Izvlačenje povezanih podataka iz strojarskih dokumenata s mreže

Topić, Jakov

Undergraduate thesis / Završni rad

2015

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Mechanical Engineering and Naval Architecture / Sveučilište u Zagrebu, Fakultet strojarstva i brodogradnje**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:235:842365

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-07-18**

Repository / Repozitorij:

Repository of Faculty of Mechanical Engineering and Naval Architecture University of Zagreb

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Jakov Topić

Zagreb, 2015.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Mentor:
Prof.dr.sc. Mario Essert

Student:
Jakov Topić

Zagreb, 2015.

Zahvala

Izjavljujem da sam ovaj rad izradio samostalno koristeći stečena znanja tijekom studija i navedenu literaturu.

Ovom prilikom želio bih se zahvaliti:

- Voditelju rada Prof.dr.sc. Mariu Essertu što mi je omogućio izradu ovog rada, te na sugestijama i pomoći pri izradi istog.
- Obitelji i prijateljima na pomoći i potpori, kako za ovaj rad, tako i kroz sve godine studiranja.

Jakov Topić

SVEUČILIŠTE U ZAGREBU

FAKULTET STROJARSTVA I BRODOGRADNJE

Središnje povjerenstvo za završne i diplomske ispite

Povjerenstvo za završne ispite studija strojarstva za smjerove:
proizvodno inženjerstvo, računalno inženjerstvo, industrijsko inženjerstvo i menadžment, inženjerstvo materijala i mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa:	
Ur.broj:	

ZAVRŠNI ZADATAK

Student: **Jakov Topić** Mat. br.: 0152195996

Naslov rada na hrvatskom jeziku: **Izvlačenje povezanih podataka iz strojarskih dokumenata s mreže**

Naslov rada na engleskom jeziku: Extracting linked open data (LOD) from engineering web documents

Opis zadatka:

Dohvaćanje informacije jedno je od temeljnih zadaća računalne tehnologije. S obzirom na ogroman broj dokumenata na globalnoj mreži – internetu, klasično spremanje informacije u bazu postupno se gasi, jer je u svakom takvom postupku nužna čovjekova intervencija i trajno nadgledanje. Cilj je unaprijediti izvlačenje informacije automatiziranim metodama, bez sudjelovanja čovjeka. Jedno od takvih novih rješenja su povezani podaci (eng. *LOD – linked open data*) s kojima se informacija dobro povezuje, uz najmanje memorijske zahtjeve – trojce informacije preko tzv. 's-o-p' ustroja (eng. *subject-object-predicate*).

U ovom radu potrebno je:

1. Opisati *LOD* paradigmu i formate spremanja povezanih podataka (*RDF, N3, N-turtle*, ...),
2. Objasniti izvlačenje informacija iz mrežnih (WEB) dokumenata preko regularnih izraza,
3. Povezati izvučenu/ekstrahiranu informaciju sa *Strunom*, web-rječnikom http://struna.ihjj.hr strojarskog nazivlja, kako bi se napisani ili mrežno dohvaćeni tehnički tekst što lakše analizirao po normativnim značenjima povezanih riječi koji se u njemu nalaze,
4. Stvoriti LOD trojce (engl. *triplets*) za obradbu i analizu (preko *SPARQL* naredbi) u bazi trojaca (eng. *triplestore*) realiziranoj u *Virtuoso open-server* mrežnom poslužitelju.

Zadatak zadan: 25. studenog 2014.

Rok predaje rada: **1. rok:** 26. veljače 2015. **2. rok:** 17. rujna 2015.

Predviđeni datumi obrane: **1. rok:** 2., 3., i 4 . ožujka 2015. **2. rok:** 21., 22., i 23. rujna 2015.

Zadatak zadao: Prof.dr.sc. Mario Essert

Predsjednik Povjerenstva: Prof. dr. sc. Zoran Kunica

Sadržaj

Popis slika

Popis tablica

1 Uvod

U ovom radu opisana je ideja i struktura sematičkog weba čiji je temelj davanje značenja podacima tako da ih razumije računalo. Također su opisani i svi gradivni elementi koji se pri njegovoj izradi koriste, kao što su XML jezik za pohranu podataka te RDF model za opis značenja podataka. U daljnjem djelu objašnjeni su i razni načini povezivanja podataka u LOD trojce (subjekt, predikat i objekt) među koje spadaju: N-triples, Turtle, RDF/XML, JSON-LD, i ostali. Nadalje, u drugom dijelu rada obrađeni su regularni izrazi općenito te regularni izrazi u programskom jeziku Python, sa cjelokupnom im sintaksom. Na kraju je izrađena ontologija za tehnički rječnik u programskom paketu Protégé, te je ona dalje prenesena u bazu trojaca unutar Virtuoso Open Server mrežnog poslužitelja. Podaci iz tehničkog rječnika ekstrahirani su i pretvoreni u LOD zapis pomoću aplikacije izrađene u Pythonu, čija struktura u ovom radu nije obrađena. Predstavljena je i stranica izrađena putem Web2py frameworka koja služi za unos i pretraživanje podataka iz baze, u ovom slučaju trojaca.

Ključne riječi:
semantički web, povezani podaci, regularni izrazi, Python, LOD(Linked Open Data), RDF, OWL, SPARQL

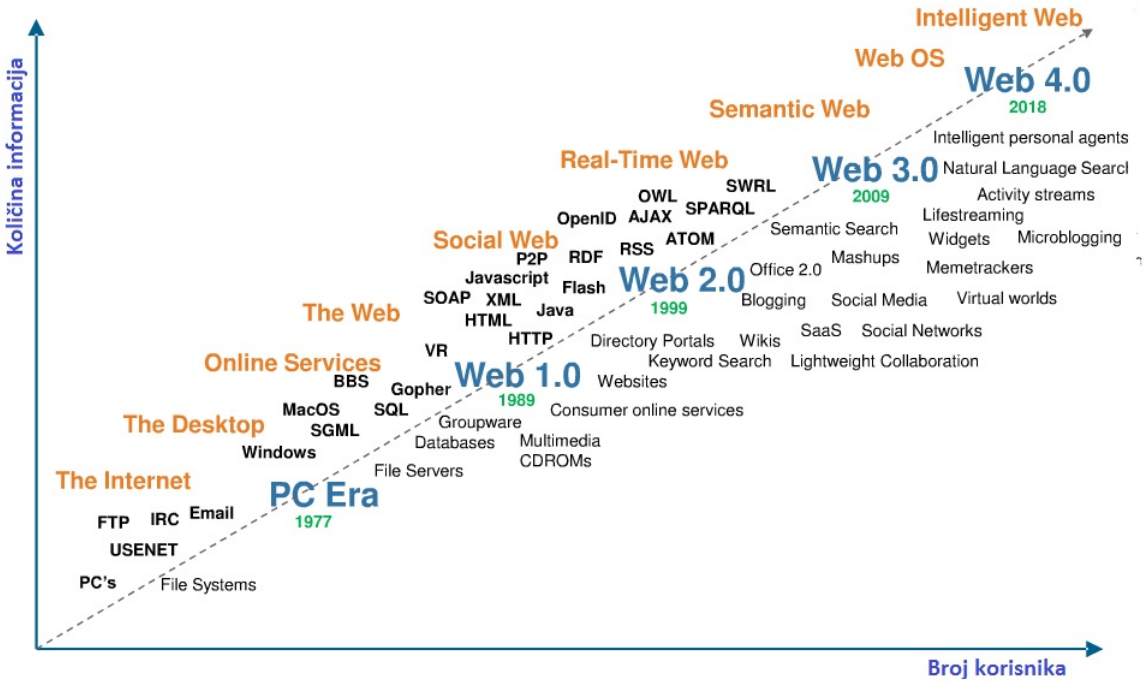
2 Semantički Web

Semantički web predstavlja sljedeću evolucijsku stepenicu u razvoju World Wide Web-a (WWW ili kraće web). U proteklom razdoblju WWW je prošao ogroman put (od alata za razmjenu i vezivanje dokumenata unutar jednog istraživačkog centra do najpopularnije i najkorištenije usluge na Internetu). Temeljeći se na nizu široko prihvaćenih standarda dostigao je neslućene razmjere. Ti vrlo važni standardi su TCP/IP protokol na koji se nadovezuje HTTP (Hypertext transfer protocol) i jezik za opis stranice HTML (Hypertext markup language). Povijesno gledano, možemo reći kako je web trenutno u svojoj drugoj generacijskoj dobi.

Prva generacija okarakterizirana je «ručnim» pisanjem/kodiranjem statičkih HTML stranica te slabim ili nikakvim dinamičkim generiranjem stranica. Sljedeći, očigledni korak je išao prema stvaranju dinamičkih stranica generiranih od strane računala koja su unaprijed iskodirana u nekom prigodnom jeziku (CGI, ASP, JSP/Servlet, PHP ...). Osnovna odlika spomenutih generacija je ta da su namjenjene za rad s ljudima/fizičkim osobama, a ne s drugim strojevima. Osobe ih mogu logički procesuirati - čitati, browsati, pretraživati te ispunjavati razne web forme.

Treća generacija cilja na novi web koji će omogućiti i strojevima da procesuiraju na višem nivou informacije koje se na njemu nalaze. Ta generacija naziva se semantički web. Semantički podrazumjeva pridjeljivanje značenja informacijama na webu. Na prvi pogled ovo može biti zbunjujuće jer je web prepun informacija, no opet se treba prisjetiti da ih samo osobe mogu razumjeti dok je strojevima, usprkos raznim «meta» tagovima, on i dalje nerazumljiv. Web trenutno ne sadržava nikakvo značenje, on dobiva značenje tek nakon ljudske interpretacije.

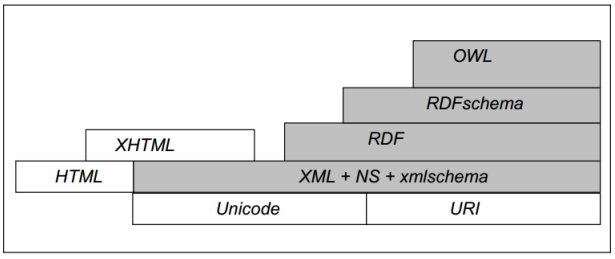
Pod patronatom World Wide Web Consortium (W3C) objedinjuje se razvoj današnjeg i budućeg weba. W3C je od kasnih 90tih godina prošlog stoljeća krenuo u promoviranje i razvoj semantičkog weba. Osnovna ideju počeo je promovirati Tim Berners-Lee, 1998. godine, kao «plan za postizanje povezanih podatkovinih aplikacija na webu u takovoj formi da oblikuju konzistentni logičku mrežu (web) podataka (sematički web)». Tim Berners-Lee se smatra ocem web-a i jedan od glavnih osoba odgovornih za novi, semantički web, čiju je viziju predstavio u dva povezana dokumenta koja opisuju semantički web iz "vrlo velikih prespektiva".



Slika 1: Razvoj web-a

2.1 Jezici semantičkog weba

Izgradnja semantičkog weba nije jednostavan proces i on se može postići samo ako su upostavljene nove razine međuoperabilnosti temeljene na otvorenim standardima. Standardi moraju pružati ne samo dobru definiciju za sintaktičku formu dokumenata, nego i za njihov sintaksni sadržaj. Ovakva semantička interoperabilnost je glavni zadatak W3C-a koja se oslanja na postojećih pet standarda te uvodi neke nove u obliku novih jezika namjenjenih webu. Shematski prikaz tih jezika dan je na slijedećoj slici:

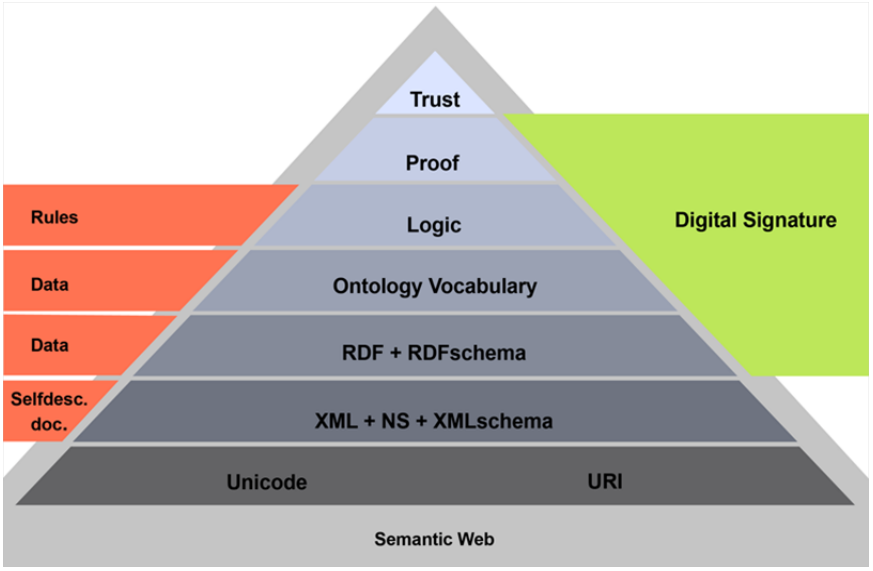


Slika 2: Prikaz i odnos jezika semantičkog weba

2.2 Slojevi semantičkog weba

Principi Semantičkog Web-a su primjenjeni na slojeve Web tehnologija i standarda:

- Unicode i URI slojevi služe da nam omoguće korištenje internacionalnog skupa znakova (engl. "character set") i pruže način za identificiranje objekata u Semantičkom Web-u.
- XML sloj sa prostorom imena i shemom definicija osigurava integraciju definicije Semantičkog Web-a sa ostalim standardima baziranim na XML-u, tj. osigurava serijalizaciju.
- RDF - sa RDF-om te RDF Shemom (RDFS) je moguće napraviti izjave o objektima sa URI-jevima i definirati rječnike kojima se može baratati preko URI-jeva. Ovo je sloj u kojem možemo dati tipove resursima i linkovima.
- Ontološki sloj podržava evoluciju rječnika, također može definirati relacije između različitih koncepata. Navedeni slojevi, zajedno sa slojem za detekciju promjena kod dokumenata (Digital Signature layer), trenutno se standardiziraju u radnoj grupi W3C.
- Slojevi na vrhu: Logic, Proof, Trust se trenutno razvijaju te se grade jednostavni demonstrativni primjeri. Logički sloj omogućava pisanje pravila, dok Proof sloj izvršava ta pravila i evaluira, zajedno sa Trust slojem, da li da se vjeruje danom dokazu (proof) ili ne.



Slika 3: Slojevi semantičkog web-a

3 Povezani Podaci (engl. Linked Data)

Povećanje količine podataka dostupnih putem intraneta, a posebno interneta (World Wide Web), donijelo je niz prednosti za čitavo čovječanstvo. Informacije za koje je nekada trebalo provesti mnogo vremena u knjižnici, uz telefon ili npr. danima čekati da stignu poštom, sada su od nas udaljene tek nekoliko "klikova" mišem. Međutim, podaci na webu uglavnom su namijenjeni ljudima, a čak i kada dolaze iz strukturiranih i dobro opisanih izvora, njihova struktura na webu najčešće nije razumljiva računalima koja ga obrađuju.

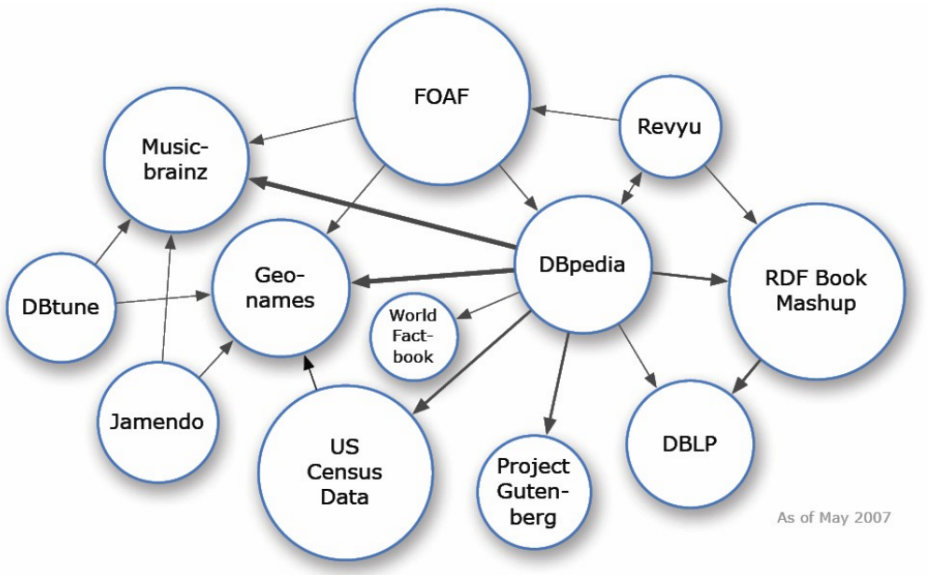
Računala vrlo uspješno mogu analizirati web stranice i raščlaniti ih na elemente kao što su zaglavlja, slike i poveznice, no u klasičnom Webu ne postoji pouzdan način za automatsku računalnu obradu semantike. Nadalje, korisnici podacima na webu uglavnom pristupaju pretraživanjem ili "surfanjem" tj. slijeđenjem poveznica. Takav pristup je spor, neučinkovit i zahtijeva puno truda i vremena. Tim Berners Lee, autor World Wide Weba, zato 2001. godine predlaže stvaranje globalne mreže podataka, koja bi se temeljila na vrlo uspješnoj ideji World Wide Weba, ali uz dodatak semantike, tj. davanja značenja podacima.

U svom radu koji predstavlja početak Semantičkog weba, Berners Lee iznosi ideju: Semantički web će donijeti strukturu sadržaju web stranica, stvarajući okruženje u kojem softverski agenti putujući s jedne stranice na drugu mogu obaviti sofisticirane zadaće za korisnike. Navodi također i osnovne gradivne elemente Semantičkog weba, kao što su eXtensible Markup Language (XML) za pohranu i Resource Description Framework (RDF) za opis značenja podataka. XML je općeprihvaćen strukturirani proširivi meta jezik za označavanje podataka i dokumenata, i de facto standard za prijenos i razmjenu podataka na internetu. RDF je jezik za prezentaciju informacija o resursima na webu. Temelji se na ideji imenovanja "pojmova" pomoću web identifikatora – Uniform Resource Identifier (URI) i njihovom opisu koristeći jednostavne izjave sastavljene od svojstava i vrijednosti. Svaka izjava u RDF-u je trojka (engl. triplet) resurs-svojstvo-vrijednost, koje možemo zamisliti kao subjekt, predikat i objekt u jednostavnoj rečenici. Vrijednost trojke može biti literal ili neki drugi resurs, što omogućava povezivanje resursa u graf, slično kao što su web stranice povezane u mrežu uz pomoć poveznica tj. hiperveza.

Razvijen je i upitni jezik SPARQL (rekurzivni akronim – SPARQL Protocol and RDF Query Language) koji omogućava dohvat i manipulaciju podacima pohranjenim u RDF grafu. Međutim, ono što je proslavilo web nisu bile toliko veze među web stranicama na istom poslužitelju, koliko veze među stranicama na različitim poslužiteljima. Slično je i u semantičkom webu, podaci na jednom poslužitelju, tj. izvoru podataka, uglavnom su međusobno dobro povezani, ali često nedostaju veze među podacima iz različitih izvora. Zbog toga Barners Lee 2006. godine pokreće inicijativu Linked Data i iznosi 4 "pravila" koja podaci na Semantičkom webu trebaju zadovoljavati kako bi se omogućio njegov rast i razvoj:

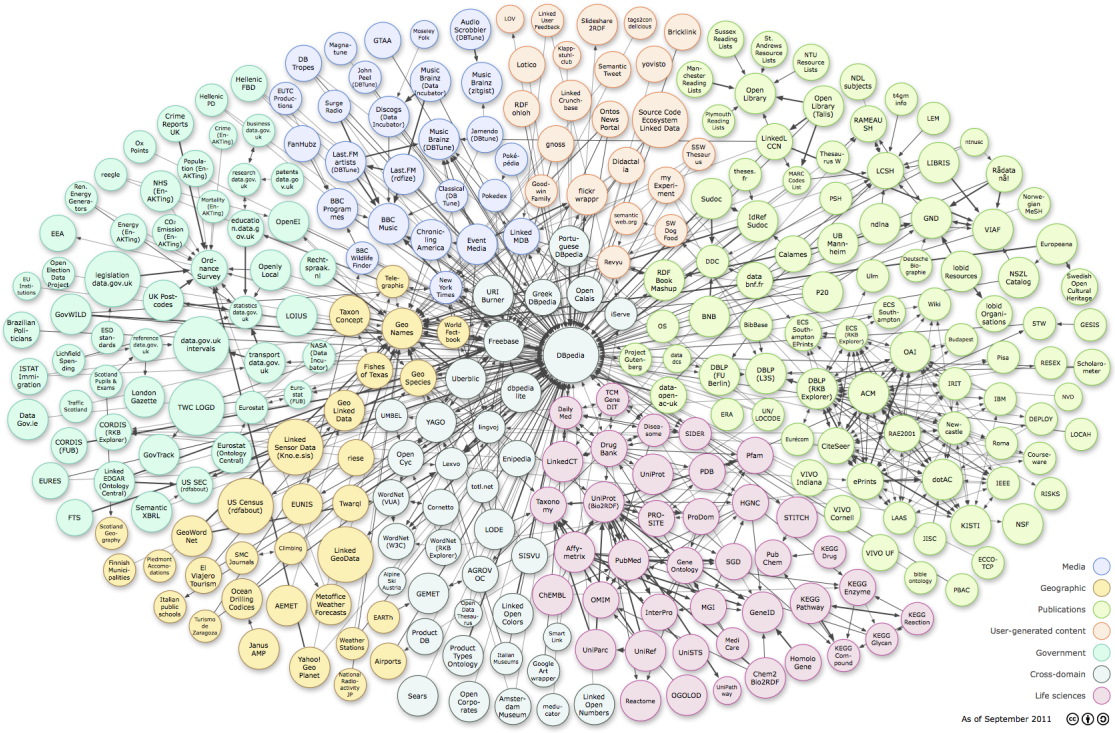
1. Korištenje URI-ja za imenovanje pojmova.
2. Korištenje HTTP URI-ja i DNS strukture, a ne alternativnih URI shema.
3. URI-ji se moraju moći razriješiti tj. pretvoriti u HTTP URL koji daje korisnu informaciju.
4. Uspostava veza prema drugim URI-jima kako bi se omogućilo otkrivanje drugih podataka.

Linked Data inicijativa polako stvara golemi globalni graf i transformira Semantički web iz niza nepovezanih "otoka" u kompaktni prostor podataka. Naziv Linked Data odnosi se na skup uputa i dobrih praksi za objavu i povezivanje strukturiranih podataka na webu. Globalni graf podataka povezanih u skladu s tim uputama naziva se Oblak povezanih podataka, LOD oblak (engl. Linked Data Cloud, Linking Open Data Cloud) ili Web podataka (engl. Web of Data – WOD). U početku se sastojao od 12 međusobno povezanih izvora, koji i danas čine njegovu jezgru:



Slika 4: Oblak povezanih podataka - svibanj 2007. godine

Ubrzo im se, objavljujući svoje podatke u skladu s pravilima inicijative, pridružio velik broj organizacija i pojedinaca. Prema trenutnim procjenama, oblak sadrži oko 330 izvora s preko 31 milijarde trojki i 500 milijuna veza. Slijedeća slika prikazuje stanje oblaka podataka iz rujna 2011. godine. Podaci pokrivaju najrazličitija područja – od knjiga i znanstvenih publikacija, preko glazbe i filmova, pa do podataka o lijekovima ili geografskim lokacijama:



Slika 5: Oblak povezanih podataka - rujan 2011. godine

3.1 Osnovni pojmovi povezanih podataka

3.1.1 Resurs, URI (Uniform Resource Identifier)

Uniform Resourse Identifier (URI) je jednoznačno određena adresa nekog izvora informacije (odnosno dokumenta) na web mreži, ili identifikator nekog izvora. Svrha korištenja URI-a je pristup (ili interakcija sa) izvorima informacija na webu korištenjem različitih protokola. Sintaksa (opći oblik pisanja) URI-a:

- URI shema (npr. http, ftp, mailto, URN, itd.)
- dvotočka
- dio koji označava određenu shemu koja se poziva

`http://dbpedia.org/resource/Mechatronics`

Valja napomenuti kako ovaj link ujedno predstavlja i URL (Uniform Resource Locator), jer klikom na njega dolazimo do stranice sa informacijama o određenom podatku, u ovom slučaju o mehatronici.

3.1.2 NameSpace (NS), Qname

U XML-u, imena elemenata definirana su od strane programera što često rezultira sukobom pri pokušaju miješanja XML dokumenata iz različitih XML aplikacija. Također se kod zapisivanja u tekstualne datoteke često pojavljuje nepreglednost zbog mnogo ponavljanja prvog dijela URI-a, odnosno domene. Kako bi izbjegli to ponavljanje i probleme sa sukobljavanjem te doprinijeli kompaktnosti i preglednosti zapisa u datotekama, uvodi se NameSpace. Uzmimo za primjer slijedeće URI-e:

```
http://dbpedia.org/resource/Mechatronics
http://dbpedia.org/resource/Robotics
http://dbpedia.org/resource/Mechanical_engineering
```

U navedenim zapisima možemo primjetiti zajednički dio http://dbpedia.org/resource/ koji predstavlja domenu, odnosno NameSpace, te različit dio koji zajedno s domenom čini URI. Kako bismo zamijenili NS sa prefiksom koristimo slijedeću sintaksu:

```
xmlns:db = http://dbpedia.org/resource/
```

Ovime smo definirali novo proizvoljno ime *db* za naš prefix i dodijelili mu NameSpace *http://dbpedia.org/resource/*. Sada možemo napisati gornje URI-je u skraćenom obliku, tj. pomoću prefiksa:

```
db:Mechatronics
db:Robotics
db:Mechanical_engineering
```

Gornji izrazi prestavljaju Qname koji predstavlja kompaktniju verziju URI-a.

3.2 XML (eXtensible Markup Language)

XML je zamišljen kao jezik za opisivanje dokumenata i podataka. Ideja je bila stvoriti jedan jezik koji će biti jednostavno čitljiv i ljudima i računalnim programima. Princip realizacije je vrlo jednostavan: odgovarajući sadržaj treba se uokviriti odgovarajućim oznakama koje ga opisuju i imaju poznato, ili lako shvatljivo značenje.

Pod izrazom "dokumenti i podaci" podrazumijevamo tekstualne dokumente ili pak skupove podataka kakvi se obično pohranjuju u baze podataka. Promatrajući XML kao tekstualni format, može se ustvrditi da je on potpuno neovisan o računalnoj platformi na kojoj se nalazi. Također je potpuno neovisan o operacijskom sustavu kojeg koristimo – XML je otvoreni standard čija je specifikacija javna i dostupna svima, tj. za njegovu standardizaciju brine se W3C (World Wide Web Consortium). Za korištenje XML-a nisu potrebne nikakve licence.

Tagove koje koristimo za opisivanje podataka nisu unaprijed definirani. XML standard jedino opisuje minimalni skup pravila koja dokument mora zadovoljavati. Korisnici XML-a moraju sami definirati dozvoljene oznake za označavanje. Danas je

XML jezik vrlo raširen i koristi se za različite namjene: odvajanje podataka od prezentacije, razmjenu podataka, pohranu podataka, povećavanje dostupnosti podataka i izradu novih specijaliziranih jezika za označavanje.

XML strukturiramo na način da radimo hijerarhiju tagova. Postoje *root*, *child* i *parent* tagovi. Slijedi primjer XML zapisa:

```
<?xml version=''1.0'' encoding=''UTF-8'' ?>
<student jmbag=''0152195996''>
  <ime>Marko</ime>
  <prezime>Marković</prezime>
  <institucija>
    <sveučilište>Sveučilište u Zagrebu</sveučilište>
      <fakultet id=''1234''>
        <ime>Fakultet strojarstva i brodogranje</ime>
        <adresa>
          <ulica>Ivana Lucića</ulica>
          <broj>5</broj>
          <pbr>HR-10000</pbr>
          <grad>Zagreb</grad>
        </adresa>
      </fakultet>
  </institucija>
</student>
```

U gornjem primjeru uočavamo da se XML dokument sastoji od dva dijela. Prvi dio je prolog ili zaglavlje.

```
<?xml version=''1.0'' encoding=''UTF-8'' ?>
```

U njemu se navode podatci koji opisuju XML dokument kao što su verzija XML preporuke prema čijim pravilima je dokument napravljen i kodna stranica. Ako se ne navede ispravna kodna stranica, programi koji barataju s XML dokumentom kada naiđu na nestandardni znak (npr. naše slovo "č") javit će grešku.

Drugi dio je sadržaj dokumenta u kojem se nalazi korisni sadržaj omeđen XML oznakama.

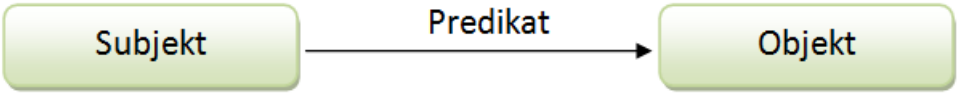
```
<student>
  <ime>Marko</ime>
    ...
</student>
```

Svaki XML dokument mora imati jedan korjenski ili root element koji uokviruje kompletan sadržaj. Taj element opisuje XML dokument i npr. kaže "ovaj XML dokument predstavlja podatke o studentu". Unutar korjenskog elementa ugniježđeni su svi ostali.

3.3 RDF (Resource Description Framework)

RDF je sintaksno neovisan, apstraktni model koji određuje standard o meta podacima (podaci o podacima) koji služe za opis resursa na webu. Kada u ovom obliku govorimo o meta podacima, njih isto tako uzimamo u najširem mogućem opsegu, ne ograničajući se ne prvotnu namjenu. RDF je standard kojeg je uspostavio i dalje razvija W3C te ga "reklamira" kao osnovu semantičkog weba, na temelju kojeg se izgrađuju svi ostali jezici semantičkog weba.

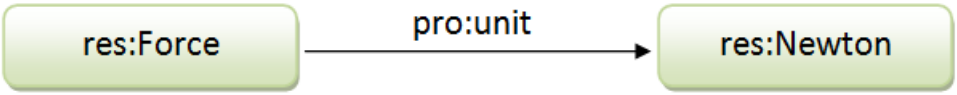
RDF se temelji na prijašnjim radovima koji su rezultirali modelima za prikaz imenovanih svojstava i njihovih vrijednosti. Temeljna konstrukcija povezanih podataka je RDF trojac oblika:



Slika 6: Općeniti oblik RDF trojca

Kao što vidimo, RDF trojac se sastoji od subjekta, predikata i objekta. Isto kao što čovjek definira vezu između subjekta i objekta u rečenici preko predikata, tako je i u RDF trojcu definirana veza između dva web resursa (ili resursa i doslovne vrijednosti) predstavljena pomoću URI-a.

Uzmimo za primjer jedan RDF trojac koji daje vezu između Sile i njezine mjerne jedinice:



Slika 7: Primjer RDF trojca

Iz gore navednog primjećuje se da se RDF data model sastoji od nekoliko članova:

1. Resurs (resource) predstavlja bilo što što se može dobaviti na webu. U biti sve što posjeduje URI je resurs. Što podrazumjeva da je to jedan HTML dokument, jedan XML dokument ili kolekcija bilo kojih od navedenih dokumenata, cijela web stranica ili jedna osoba... URI specifikacija pruža dovoljno proširivosti da možemo reći kako bilo što može posjedovati jedinstveni identifikator.

2. Atributi, još poznati pod drugim imenom kao svojstva (properties), predstavljaju specifičan aspekt nekog resursa. Svaki atribut posjeduje vlastito značenje koje dopušta određeni opseg vrijednosti ili može biti povezan samo s određenim resursima.

3. Vrijednosti koja može biti ili literal ili neki drugi resurs ili bilo koji drugi primitiv definiran XML-om
4. Sama tvrdnja (statement) je spomenuta trojka koja se sastoji od trojke: subjekta, predikata i objekta.

3.3.1 RDF/XML

RDF/XML je sintaksa, definirana od strane W3C-a, za izražavanje RDF grafa kao XML dokumenta. RDF graf može se smatrati skupom staza u obliku čvorova: predikatna grana, čvor, predikatna grana, čvor, predikatna grana, ... čvor, koji pokrivaju cijeli graf. U RDF/XML-u one su pretvorene u sekvence elemenata unutar elemenata koji se izmjenjuju između elemenata za čvorove i predikatnih grana. To se naziva niz čvor/grana poveznica. Čvor na početku sekvence prelazi u krajnji vanjski (*outermost*) element, sljedeća predikatna grana prelazi u *child* element, i tako dalje.

Slijedi primjer RDF/XML zapisa:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:pro="http://dbpedia.org/property/">

<rdf:Description rdf:about="http://dbpedia.org/resource/Force">
  <pro:unit>Newton</pro:unit>
  <pro:symbol>F</pro:symbol>
  ...
</rdf:Description>

<rdf:Description rdf:about="http://dbpedia.org/resource/Power">
  <pro:unit rdf:resource="http://dbpedia.org/resource/Watt"/>
  <pro:symbol>P</pro:symbol>
  ...
</rdf:Description>
.
.
</rdf:RDF>
```

Kao što je vidljivo, na početku je definiran glavni tag *rdf:RDF*, te također da se radi o XML datoteci. Unutar glavnog čvora nalaze se RDF trojci, dok se u njegovim atributima mogu definirati prefiksi NS-a.

RDF trojac je strukturiran tako da je najprije definiran čvor koji predstavlja subjekt, s tagom *rdf:Description* koji označava da opisujemo nešto, a kao rdf:about atribut tog taga, upisujemo subjekt pod vrijednost atributa unutar navodnika. Zatim unutar njega slijedi tag koji predstavlja predikat, npr. tag *pro:unit*, a on kao vrijednost atributa sadrži URI objekta *rdf:resource="http://dbpedia.org/resource/Watt* ili se između otvorenog i zavorenog dijela taga nalazi doslovna vrijednost objekta.

3.3.2 N-Triples

N-triples je format za pohranjivanje i prijenos podataka. To je običan tekstualni zapis u linijama koji predstavlja serijalizacijski format za RDF grafove, a podskup je od Turtle (Terse RDF Triple Language) formata. Svaki redak datoteke ima ili oblik komentara ili izjave. Izjava se sastoji od tri dijela odijeljenih razmakom (subjekta, predikata i objekta) i završava s točkom.

Subjekti mogu biti u obliku URI-a ili praznog čvora, predikati moraju biti URI, a objekti mogu biti URI-ji, prazni čvorovi ili doslovne vrijednosti. Ako se kao element upisuje URI tada se on upisuje između izlomljenih zagrada "<" i ">", dok doslovne vrijednosti pišemo unutar navodnika. Prazni čvorovi se označavaju alfanumeričkim stringom ispred kojeg se nalazi donja crtica i dvotočka "_:".

Primjer N-triples formata:

```
<http://dbpedia.org/resource/Force>
<http://dbpedia.org/property/unit> "Newton".
```

3.3.3 Turtle (Terse RDF Triple Language)

Turtle je format za izražavanje podataka u opisu RDF modela podataka sa sintaksom sličnom SPARQL. Zapravo, on nam pruža način za grupiranje tri URI-ja kako bi napravili trojku, te također pruža načine za skraćivanje takve informacije, npr. izlučivanje zajedničkih dijelova URI-ja pomoću prefiksa. Slijedi primjer definiranja prefiksa i najjednostavniji zapis jedne trojke u turtle formatu:

```
@prefix res:<http://dbpedia.org/resource/>
@prefix pro:<http://dbpedia.org/property/>
@prefix ont:<http://dbpedia.org/ontology/>

res:Force pro:unit res:Newton.
```

Često će se isti subjekt referencirati na veći broj predikata. Tada kreiranje *predicateObjectList-e* odgovara nizu predikata i objekata odijeljenih znakom točka-zarez ";" koji se nalaze na kraju svake trojke, dok se nakon svih ponovljenih nalazi točka. Taj zapis izražava niz RDF trojki s istim subjektom, te svaki predikat i objekt dodijeljen pojedinoj trojki. Sljedeća dva primjera predstavljaju ekvivalentan način zapisa trojki o sili:

```
res:Force pro:unit res:Newton;
          pro:isVector "True";
          pro:symbol "F".

res:Force pro:unit res:Newton.
res:Force pro:isVector "True".
res:Force pro:symbol "F".
```

Kao i sa predikatima, često se i objekti ponavljaju s istim subjektima i predikatima. Tada kreiranje *objectList-e* odgovara nizu objekta odvojenih znakom zareza "," koji se nalazi na kraju svake trojke, dok se na kraju svih ponovljenih isto nalazi točka. Slijedi primjer zapisa trojke sa zajedničkim subjektom i prekidatom:

```
res:Tensile_strength pro:unit res:Pascal_(unit),
                              "Mpa".
```

3.3.4 JSON-LD (JavaScript Object Notation for Linking Data)

JSON-LD je lagana sintaksa za serijalizaciju povezanih podataka u JSON-u. Njegov dizajn omogućuje postojećem JSON-u da se tumači kao povezani podatak s minimalnim izmjenama. Uz sve značajke koje pruža JSON, JSON-LD uvodi:

- univerzalni identifikatorski mehanizam za JSON objekte putem korištenja URI-ja.
- način da se ukloni neizvjesnost značenja ključeva koji se dijele između različitih JSON dokumenata njihovim mapiranjem u URI-je pomoću @context.
- mehanizam u kojem se vrijednost u JSON objektu može odnositi na JSON objekt na drugom mjestu na webu.
- mogućnost da bilježi stringove napisane u različitim jezicima.
- način za povezivanje vrste podataka s vrijednostima kao što su datum i vrijeme.
- Sposobnost izražavanja jednog ili više usmjerenih grafova, kao što su društvene mreže, u jednom dokumentu.

Općenito govoreći, model podataka korišten za JSON-LD je označen, usmjereni graf. Taj graf sadrži čvorove, koji su međusobno povezani. Čvor je obično podatak kao što je string, broj, datum ili vrijeme, ili URI. Postoji također i posebna klasa čvora koja se naziva prazan čvor, a obično se koristi za izražavanje podataka koji nema globalni identifikator kao što je URI. Prazni čvorovi su identificirani pomoću identifikatora praznog čvora. Ovaj jednostavan model podataka je nevjerojatno fleksibilan i moćan, te sposoban za modeliranje gotovo bilo koje vrste podataka.

Slijedi primjer jednostavnog JSON dokumenta:

```
{
  "name": "Albert Einstein",
  "homepage": "http://albert.einstein.org/",
  "image": "http://albert.einstein.org/images/albert.png"
}
```

Za ljude je očito da su to podaci o osobi čije ime je "Albert Einstein", te da *homepage* svojstvo sadrži URL početne stranice te osobe. Problem se javlja zato što stroj nema takvo intuitivno razumijevanje, a ponekad je čak i za ljude teško riješiti nerazumijevanje u takvim prikazima. Taj se problem može riješiti pomoću jednoznačnih identifikatora za označavanje različitih koncepata umjesto tokena, kao što su "ime", "početna stranica", itd. Ti jednoznačni indentifikatori jesu URI-ji, a slijedeći primjer prikazuje uzorak JSON-LD dokumenta koji u potpunosti koristi URI-je umjesto izraza:

```
{
  "http://schema.org/name": "Albert_Einstein",
  "http://schema.org/url": {"@id": "http://albert.einstein.org/"},
  "http://schema.org/image": {"http://albert.einstein.org/images
  /albert.png"}
}
```

U gornjem primjeru, svaki objekt jednoznačno je identificiran od strane URI-ja i sve vrijednosti koje predstavljaju URI-je jasno su označene kao takve od strane @id ključne riječi. Iako je ovo valjan JSON-LD dokument koji je vrlo specifičan o svojim podacima, ujedno je i suviše preopširan te bi programerima bilo teško raditi s njime. Kako bi riješili taj problem, JSON-LD uvodi pojam kontekst, što prikazuje slijedeći primjer.

Kontekst za uzorak dokumenta u prethodnom poglavlju:

```
{
  "@context":
  {
    "name": "http://schema.org/name", # Ovo znači da 'name'
             predstavlja kraticu za 'http://schema.org/name'
    "image": {
      "@id": "http://schema.org/image", # Ovo znači da 'image'
               predstavlja kraticu za 'http://schema.org/image'
      "@type": "@id" # To znači da se string vrijednost povezana
         sa 'image' treba tumačiti kao identifikator koji je URI
    },
    "homepage": {
      "@id": "http://schema.org/url",
      "@type": "@id"
    }
  }
}
```

Konteksti mogu ili biti izravno ugrađeni u dokument ili se možemo referencirati na njih. Pod pretpostavkom da kontekst dokument u prethodnom primjeru može biti preuzet sa *http://json-ld.org/contexts/person.jsonld*, može se pozivati dodavanjem jedne linije što će rezultirati da JSON-LD dokument bude izražen puno sažetije, kao što je prikazano u primjeru ispod:

```
{
  "@context": "http://json-ld.org/contexts/person.jsonld",
  "name": "Albert Einstein",
  "homepage": "http://albert.einstein.org/",
  "image": "http://albert.einstein.org/images/albert.png"
}
```

U JSON-LD dokumentima, konteksti mogu također biti navedeni *inline*. To ima za prednost da dokumenti mogu biti obrađeni i u nedostatku veza s internetom. U konačnici, to je odluka modeliranja te različiti slučajevi uporabe mogu zahtijevati različito rukovanje. Slijedi primjer *inline* definicije konteksta:

```
{
  "@context":
  {
    "name": "http://schema.org/name",
    "image": {
      "@id": "http://schema.org/image",
      "@type": "@id"
    },
    "homepage": {
      "@id": "http://schema.org/url",
      "@type": "@id"
    }
  },
  "name": "Albert Einstein",
  "homepage": "http://albert.einstein.org/",
  "image": "http://albert.einstein.org/images/albert.png"
}
```

Uz spomenute riječi **@context** i **@id** JSON-LD navodi niz simbola i ključnih riječi koji su temeljni dio jezika:

@value Koristi se za jedinstveno identificiranje stvari koje su u dokumentu opisane s URI-jima ili identifikatorima praznih čvorova.

@language Koristi se za određivanje jezika za određenu vrijednost stringa ili zadani jezik za JSON-LD dokument.

@type Koristi se za postavljanje vrste podataka čvora ili upisane vrijednosti.

@base Koristi se za postavljanje baznog URI-ja.

Te još **@container**, **@list**, **@set**, **@reverse**, **@graph**, **@index**, i ostali koji neće biti obrađeni u ovom poglavlju.

3.4 Ontologijski jezik (engl. *Ontology Language*)

U računalnoj znanosti i umjetnoj inteligenciji, ontologijski jezici su formalni jezici koji se koriste za izgradnju ontologija. Oni omogućuju kodiranje znanja o specifičnim domenama i često uključuju pravila rasuđivanja koja podržavaju obradu tog znanja. Jezici ontologije su obično deklaratitvni jezici, gotovo uvijek su generalizacije *frame* jezika, a obično se temelje na bilo logici prvog reda ili opisnoj logici.

Ontologija nam služi za točno definiranje veza među resursa, dok nam za povezivanje resursa služi RDF. Kako bi definirali semantičke odnose između resursa potrebna nam je RDF Schema.

3.4.1 RDFS (RDF Schema)

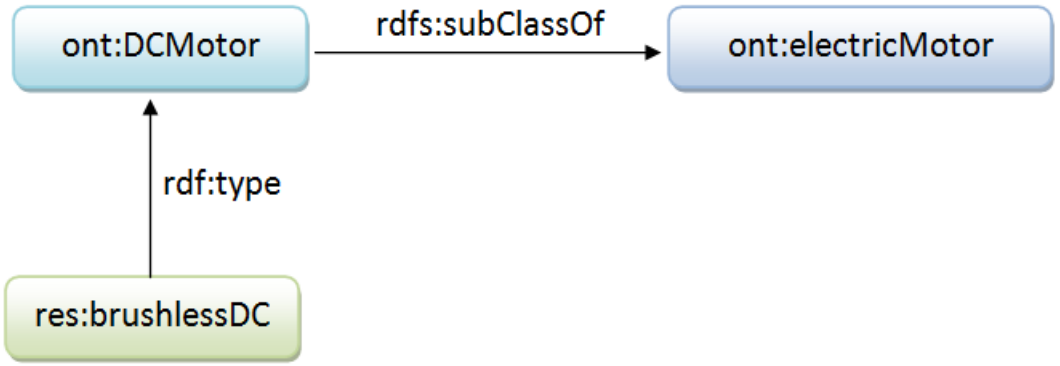
RDFS je semantičko proširenje RDF-a, tj. skup klasa s određenim značajkama RDF proširivog jezika koji pruža osnovne elemente za opis ontologija, namijenjenih za strukturiranje RDF resursa. Ovi resursi mogu se spremiti u trojke kako bi se do njih došlo pomoću SPARQL *query* jezika. Mnoge RDFS komponente uključene su u više izražajan Web Ontology Language (OWL).

RDFS konstrukti su RDFS klase, te pridružena svojstva izgrađena na ograničenom RDF rječniku.

Klase:

- **rdfs:Resource** je klasa svega. Sve stvari koje opisuje RDF su resursi.
- **rdfs:Class** - proglašava resurs kao klasu za druge resurse.
- **rdfs:Literal** - doslovne vrijednosti kao što su stringovi i brojevi, npr. vrijednosti nekog svojstva.
- **rdfs:Datatype** - klasa vrste podataka. rdfs:Datatype je ujedno instanca i podklasa od rdfs:Class. Svaka instanca od rdfs:Datatype je i podklasa od rdfs:Literal.
- **rdf:XMLLiteral** - klasa XML doslovne vrijednosti.
- **rdf:Property** - klasa svojstava.

Tipičan primjer *rdfs:Class* je *ont:DCMotor* u primjeru prikazanom na donjoj slici. Instanca od *ont:DCMotor* klase je resurs *res:brushlessDC* koji je povezan s njom pomoću *rdf:type* svojstva.



Slika 8: Hijerarhija RDF trojca

Ovime smo rekli da je brushlessDC ujedno i električni motor, iako nisu direktno povezani.

Svojstva:

Svojstva su instance klase *rdf:Property* i opisuju odnos između resursa subjekta i resursa objekta, što znači da su kao takva ujedno i predikati.

- **rdfs:domain** - je instanca od rdf:Property koja se koristi za tvrdnju da su vrijednosti svojstva instance od jedne ili više klasa.
- **rdfs:range** - je instanca od rdf:Property koja se koristi za navod da bilo koji resurs koji ima dano svojstvo jest instanca jedne ili više klasa.
- **rdf:type** - je svojstvo koje se koristi za tvrdnju da je određeni resurs instanca neke klase.
- **rdfs:isSubClassOf** - omogućuje da se proglase hijerarhije klasa.
- **rdf:subPropertyOf** - je instanca od rdf:Property koja se koristi za tvrdnju da su svi resursi povezani s jednim svojstvom također povezani i s nekim drugim.
- **rdf:label** - je instanca od rdf:Property koja se može koristiti za pružanje ljudski čitljive verzije imena resursa.
- **rdf:comment** - je instanca od rdf:Property koja se može koristiti za pružanje ljudski čitljivog opisa resursa.

3.4.2 OWL (Web Ontology Language)

OWL je ontološki jezik namijenjen opisu pojmova i odnosa između pojmova. Izrastao je na ideji semantičkog weba u kojemu informacija za web ima eksplicitno značenje, a ne samo oznake za prikaz. Implementacija ideje semantičkog weba ostvarena je kroz višerazinsku arhitekturu u kojoj su prve razine ostvarene s XML-om i RDFS-om. OWL predstavlja korak dalje, s ciljem da "formalno opiše značenje terminologije upotrebljene u web dokumentima". Teoretsku osnovu za OWL pružaju opisne logike - formalni jezici za predstavljanje znanja.

OWL je izgrađen na RDF-u i RDF Schemi te koristi RDF-ovu XML-baziranu sintaksu ali dodaje više vokabulara za opisivanje svojstava i klasa: između ostalog, odnose između klasa (npr. "disjointness"), kardinalitet (npr. "točno jedan"), jednakosti, bogatije tipizacija svojstava, karakteristike svojstava (npr. "simetrija"), te numerirane klase. Globalni *NameSpace* OWL rječnika je *http://www.w3.org/2002/07/owl#*, a uobičajeni prefiks koji se dodjeljuje je *owl*.

Postoje tri podjezika OWL-a, a to su:

- OWL Lite
- OWL DL
- OWL Full

Svaki od ovih podjezika je proširenje svog jednostavnijeg prethodnika, kako u onome što može legalno izraziti tako i u onome što može valjano zaključiti:

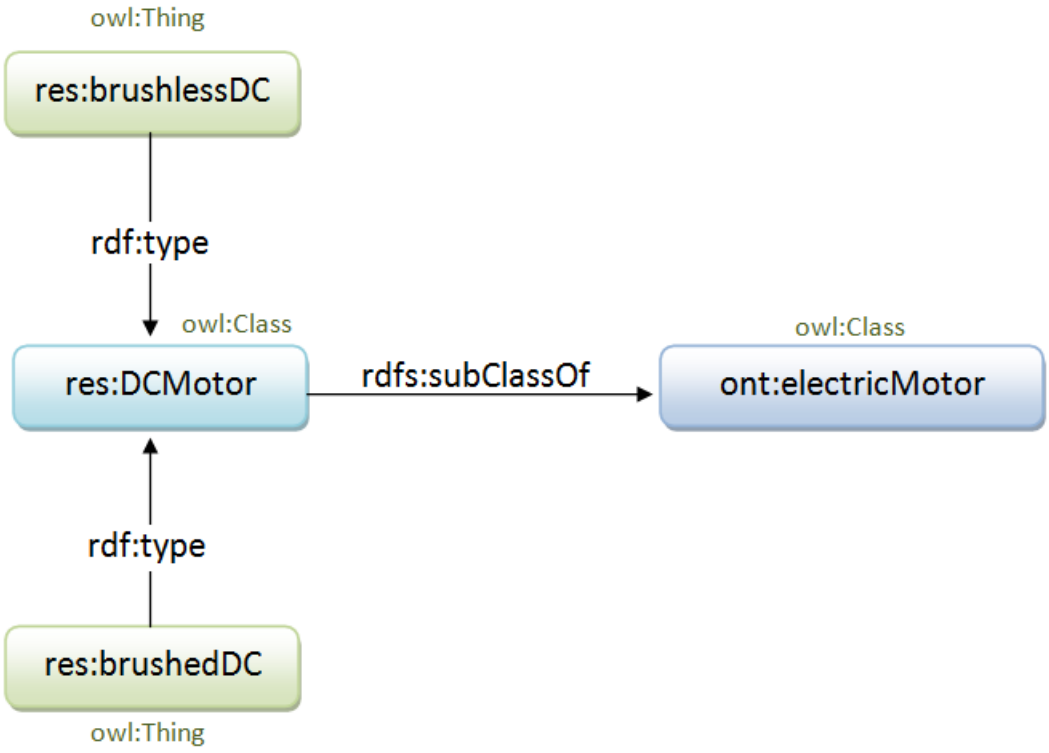
- Svaka pravna OWL Lite ontologija je pravna OWL DL ontologija.
- Svaka pravna OWL DL ontologija je pravna OWL Full ontologija.
- Svaki valjani OWL Lite zaključak je valjani OWL DL zaključak.
- Svaki valjani OWL DL zaključak je valjani OWL Full zaključak.

Kako definirati da je *DCMotor* klasa te ujedno i podklasa od *electricMotor* u OWL dokumentu prikazuje slijedeći primjer:

```
<rdf:RDF xmlns:owl ="http://www.w3.org/2002/07/owl#"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

  <owl:Class rdf:about="DCMotor">
    <rdfs:subClassOf rdf:resource="htpp://www.dbpedia.org/resource/
    electricMotor" />
    ...
  </owl:Class>

</rdf:RDF>
```

Osim definiranja klase, želimo biti u mogućnosti i opisati njene članove, što su pojedinci u našoj domeni stvari (things). Slijedeći primjer definira pojedince *brushlessMotor* i *brushedMotor* kao tipove električnog motora *DCMotor*:

```
<owl:Thing rdf:ID="http://www.dbpedia.org/resource/brushlessMotor"/>
<owl:Thing rdf:ID="http://www.dbpedia.org/resource/brushedMotor"/>

<owl:Thing rdf:about="http://www.dbpedia.org/resource/brushlessMotor"/>
   <rdf:type rdf:resource="http://www.dbpedia.org/resource/DCMotor"/>
</owl:Thing>

<owl:Thing rdf:about="http://www.dbpedia.org/resource/brushedMotor"/>
   <rdf:type rdf:resource="http://www.dbpedia.org/resource/DCMotor"/>
</owl:Thing>
```



Slika 9: grafički prikaz OWL zapisa

Najnoviji standard ontologijskog jezika je OWL2 koji je kompatibilan sa prošlom verzijom OWL-a i preporučen od strane W3C-a 2009. godine.

4 Regularni Izrazi

4.1 Uvod

U računarstvu i informatici, regularni izraz ili "pravilni/ispravni izraz" - često i engleske skraćenice "regexp" ili "regex") jest niz znakova (zvanih *metaznakovi*, nadznakovi) koji opisujuju druge nizove znakova u skladu s određenim sintaksnim pravilima. Prvenstvena svrha regularnog izraza je opisivanje uzorka za pretraživnje nizova znakova. Koristeći ih, među ostalim primjenama, moguće je učiniti sljedeće:

- Provjeriti poštuje li unešeni niz znakova zadani uzorak, npr. može se s njim provjeriti je li vrijednost unesena u HTML obrazac valjana e-mail adresa.
- Pretražiti pojavljivanje uzorka u dijelu teksta; npr. provjeriti pojavljuje li se riječ "tlak" ili riječ "pritisak" u dokumentu, i to samo s jednim, a ne višestrukim, ispitivanjem.
- Izdvojiti određene dijelove teksta; npr. izvući poštanski broj iz neke adrese.
- Zamijeniti dijelove teksta; npr. promijeniti sve pojave riječi "tlak" ili "pritisak" s riječju "pressure".
- Podijeliti veći tekst u manje dijelove, npr. razdijeliti tekst na bilo kojem mjestu gdje se pojavljuje točka, zarez ili znak za novi red.

4.2 Povijest, relevantnost i svrha

Regularni izrazi su sveprisutni. Mogu se naći u najnovijim software-skim rješenjima kao i onima iz 70-ih godina prošlog stoljeća. Niti jedan moderni programski jezik ne može se nazivati potpunim sve dok ne podržava regularne izraze. Regularni izrazi mogu biti vrlo teški za svladati i vrlo kompleksni za protumačiti, ako nisu napisani vrlo, vrlo pozorno. Kao rezultat ove složenosti, može se naći poznata izreka:

> "Neki ljudi, kada se suoče s problemom obrade znakova, misle: '*Znam kako, problem ću riješiti koristeći regularne izraze.*' No, od tog časa, više nemaju samo jedan problem, nego dva problema." *(Jamie Zawinski, 1997)*

Premda se regularni izrazi mogu naći u najnovijim i najboljim programskim jezicima današnjice i vjerojatno se budu nalazili još dugi niz godina, njihova povijest seže u 1943. godine kada su neurofizičari *Warren McCulloch* i *Walter Pitts* objavili *"A logical calculus of the ideas immanent in nervous activity"*. Ovaj rad ne samo da

je predstavio početak regularnih izraza, već je i predložio prvi matematički model neuronske mreže.

Sljedeći korak se dogodio 1956. godine, ovaj put od strane matematičara. Stephen Kleene je napisao rad *"Representation of events in nerve nets and finite automata"*, u kojem je po prvi puta koristio izraze "regular sets" i "regular expressions". Dvanaest godina kasnije, 1968. godine, legendarni začetnik računalnih znanosti uzeo je Kleene-ov rad i proširio ga, te objavio svoje studije u radu *"Regular Expression Search Algorithm"*. Taj inženjer bio je Ken Thompson, poznat po projektiranju i implementaciji Unix-a, programskog jezika B, UTF-8 kodiranja, itd. Naime, Ken Thompsonov rad nije završio samo s objavljivanjem rada, već je uključio i podršku za regularne izraze u svojoj verziji QED-a (*Quick Editor*). Za pretraživanje s regularnim izrazom u QED-u, bilo je potrebno napisati sljedeće:

```
g/<regular expression>/p
```

U prethodnoj liniji kôda, **g** označava globalno pretraživanje a **p** znači ispis.

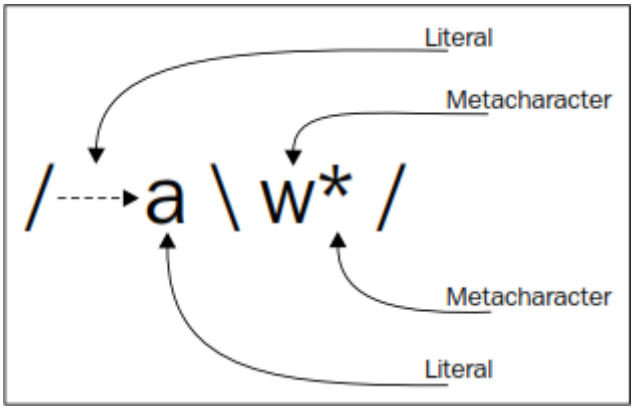
Sljedeći izvanredni događaj bilo je izdavanje prve *regex* knjižnice (Henry Spence), te potom, stvaranje skriptnog jezika *Perl* (Larry Wall) koji je proslavio regularne izraze od osobnih do moćnih strojeva (*'mainstream'*-a). Implementacija u Perlu uvela je mnoge izmjene u sintaksi izvornih regularnih izraza, stvarajući tzv. *Perl flavor*. Mnoge kasnije i današnje implementacije u ostalim jezicima ili alatima temelje se upravo na Perlovim regularnim izrazima.

Danas, standardni Python modul za regularne izraze **re** podržava samo regularne izraze sa Perl stilom. Postoji i nastojanje za pisanjem novog **regex** modula s boljom podrškom na https://pypi.python.org/pypi/regex. Ovaj novi modul bi s vremenom trebao zamijeniti Pythonovu implementaciju **re** modula.

4.3 Sintaksa

Svaki stariji programer (onaj koji je upoznao prve operacijske sustave, tzv. (DOS) se bez sumnje barem nekada koristio regularnim izrazom, iako možda nije znao da se o njemu radi. Često bi u konzoli operativnog sustava koristio zvjezdicu (*) ili upitnik (?) za pronalaženje nekih datoteka. Tako će, na primjer, uzorak koji sadrži upitnik kao što je "file?.xml" u naredbi "dir file?.xml" naći datoteke s nazivom file1.xml, file2.xml i file3.xml (ako dakako postoje u mapi/folderu u kojem naredba djeluje), ali neće ispisati datoteku s nazivom file99.xml, koja se tamo može nalaziti. Razlog je što metaznak '?' označuje niti jedan ili samo jedan znak, a ne dva ili više znakova, pa će se podudarati samo s onim imenima datoteka koje imaju niz znakova 'file' iza kojih slijedi samo jedan (bilo koji) znak ili da ga uopće nema i na koncu znakovi '.xml'.

U prethodnom izrazu, pojavljuju se dvije vrste komponente: **literal** ('file' i '.xml') i **metaznakovi** (? ili *). Sljedeća slika (10.1.) prikazuje primjer regularnog izraza u kojem se jasno vidi razlika između literala i metaznakova:



Slika 10: Regex - korištenje literala i metaznakova

Treba napomenuti kako su regularni izrazi koji se koriste u programskim jezicima mnogo moćniji od jednostavnih izraza koje obično pronalazimo u naredbenoj liniji temeljnog operativnog sustava, ali i jedni i drugi dijele istu definiciju: *"Regularni izraz je uzorak teksta koji se sastoji od običnih znakova (primjerice, slova **a** do **z** ili brojeva **0** do **9**) i posebnih znakova poznatih kao metaznakova te opisuje one nizove znakova koji će mu odgovarati kada se primijeni u nekom tekstu."*

4.4 Literali

Literali su najjednostavniji oblici uzorka u regularnim izrazima, oni će jednostavno uspjeti svaki put kad se utvrdi da je literal pronađen. Ako se primijeni regularni izraz " *programa* " kao uzorak za traženje, u sljedećem tekstu (stringu), naći će se jedno podudaranje:

"Programiranje je umjetnost i umijeće u stvaranju programa za računala."

Međutim, dobit će se više rezultata, ako se na isti tekst primijeni regularni izraz "*dioda*":

"Svjetleća dioda je poluvodička dioda koja emitira svjetlo čim se kroz nju pusti električna energija."

Metaznakovi se često pojavljuju zajedno s literalima u istom regularnom izrazu. To može unijeti pomutnju, ako se ne uoče i razumiju pravila i razlikovanje običnih znakova od metaznakova. Na primjer, primijeni li se izraz "(simbol N)" za pretraživanje sljedećeg teksta, može se vidjeti kako zagrade nisu uključene u rezultat:

"SI jedinica za silu je newton (simbol N)."

To se događa zato što su zagrade *metaznakovi*, a oni imaju posebno značenje, posebnu interpretaciju u regularnim izrazima. Međutim, metaznakovi se mogu koristiti

kao literali, kao obični znakovi, ali onda je u Python programskom kôdu potrebno:

- ispred metaznaka umetnuti lijevu kosu crtu (engl. *backslash*).
- ili koristiti metodu *re.escape* za izbjegavanje ne-alfanumeričkih znakova koji se mogu pojaviti u izrazu.

U regularnim izrazima postoji ukupno dvanaest *metaznakova* koje treba izbjegavati, ako se planiraju koristiti sa svojim doslovnim značenjenjem. To su:

- lijeva kosa crta "\"
- krović "^"
- točka "."
- znak dolara "$"
- povezni 'ILI' znak "|"
- upitnik "?"
- zvjezdica "*"
- znak plusa "+"
- otvorene zagrade "("
- zatvorene zagrade ")"
- otvorene uglaste zagrade "["
- zatvorene uglaste zagrade "]"
- otvorene vitičaste zagrade "{"
- zatvorene vitičaste zagrade "}"

U nekim slučajevima, sustavi regularnih izraza učiniti će sve kako bi razumjeli da li ti metaznakovi trebaju imati doslovno značenje, čak i ako nisu izbjegnuti; primjerice otvorena vitičasta zagrada "{" biti će tretirana kao metaznak, samo ako iza nje slijedi broj koji ukazuje na ponavljanje.

4.5 Klase znakova

Razredi ili klase znakova, (također poznate kao "skupovi znakova") omogućuju raspolaganje s jednim od definiranih znakova unutar skupa. Za definiranje razreda znakova, prvo trebamo napisati metaznak "[", a zatim sve znakove iz skupa koji se mogu pojaviti, te na koncu zatvoriti skup s metaznakom "]". Npr. definirani regularni izraz "filt[ae]r" će se podudarati s riječi "*filtar*", ali i "*filter*". Kao rezultat dobit će se sljedeće:

Električki filtar (ili filter) je elektronički sklop čija je funkcija da na određeni način promijeni karakteristiku frekvencijskog spektra ulaznog signala."

Može se također koristiti i raspon znakova. To se postiže uporabom simbola crtice (-) između dvaju (srodnih) znakova; npr. za izraz koji će se podudarati s bilo kojim malim slovom može se koristiti skup znakova [a-z]. Isto tako za podudaranje s bilo kojim jednoznamenkastim brojem može se definirati skup znakova [0-9]. Rasponi skupova znakova mogu se kombinirati tako da znak koji se obrađuje ima mogućnost zadovoljavati samo jedan od skupa raspona, a da pri tome nije potrebno nikakvo posebno odvajanje. Npr. ako se želi podudaranje s bilo kojim malim ili velikim alfanumeričkim znakom, može se koristiti izraz "[0-9a-zA-Z]", ili alternativni zapis "[0-9[a-z[A-Z]]]".

Postoji i još jedna mogućnost - *negacija raspona*. Možemo izokrenuti značenje skupa znakova postavljanjem metaznaka "^" odmah nakon metaznaka "[". Ako imamo skup znakova kao što je [0-9] koji označava bilo koji jednoznamenkasti broj, takav negirani skup [^0-9] podudarat će se sa svime što nije znamenka.

4.6 Predefinirane klase znakova

Upotreba znakovnih skupova vrlo brzo je pokazala kako su neki od njih vrlo korisni i umjesto da se stalno prepisuju, dodijeljena im je kratica, prečac. Trenutačno postoji velik broj predefiniranih razreda znakova što omogućuje izrazima koji ih koriste da budu puno čitljiviji i uredniji. Ovi znakovi nisu korisni samo kao dobro poznati prečaci za tipične skupove znakova, već također imaju i različita značenja u različitim kontekstima. Znakovna klasa "\w", koja odgovara bilo kojem alfanumeričkom znaku, podudarat će se s drugačijim skupom znakova ovisno o konfiguriranom lokalnom jeziku (eng. "*configured locale*") i "Unicode" podršci. Sljedeća tablica prikazuje predefinirane klase znakove podržane u Python-u:

Element	Opis (za regex s defaultnim flag-om)
.	Ovaj element odgovara bilo kojem znaku osim newline \n
\d	Ovaj element odgovara bilo kojoj decimalnoj znamenki; to je ekvivalent za klasu [0-9]
\D	Ovaj element odgovara bilo kojem znaku koji nije decimalna znamenka; to je ekvivalent za klasu [^0-9]
\s	Ovaj element odgovara bilo kojem znaku razmaka; to je ekvivalent za klasu [\t\n\r\f\v]
\S	Ovaj element odgovara bilo kojem znaku koji nije znak razmaka; to je ekvivalent za klasu [^\t\n\r\f\v]
\w	Ovaj element odgovara bilo kojem alfanumeričkom znaku; to je ekvivalent za klasu [a-zA-Z0-9_]
\W	Ovaj element odgovara bilo kojem znaku koji nije alfanumerički znak; to je ekvivalent za klasu [^a-zA-Z0-9_]

Tablica 1: Predefinirane klase znakova

4.7 Izbor ili alternacija (engl. "Alternation")

Izbor ili alternacija između dva ili više elemenata iz skupa regularnih izraza, postiže se s pomoću simbola "|". Npr. definiranje regularnog izraza kojim se želi podudaranje s riječi "*nizak*" ili riječi "*srednji*", željeni rezultat postiže se izrazom "nizak | srednji". Na isti se način, izraz može proširiti novim vrijednostima iz skupa koje proširuju regularni izraz, npr. "nizak | srednji | visok".

Prilikom korištenja alternacije u složenijim regularnim izrazima, moguće je alternaciju smjestiti unutar zagrada '( )' kako bi se izbor proveo samo unutar njih, a ne na cijelom izrazu. Npr. ako se napiše pogrešni izraz "*Pritisak: nizak | visok*" očekivat će se podudaranje s "*Pritisak: nizak*" ili "*Pritisak: visok*". Međutim, podudaranje je u ovom slučaju: "*Pritisak: nizak*" ili "*visok*", jer je alternacija primjenjena na cijeli regularni izraz, umjesto samo na *"nizak | visok*" dio. Ispravan regularni izraz treba, za takvu namjenu, biti: "*Pritisak: (nizak | visok)*".

4.8 Kvantifikatori (engl. "Quantifiers")

Kvantifikatori su brojevni mehanizmi s kojima se metaznakovi ili skupovi znakova mogu ponavljati. Osim na znakove i skupove znakova, kvantifikatori se mogu primjeniti i na grupe. Simbole kvantifikatora, te njihov opis, sadrži sljedeća tablica:

Simbol	Ime	Kvantifikacija prethodnog znaka
?	Question mark	Uvjetno (0 ili 1 ponavljanje)
*	Asterisk	Nula ili više puta
+	Plus	Jednom ili više puta
{m,n}	Curly braces	Između n i m puta
{n}	Curly braces	Točno n puta
{,n}	Curly braces	Najviše n puta
{n,}	Curly braces	Minimalno n puta

Tablica 2: Kvantifikatori

Neka se kao primjer uzme zadatak u kojemu je potrebno definirati regularni izraz koji će se podudarati s telefonskim brojem koji može bit zapisan u bilo kojem od sljedeća tri formata: *123-456-789*, *123 654 879*, ili *987654321*. Može se npr. konstruirati ovakav regularni izraz: "\d+[-\s]?\d+[-\s]?\d+". On će se podudarati s telefonskim brojem samo ako se na početku stringa nalazi jedna ili više znamenki, iza kojih može ali i ne mora slijediti crtica "-" ili prazan prostor, pa onda opet isto, te na koncu mora biti jedna ili više znamenki. Ovaj regularni izraz podudarao bi se sa svim gore navedenim formatima telefonskog broja, ali samo ako se ispred njega ne bi nalazili neki drugi brojevi. Oni bi pokvarili naše traženje. Zato je potrebno finije podesiti regularni izraz, tako da krajnje lijevi skup znamenki može sadržavati maksimalno do tri znaka, dok ostatak brojevnih skupina treba sadržavati točno tri znamenke. Tako uređeni regularni izraz izgledao bi ovako "\d{1,3}[-\s]?\d{3}[-\s]?\d{3}". On će davati ispravno rješenje u bilo kojem slučaju.

4.9 Pohlepni i reluktantni kvantifikatori

Još uvijek nije definirano što će se podudarati ako se primijeni regularni izraz s kvantifikatorom kao što je ovaj "'.+'" (uočite jednostruke navodnike kao dio izraza) na sljedeći tekst: "*hrvatski 'čelik', engleski 'steel'*". Moglo bi se očekivati da će kao rezultat biti pronađene riječi '*čelik*' i '*steel*', ali ono što će se zapravo dobiti bit će dio rečenice koji glasi "*'čelik', engleski 'steel*". Takvo ponašanje naziva se pohlepno i jedno je od dvaju mogućih ponašanja kvantifikatora u Pythonu: pohlepno i nepohlepno (također poznato kao reluktantno).

- Pohlepno ponašanje se pretpostavlja u svim kvantifikatorima. Bitno je istaknuti kako će se pohlepan kvantifikator pokušati uskladiti s koliko god je više moguće znakova, kako bi imao najveći mogući rezultat podudaranja.
- Ne-pohlepno ponašanje može se zatražiti dodavanjem dodatnog upitnika nakon kvantifikatora, npr. "??", "*?" ili "+?".

Pravilan regularni izraz za gornji primjer glasio bi: "'.+?'", koji bi dao rezultat: '*čelik*', '*steel*'.

4.10 Granična podudarala (engl. "Boundary matchers")

Granično podudaralo je identifikator koji odgovara određenoj poziciji unutar promatranog ili unešenog niza znakova. Sljedeća tablica prikazuje granična podudarala, koja su dostupna u Python programu:

Podudarivač	Opis
^	Odgovara početku retka
$	Odgovara kraju retka
\b	Odgovara granici riječi
\B	Odgovara suprotnom od \b. Sve što nije granica riječi
\A	Odgovara početku ulaza
\Z	Odgovara kraju ulaza

Tablica 3: Granična podudarala

Treba napomenuti kako će se granična podudarala drugačije ponašati u različitim kontekstima. Npr. graničnik riječi (\b) će izravno ovisiti o konfiguriranom lokalnom jeziku, budući da različiti jezici mogu imati različite graničnike riječi. Na isti način ponašat će se granice linije/retka (početak i kraj), koje će ovisiti o zadanim oznakama, zastavicama (eng. *flag*).

Česta je potreba da se u radu sa stringovima želi biti siguran da iza nekog imena postoje samo abecedni znakovi ili praznine do kraja linije. To se postiže usklađivanjem cijelog retka, od početka do kraja, s postavljenim skupom znakova čije je ponavljanje omogućeno neograničeni broj puta do kraja retka, kako prikazuje sljedeći regularni izraz: "^Ime:[\sa-zA-Z]+$".

Još jedno posebno granično podudaralo je graničnik riječi \b. Vrlo je koristan kad se želi raditi s izoliranim, samostalnim riječima. Pritom se ne želi stvarati skupove znakova sa svakim pojedinim znakom koji može biti razdijelnik riječi (npr. space-ovi, zarezi, dvotočke, crtice, itd.). Može se, primjerice, bili siguran da se riječ "materijal" pojavljuje u tekstu pomoću sljedećeg regularnog izraza: "\bmaterijal\b". Za vježbu se može postaviti pitanje zašto je prethodni izraz bolji od, na primjer, izraza "materijal". Razlog je u tome što će onaj izraz s razdjelnikom odgovarati samo izoliranoj riječi, a ne bilo kojoj riječi koja sadrži "materijal", kao što su 'materijalizam', 'biomaterijal', i slično, dok će izraz "\bmaterijal\b" odgovarati samo riječi 'materijal'.

5 Regularni izrazi u Pythonu

Regularni izrazi u Pythonu podržani su unutar **re** modula. Kao i sa svim modulima u Pythonu, prije poziva neke od njegovih funkcija, modul treba učitati, uvesti (engl. "import"):

```
1 >>> import re
```

Nakon što je modul učitan, mogu se graditi uzorci regularnih izraza. Najbrže izvođenje postiže se kompilacijom, pretvaranjem u binarni zapis (*engl. "bytecode"*), koji se kasnije izvršava s brzim fukcijama pisanim u C-jeziku:

```
1 >>> pattern = re.compile(r'\bistosmjerni\b')
```

Nakon stvaranja uzorka, moguće ga je primjeniti na nekom nizu znakova, npr. provjeriti hoće li se naći podudarnost:

```
1 >>> pattern.match("istosmjerni motor")
2 <_sre.SRE_Match at 0x108acac60>
```

Za zadani uzorak regularnog izraza, potraga u tekstu "istosmjerni motor" dala je pozitivno rješenje kao što prikazuje redak <_sre.SRE_Match at 0x108acac60>. To znači da je podudarni objekt pronađen i spremljen na memorijsku adresu (0x108acac60). U Pythonu postoje dva različita objekta unutar *regex* modula:

- *RegexObject*: poznat kao *PatternObject* – predstavlja kompajlirani regularni izraz.
- *MatchObject*: predstavlja pronađeni, podudarni uzorak u traženom nizu znakova.

5.1 Pretraživanje

Python sadrži više metoda za pretraživanje, a to su: *match()*, *search()*, *findall()* i , *finditer()*.

5.1.1 match (string[, pos[, endpos]])

Ova metoda pokušava uskladiti kompilirani uzorak samo na početku string-a, te ako postoji podudaranje, vraća *MatchObject*. Tako se, na primjer, može provjeriti da li neki string počinje s <HTML> ili ne:

```
>>> pattern = re.compile(r'<HTML>')
>>> pattern.match("<HTML><head>")
<_sre.SRE_Match at 0x108076578>
```

U prethodnom primjeru, prvo se složio i kompilirao uzorak, a zatim je došlo do podudaranja u "<HTML><head>" string-u. Međutim, kad string ne bi počinjao s traženim <HTML> nizom, pretraga ne bi vratila objekt, nego prazni tip podataka (*None*):

```
>>> pattern.match(" <HTML>")
    None
```

Kao što se može vidjeti, nema podudaranja uzorka i teksta jer tekst/string započinje s razmakom (za razliku od uzorka), a pretraga počinje od početka.

Međutim, uporabom **pos** paramentra može se odrediti mjesto odakle će započeti traženje, kao prikazuje sljedeći kôd:

```
>>> pattern = re.compile(r'<HTML>')
>>> pattern.match("  <HTML>")
    None
>>> pattern.match("  <HTML>", 2)
   <_sre.SRE_Match at 0x1043bc850>
```

Treba imati na umu da *pos* veći od 0 ne znači da traženi uzorak počinje na tom indeksu, npr:

```
>>> pattern = re.compile(r'^<HTML>')
>>> pattern.match("<HTML>")
   <_sre.SRE_Match at 0x1043bc8b8>
>>> pattern.match("  <HTML>",  2)
    None
```

U prethodnom kôdu, stvoren je uzorak koji bi se podudarao samo sa stringovima u kojima je prvi string nakon početka reda <HTML> (primjetite znak '^'). Nakon toga, u provjeri podudara li se string <HTML> s početkom na drugom znaku (<) stringa, dobiveno je da nema podudaranja, jer uzorak prvo pokušava uskladiti metaznak (^) na poziciji s indeksom 2 i u tomu ne uspjeva (početak linije stringa nije podudarno s '<'). Podudaranje bi uspjelo tek primjenom kriške ("*slice-a*") uzorka s kojom bi se odrezao početak, pa bi uzorak počinjao od pozicije 2, kako je prikazano u sljedećem kôdu:

```
>>> pattern.match("  <HTML>"[2:])
   <_sre.SRE_Match at 0x1043bca58>
```

"Kriška" stringa stvara, odnosno daje, novi string koji uključuje i novi početak stringa, kojim nastaje podudaranje s metaznakom (^) i njemu. Argiment **pos** pak samo pomiče indeks do polazne točke za pretraživanje u stringu.

Postoji također i treći argument u *match()* metodi, **endpos**, koji postavlja granicu do koje će se uzorak pokušati usklađivati sa stringom, te je ekvivalent za *slicing*:

```
>>> pattern = re.compile(r'<HTML>')
>>> pattern.match("<HTML>"[:2])
    None
>>> pattern.match("<HTML>", 0, 2)
    None
```

Dakle, u sljedećem slučaju, problema spomenutog kod *pos* nema, već postoji podudaranje čak i kad se koristi ($) metaznak:

```
>>> pattern = re.compile(r'<HTML>$')
>>> pattern.match("<HTML> ", 0, 6)
<_sre.SRE_Match object at 0x1007033d8>
>>> pattern.match("<HTML> "[:6])
<_sre.SRE_Match object at 0x100703370>
```

5.1.2 search(string[, pos[, endpos]])

Ova metoda pokušava uskladiti uzorak na bilo kojoj lokaciji u stringu, a ne samo na početku, pa ako postoji podudaranje, vraća *MatchObject*.

```
>>> pattern = re.compile(r"motor")
>>> pattern.search("asinkroni motor")
   <_sre.SRE_Match at 0x1080901d0>
```

Treba naglasiti kako parametri **pos** i **endpos** imaju isto značenje kao što imaju i u *match* operaciji. Međutim, '^' simbol, uz MULTILINE argument, podudara se s početkom stringa i s početkom svake linije nekog teksta koji se pretražuje, što bitno mijenja ponašanje ove pretraživačke operacije. U sljedećem primjeru, prvo pretraživanje podudara se s <HTML> jer se nalazi na početku stringa, dok drugo ne odgovara, jer string počnje s razmakom. I konačno, u trećem pretraživanju, imamo podudaranje jer je <HTML> pronađen odmah nakon nove linije, zahvaljujući zastavici (*engl. flag*) **re.MULTILINE**.

```
>>> pattern = re.compile(r'^<HTML>', re.MULTILINE)
>>> pattern.search("<HTML>")
   <_sre.SRE_Match at 0x1043d3100>
>>> pattern.search(" <HTML>")
    None
```

```
>>> pattern.search("   \n<HTML>")
   <_sre.SRE_Match at 0x1043bce68>
```

5.1.3 findall(string[, pos[, endpos]])

Prethodne operacije tražile su samo jedno podudaranje. Funkcija *findall()* vraća listu sa svim preklapajućim pojavama uzorka, a ne **MatchObject** kao u *match()* i *search()* operacijama. U sljedećem primjeru, traži se svaka riječ u stringu, pa se stoga dobiva lista u kojoj se svaki njen element (u ovom slučaju riječ) podudara s uzorkom.

```
>>> pattern = re.compile(r"\w+")
>>> pattern.findall("asinkroni motor")
    ['asinkroni', 'motor']
```

Nadalje, treba imati na umu da su i prazna podudaranja isto dio rezultata, što prikazuju sljedeći primjeri:

```
>>> pattern = re.compile(r'(M\d\d)*')
>>> pattern.findall("M20,M24")
    ['M20', '', 'M24', '']
```

Postavlja se pitanje zašto se to dogodilo? Odgovor se krije u '*' kvantifikatoru, koji omogućuje 0 ili više ponavljanja prethodnog regularnog izraza, a isto to bi se dogodilo i sa '?' kvantifikatorom.

```
>>> pattern = re.compile(r'(M\d\d)?')
>>> pattern.findall("M20,M24")
    ['M20', '', 'M24', '']
```

Prvo se *regex* podudarao sa izrazom "M20", te je zatim slijedio znak "," na kojemu je došlo do podudaranja zbog '*' kvantifikatora, pa je stoga bio vraćen prazan string. Nakon toga, došlo je do podudaranja s drugim izrazom "M24" i na kraju je slijedio pokušaj podudaranja sa znakom '$'. Treba napomenuti da je znak '$', iako je nevidljiv, važeći znak za *regex* obradbe.

U slučaju postojanja grupa u uzorku, one će biti vraćene kao *n-torke* (*engl. "tuples"*). String se također obrađuje s lijeva na desno, tako da se grupe vraćaju u istom redoslijedu kao i u uzorku. Sljedeći primjer pokušava uskladiti uzorak napravljen od dvije rijči sa stringom, te stvara grupu za svaku pronađenu riječ, što rezultira listom n-torki u kojoj svaka n-torka sadrži po dvije grupe.

```
>>> pattern = re.compile(r"(\w+) (\w+)")
>>> pattern.findall("duktilan materijal krhak materijal")
    [('duktilan', 'materijal'), ('krhak', 'materijal')]
```

Uzorak bi radio s bilo kojim parnim nizom riječi, a za neparni bi zadnju riječ izostavio u listi (ne bi postajala podudarnos s njom, jer su u uzorku dvije grupe regularnih izraza).

5.1.4 finditer(string[, pos[, endpos]])

Ova funkcija je u suštini ista kao i *findall()*, ali vraća iterator u kojem je svaki element **MatchObject** kako bismo mogli koristiti operacije koje on pruža. Stoga, vrlo je koristan kad se želi informacija za svako pojedinačno podudaranje uzoraka, primjerice pozicija u kojoj se podniz podudario s uzorkom. Može se uzeti već korišteni primjer u kojem se uzorak podudara s parom riječi:

```
>>> pattern = re.compile(r"(\w+) (\w+)")
>>> it = pattern.finditer("duktilan materijal krhak
    materijal")
>>> match = it.next()
>>> match.groups()
    ('duktilan', 'materijal')
>>> match.span()
    (0, 18)
```

Iz prethodnog primjera vidimo da iterator pamti sva podudaranja, ali objavljuje ih jednog po jednog uz pomoć *next()* metode. Za svaki element u iteratoru spremljen je *MatchObject*, pa se mogu dobiti grupe koje se podudaraju s uzorkom, u ovom slučaju dvije. Također se preko funkcije *span()* mogu dobiti pozicije podudaranja uzorka sa stringom.

```
>>> match = it.next()
>>> match.groups()
    ('krhak', 'materijal')
>>> match.span()
    (19, 34)
```

Uzastopnom primjenom *next()* funkcije, dohvaćaju se idući objekti u iteratoru, što pokazuje gornji primjer. To je moguće, sve dok iterator posjeduje sljedeći objekt:

```
>>> match = it.next()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
StopIteration
```

Konačno, ako su objekti u iteratoru iscrpljeni, podiže se iznimka *StopIteration*, koja ukazuje da iterator više nema spremljenog niti jednog elemenata.

5.2 Promjena stringa

Operacije za modifikaciju stringova služe ili za podjelu stringova na dijelove ili za zamjenu nekih njihovih djelova s pomoću željenog uzorka.

5.2.1 split(string, maxsplit=0)

U gotovo svim programskim jezicima, može se pronaći *split* operacija nad stringovima, ali velika razlika je u tomu što je *split()* funkcija u **re** modulu znatno moćnija, jer koristi *regex* izraze. U ovom slučaju, string se dijeli na temelju podudaranja s uzorkom što prikazuje sljedeći primjer.

```
>>>re.split(r"\n", "Mjerna jedinica: newton.\nSimbol: N.")
['Mjerna jedinica: newton.', 'Simbol: N.']
```

U gornjem primjeru došlo je do podudaranja u izrazu "\n" pa je string podijeljen koristivši njega kao separatora.

Neka se pogleda složeniji primjer u kojem se izvlače riječi iz stringa:

```
>>> pattern = re.compile(r"\W")
>>> pattern.split("elektromotorni pogon")
['elektromotorni', 'pogon']
```

U prethodnom primjeru definiran je uzorak koji odgovara bilo kojem ne-alfanumeričkom znaku, pa se stoga u ovom slučaju podudaranje događa upravo u tom razmaku, te je to je razlog zašto je string podijeljen u riječi.

Parametar **maxsplit** određuje koliko se maksimalno podjela može obaviti te vraća preostali dio u rezultatu:

```
>>> pattern = re.compile(r"\W")
>>> pattern.split("Strojarstvo je primijenjena znanost", 2)
['Strojarstvo', 'je', 'primjenjena znanost']
```

Treba napomenuti kako uzorak podudaranja nije uključen u rezultat, pa ako ga se želi uključiti, potrebno je koristiti grupe:

```
>>> pattern = re.compile(r"(-)")
>>> pattern.split("elektro-energetika")
['elektro', '-', 'energetika']
```

Ako se grupa podudara s početkom stringa, onda rezultat sadrži prazan string kao prvi rezultat:

```
>>> pattern = re.compile(r"(\W)")
>>> pattern.split(" elektromotorni pogon")
['', ' ', 'elektromotorni', ' ', 'pogon']
```

5.2.2 sub(repl, string, count=0)

Funkcija *sub()* vraća rezultirajući string nakon izmjene podudarajućeg uzorka u izvornom stringu sa željenom zamjenom, a ako nije pronađen, onda vraća izvorni string. Neka se, kao primjer, uzme zamjena "<->" u stringu sa "M20":

```
>>> pattern = re.compile(r"(<->)+")
>>> pattern.sub("M20", "Za sklapanje proizvoda potreban je
    vijak <-> i matica <->.")
  'Za sklapanje proizvoda potreban je vijak M20 i matica M20
     .'
```

Treba uočiti da se *regex* podudara sa svim pojavama izraza "<->" te ih zamjenjuje, u ovom slučaju sa izrazom "M20". Zamjena se događa s lijeva nadesno, ako su pojave ne-preklapajuće, što zorno pokazuje sljedeći primjer:

```
>>> re.sub('00', '-', 'vijak00001')
    'vijak--1'
```

Argument **repl** također može biti i funkcija, te u tom slučaju ona prima **MatchObject** kao argument, a string kojeg vraća je željena zamjena. Neka se pretpostavi sustav u kojem postoje dvije vrste naredbi, od kojih neke započinju s crticom, a neke sa slovom:

- -1234
- A193, B123, C124

Zadatak je promijeniti ih u slijedeće:

- A1234
- B193, B123, B124

Ukratko, one koje počinju s crticom bi trebale počeinjati s A, a ostale (koje poćinju s nekim slovom) bi trebale počinjati s B.

```
>>>def normaliziraj(matchobj):
        if matchobj.group(1) == '-': return "A"
        else: return "B"
>>> re.sub('([-|A-Z])', normaliziraj, '-1234 A193 B123 B124')
'A1234 B193 B123 B124'
```

Funkcija *normaliziraj()* poziva se za svaki podudarajući uzorak, pa ako je prva podudarajuća grupa imala znak crtice ('-'), onda funkcija vraća 'A', dok u svakom drugom slučaju vraća 'B'.

Funkcija *sub()* osim same funkcije zamjene jednog stringa s drugim, ima i snažno obilježje koje se zove *povratnost* ili *Backreference*. Ono se može pokazati na idućem primjeru, u kojem će se transformirati jedan format(*markdown*) u drugi (HTML). Da bi primjer bio kratak, samo će se podebljati tekst (oznaka "..." za 'bold'(podebljano)):

```
>>> text = "*HR*-mikrokontroler, *EN*-microcontroller."
>>> pattern = re.compile(r'\*(.*?)\*')
>>> pattern.sub(r"<b>\g<1></b>", text)
'<b>HR</b>-mikrokontroler, <b>EN</b>-microcontroller'
```

Prvo se, dakako, kompilira uzorak koji se podudara sa svakom riječi između dvije zvjezdice (*), između kojih se bilježi svaka riječ (grupa sa '.*' ali s nepohlepnim kvantifikatorom, inače bi bili dohvaćeni svi znakovi do konca stringa). Oznaka \g<broj> koristi opći (global) simbol 'g' (kojim se zamjenjuje svaki pojavak uzorka) i zagrade (<>) koje uključuju broj grupe (u ovom slučaju može se upisati samo 1, jer imamo samo jednu grupu). Ako bi se koristio izraz '\g1' (bez zagrada) i nakon toga željela nadodati npr. znamenka 1, kao rezultat bi se podigla iznimka *error: bad group name* što prikazuje sljedeći primjer:

```
#NEISPRAVAN IZRAZ
>>> text = "*HR*-mikrokontroler, *EN*-microcontroller."
>>> pattern = re.compile(r'\*(.*?)\*')
>>> pattern.sub(r"<b>\g11<\\b>", text)
 error: bad group name

#ISPRAVAN IZRAZ
>>> pattern = re.compile(r'\*(.*?)\*')
>>> pattern.sub(r"<b>\g<1>1<\\b>", text)
    '<b>HR1<\\b>-mikrokontroler, <b>EN1<\\b>-microcontroller'
```

Funkcija *sub()* ima i treći argument **count** koji služi za ograničenje broja zamjena, pa će se za 'count=1' dogoditi samo jedna zamjena.

5.2.3 subn(repl, string, count=0)

Funkcija *subn* ima ista svojstva kao i *sub()* funkcija, osim što vraća n-torku s novim, zamijenjenim stringom i podatkom o broju zamjena. S podacima iz prethodnog primjera, funkcija *subn()* vratit će obrađeni string u *dvojcu*, u kojem će drugi argument biti broj 2, jer su se dogodile dvije zamjene:

```
>>> text = "*HR*-mikrokontroler, *EN*-microcontroller."
>>> pattern = re.compile(r'\*(.*?)\*')
>>> pattern.subn(r"<b>\g<1></b>", text)
('<b>HR</b>-mikrokontroler, <b>EN</b>-microcontroller', 2)
```

5.3 MatchObject

Ovaj objekt predstavlja podudarni uzorak; dobiva se svaki put kad se uspješno izvrši jedna od sljedećih funkcija: *match()*, *search()* i *finditer()*. Uz njega se pojavljuju i funkcije za rad s označenim grupama unutar uzorka, za dobivanje informacija o položaju podudaranja, i slično.

5.3.1 group([group1, ...])

Funkcija *group()* vraća podgrupe podudaranja. Ako je pozvana bez argumenata ili s nulom, vratit će ukupno podudaranje, a ako su dani jedan ili više identifikatora grupa, biti će vraćena podudaranja samo s njima.

```
>>> pattern = re.compile(r"(\w+) (\w+)")
>>> match = pattern.search("vijak M20")
```

Uzorak se podudara sa cijelim stringom i bilježi dvije grupe, *vijak* i *M20*. Pozivi bez argumenta ili s argumentom '0' dat će sljedeće:

- Bez argumenta ili s nulom:

```
>>> match.group()
'vijak M20'
>>> match.group(0)
'vijak M20'
```

- S *group1* argumentom većim od nule, vraća se odgovarajuća grupa:

```
>>> match.group(1)
'vijak'
>>> match.group(2)
```

```
'M20'
```

- Ako grupa ne postoji, biti će podignut *IndexError*:

```
>>> match.group(3)
...
IndexError: no such group
```

- S više argumenata pak, vraćaju se odgovarajuće grupe:

```
>>> match.group(0, 2)
    ('vijak M20', 'M20')
```

Grupe također mogu biti imenovane. Ako uzorak ima imenovane grupe, pristupa im se uz pomoć imena, ali i indeksa:

```
>>> pattern = re.compile(r"(?P<first>\w+) (?P<second>\w+)")
>>> match = pattern.search("vijak M20")
>>> match.group('first')
'vijak'
```

Imenovanim grupama može se pristupiti brojem, po njihovom indeksu, kao što je prikazano u sljedećem kôdu:

```
>>> match.group(1)
'vijak'
```

Mogu se istodobno koristiti čak i oba tipa pristupa:

```
>>> match.group(0, 'first', 2)
('vijak M20', 'vijak', 'M20')
```

5.3.2 groups([default])

Funkcija *groups()* slična je *group()* funkciji samo što ona vraća n-torku sa svim podgrupama u podudaranju, umjesto da vraća samo jednu ili neke od grupa. To se može prikazati na primjeru iz prethodnog odlomka:

```
>>> pattern = re.compile("(\w+) (\w+)")
>>> match = pattern.search("vijak M20")
>>> match.groups()
    ('vijak', 'M20')
```

Očevidno funkcija *groups()* odgovara pozivu *group(1, zadnja_grupa)*. U slučaju da postoje grupe koje se ne podudaraju, vraća se *default*, dodatni argument. Ako pak pretpostavljeni argument nije određen, onda se vraća prazno, *None*:

```
>>> pattern = re.compile("(\w+) (\w+)?")
>>> match = pattern.search("vijak ")
>>> match.groups()
    ('vijak', None)
>>> match.groups("dosjedni")
    ('vijak', 'dosjedni')
```

5.3.3 groupdict([default])

Metoda *groupdict* se koristi u slučajevima u kojima su se koristile imenovane grupe. Ona vraća rječnik sa svim grupama koje su pronađene:

```
>>> pattern = re.compile(r"(?P<element>\w+) (?P<navoj>\w+)")
>>> pattern.search("vijak M20").groupdict()
{'element': 'vijak', 'navoj': 'M20'}
```

Treba napomenuti da ukoliko ne postoje imenovane grupe, ova metoda vraća prazan rječnik.

```
>>> pattern = re.compile(r"(\w+) (\w+)")
>>> pattern.search("vijak M20").groupdict()
{}
```

5.3.4 start([group])

Ponekad je korisno znati indeks na kojem je došlo do podudaranja s uzorkom. Za to se brine funkcija *start()*. Kao i sa svim operacijama koje se odnose na grupe, ako je argument *group* nula, onda operacija radi s cijelim stringom:

```
>>> pattern = re.compile(r"(?P<first>\w+) (?P<second>\w+)?")
>>> match = pattern.search("vijak M20")
>>> match.start(1)
0
>>> match.start(2)
6
```

Ako postoje grupe koje se ne podudaraju, onda se vraća '-1':

```
>>> match = pattern.search("vijak ")
>>> match.start(2)
-1
```

5.3.5 end([group])

Funkcija *end()* ponaša se isto kao i *start()*, osim što vraća kraj podstringa dobivenog podudaranjem stringa s grupom iz uzorka:

```
>>> pattern = re.compile(r"(?P<first>\w+) (?P<second>\w+)?")
>>> match = pattern.search("vijak ")
>>> match.end (1)
5
```

5.3.6 span([group])

Funckija *span()* vraća n-torku s vrijednostima(indeksima) početka i kraja. Ova funkcija se često koristi u tekstualnim editorima u kojima služi za lociranje i označavanje dobivene pretrage. Sljedeći kôd prikazuje primjer ove operacije:

```
>>> pattern = re.compile(r"(?P<first>\w+) (?P<second>\w+)?")
>>> match = pattern.search("vijak ")
>>> match.span(1)
(0, 5)
```

5.3.7 expand([template])

Funkcija *expand()* vraća string nakon što ga je izmjenila s povratnom referencom (*engl. backreference*) u početnom stringu. Vrlo je slična već opisanoj funkciji *sub*:

```
>>> text = "Tvrtka *Atmel* proizvodi razne tipove
    mikrokontrolera."
>>> match = re.search(r'\*(.*?)\*', text)
>>> match.expand(r"<b>\g<1><\\b>")
  '<b>Atmel<\\b>'
```

5.4 Operacije re modula

5.4.1 escape()

Funkcija *escape()* daje doslovno značenje literalima koji se mogu pojaviti u izrazima, ne shvaća ih kao metaznakove, već kao obične znakove.

```
>>> re.findall(re.escape("^"), "^Python^")
['^', '^']
```

5.4.2 purge()

Funkcija *purge()* briše spremište (*eng. cache*) regularnih izraza, pa ju je potrebno povremeno koristiti kako bi se oslobodila memorija nakon korištenja operacija iz *re* modula.

5.5 Kompilacijske oznake (engl. "Compilation flags")

Prilikom kompiliranja uzorka regularnog izraza u **PatternObject**, moguće je djelovati na promjenu uobičajenog ponašanja uzoraka. Da bi se to postiglo, moraju se koristiti kompilacijske zastavice, koje se mogu kombinirati s pomoću ILI (eng. OR) operatora "|".

Flag	Python	Opis
re.IGNORECASE ili re.I	2.x 3.x	Ovaj uzorak će se podudarati s velikim i malim slovima.
re.MULTILINE Ili re.M	2.x 3.x	Ovaj flag mijenja ponašanje dvaju metaznakova: • ^: Koji se sada podudara sa početkom stringa i početkom svake nove linije. • $: Koji se u ovom slučaju podudara sa krajem stringa i krajem svake linije. Konkretnije, podudara se neposredno prije *newline* karaktera.
re.DOTALL ili re.S	2.x 3.x	Metaznak "." će se podudarati sa bilo kojim znakom, čak i sa znakom za novi red.
re.LOCALE ili re.L	2.x 3.x	Ovaj flag čini \w, \W, \b, \B, \S i \S ovisnima o trenutnom lokalnom jeziku. • valja napomenuti da prilikom korištenja re.L i re.U zajedno (re.L | re.U) koristi samo *Locale*
re.VERBOSE Ili re.X	2.x 3.x	On omogućuje pisanje regularnih izraza koji su lakši pročitati i razumjeti. Za to tretira neke znakove na poseban način: • Razmak je ignoriran, osim kada se nalazi u klasi znakova ili je prethođen s *backslashom* • Svi znakovi desno od # zanemaruju se kao da su komentar, osim kada se nalazi u klasi znakova ili je prethođen s *backslashom*
re.DEBUG	2.x 3.x	Daje nam informacije o kompilacijskom uzorku.
re.UNICODE Ili re.U	2.x 3.x	Ovaj flag čini \w, \W, \b, \B, \d \D, \s i \S ovisnima o svojstvima znakova iz *Unicode* baze podataka.
re.ASCII Ili re.A (samo Python 3)	3.x	On čini da \w, \W, \b, \B, \d \D, \s i \S obavljaju samo ASCII podudaranja.

Tablica 4: Kompilacijski flagovi

5.6 Grupiranje

Grupiranje je moćan alat koji omogućuje obavljanje operacija kao što su:

- Stvaranje podizraza na koje se mogu primijeniti kvantifikatori. Primjerice, ponavljanje podizraza umjesto samo jedanog znaka.
- Ograničavanje opsega alternacije (odabira). Umjesto alternacije cijelog izraza, može se definirati točno što mora biti alternirano.
- Izvlačenje podataka iz podudarenih uzorka. Npr. vađenje datuma iz popisa narudžbi.
- Korištenje izvađene informacije opet u regexu, što je vjerojatno najkorisnije svojstvo. Jedan od primjera bilo bi pronalaženje ponovljenih riječi.

Postiže se korištenjem dva metaznaka, tj. zagrada '()' unutar kojih se nalazi uzorak. Najjednostavniji primjer uporabe zagrada bio bi izgradnja podizraza. Neka se pretpostavi popis proizvoda u kojem se ID svakog proizvoda sastoji od dvije ili tri sekvence jedne znamenke, nakon čega slijedi crtica te na kraju još jedan alfanumerički znak:

```
1 >>>re.match(r"(\d-\w){2,3}", ur"1-a2-b")
2 <_sre.SRE_Match at 0x10f690738>
```

U gornjem primjeru zagrade ukazuju *regex engine-u* na to da uzorak unutar njih treba tretirati kao cjelinu.

Još jedan jednostavan primjer njihove uporabe je ograničavanje opsega alternacije. Neka se pretpostavi izraz koji će odgovarati riječima *stroja* ili *strojne*:

```
1 >>>re.search("stroja|ne", "strojne")
2 <_sre.SRE_Match at 0x1043cfe68>
3 >>>re.search("stroja|ne", "stroja")
4 <_sre.SRE_Match at 0x1043cfed0>
```

U gornjem izrazu došlo je do podudaranja, ali problem je u tome što se taj izraz također podudara s *ne*. Stoga, slijedi izraz s klasom znakova:

```
1 >>>re.search("stroj[ane]", "stroja")
2 <_sre.SRE_Match at 0x1043cf1d0>
3 >>>re.search("stroj[ane]", "strojne")
4 <_sre.SRE_Match at 0x1043cf850>
```

Može se vidjeti da izraz djeluje, ali također nije potpuno ispravan jer se podudara i sa "strojn" i "stroje", a te riječi nemaju nikakvo značenje. Slijedi još jedan izraz, ovog puta s alternacijom unutar klase znakova:

```
>>>re.search("stroj[a|ne]", "strojn")
<_sre.SRE_Match at 0x1043cfb28>
```

Na kraju, rješenje ovog primjera leži u korištenju zagrada, što prikazuje slijedeći kôd:

```
>>>re.search("stroj(a|ne)", "stroja")
<_sre.SRE_Match at 0x10439b648>
>>>re.search("stroj(a|ne)", "strojne")
<_sre.SRE_Match at 0x10439b918>
>>>re.search("stroj(a|ne)", "strojn")
    None
>>>re.search("stroj(a|ne)", "stroje")
    None
```

5.6.1 Povratne reference

Jedna od najmoćnijih funkcionalnosti koju grupiranje posjeduje jest mogućnost korištenja zabilježenih grupa unutar uzorka ili nekih drugih operacija, a to je upravo ono što povratne reference (*engl. backreference*) pruže. Slijedi primjer regularnog izraza koji opisuje način na koji se povratne reference koriste, a cilj primjera je pronaći duplicirane riječi, kao što prikazuje slijedeći kôd:

```
>>>pattern = re.compile(r"(\w+) \1")
>>>match = pattern.search(r"LED dioda dioda")
>>>match.groups()
('dioda',)
```

U gornjem primjeru najprije je zabilježena grupa koja se sastoji od jednog ili više alfanumeričkih znakova, nakon čega se uzorak pokušava podudariti s znakom razmaka, i naposljetku s povratnom referencom označenom s izrazom \1. Taj izraz znači kako taj dio regularnog izraza mora odgovarati točno onome čemu je odgovarala prva grupa. Treba napomenuti kako postoje ograničenje, a ono je to da se povratne reference mogu koristiti samo s prvih 99 grupa.

Još jedan primjer u kojoj povratne reference zaista dolaze do izražaja jest onaj iz prethodnog poglavlja, u kojem je bila pretpostavljena struktura popisa proizvoda. Ovog puta redoslijed ID-a biti će promjenjen na način da prvo ide alfanumerički znak, pa crtica i na kraju pozivni broj zemlje:

```
>>>pattern = re.compile(r"(\d+)-(\w+)")
>>>pattern.sub(r"\2-\1", "1-a, 20-baer, 34-afcr")
'a-1, baer-20, afcr-34'
```

5.6.2 Imenovane grupe

Imenovane grupe postoje zato što je korištenje brojeva za označavanje grupa vrlo često zamorno i zbunjujuće, te oni također nemaju mogućnost davanja značenja ili konteksta nekoj grupi. Neka se pretpostavi *regex* koji sadrži nekoliko povratnih referenci (npr. 10) s pomoću brojeva, a da treća nije valjana. Njezino uklanjanje iz *regexa* značilo bi da je potrebno promijeniti indeks svake povratne reference počevši od uklonjene pa nadalje. Kako bi se riješio taj problem, Guido van Rossum je 1997. godine osmislio imenovane grupe za Python 1.5 koje imaju mogućnost davanja imena grupama, tako da se na njih referencira uz pomoć njihovih imena u bilo kojoj operaciji u kojoj su uključene grupe. Njihova sintaksa je (?P<name>pattern), u kojoj P dolazi od Python-specifične ekstenzije.

Slijedeći primjer prikazuje koliko su povratne reference, uz korištenje imenovanih grupa, jednostavnije za korištenje i održavanje:

```
>>>pattern = re.compile(r"(?P<country>\d+)-(?P<id>\w+)")
>>>pattern.sub(r"\g<id>-\g<country>", "1-a, 20-baer, 34-afcr
    ")
'a-1, baer-20, afcr-34'
```

Gornji primjer prikazuje kako se za referenciranje na grupu pomoću imena u **sub** operaciji, mora koristiti sintaksa \g<ime>. Imenovane grupe se također mogu koristiti unutar samog uzorka, što prikazuje slijeći kôd:

```
>>>pattern = re.compile(r"(?P<rijec>\w+) (?P=rijec)")
>>>match = pattern.search(r"LED dioda dioda")
>>>match.groups()
('dioda',)
```

Što je jednostavnije i puno čitljivije nego s korištenjem brojeva kao referenci.

Uporaba	Sintaksa
Unutar uzorka	(?P=name)
U *repl* stringu unutar *sub* operacije	\g<name>
U bilo kojoj operaciji *MatchObjekta*	Npr. match.group("name")

Tablica 5: Sintaksa za imenovane grupe

5.6.3 Ne-hvatajuće grupe (engl. "Non-capturing groups")

Kao što je prije spomenuto, hvatanje sadržaja nije jedina primjena za koju grupe služe. Postoje slučajevi kada se žele koristiti grupe, ali nema interesa za vađenje njihovog sadržaja, na primjer, kod alternacije. Zato postoji način za stvaranje grupa bez hvatanja. Slijedi primjer korištenja grupa koji hvata njen sadržaj:

```
1 >>>re.search("energij(a|e)", "energija")
2 <_sre.SRE_Match at 0x10e90b828>
3 >>>re.search("energij(a|e)", "energija").groups()
4 ('a',)
```

Saržaj grupe koji je uhvaćen u gornjem primjeru nema nikakvog smisla, stoga, za pretraživanje stringa bez hvatanja potrebno je koristiti sintaksu (?:pattern).

```
1 >>>re.search("energij(?:a|e)", "energija")
2 <_sre.SRE_Match at 0x10e912648>
3 >>>re.search("energij(?:a|e)", "energija").groups()
4 ()
```

Treba napomenuti kako se na takve grupe ne možemo referencirati.

5.6.4 Posebni slučajevi

Python pruža neke posebne oblike grupa uz pomoć kojih je moguće izmjeniti regularni izraz ili čak učiniti da se uzorak podudari samo ako se prethodna grupa podudarila, a neki od njih su:

1. **Grupne zastavice (engl. "Group flags")**

   Način da se flagovi primjene na grupu koristeći poseban oblik grupiranja: (?iLmsux) gdje slijedeća tablica prikazuje značenje pojedinih slova.

```
1 >>>re.findall(r"(?u)\w+", ur"\~{n}")
2 [u'\xf1']
```

Slovo	Flag
i	re.IGNORECASE
L	re.LOCALE
m	re.MULTILINE
s	re.DOTALL
u	re.UNICODE
x	re.VERBOSE

Tablica 6: Grupni flagovi

2. **Da-uzorak | Ne-uzorak (engl. "Yes-pattern | No-pattern")**

   Ovo je vrlo koristan slučaj grupa koji se pokušava podudarati s uzorkom u slučaju da je pronađen prethodni. S druge strane, ne pokušava se uskladiti s uzorkom u slučaju da prethodna grupa nije pronađena. Ukratko, funkcija ovih grupa je ista kao i *if statement*. Sintaksa za ovu operaciju je:

   ```
   (?(id/name)yes-pattern|no-pattern)
   ```

   Ovaj izraz znači: ako se grupa s ovim ID-om već podudarila, tada se na tom mjestu stringa *yes-pattern* mora podudariti. Ukoliko se grupa nije podudarila, onda se *no-pattern* mora podudariti.

   Neka se pretpostavi primjer popisa proizvoda u kojem se u ovom slučaju ID može izvršiti na dva načina:

   - Pozivni broj države (dvije znamenke), crtica, tri ili četiri alfanumerička znaka, crtica, i pozivni broj mjesta (2 znamenke). Npr. *34-adrl-01*.
   - Tri ili četiri alfanumeričkih znakova. Npr. *adrl*.

   Stoga, kada postoji pozivni broj države, mora biti pronađen i pozivni broj mjesta:

```
1 >>>pattern = re.compile(r"(\d\d-)?(\w{3,4})(?(1)(-\d\d))
    ")
2 >>>pattern.match("34-erte")
3 None
4 >>>pattern.match("34-erte-22")
5 <_sre.SRE_Match at 0x10f68b7a0>
6 >>>pattern.match("erte")
7 <_sre.SRE_Match at 0x10f68b828>
```

Uz dodavanje još jednog ograničenja na prethodni primjer: ako nema pozivnog broja države mora postojati ime od tri ili četri slova na kraju stringa, npr. *adrl-sala*:

```
>>>pattern = re.compile(r"(\d\d-)?(\w{3,4})-(?(1)(\d\d)
    |[a-z]{3,4})$")
>>>pattern.match("34-erte-22")
<_sre.SRE_Match at 0x10f6ee750>
>>>pattern.match("erte-abcd")
<_sre.SRE_Match at 0x10f6ee880>
>>>pattern.match("34-erte")
None
```

Također treba napomenuti kako *no-pattern* nije obavezan.

5.7 Pogled uokolo

Postoje različiti mehanizmi podudaranja znakova. Znak koji je već podudaren ne može se ponovno uspoređivati, a jedini način da omogućimo podudaranje nadolazećem znaku je taj da odbacimo trenutnog. Iznimke su nekoliko metaznakova, takozvani *zero-width assertions*. Ovi znakovi ukazuju na pozicije, a ne na stvarni sadržaj, npr. znak (^) koji označava početak linije ili znak ($) koji označava kraj linije. Oni samo osiguravaju da je položaj u unosu ispravan, bez da zapravo konzumiraju znak ili se podudaraju s njime.

Moćnija vrsta *zero-width assertions-a* je pogled uokolo (*look around*), mehanizam s kojim je moguće podudarati se s određenim prethodnim (*look behind*) ili onostranim (*look ahead*) vrijednostima za trenutnu poziciju. Oni učinkovito čine tvrdnju bez konzumiranja znakova, te vraćaju samo pozitivan ili negativan rezultat podudaranja.

Look around mehanizam je vjerojatno najnepoznatija, a u isto vrijeme i najmoćnija tehnika u regularnim izrazima. Ovaj mehanizam omogućuje stvaranje moćnih regularnih izraza koji se ne mogu napisati drugačije, bilo zbog složenosti koju bi predstavljali ili jednostavno zbog tehničkih ograničenja regularnih izraza bez *look around-a*.

Look ahead i *look behind* se također mogu podijeliti u još dvije podvrste: pozitivni i negativni.

- Pozitivni *look ahead*: Ovaj mehanizam je predstavljen kao izraz kojemu prethodi znak upitnika i znak jednakosti ?= unutar zagrada. Npr. (?=Regex) će odgovarati samo ako se dani *Regex* podudara s nadolazećim ulazom.

- Negativni *look ahead*: Ovaj mehanizam je određen kao izraz kojemu prethodi znak upitnika i znak uskličnika ?! unutar zagrada. Npr. (?!Regex) će odgovarati samo ako se dani *Regex* ne podudara sa nadolazećim ulazom.

- Pozitivni *look behind*: Ovaj mehanizam je predstavljen kao izraz kojemu prethodi znak upitnika, znak manje-nego i znak jednakosti ?<= unutar zagrada. Npr. (?<=Regex) će odgovarati samo ako se dani *Regex* podudara sa prethodnim ulazom.

- Negativni *look behind*: Ovaj mehanizam je predstavljen kao izraz kojemu prethodi znak upitnika, znak manje-nego i znak upitnika ?<! unutar zagrada. Npr. (?<!Regex) će odgovarati samo ako se dani *Regex* ne podudara sa prethodnim ulazom.

Također treba napomenuti kako su zbog svoje *zero-width* prirode ove dvije *look around* operacije veoma složene i teško razumljive.

5.7.1 Pogled unaprijed

Ovaj mehanizam se pokušava podudariti s nadolazećim podizrazom koji je dan kao argument unutar zagrada. Predstavljen je kao izraz kojemu prethodi znak upitnika i znak jednakosti ?= unutar zagrada (?= regex). Slijedeći primjer prikazuje usporedbu između njegovog principa rada i običnog regularnog izraza:

```
1 >>>pattern = re.compile(r'mehanikom')
2 >>>result = pattern.search("Robotika je srodna sa
     elektronikom, mehanikom i sa softverom")
3 >>>print result.start(), result.end()
4 36 45
```

Sa *look ahead* mehanizmom:

```
1 >>>pattern = re.compile(r'(?=mehanikom)')
2 >>>result = pattern.search("Robotika je srodna sa
     elektronikom, mehanikom i sa softverom")
3 >>>print result.start(), result.end()
4 36 36
```

Kao rezultat dobivena je pozicija u indeksu 36, tj. i početak i kraj pokazuju na isti indeks, to je zbog toga što *look around* ne konzumira znakove te također ne određuje niti sadržaj rezultata.

Slijedeći primjer je zadatak kojemu je cilj podudariti se s bilo kojom riječi nakon koje slijedi znak zareza ',':

```
1 >>>pattern = re.compile(r'\w+(?=,)')
2 >>>pattern.findall("Potrebne komponente: tranzistor,
    kondenzator, i dioda.")
3 ['tranzistor', 'kondenzator']
```

U prethodnom primjeru napisan je regularni izraz koji prihvaća bilo koje ponavljanje alfanumeričkih znakova nakon kojih slijedi znak zareza koji se ne koristi kao dio rezultata. Dakle, samo *tranzistor* i *kondenzator* bili su dio rezultata kako *dioda* nije imala zarez nakon imena i to je jedna od najbitnijih značajki ovog mehanizma.

Treba napomenuti kako se *look ahead* mehanizam gleda kao poseban podizraz na kojeg se može utjecati sa svom snagom regularnih izraza. Stoga, može se koristiti svakakva vrsta konstrukcije izraza, kao i alternacija:

```
1 >>>pattern = re.compile(r'\w+(?=,|\.)')
2 >>>pattern.findall("Potrebne komponente: tranzistor,
    kondenzator, i dioda.")
3 ['tranzistor', 'kondenzator', 'dioda']
```

5.7.2 Negativni pogled unaprijed

Negativan *look ahead* mehanizam predstavlja istu vrstu *look ahead-a* ali uz zamjetnu razliku: rezultat je valjan samo ako podizraz ne odgovara. Predstavljen je kao izraz kojemu prethodi znak upitnika i znak uskličnika ?! unutar zagrada (?! regex). Koristan je kada se želi izraziti ono što se ne bi trebalo dogoditi, npr. kada se želi pronaći bilo koji *vijak* koji nije *M20*:

```
1 >>>pattern = re.compile(r'vijak(?!\sM20)')
2 >>> result = pattern.finditer("vijak M8, vijak M20, vijak
    M24")
3 >>>for i in result:
4 ...print i.start(), i.end()
5 ...
6 0 5
7 21 26
```

5.7.3 Pogled uokolo i substitucije

Zero-width priroda *look around* operacija osobito je korisna u zamjenama, tj. zahvaljujući njima, moguće je obavljati transformacije koje bi inače bile izuzetno složene za čitanje i pisanje. Tipičan primjer *look ahead-a* i supstitucije je konverzija broja

koji se sastoji samo od numeričkih znakova, npr. 1234567890, u zarezom odvojen broj: 1,234,567,890. Za formiranje ovog regularnog izraza, potrebno je slijediti određenu strategiju. Ono što treba učiniti jest grupiranje brojeva u blokove od tri znamenke koji će se potom moći zamijeniti s istom grupom plus znak zareza.

Pokušaj s gotovo naivnim pristupom slijedećeg regularnog izraza:

```
>>>pattern = re.compile(r'\d{1,3}')
>>>pattern.findall("Broj je: 12345567890")
['123', '455', '678', '90']
```

Kao što je vidljivo iz gornjeg primjera, ovaj pokušaj nije uspjeo. Ostvareno je grupiranje u blokovima od tri znamenke s lijeva na desno, a treba ih uzeti s desna na lijevo. Potrebno je pronaći jednu, dvije ili tri znamenke nakon kojih mora slijediti bilo koji broj blokova od tri znamenke, sve dok se ne pronađe nešto što nije znamenka. To će imati sljedeći učinak na broj. Kada će pokušavati pronaći jednu, dvije, ili tri znamenke, regularni izraz će započeti sa uzimanjem samo jedne, a to će biti brojka 1. Zatim, će pokušati uhvatiti blokove od točno tri broja, primjerice, 234, 567, 890, sve dok ne pronađe znak koji nije znamenka, a to je kraj ulaza.

```
>>>pattern = re.compile(r'\d{1,3}(?=(\d{3})+(?!\d))')
>>>results = pattern.finditer('1234567890')
>>>for result in results:
...     print result.start(), result.end()
...
0 1
1 4
4 7
```

Ovaj put, korišten je ispravan pristup zato što su identificirani ispravni blokovi: 1, 234, 567 i 890. Sada je još potrebno iskoristiti supstituciju za zamijenu svakog podudaranja na kojeg je regularni izraz naišao s istim rezultatom podudaranja plus znak zareza:

```
>>>pattern = re.compile(r'\d{1,3}(?=(\d{3})+(?!\d))')
>>>pattern.sub(r'\g<0>,', "1234567890")
'1,234,567,890'
```

5.7.4 Pogled unatrag

Look behind se sa sigurnošću može definirati kao suprotnost operacije *look ahead*. On se pokušava podudariti s podizrazom danim kao argument, te ujedno i sa izrazom koji se nalazi iza tog podizraza. Također ima *zero-width* prirodu, i stoga, on isto nije dio rezultata. Predstavljen je kao izraz kojemu prethodi znak upitnika, znak

manje-nego, i znak jednakosti ?<= unutar zagrada: (?<= regex).

Moguće ga je, primjerice, koristiti u primjeru sličnom onome koji je korišten u negativnom *look ahead-u*, ali ovaj put za pronalaženje samo izraza *vijak M20*:

```
>>>pattern = re.compile(r'(?<=vijak\s)M20')
>>>pattern.findall("vijak M8, vijak M20, vijak M24")
['M20']
```

Treba napomenuti kako u Pythonovom *re* modulu postoji temeljna razlika između toga kako se provode *look ahead* i *look behind* operacije. Zbog velikog broja duboko ukorijenjenih tehničkih razloga, *look behind* mehanizam ima mogućnost podudaranja samo sa uzorcima fiksne širine, a uzorci fiksne širine ne mogu sadržavati kvantifikatore za izraze promjenjive duljine. Ostale konstrukcije izraza promjenjive duljine kao što su povratne reference također nisu dopuštene. Alternacija je dopuštena, ali samo ako varijante imaju istu duljinu. Opet, ta ograničenja nije prisutna u spomenutom Python-ovom *regex* modulu.

Ono što će se dogoditi ako se koristi alternacija s različitim duljinama varijanta unutar *look behind* izraza prikazuje slijedeći primjer:

```
>>>pattern = re.compile(r'(?<=(vijak|matica)\s)M20')
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "C:\Python27\Lib\re.py", line 190, in compile
    return _compile(pattern, flags)
  File "C:\Python27\Lib\re.py", line 244, in _compile
    raise error, v # invalid expression
error: look-behind requires fixed-width pattern
```

Dobivena je iznimka koja znači da *look behind* zahtijeva uzorak fiksne širine.

5.7.5 Negativni pogled unatrag

Negativan *look behind* mehanizam predstavlja istu vrstu glavnog *look behind* mehanizma, ali ima važeći rezultat samo ako dani podizraz ne odgovara. Predstavljen je kao izraz kojemu prethodi znak upitnika, znak manje-nego, i znak upitnika ?<! unutar zagrada: (?<! regex). Treba napomenuti kako negativni *look behind* ne samo da dijeli većinu svojstva *look behind* mehanizma, već također dijeli i sva ograničenja.

Slijedi primjer kojemu je zadatak pronaći tekstualne lokacije svih vrsta dioda koje nisu *LED* diode:

```
>>>pattern = re.compile(r'(?<!LED\s)dioda')
>>>results = pattern.finditer("Zener dioda, LED dioda,
   Schottky dioda, ...")
```

```
>>>for result in results:
...     print result.start(), result.end()
...
6 11
33 38
```

5.7.6 Pogled uokolo i grupe

Jedna od korisnih upotreba *look around* mehanizma je unutar grupa. Kada se koriste grupe, često je potrebno da bude usklađen i vraćen vrlo specifičan rezultat unutar nje. Slijedi primjer formata na kojem će biti korišten *look around* mehanizam:

```
INFO 2013-09-17 12:13:44,487 authentication failed
```

Regularni izraz kojim se dobivaju obje vrijednosti glasi:

```
>>>pattern = re.compile(r'\w+\s[\d-]+\s[\d:,]+\s(.*\sfailed)')
>>>result = pattern.search("INFO 2013-09-17 12:13:44,487 authentication failed")
>>>result.group(1)
'authentication failed'
```

Međutim, ako je potrebno da izraz odgovara samo kada neuspjeh nije neuspjeh provjere autentičnosti, onda to ostvarujemo dodavanjem negativnog *look behind-a*:

```
>>>pattern = re.compile(r'\w+\s[\d-]+\s[\d:,]+\s(.*(?<!authentication\s)failed)')
>>>pattern.findall("INFO 2013-09-17 12:13:44,487 authentication failed")
[]
>>>pattern.findall("INFO 2013-09-17 12:13:44,487 login failed")
['login failed']
```

6 Povezani podaci u praksi

6.1 Virtuoso univerzalni server

Virtuoso je višenamjenski, multi-protokolni mrežni poslužitelj kojeg je napravio *OpenLink Software*, a uključuje upravljanje podacima kao što su SQL, RDF, XML, Free Text, web aplikacije, povezani podaci i ostalo.

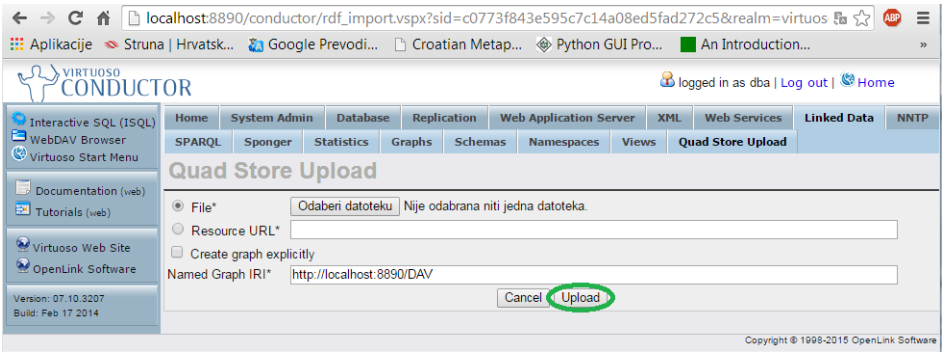
Virtuoso također sadrži ugrađenu web-baziranu aplikaciju koja se naziva *Virtuoso Conductor*, a pruža sučelje za upravljanje bazama podataka koje uobičajeno izvodi administrator (DBA, *engl. Database Administrator*):

Slika 11: Prikaz Virtuoso Conductor sučelja

Unutar opsega ovog rada biti će korišten i objašnjen samo dio *Virtuosa* koji se tiče ubacivanja povezanih podataka u *triplestore* bazu podataka, te njihovog pretraživanja uz pomoć naredbi iz SPARQL editora.

6.2 Ubacivanje trojaca u *triplestore* bazu

Ubacivanje datoteka koje sadrže LOD trojce, tj. datoteka koje imaju ekstenziju bilo .rdf, .xml, .owl, te ostale koje zadovoljavaju jedan od formata standardnih načina serijalizacije kao što su RDF/XML, N-triples i Turtle, u *triplestore* bazu vrši se unutar *Linked Data* $\Rightarrow$ *Quad Store Upload* sučelja pritiskom na gumb "Upload":

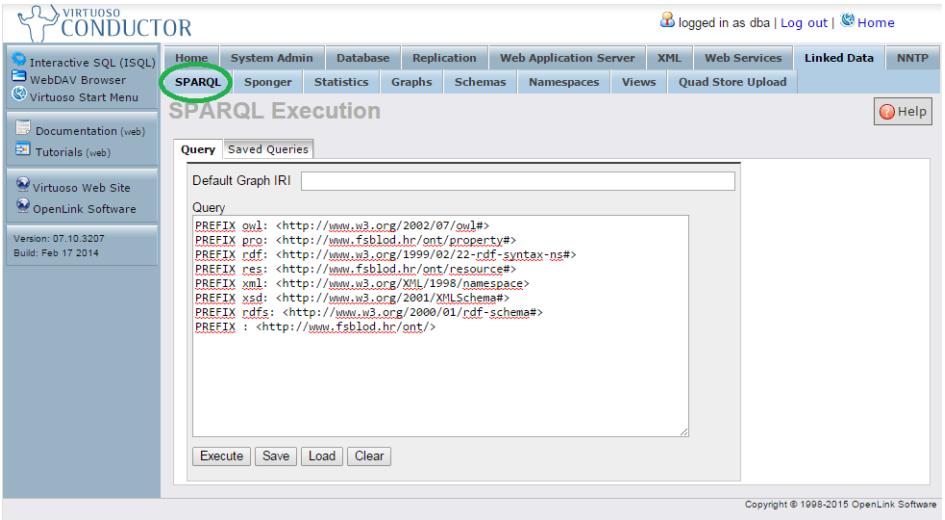


Slika 12: Prikaz Virtuoso Conductor, Quad Store Upload sučelja

6.3 SPARQL

Sada kada je objašnjena osnovna ideja semantičkog weba i povezanih podataka, te njihova obilježja i način pohrane, potrebno je te podatke i isčitati iz baze. W3C je postavio SPARQL jezik kao standard za pretragu podataka u RDF modelu.

Pretraživanje baze vrši se unutar *Virtuoso-vog* SPARQL editora kojeg prikazuje slijedeća slika:



Slika 13: Prikaz Virtuoso Conductor, SPARQL editora

SPARQL je za semantički web, odnosno RDF baze podataka, ono što je SQL za relacijske baze podataka. Jezik za pretragu podataka u RDF modelu, koji nije ograničen samo na baze podataka. S njim je moguće pretraživati bilo koji format zapisanih RDF tojaca, poput XML datoteka.

6.3.1 Sintaksa

Kod upita putem SPARQL-a postoje varijable. One su namijenjene kako bi u njih spremali rezultat pretrage ili ih koristili samo privremeno kako bi filtrirali rezultate. Varijable se označavaju upitnikom ispred imena i nije ih potrebno deklarirati unaprijed.

Slično kao i u SQL-u, upit postavljamo sa kombinacijom naredbi SELECT i WHERE.

U SELECT dijelu upisujemo varijable koje želimo pronaći, odnosno varijable koje želimo zadržati kao rezultat. Važno je napomenuti kako u nastavku možemo deklarirati i dodatne varijable koje će nam pmoći u pretrazi, ali ako nisu spomenute u SELECT dijelu, one neće biti u rezultatima. Ako želimo zadržati sve varijable korištene u pretrazi, tada poput SQL-a možemo upisati zvjezdicu "*" umjesto varijabli.

WHERE predstavlja dio upita u kojeg se upisuju trojci s varijablama. Oni se pišu između vitičastih zagrada, "{" i "}", te kao u Turtle sintaksi, elementi trojaca, odnosno subjekt, objekt i predikat su odvojeni praznim mjestom, a na kraju linije upisuje se točka. URI se stavljaju između izlomljenih zagrada "<" i ">", doslovne vrijednosti u navodnike, a Qname se piše bez ikakvih znakova. Također, koriste se točka-zarez ";" za ponavljanje subjekta (sa različitim predikatom i objektom) ili zarez "," za ponavljanje subjekta i predikata (sa različitim objektom). Umjesto nepoznatog elementa trojca upisuje se varijabla.

Također, slično kao i Turtle format, moguće je odrediti prefikse koji omogućuju kompaktniji prikaz trojaca. Određeni su prefiksi *res* i *pro*, kojima glavnu domenu predstavlja URI *http://www.fsblod.hr/ont/*, te ostali, nužni za definiranje veza i odnosa između resursa kao što su *owl. rdf*, *rdfs*, itd.

Kroz sljedeće primjere biti će prikazani SPARQL upiti prema bazi podataka koja sadrži pojmove, zajedno s njihovim svojstvima, iz tehničkog rječnika.

6.3.2 Primjeri pretraživanja

Ukoliko se iz baze žele preuzeti svi pojmovi, njihove definicije i grane djelatnosti kojoj pripadaju, potrebno je napisati sljedeću SPARQL naredbu. Pod SELECT dio je također dodana naredba DISTINCT koja u slučaju ponovljenih vrijednosti u rezultatima filtira sve jednake i ostavlja samo jednu.

Query

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX pro: <http://www.fsblod.hr/ont/property#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX res: <http://www.fsblod.hr/ont/resource#>
PREFIX xml: <http://www.w3.org/XML/1998/namespace>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : <http://www.fsblod.hr/ont/>

SELECT DISTINCT ?pojam ?definicija ?grana WHERE {
?individua rdf:type owl:NamedIndividual .
?individua rdfs:label ?pojam .
?individua pro:definicija ?definicija .
?individua pro:grana ?grana .
}
```

Execute Save Load Clear

pojam	definicija	grana
"matica uzorka"	"dio ispitnoga uzorka koji sadržava sve sastojke osim analita"	"analitička kemija "
"A-pužni vijak"	"pužni vijak kojemu je presječna krivulja boka zuba i ravnine okomite na os Arhimedova spirala"	"opće strojarstvo (konstrukcije) "
"Carnotov stroj"	"idealni stroj koji bi radio na načelu Carnotova termodinamičkoga kružnog procesa"	"opća i klasična fizika "
"Clarkova elektroda"	"elektroda koja omogućuje ampermetrijsko mjerenje aktiviteta otopljenoga kisika"	"analitička kemija "
"E-pužni vijak"	"pužni vijak oblikom istovjetan cilindričnomu zupčaniku s evolventnim kosim zubima velikoga kutnog nagiba boka zuba"	"opće strojarstvo (konstrukcije) "
"bakrova/bakrosulfatna referencijska elektroda"	"referencijska bakrena elektroda uronjena u zasićenu otopinu bakrova sulfata"	"reakcijsko inženjerstvo "
"cijevna dioda"	"cijevni evakuirani elektronički element strujnoga kruga u kojemu elektroni izlaze iz ugrijane katode, a skupljaju se na anodi"	"opća i klasična fizika "
"cilindrična matica s urezom"	"cilindrična matica koja s jedne čelne strane ima utor pravokutnoga poprečnog presjeka duž cijeloga promjera"	"opće strojarstvo (konstrukcije) "
"cilindrični pužni vijak"	"cilindrični zupčanik s jednim kosim zubom ili s više kosih zuba"	"opće strojarstvo

Slika 14: SPARQL DISTINCT

U SELECT djelu su također navedene samo varijable ?pojam, ?definicija i ?grana jer nije potrebno ispisivati ostale (?individua) koje su služile samo za pretragu i sadrže URI-e. U prvom RDF trojcu upita u varijablu ?individua spremljene su sve vrijednosti koje s *owl:NamedIndividual* veže ontologija *rdf:type*, odnosno URI vrijednosti koje predstavljaju pojmove. Zatim su u drugom redu iz tih filtriranih pojmova izvučene njihove doslovne vrijednosti (*rdfs:label*), koje su spremljene u varijablu ?pojam. U trećem i četvrtom retku na isti način izvučena je definicija i grana (djelatnosti) pojedinih pojmova preko odgovarajuće ontologije (*pro:definicija i pro:grana*), te su također spremljene u varijable ?definicija i ?grana kako bi ih se moglo ispisati.

U sljedećem primjeru, prethodni će upit biti upisan u obliku ponovljenih RDF trojaca, jer se varijabla, odnosno subjekt u njima ponavlja. Prvi RDF u kojemu je varijabla ?individua subjekt piše se normalno, a na kraju se umjesto točke upisuje točka-zarez ";". Sada u preostala dva nije potrebno ponavljati subjekt, već se samo

piše predikat i objekt koji se odvajaju također točka-zarezom ";", a na kraju svih ponovljenih stavlja se točka.

Query

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX pro: <http://www.fsblod.hr/ont/property#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX res: <http://www.fsblod.hr/ont/resource#>
PREFIX xml: <http://www.w3.org/XML/1998/namespace>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : <http://www.fsblod.hr/ont/>

SELECT DISTINCT ?pojam ?definicija ?grana WHERE {
?individua rdf:type owl:NamedIndividual ;
           rdfs:label ?pojam ;
           pro:definicija ?definicija ;
           pro:grana ?grana .
} ORDER BY ?pojam LIMIT 8 OFFSET 5
```

Execute Save Load Clear

pojam	definicija	grana
"cijevna dioda"	"cijevni evakuirani elektronički element strujnoga kruga u kojemu elektroni izlaze iz ugrijane katode, a skupljaju se na anodi"	"opća i klasična fizika "
"cilindrična matica s urezom"	"cilindrična matica koja s jedne čelne strane ima utor pravokutnoga poprečnog presjeka duž cijeloga promjera"	"opće strojarstvo (konstrukcije) "
"cilindrični pužni vijak"	"cilindrični zupčanik s jednim kosim zubom ili s više kosih zuba"	"opće strojarstvo (konstrukcije) "
"dioda"	"elektronički element koji propušta struju samo u jednome smjeru"	"opća i klasična fizika "
"dosjedni vijak"	"vijak kojemu je dio tijela u dosjedu s provrtom u kojemu se nalazi"	"opće strojarstvo (konstrukcije) "
"dvopolna elektroda"	"elektroda smještena u električnome polju između anode i katode koja nije električno povezana s ostalim dijelovima elektrolitskoga članka"	"proizvodno strojarstvo "
"elastični vijak"	"vijak kojemu pojedini dijelovi tijela imaju manji promjer od unutarnjega promjera navoja"	"opće strojarstvo (konstrukcije) "
"elektroda"	"vodič kroz koji struja ulazi u vakuum, nekovinsko, kruto, tekuće ili plinovito sredstvo ili iz takva sredstva izlazi"	"elektrostrojarstvo "

Slika 15: SPARQL ORDER BY, LIMIT, OFFSET

Upit je također proširen naredbama ORDER BY kojom su rezultati poredani prema varijabli ?pojam, odnosno po abecedi. Kada bi bilo potrebno rezultate poredati u obrnutom redoslijedu, upisuje se ORDER BY DESC.

Naredba LIMIT ograničava broj dobivenih rezultata, u ovom slučaju na 8. Ta naredba je vrlo korisna kada se pretražuju ogromne baze podataka.

OFFSET naredbom određuje se od kud se želi početi s ispisivanjem rezultata, a u ovom slučaju je to od petog pa nadalje.

Za prikaz naredbe OPTIONAL, prethodni upit biti će proširen tako da iz baze budu dohvaćene istoznačnice pojmova. Upravo iz razloga što one nisu poznate kod svih pojmova, odnosno RDF trojci nisu zapisani (nedostaje podatak), običan SQL upit ne može prikazati te rezultate, tj. došlo bi do greške kod varijabli jer vrijednosti ne postoje. Tada se naredbom OPTIONAL koja se piše jednako kao i WHERE, ali unutar WHERE-a, prikazuju dodatni RDF trojci, kod kojih je pronađen i taj podatak (u ovom slučaju istoznačnice).

Query

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX pro: <http://www.fsblod.hr/ont/property#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX res: <http://www.fsblod.hr/ont/resource#>
PREFIX xml: <http://www.w3.org/XML/1998/namespace>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : <http://www.fsblod.hr/ont/>

SELECT DISTINCT ?pojam ?definicija ?istoznacnice WHERE {
?individua rdf:type owl:NamedIndividual ;
           rdfs:label ?pojam ;
           pro:definicija ?definicija .
OPTIONAL { ?individua pro:istoznacnice ?istoznacnice . }
} ORDER BY ?pojam
```

Execute Save Load Clear

pojam	definicija	istoznacnice
"A-pužni vijak"	"pužni vijak kojemu je presječna krivulja boka zuba i ravnine okomite na os Arhimedova spirala"	"A-puž , ZA-puž, ZA-pužni vijak"
"Carnotov stroj"	"idealni stroj koji bi radio na načelu Carnotova termodinamičkoga kružnog procesa"	
"Clarkova elektroda"	"elektroda koja omogućuje ampermetrijsko mjerenje aktiviteta otopljenoga kisika"	
"E-pužni vijak"	"pužni vijak oblikom istovjetan cilindričnomu zupčaniku s evolventnim kosim zubima velikoga kutnog nagiba boka zuba"	"E-puž , ZE-puž, ZE-pužni vijak"
"bakrova/bakrosulfatna referencijska elektroda"	"referencijska bakrena elektroda uronjena u zasićenu otopinu bakrova sulfata"	"bakar / bakar sulfat referentna elektroda"
"cijevna dioda"	"cijevni evakuirani elektronički element strujnoga kruga u kojemu elektroni izlaze iz ugrijane katode, a skupljaju se na anodi"	"vakuumska dioda"
"cilindrična matica s urezom"	"cilindrična matica koja s jedne čelne strane ima utor pravokutnoga poprečnog presjeka duž cijeloga promjera"	
"cilindrični pužni vijak"	"cilindrični zupčanik s jednim kosim zubom ili s više kosih zuba"	"cilindrični puž"
"dioda"	"elektronički element koji propušta struju samo u jednome smjeru"	
"dosjedni vijak"	"vijak kojemu je dio tijela u dosjedu s provrtom u kojemu se nalazi"	

Slika 16: SPARQL OPTIONAL

Sljedeća naredba je FILTER. Ona se također piše unutar WHERE djela, a u zagrade se upisuju uvjeti prema kojima se žele filtrirati rezultati. U danom primjeru dani su uvjeti uz koje će se ispisati svi pojmovi čija imena započinju s "d". Dodatna opcija "i" unutar *regexa* određuje da se pri filtriranju velika i mala slova ne uzimaju u obzir.

```
SELECT DISTINCT ?pojam ?definicija ?grana WHERE {
?individua rdf:type owl:NamedIndividual ;
           rdfs:label ?pojam ;
           pro:definicija ?definicija ;
           pro:grana ?grana .
FILTER (regex(?pojam , "^D", "i"))
} ORDER BY ?pojam
```

Execute Save Load Clear

pojam	definicija	grana
"dioda"	"elektronički element koji propušta struju samo u jednome smjeru"	"opća i klasična fizika "
"dosjedni vijak"	"vijak kojemu je dio tijela u dosjedu s provrtom u kojemu se nalazi"	"opće strojarstvo (konstrukcije) "
"dvopolna elektroda"	"elektroda smještena u električnome polju između anode i katode koja nije električno povezana s ostalim dijelovima elektrolitskoga članka"	"proizvodno strojarstvo "

Slika 17: SPARQL FILTER

Svi ovi upiti su usmjereni na bazu podataka koja sadržava RDF trojce. SPARQL podržava i pretragu datoteka koje na neki od standardiziranih načina serijalizacije također sadrže RDF trojce. U tom slučaju ispred naredbe SELECT uz pomoć naredbe FROM pozivamo lokalnu datoteku ili datoteku na serveru.

Uz SELECT naredbu, u SPARQL jeziku postoje još i tipovi pretraga kao što su ASK, CONSTRUCT i DESCRIBE.

Krajem 2012. godine predložen je SPARQL 1.1 standard koji sadržava još veća proširenja osnovnog SPARQL jezika, te dodaje vrlo bitne funkcije za manipuliranje RDF bazama. Uvode se INSERT i DELETE, osnovne naredbe za umetanje, brisanje i izmjenu podataka u bazi, te na taj način SPARQL postaje *Read/Write* jezik. Iako još nije službeno prihvaćen kao standard, mnogo komercijalnih, ali i besplatnih programa namijenjenih korištenju povezanih podataka i semantičkom web-u ga već koriste.

Sljedeći primjer prikazuje sintaksu INSERT INTO naredbe i sam RDF trojac koji je sastavljen od subjekta zapisanog pomoću Qname URI-a. Predikat je također Qname URI-a koji govori da se radi o pojmu. Objekt je doslovna vrijednost pojma. U izlomljene zagrade potrebno je upisati RDF graf (*RDF Graph*) u kojeg ubacujemo trojac, a u ovom slučaju je to lokalni zadani, pa se ostavlja prazno.

Query

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX pro: <http://www.fsblod.hr/ont/property#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX res: <http://www.fsblod.hr/ont/resource#>
PREFIX xml: <http://www.w3.org/XML/1998/namespace>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : <http://www.fsblod.hr/ont>

INSERT INTO <> { res:svjetleca_dioda rdfs:label "svjetleća dioda" }
```

Slika 18: SPARQL INSERT INTO

Zadnja naredba koja će biti prikazana u ovom radu je DELETE FROM. Sintaksa za nju je ista kao i INSERT INTO, samo što se unutar zagrada upisuju RDF trojci prema kojima se vrši brisanje iz baze. Ako se kao varijable navedu subjekt, predikat i objekt, briše se cijela baza.

Query

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX pro: <http://www.fsblod.hr/ont/property#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX res: <http://www.fsblod.hr/ont/resource#>
PREFIX xml: <http://www.w3.org/XML/1998/namespace>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : <http://www.fsblod.hr/ont>

DELETE FROM <> { ?s ?p "dioda" }
```

Slika 19: SPARQL DELETE FROM

6.4 Izrada web stranice

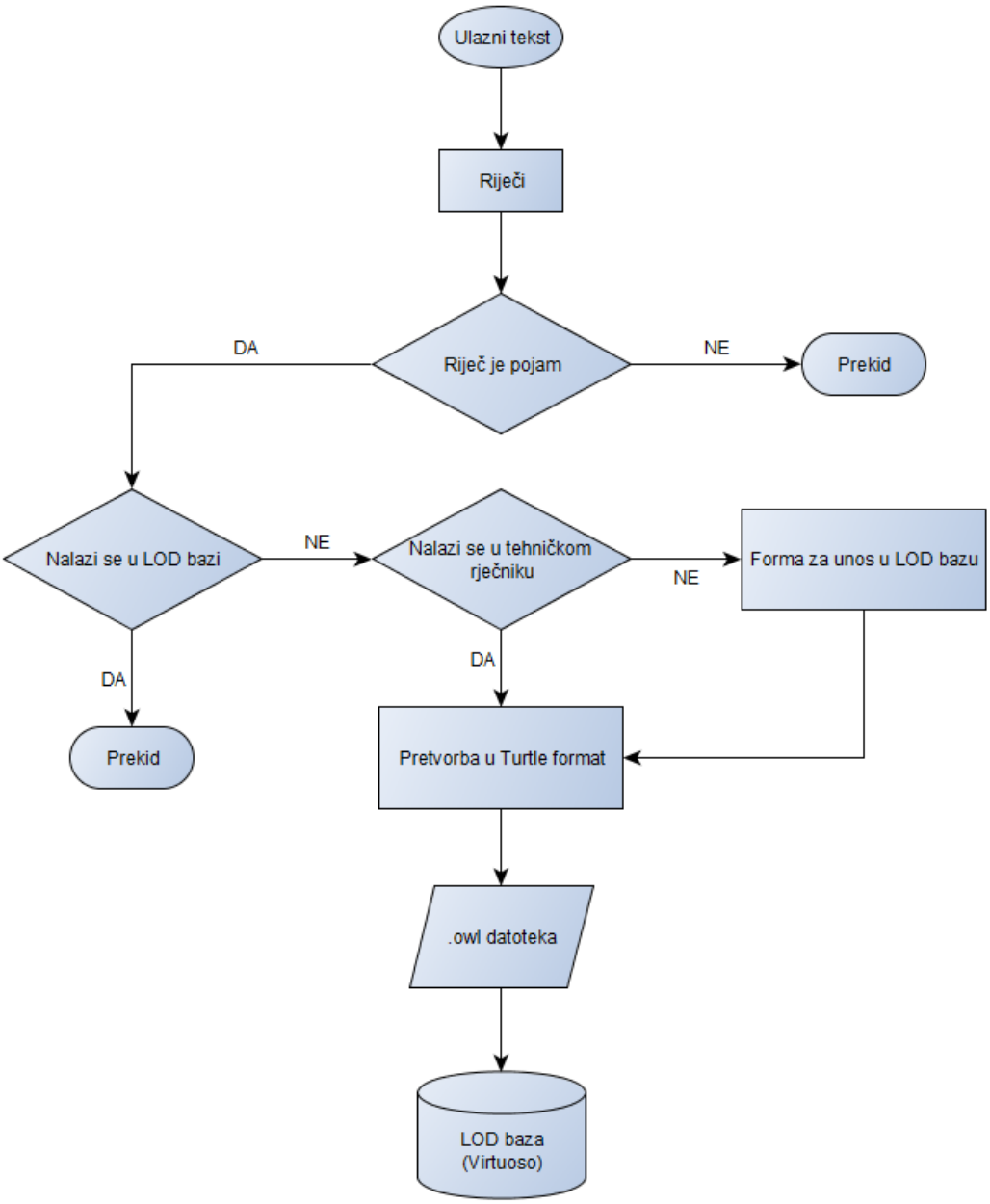
Ovdje će biti prikazana i objašnjena web stranica koja je izrađena u *Web2py Framework*-u. Slika 20. prikazuje naslovnu stranicu:

Slika 20: Naslovna stranica

Zamišljeni algoritam za obradu tehničkih tekstova sastojao bi se od sljedećeg:

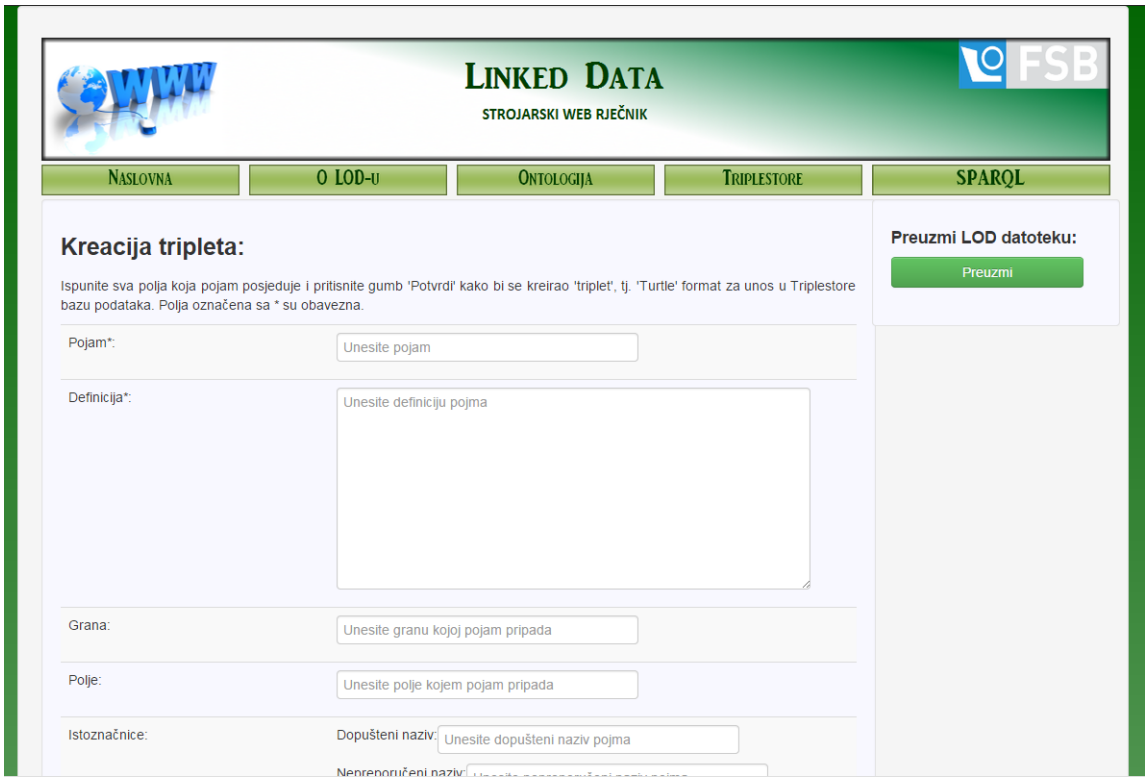
1. Najprije bi se iz teksta izvukle sve riječi uz pomoć regularnih izraza.
2. Zatim bi se iz tih riječi izdvojile sve one koje su pojam, tj. koje imaju neko značenje (nisu veznici, itd.).
3. Nakon toga, sljedila bi provjera da li određeni pojam već pripada *triplestore* bazi, te ukoliko pripada, došlo bi do prekida programa.
4. Ako se ustanovilo da pojam ne pripada *triplestore* bazi, sljedila bi provjera pripadnosti pojma tehničkom rječniku. Ukoliko pojam pripada rječniku, vršila bi se pretvorba svih podataka koji su za njega u tehničkom rječniku pronađeni u "Turtle" format. Ako pak ne pripada, prikazala bi se forma za unos podataka u *triplestore* bazu, te bi se nakon njenog popunjavanja također vršila pretvorba u "Turtle" format.
5. Na kraju, kreirani tripleti bili bi ubačeni u datoteku pod nazivom "Turtle.owl", koju bi se dalje moglo učitati iz bilo koje baze koja ima mogućnost čitanja "Turtle" zapisa. U ovom slučaju, to bi bila *triplestore* baza podataka koju posjeduje *OpenLink Virtuoso* univerzalni server.

Slijedi grafički prikaz gore navedenog algoritma:



Slika 21: Principna shema algoritma za obradu tehničkih tekstova

Unutar *Triplestore* djela stranice nalaze se: forma za kreiranje tripleta iz podataka unesenih u pojedina polja, gumb za preuzimanje "Turtle.owl" datoteke, koja sa svojim definiranim trojcima predstavlja ulazni podatak za *triplestore* bazu podataka, te dio pod nazivom "Obrada tehničkih tekstova", koji sadrži tekstualno polje za obradu unešenih tekstova (može sadržavati večiku količinu podataka, tj. pojmova). Izgled *Triplestore* djela stranice prikazuju slike 22 i 23:

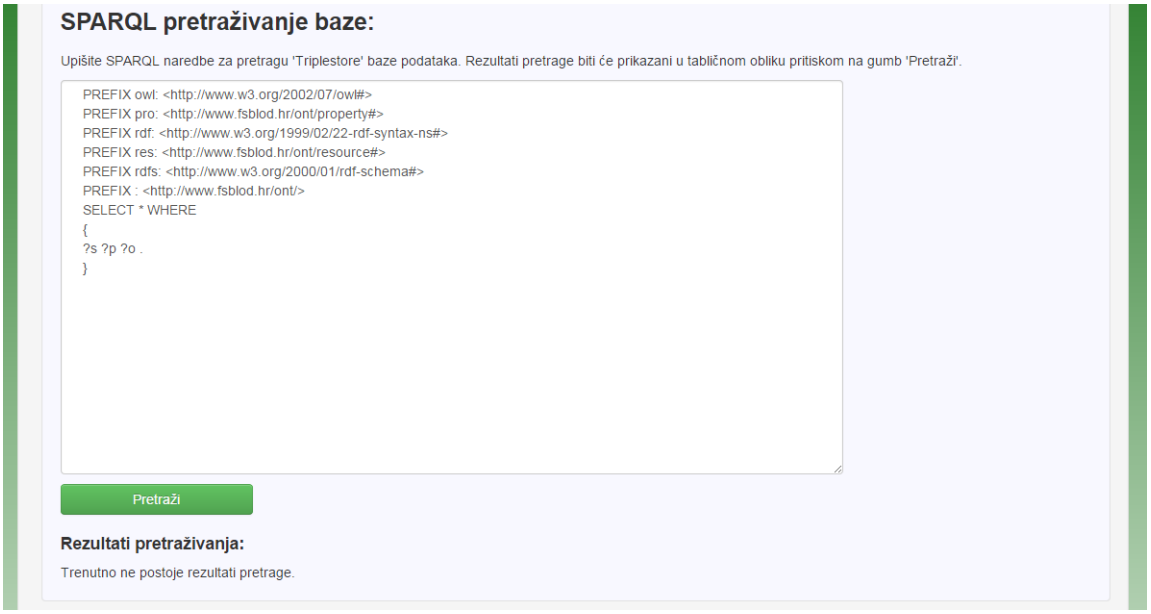


Slika 22: Triplestore dio stranice: forma za kreiranje trojca

Slika 23: Triplestore dio stranice: obrada tehničkih tekstova

SPARQL dio stranice sadrži formu koja se sastoji od tekstualnog polja za unos naredbi za pretraživanje, i gumba "Pretraži" s kojim pretraga započinje. Ispod te

forme pod djelom "Rezultati pretrage", ispisuju se rezultati pretrage u tabličnom obliku:



Slika 24: SPARQL dio stranice

7 Zaključak

Iako su semantički web i povezani podaci kao ideja već dugo poznati, tek zadnjih godina se pokazala kao stvarna i nezaobilazna potreba. Zbog širenja interneta i rasta broja informacija na njemu, sve teže je filtrirati one potrebne. Osim toga, proizvode se sve viđe uređaja čija se funkcionalnost oslanja na internet poput pametnih telefona, tableta, prijenosnih računala, autiju, pa čak i hladnjaka. U cijeloj toj mreži, brzina dolaženja do podataka i njihova kvaliteta postaje vrlo bitan faktor.

Povezani podaci osim što ljudima olakšavaju upotrebu interneta, također imaju vrlo kvalitetne temelje za algoritme i strojeve koji koriste umjetnu inteligenciju. Upravo semantika kod povezanih podataka olakšava algoritmima kod kvalitetnog zaključavanja.

Semantički web ne mijenja strukturu današnjeg interneta, već on upravo postaje struktura interneta. U web 3.0 standardu i dalje će se koristiti standardne web tehnologije, ali internet svojim širenjem danas ostavlja mnogo praznog i neispokrištenog mjesta, koje će povezani podaci ispuniti dodajući smisao između tog velikog broja nepovezanih podataka.

8 Literatura

Felix López, Victor Romero: "Mastering Python Regular Expressions"
Marko Cundeković: "Povezani podaci u novom ustroju Interneta"
http://docs.python.org/2/library/re.html
http://docs.python.org/2/howto/regex.html
www.zemris.fer.hr/predmeti/krep/Pavic.pdf
http://en.wikipedia.org/wiki/Semantic_Web
http://www.w3.org/standards/techs/rdf#w3c_all
http://www.w3.org/standards/techs/linkeddata#w3c_all
http://intranet.fesb.hr/Portals/0/docs/nastava/kvalifikacijski/Karmen%
20Klarin%20KDI%20v14.pdf
http://en.wikipedia.org/wiki/Web_Ontology_Language
http://protege.stanford.edu/
http://linkeddata.org/
http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/