best Practices und Tipps

Performance-Optimierung

1. **Spezifische Muster zuerst:** Setzen Sie restriktive Bedingungen (die die Anzahl der möglichen Lösungen stark einschränken) früh im WHERE -Block.
2. **LIMIT verwenden:** Begrenzen Sie Ergebnisse bei explorativen Abfragen, um nicht von der Datenmenge überwältigt zu werden.
3. **INDEX-freundliche Abfragen:** Vermeiden Sie REGEX oder komplexe FILTER -Operationen auf ungebundene Variablen oder sehr große Zwischenergebnismengen, wenn einfachere Muster oder spezifischere String-Funktionen ausreichen.
4. **OPTIONAL sparsam und gezielt einsetzen:** Unnötige oder zu breite OPTIONAL -Blöcke können die Performance beeinträchtigen.
5. **Projektionen (SELECT) minimieren:** Fordern Sie nur die Variablen an, die Sie wirklich benötigen. SELECT * kann bei breiten Daten ineffizient sein.

Debugging von SPARQL-Abfragen

1. **Schrittweise Entwicklung:** Beginnen Sie mit einfachen Mustern und fügen Sie Komplexität schrittweise hinzu.
2. **Zwischenergebnisse prüfen:** Verwenden Sie temporäre SELECT -Abfragen mit LIMIT , um Teile Ihrer Abfrage zu testen und zu sehen, welche Variablen gebunden werden.
3. **OPTIONAL isoliert testen:** Wenn ein OPTIONAL -Block nicht das erwartete Ergebnis liefert, testen Sie das Muster innerhalb des OPTIONAL separat als eigenständige Abfrage.
4. **Syntax prüfen:** Achten Sie auf korrekte Punktsetzung (Punkte am Ende von Tripel-Patterns, Semikolons), Klammern und Präfixe.
5. **Datentypen beachten:** Fehler können entstehen, wenn Operationen auf inkompatiblen Datentypen ausgeführt werden (z.B. arithmetische

Operationen auf Strings).

Zusammenfassung und Ausblick

Was wir gelernt haben

- **Grundlagen:** SELECT , WHERE , FILTER , PREFIX
- **Werte direkt binden:** VALUES
- **Erweiterte Filter:** String-Funktionen (CONTAINS , STRSTARTS , REGEX), Datentyp- (DATATYPE) und Sprachfilter (LANGMATCHES)
- **Negation:** MINUS , FILTER NOT EXISTS
- **Verknüpfungen:** Implizite Joins durch gemeinsame Variablen
- **Property Paths:** Grundlagen für Pfadabfragen (+ , * , / , |)
- **Aggregation:** COUNT , AVG , MIN , MAX , SUM , GROUP BY
- **Optionale Muster:** OPTIONAL
- **Alternative Muster:** UNION , BIND
- **Sortierung und Begrenzung:** ORDER BY , LIMIT , OFFSET
- **Verschiedene Abfragetypen:** CONSTRUCT , ASK , DESCRIBE
- **Externe Endpoints:** Abfragen an DBpedia und Wikidata

Weiterführende Themen

- **SPARQL 1.1 Features:** Fortgeschrittene Property Paths, Subqueries (SELECT innerhalb von SELECT), BINDINGS , SERVICE für föderierte Abfragen (kurz gezeigt).
- **SPARQL Update:** INSERT DATA , DELETE DATA , MODIFY zum Verändern von RDF-Daten. (hier noch nicht behandelt)
- **Reasoning und Inferenz:** Nutzung von RDFS/OWL-Schemata, um implizites Wissen in Abfragen einzubeziehen (z.B. `rdfs:subClassOf`). (hier noch nicht behandelt)
- **Named Graphs:** Abfragen über mehrere benannte Graphen innerhalb eines RDF-Stores. (oben erwähnt, aber hier noch nicht behandelt)
- **Performance Tuning:** Fortgeschrittene Optimierungsstrategien für große Datenmengen und komplexe Abfragen. (hier noch nicht behandelt)

Nützliche Ressourcen

- [W3C SPARQL 1.1 Query Language Specification](#)
- [SPARQL.org](#) (Viele nützliche Dienste)
- [Bob DuCharme: Learning SPARQL](#)
- [Apache Jena SPARQL Tutorial](#)
- [Wikidata Query Service](#) (Mit vielen Beispielen)
- [DBpedia SPARQL Endpoint](#)
- [Anton's SPARQL Parse Tree Viewer](#)

SPARQL ist ein mächtiges Werkzeug für die Arbeit mit semantischen Daten. Mit den hier erlernten Grundlagen können Sie komplexe Abfragen auf RDF-Daten durchführen und die Potentiale des Semantic Web nutzen.

In []: