Univerza v Ljubljani

Filozofska fakulteta

Oddelek za slovenistiko

Robert Jakomin

Algoritem za analizo kratke povedi v slovenščini

Diplomsko delo

Mentor: doc. dr. Primož Jakopin

Ljubljana, september 2009

Zahvala

Najprej se želim zahvaliti svojemu mentorju doc. dr. Primožu Jakopinu za odlično idejo diplomskega dela, napotke in pomoč pri računalniški pretvorbi in morfološki analizi korpusa povedi za analizo. Posebna zahvala gre tudi doc. dr. Matiji Maroltu iz Laboratorija za grafiko in multimedije Fakultete za računalništvo in informatiko, ki me je uvedel v profesionalno delovno okolje v laboratoriju, in raziskovalcema Cirilu Bohaku ter Jerneju Južni – oba sta mi pomagala z učinkovitimi nasveti za programsko realizacijo diplomskega dela. Zahvaljujem se tudi doc. dr. Andreji Žele za nekaj konstruktivnih pogovorov in elektronsko različico njenega vezljivostnega slovarja. Hvaležen sem tudi raziskovalcema dr. Domnu Marinčiču in Roku Piltaverju z Inštituta Jožef Stefan, ki sta mi dala nekaj dobrih idej za uspešnejšo implementacijo algoritma.

Posebna zahvala gre tudi mnogim vnetim računalničarjem, ki na spletu objavljajo brezplačne učbenike, programe, primere in odgovore na najpogostejše težave javanskih programerjev – brez njih tega diplomskega dela zagotovo ne bi bilo.

Zahvaljujem se tudi svojim staršem in Martini za podporo in razumevanje.

Kazalo

1. Uvod

Računalništvo v zadnjih desetletjih uspešno prodira v vse segmente družbe in znanosti, tudi jezikoslovje že dolgo ni več izjema. Na prvi pogled se stroki morda ne zdita tako povezani, vendar je računalništvo kompleksna stroka, na katero lahko gledamo in jo uspešno razlagamo z različnih zornih kotov z zelo raznolikimi modeli, od konkretnega fizikalno-kemijskega in elektrotehniškega prek abstrahiziranega računalniškoarhitekturnega pa vse do formalnih matematičnega in jezikoslovnega. Vsi ti modeli niso povsem alternativni drug drugemu: koncept računalniškega programa je navsezadnje neodvisen od elektronske realizacije. V svojem bistvu gre za program, to je za z navodili podan postopek, ki bi ga lahko izvajal kakršenkoli avtomat, elektronski, elektromehanski ali celo povsem mehanski – lahko bi bil navsezadnje tudi stisnjen zrak. Z višanjem stopnje abstrakcije programskega jezika se dosega večjo neodvisnost od konkretne tehnologije implementacije strojne pa tudi nižjeležeče programske opreme, kot je operacijski sistem. Na nivoju višjih programskih jezikov, tj. na nivoju t. i. tretje[1] in četrte[2] generacije, programski jeziki na nek način zelo počasi napredujejo k bistveno kompleksnejšim in formalno nezajemljivim naravnim jezikom.

Programski del računalništva tako kaže jasne vzporednice tudi z jezikoslovjem, ne le z diskretno oziroma formalno matematiko; bilo bi nenavadno, če se ne bi stroki uspešno povezovali. Računalništvo in računalniško podprto jezikoslovje se je doslej že uspešno uveljavilo na mnogih področjih jezikoslovja, še posebej na področju splošnega in sinhronega. Na področju zadnjega se poleg fonetičnih analiz uspešno uporablja tudi na področju skladnje – pri preučevanju množice računalniško obdelanih besedil, t. i. korpusov. Korpusi ponujajo hitro iskanje besednih zvez po ogromnih zbirkah besedil v rangu nekaj 100 milijonov besed – obseg, ki je bil še pred nekaj desetletji za jezikoslovca povsem nepredstavljiv in neobvladljiv. Korpusi so zelo učinkovito sredstvo zlasti za preučevanje morfologije, skladnje, frazeologije in besediloslovja, so

[1] Sem štejemo pretežno imperativne programske jezike, ki so nastajali od 50. let 20. stoletja dalje, npr.: Fortran, C/C++, Basic, Pascal, Java, C# itd.

[2] Sem štejemo jezike, ki so nastajali od 70. let 20. stoletja dalje, to so npr.transformacijski jezik za poizvedbe v bazah podatkov SQL (z množico dialektov), statistični R, matematični MATLAB itd.

nepogrešljiv pripomoček za leksikologijo, saj poleg pojavnosti ponujajo tudi informacijo o številu pojavitev posamezne fraze. Problem korpusov pa so pogosto uporabniku neprijazni uporabniški vmesniki (še posebej za uporabnika, ki ni vešč programiranja oz. vsaj sestavljanja regularnih izrazov) in morfološko-skladenjska neoznačenost besedila. Oba problema sta za sintetične jezike, kot je tudi slovenščina, še posebno pereč problem – pri analitičnih jezikih, kot je npr. angleščina in vsaj deloma tudi ostali germanski jeziki, to ni taka težava, saj je temelj informacije o skladenjski organiziranosti vsebovan v besednem redu, ne pa v morfoloških elementih, kot so končnice.

Za slovenščino sicer že obstaja nekaj morfoloških označevalnikov (npr. JAKOPIN 2001; označevalnik podjetja Amebis, ki pa ni javno dostopen), toda njihova težava je, da za večino pojavitev besed v besedilu obstaja več možnosti – dokončen odgovor na to, katera je prava, pa bi dali šele skladenjska, pomenska in sobesedilna analiza. Ker je skladenjska analiza za slovenščino šele v razvoju (MARINČIČ 2008; Amebis – projekt ni javno dostopen) in me omenjena tematika zaradi interdisciplinarne povezave računalništva (programiranja) in jezikoslovja še posebej zanima, sem se z veseljem lotil algoritma za skladenjsko analizo kratke povedi v slovenščini (samo do treh besed), ki bi se ga kasneje eventualno dalo razširiti tudi na daljše povedi, poleg algoritma pa bi se že v okviru diplomskega dela izdelala tudi konkretna implementacija v programskem jeziku z naprednejšim, a čim bolj enostavnim iskalnikom.

2. Razvoj in implementacija

2.1. Izbira programskega jezika in tehnologije

Na začetku sem imel na voljo v tekstni datoteki korpus z nekaj več kot 57.000 povedmi, dolžine do treh besed, iz dnevnika DELO za leto 2007 (vseh povedi je bilo 1,040.000). Enobesednih povedi je bilo 13, dvobesednih povedi 872, povedi dolžine treh besed pa 56275. Korpus povedi je pridobljen iz besedilnega korpusa Nova beseda. V drugi tekstni datoteki pa so z oblikoslovnim označevalnikom Primoža Jakopina označene pojavitve oblik besed iz tega korpusa (JAKOPIN 2001). Namen diplomskega dela je bilo stavčno analizirati povedi iz korpusa in rezultat shraniti, zatorej je bila uspešnost algoritma pomembnejša od hitrosti izvajanja. Pozneje bi bila zaželena tudi možnost analize povedi, vnesenih s strani uporabnika. Stavčni analizator naj bi bil javno dostopen, najbolje prek spleta, uporabnik bi do analiziranih besed dostopal prek filtrov, realiziranih z grafičnim vmesnikom. Iskanje po korpusu z neposrednim vnosom zmogljivih posebnih (regularnih) izrazov je za ciljnega uporabnika, ki ni računalničar kaj šele programer, precej težko, vsak iskalnik ima pogosto svojo sintakso, tudi sintaksa razmeroma univerzalnih regularnih izrazov iz okolij UNIX ni vedno lahka, še posebej za pričakovane kompleksnejše poizvedbe po analiziranem korpusu. Enostavnejše poizvedbe, za katere lahko uporabimo nadomestne znake (npr. znaka '*' ali '?'), ki jih uporabniki večinoma dobro poznajo, pa niso dovolj zmogljive za želene kombinacije iskalnih pogojev.

Na podlagi vseh zahtev sem se odločil za programsko tehnologijo Java, saj je odlično dokumentirana, prosto dostopna, vključno z dvema dobrima razvojnima okoljema[3], omogoča prenosljivost med operacijskimi sistemi[4], je objektno orientirana[5], vključuje tudi dobri programski knjižnici za delo z grafičnim vmesnikom (AWT in Swing), knjižnice za povezavo z bazami podatkov in zmogljivo implementacijo regularnih izrazov sistemov UNIX; je razmeroma

[3] Netbeans (http://www.netbeans.org/) in Eclipse (http://www.eclipse.org/).

[4] Javanski programi ne tečejo neposredno na operacijskem sistemu uporabnika, temveč na posebnem navideznem (programsko realiziranem) stroju.

[5] Objektno orientiran pristop je danes najpogostejši pristop pri programiranju – nekoliko večjo porabo pomnilnika odtehta lažja berljivost izvorne kode, posledično lažje vzdrževanje programov ter večja podobnost programov oz. programskih objektov z objekti realnega sveta, ki ga program modelira.

hitra (dinamično prevajanje omogoča kombinirano interpretativno-prevedeno izvajanje), poleg tega pa je enostavna za uporabo na spletu v obliki posebnih spletnih programov, apletov, ki tečejo na strani odjemalca, tako da pisanje posebne spletne strani ni potrebno, poenostavi pa tudi morebitno kasnejše dodajanje vmesnika za sprotno analizo od uporabnika vnesenih povedi – analiza lahko tako poteka na odjemalčevem računalniku brez obremenjevanja in čakanja na strežnik. Java je sicer počasnejša od programskega jezika C/C++, nekoliko tudi od konkurenčnih C#/VB.NET, ki sicer prav tako tečeteta na podobnem navideznerm stroju kot Java (CLR), Java zasede tudi več pomnilniškega prostora, kar pa je pa je danes vse manjši problem. Menim, da vse omenjene slabosti odtehta javanska prenosljivost, odprtokodnost in razširjenost njene uporabe.

Glede hrambe analiziranih podatkov sem se odločal med tremi možnostmi: shranjevanje v binarni ali tekstovni datoteki, shranjevanje v XML-datoteki in shranjevanje v bazi podatkov. Prva možnost je razmeroma počasna za iskanje, pohitritev oz. preoblikovanje v bazi podatkov podobno obliko bi zahtevalo preveč dela, enako iskanje. Druga možnost, XML, je danim podatkom (tj. analiziranim povedim) najprimernejša, saj omogoča preprost vnos hierarhije po vozliščih XML-drevesa in ne potrebuje dodatnega programa oziroma sistema za upravljanje baze, je pa razmeroma počasna pri iskanju – podatki se hranijo v tekstovni, in ne binarni obliki, poleg tega za Javo še ne obstaja vgrajena privzeta implementacija poizvedbenega XML-jezika XQuery[6], v Javo vgrajeni XPath pa ne omogoča dovolj možnosti za iskanje. Tako sem se odločil za podatkovno bazo SQL, in sicer za sistem za upravljanje podatkovnih baz MySQL, ki je prosto dostopen, razmeroma hiter in dobro dokumentiran. Hierarhija je realizirana prek tujih ključev s t. i. relacijami »eden-do-mnogih« (angl. one-to-many relation).

Odločil sem za uporabo angleških imen spremenljivk, delov programa in komentarjev, saj je to običajna praksa profesionalnega programiranja – tako kodo je možno enostavneje izmenjati s programerji v tujini, enostavneje je poslati del izvorne kode programa za morebitno tehnično pomoč pri programiranju: programer se namreč z nerazumljivimi imeni spremenljivk precej težje poglobi v tujo kodo; za uporabnika pa to ne predstavlja nobene težave: obvestila programa in grafični vmesnik so v slovenščini.

[6] Obstajajo pa delno prosto dostopne, kot je Saxon: http://saxon.sourceforge.net/.

2.2. Zgradba programa

Program lahko razdelimo na štiri glavne dele:

1. del za analizo povedi:
    - **Category.java** – razred, namenjen za tvorbo objekta, ki vsebuje podatke o kategorijah posameznega objekta Lemma; razred se zaradi poenostavitve pri programiranju ne deduje, vsi objekti besednih vrst imajo skupne kategorije, neobstoječe za posamezno besedno vrsto imajo vrednost »null« (v obliki za bazo MySQL); vsebuje tudi konstante tipov povedi, stavkov in stavčnih členov ter vzorce regularnih izrazov za iskanje po besednih vrstah in kategorijah;
    - **CmpSentence.java** – razred, namenjen za analizo zloženih povedi: vsaka poved vsebuje več objektov Element (stavkov) in seznam vseh besed povedi (objekti Word);
    - **Corpus.java** – razred, namenjen za razčlenitev korpusa iz tekstovne datoteke: vsebuje objekt Dictionary;
    - **Dictionary.java** – razred, namenjen za razčlenitev slovarja iz tekstovne datoteke: vsebuje razpršeno tabelo, v kateri je shranjen slovar vseh oblik besed (ključi so oblike besed, vrednosti pa seznami objektov Lemma, tj. vse možnosti analize posamezne oblike);
    - **Element.java** – razred, namenjen za tvorbo objektov podenot znotraj povedi (stavki), znotraj stavkov (stavčni členi) in znotraj stavčnih členov (deli stavčnih členov); vsebuje niza za označitev tipa in podtipa ter razpršeno tabelo podenot;
    - **Lemma.java** – razred, namenjen za tvorbo objektov, ki vsebujejo kategorije in osnovno obliko[7] posamezne oblike besede; vsaka instanca (objekt) predstavlja eno možnost analize posamezne oblike;

[7] Osnovna besedna oblika je za samostalniško besedo imenovalnik ednine, za pridevniško imenovalnik moškega spola ednine (osnovna stopnja), za glagole nedoločnik, za prislove pa osnovna stopnja pri stopnjevanju.

- **Word.java** – razred, namenjen za označitev posamezne besede: vsebuje niza označitev tipa besede in tipa stavčnega člena, ki mu beseda pripada;

2. grafični uporabniški vmesnik:
    - **GUIApplet.java** – razred, ki vsebuje grafični vmesnik (panel) za spletno aplikacijo – aplet;
    - **GUIApplication.java** – razred, ki vsebuje grafični vmesnik (panel) za namizno aplikacijo;
    - **SAApplet.java** – razred za zagon apleta;
    - **SAApplication.java** – razred za zagon namizne aplikacije;
3. vmesnik za shranjevanje podatkov v bazo in poizvedbe:
    - **DBConnection.java** – razred, ki vsebuje in sestavlja SQL-stavke za poizvedbe in shranjevanje podatkov v bazo;
    - **DBRequest.java** – razred za prenos podatkov, ki jih posreduje aplet servletu, da ta opravi poizvedbo v bazi;
    - **DBServletConnection.java** – razred za komunikacijo med apletom in servletom;
    - **SAServlet.java** – razred za strežniško aplikacijo – servlet (potreben za aplet), ki komunicira z bazo (prek DBConnection.java).
4. tabele v bazi podatkov MySQL:
    - **cmpsentences** – tabela zloženih povedi;
    - **dictionary** – tabela možnih kombinacij kategorij;
    - **sentences** – tabela vseh stavkov – podenot zložene povedi;
    - **words** – tabela vseh besed: vsaka beseda predstavlja svojo vrstico; z ostalimi tabelami je povezana s tujimi ključi: cmpSentID (iz tabele *cmpsentences*), sentID (iz tabele *sentence*) in dictID (iz tabele *dictionary*).

2.3. Opis poteka programa

Ob zagonu samostojne aplikacije se najprej v pomnilniku ustvari slovar besednih oblik, implementiran je z razpršeno tabelo (Hashtable) za učinkovito iskanje. Ključ v tabeli je niz –

besedna oblika, vrednost pa seznam objektov Lemma – vse možnosti analize posamezne oblike. Podatke o oblikah se prebere z razčlenjevanje nizov v datoteki z oznakami oblik »besede_z_oznakami.txt«, ki vsebuje analize v obliki izpisa oblikoslovnega označevalnika Inštituta Frana Ramovša ZRC SAZU (gl. vir JAKOPIN 2001). Po konstrukciji slovarja se začne gradnja korpusa v pomnilniku. Zaradi precejšne obsežnosti podatkov se v pomnilniku ne hrani podatkov za vse povedi – hrani se le podatke za posamezno poved, ki se v nekem trenutku analizira oz. shranjuje v bazo podatkov. Povedi se razčlenjuje iz datoteke »korpus.txt«, za vsako besedo posamezne povedi se konstruira objekt Word; ti se hranijo v seznamu tipa ArrayList. Za vsako besedo se v slovarju oblik poišče ustrezne možnosti, na tej stopnji pa program še ne ve, katere bi lahko bile prave.

Vsako poved se analizira sproti (podroben opis algoritma za stavčno analizo je podan v naslednjem poglavju). Po analizi posamezne povedi in enoznačni določitvi besednih vrst in stavčnih členov besed se poved zapiše v bazo. Zapis v bazo je realiziran prek zapisa v tekstovno datoteko, ki po koncu analize vseh povedi zapiše oz. prevede v bazo podatkov MySQL (MySQL-stavek »LOAD DATA INFILE«). Ta način je precej hitrejši od sprotnega shranjevanja vsake povedi posebej – razlog za zahtevo po večji hitrosti shranjevanja je v številnih testiranjih in preverjanjih algoritma. Pred zapisom analiziranih povedi v bazo se stare podatke iz baze izbriše.

Po koncu analize se odpre grafični vmesnik, v katerem lahko uporabnik izbere želene kriterije za iskanje in izpis analiziranih povedi (podrobnejši opis je v 4. poglavju). Pri zagonu apleta se na začetku povedi ne analizirajo ponovno, naloži se le slovar (zgolj v informacijo uporabniku, katere besedne oblike program sploh prepozna), grafični vmesnik pa je enak. Ko uporabnik klikne na gumb »išči«, se pošlje poizvedba v bazo, ki nato vrne ustrezne vrstice iz tabele *words* in ostalih s stiki povezanih tabel (v obliki povezanega seznama ResultSet) – pri samostojni aplikaciji se to izvrši neposredno, pri apletu pa se najprej serializira tabela kriterijev, natančneje podatkovni seznam (tipa Vector) v modelu tabele, ki se nato pošlje strežniški aplikaciji – servletu, ta pa pošlje poizvedbo bazi. Vrnjen povezani seznam zadetkov (ResultSet) servlet prepiše v model za grafično kontrolo Jlist in ga serializiranega pošlje apletu. Razlog za posredniško aplikacijo – servlet – je v varnosti, saj je zaradi številnih vdorov v baze podatkov praksa sistemskih administratorjev, da vrat za dostop do baze podatkov ne odpirajo navzven, tako da neposreden dostop apleta do baze prek spleta ni možen.

Vrstice v seznamu povedi so sestavljene iz posameznih vrstic tabele *words* – ob kliku na posamezno vrstico se sproži iskanje v bazi glede na identifikacijsko številko povedi: program po kliku prikaže podrobnosti o posamezni povedi. Stavčni členi so v seznamu povedi označeni z več vrstami oklepajev: najprej sem nameraval uporabiti HTML-jevsko označevanje z barvami, vendar je razčlenjevanje oznak povsem prepočasno za normalno delo s korpusom povedi takega obsega. Da se izpisi še pospešijo, sem uporabil tudi konkatenacijo oklepajev in nizov besed v bazi podatkov (v stavkih *SELECT*), javanska je bila kljub uporabi StringBufferjev počasnejša (problem je verjetno veliko število samostojnih nizov).

2.4. Hramba analiziranih povedi v bazi podatkov

Podatki analiziranih povedi so shranjeni v štirih tabelah: *cmpsentences*, *sentences*, *words* in *dictionary*. Vsaka beseda predstavlja svojo vrstico v tabeli *words*, besede določene povedi so povezane s tujim ključem iz tabele *cmpsentences*. V tabeli ***cmpsentences*** so naslednji atributi (stolpci): **cmpSentID** – identifikacijska številka posamezne povedi; **length** – dolžina posamezne besede (v veliki večini primerov je to 3, obstaja nekaj izjem dolžine 2 in 1 zaradi napak pri izbiranju povedi iz korpusa oziroma napačne členitve nizov pri ločilih); **frequency** – frekvenca posamezne povedi, podatek je pridobljen iz korpusa Nova beseda; **type** – tip povedi, možni so trije: glagolska (G), neglagolska samostalniška (nG) in neglagolska nesamostalniška (nnG); **noOfVerbSent** – število glagolskih stavkov v povedi; **structure** – struktura povedi (atribut na trenutni stopnji razvoja algoritma še ni v uporabi).

V tabeli ***sentences*** so atributi: **sentID** – identifikacijska številka posameznega stavka; **type** – vrsta stavka, možnosti so: glagolski (G), samostalniški pastavek (PS), drugi pastavki (PO); če je stavek zanikan, ima na koncu znak '-', npr. 'PS-'; **relation** – razmerje do glavnega stavka: priredno, podredno ali soredno (atribut na trenutni stopnji razvoja algoritma še ni v uporabi).

V tabeli ***words*** so atributi: **id –** identifikacijska številka besede, **position** – vrstni red besede v stavku; **frontPunct** – ločila pred besedo (npr. narekovaj), **word** – beseda; **rearPunct** – ločila za besedo (npr. vejica, pika); **cmpSentID** – tuji ključ povedi (iz tabele *cmpsentences*), v katero spada beseda; ***sentID*** – tuji ključ stavka (iz tabele *sentences*), v katerega spada beseda; a**rtType** –

vrsta stavčnega člena; **neutralForm** – osnovna besedna oblika; **dictID** – tuji ključ kategorij besede oz. besedne oblike (iz tabele *dictionary*).

id	position	frontPunct	word	rearPunct	cmpSentID	sentID	artType	neutralForm	dictID
31	0		Abotna		11	11	S	aboten	35
32	1		zmaga		11	11	P	zmagati	79
33	2		denarja	.	11	11	O	denar	14
34	0		Abramovič		12	12	*NULL*	*NULL*	*NULL*
35	1		kupuje		12	12	P	kupovati	79
36	2		Beckhama	?	12	12	*NULL*	*NULL*	*NULL*
37	0		Absolutiziranje		13	13	S	absolutiziranje	41
38	1		rojeva		13	13	P	rojevati	79
39	2		malike	.	13	13	O	malik	71
40	0		Absolutne		14	14	S	absoluten	19
41	1		garancije		14	14	*NULL*	garancija	7
42	2		ni	.	14	14	P	ne biti o	966
43	0		Absolutne		15	15	S	absoluten	19
44	1		varnosti		15	15	S	varnost	5
45	2		ni	.	15	15	P	ne biti o	966

Slika 1: Zapis povedi v bazi podatkov MySQL (tabela *words*)

V tabeli ***dictionary*** pa so atributi: **dictID** – identifikacijska številka posamezne kombinacije kategorij; **wordType** – besedna vrsta; **gender –** spol; **grammNo** – število; **grammCase** – sklon; **person** – oseba; **comparison** –primerjalna stopnja; **definiteness –** določnost; **antecedentGrammNo** – število reference zaimka; **antecedentGender** – spol reference zaimka; **relation** – razmerje predloga.

dictID	wordType	gender	grammNo	grammCase	person	comparison	definiteness	antecedentGrammNo	antecedentGender	relation
1208	ZO	m	e	1	c	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1209	ZK	ž	e	1	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1210	ZO	ž	e	1	c	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1211	ZO	ž	e	1	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1219	PL	m	e	6	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1220	ZK	m	e	1	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1221	ZK	m	e	4	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1224	ZO	m	e	1	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1248	PČ	s	e	6	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1282	PČ	m	e	6	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1435	ZSVP	m	e	6	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1482	ZO	*NULL*	e	1	b	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1490	ŠM	m	e	4	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*
1491	ZSV	m	e	1	b	*NULL*	*NULL*	e	*NULL*	*NULL*
1492	ZSV	m	e	4	b	*NULL*	*NULL*	e	*NULL*	*NULL*

Slika 2: Zapis možnih morfoloških oznak v bazi podatkov MySQL (tabela *dictionary*)

3. Analiza algoritma za stavčno analizo

3.1. Opis zasnove

Algoritem je osredotočen na stavčno analizo glagolske povedi in ne upošteva konteksta povedi, deluje na podlagi slovničnih lastnosti (kategorij) posameznih besed. Za uspešnost njegovega delovanja je ključen čim boljši oblikoslovni označevalnik. Uporabljen je bil oblikoslovni označevalnik Primoža Jakopina (JAKOPIN 2001), ki se je dobro obnesel, slabši je bil le pri prepoznavi lastnih imen. Temeljna besedna vrsta za delovanje algoritma je glagol oz. glagolska besedna zveza, na podlagi katere algoritem identificira stavčne meje in ostale stavčne člene, temeljna skladenjska lastnost za delovanje pa je ujemanje med osebkom in povedkom ter med jedrom in prilastki. Algoritem je bil zasnovan z namenom nadaljnjega razvoja in uporabe na daljših povedi. Analiza daljših povedi je sicer možna že na trenutni stopnji razvoja, ni pa še bila preizkušena.

3.2. Psevdokoda algoritma za analizo povedi

Eden izmed najbolj učinkovitih načinov za predstavitev algoritma je zapis v psevdokodi. Psevdokoda je način, kako predstaviti algoritem, ki sicer upošteva semantiko, ni pa nujno sintaktično pravilna in je namenjena izključno razumevanju delovanja algoritma. Ni omejena na določen programski jezik, napisana mora biti tako, da je na njeni podlagi mogoča implementacija v kateremkoli programskem jeziku. V nadaljevanju je v psevdokodi zapisan glavni del algoritma, za lažje razumevanje je psevdokoda v slovenščini (imena v programu so v angleščini), tudi struktura objektov je nekoliko poenostavljena.

```
var[] povedi = razcleniDatotekoKorpusaNaPovedi();

foreach(poved : povedi){

        if(poved.vsebujeGlagol()){
                poved.tip = Tip.glagolska;
                poved.poisciStavcneMeje();
                poved.analizirajStavke();

        }
        else{
                 if(poved.vsebujeSamostalnik())
                         poved.tip = Tip.samostalniskaNeglagolska;
```

```
                else
                        poved.tip = Tip.nesamostalniskaNeglagolska;

                poved.analizirajNeglagPoved();
        }
}


class Poved{

var tip;
var[] besede;
var[] stavki;

        poisci_stavcne_meje(){
                var stavek = this.novStavek();

                foreach(beseda : besedePovedi){
                        if(beseda.jeNikalniClen()){
                                this.tip = Tip.glagolskaZanikana;
                                beseda.stavcniClen = Clen.povedek;
                        }
                        else if(beseda.jeGlagol()){
                                if(!stavek.preveriCeLahkoDelPovedka(beseda)){
                                        poisciStavcnoMejoOdDo(mejaNazadnjeDolocenegaStavka, beseda);

                                        if(stavcnaMejaNajdenaDoNovegaGlagola())
                                                stavek = this.novStavek();
                                        else //algoritem ne zna najti konca stavka pred novim glagolom
                                                stavek.odstraniPrejsnjiGlagol(); //verjetno gre za drugo b. vrsto
                                }
                                else{
                                        stavek.dodajGlagolVPovedek(beseda);
                                }
                        }
                }
                poisciStavcnoMejoOdDo(mejaPrejsnjegaStavka, zadnjaBesedaPovedi());
        }

        analizirajStavke(){

                foreach(stavek : stavkiZlozenePovedi){
                        stavek.poisiciSamostInPridBesedneZveze();
                        stavek.poisciPrislDol();
                }
        }

        poisiciSamostInPridBesedneZveze(){

                stavek.posiciOsebek();
                stavek.poisciPovDolocilo();

                if(!stavek.jeOsebekNajden()){
                        if(stavek.!jePovDolociloNajdeno() && stavek.vsebujeLeGlagolBitiObstajanja())
                                stavek.poisciNegiranOsebek();
                        else
```

```
                    stavek.poisciPosamostaljenOsebek();
                }

                if(!stavek.ceVStavkuLeGlagolBitiObstajanja()){
                    stavek.poisciPredmete();
                    stavek.poisciPosamostaljenjePredmete();
                }
            }
}


class Stavek{
        poisciOsebek(){
            for(i = 0; i < besedeStavka.length; i++){
                beseda = besedeStavka.dobiBesedo(i);

                if(beseda.jeZeAnalizirana() || beseda.moznostiOznacitve.length == 0)
                    continue;
                else{
                    foreach(moznaOblikOznaka : beseda.dobiMozneOblikOznake()){
                        if(moznaOblikOznaka.jeLahkoTipa(Tip.predlog)){
                            i = poisciKonecPredlozneZveze();
                            nadaljujZunanjoZanko();
                        }
                        else if(moznaOblikOznaka.jeLahkoTipa(Tip.osebek)){
                            if(beseda.seUjemaZ(povedek, moznaOblikOznaka)){
                                beseda.tip = Tip.osebek;
                                najdiPrilastke(i, moznaOblikOznaka);
                                return;
                            }
                        }
                    }

                }
            }
        }
        […]
}
```

Na podoben način kot metoda *poisciOsebek()* so realizirane tudi ostale metode za iskanje stavčnih členov (pri predmetih in prislovnih določilih se seveda ob prvem najdenem primeru metoda ne konča, kot se pri osebku in pri povedkovem določilu); natančna implementacija je v poglavju Izvorna koda. Metoda *analizirajNeglagPoved()* samo za vsako besedo predpostavi, da je prava oblika kar prva od možnih, stavkov ali naštevanj ne išče.

3.3. Časovna zahtevnost algoritma

Časovna zahtevnost algoritma je reda O(n) oz. natančneje f(x) = k * n = Θ(n), pri čemer je *n* število besed v povedi, *k* pa konstantno število iteracij pri analizi besed določene povedi[8]. Z naraščanjem števila besed in kompleksnosti povedi se število iteracij ne poveča.

3.4. Omejitve algoritma

Algoritem je zelo odvisen od oblikoslovnega označevalnika: če oblikoslovni označevalnik neke besede ne prepozna, npr. lastnega imena, je tudi algoritem ne more uspešno vključiti v analizo. Problem predstavljajo tudi vse možnosti za neko besedno obliko: pri nekaterih uporabljeni oblikoslovni označevalnik ne prepozna vseh možnosti, zaradi česar je analiza napačna (npr. poved »Absolutne garancije ni«). Poleg tega oblikoslovni označevalniki navadno prepoznajo le pravopisno in slovnično pravilno zapisane besede knjižnega jezika, algoritem pa je poleg tega razmeroma odvisen tudi od pravilno postavljenih ločil v besedilu – za raziskave korpusov neknjižnih besedil tako ni primeren. Algoritem slabo prepoznava tudi vrinjene neglagolske stavke, ne prepozna pa tudi naštevanj, saj jih v trobesednih glagolskih povedih skoraj ni. Pri nadaljnjem razvoju in uporabi algoritma za daljše povedi bi bilo tako nujno v algoritem vključiti dognanja Domna Marinčiča pri njegovih raziskavah strojnega razčlenjevanja besedila (MARINČIČ 2008). Algoritem nadalje zaradi neupoštevanja konteksta povedi ne more povsem natančno ločiti med osebkom in povedkovim določilom – predpostavi se, da je osebek prvi stavčni člen, povedkovo določilo pa mu sledi (kar pa ni vedno res zaradi prostega besednega reda v slovenščini, členitve po aktualnosti).

Algoritem za vsako poved določi le eno možnost analize oz. interpretacije, in sicer prvo, do katere pride; to sicer ni večji problem za nadaljnji razvoj, saj bi se dalo formalno enako poved večkrat vnesti v bazo podatkov, vsakič z drugačno analizo oz. interpretacijo; več analiz pa bi

[8] Notacija t. i. *velikega O* se na področju informatike in matematike uporablja za označevanje asimptotske tendence (hitrosti naraščanja) poljubne funkcije; na področju informatike gre navadno za funkcijo časovne ali prostorske zahtevnosti algoritma (porabe procesorskega časa ali pomnilnika). $g(x) = O(f)$ pomeni, da funkcija *g* asimptotsko gledano ne raste hitreje od *f*, $g(x) = \Theta(f)$ pa, da *g* in *f* asimptotsko gledano rasteta enako.

lahko dobili z več iteracijami po različnih možnostih za označitev posameznih besed. Bi pa to občutno povečalo časovno zahtevnost algoritma. Na trenutni stopnji razvoja algoritma to še ni smiselno, saj je takih povedi, ki bi jih algoritem lahko uspešno analiziral na več možnih načinov, malo. Zaradi svoje formalne narave algoritem ne more biti uspešen tudi pri ločevanju med predmeti in prislovnimi določili – na trenutni stopnji razvoja se kot predmeti označijo vse nepredložne samostalniške (ali posamostaljene) besedne zveze v odvisnih sklonih, kot prislovna določila pa vse predložne. Za uspešnejše ločevanje med predmeti in prislovnimi določili bi bilo v nadaljnjem razvoju moč uporabiti podatke o vezljivosti glagolov iz elektronskega vezljivostnega slovarja Andreje Žele (ŽELE 2008). Manjši problem bi sicer predstavljali izpeljani glagoli, ki jih slovar ne navaja, vendar bi se tudi to dalo rešiti s sestavitvijo podalgoritma za identifikacijo izpeljanih glagolov.

4. Primeri uporabe algoritma

4.1. Opis programa za pregled analiziranih povedi

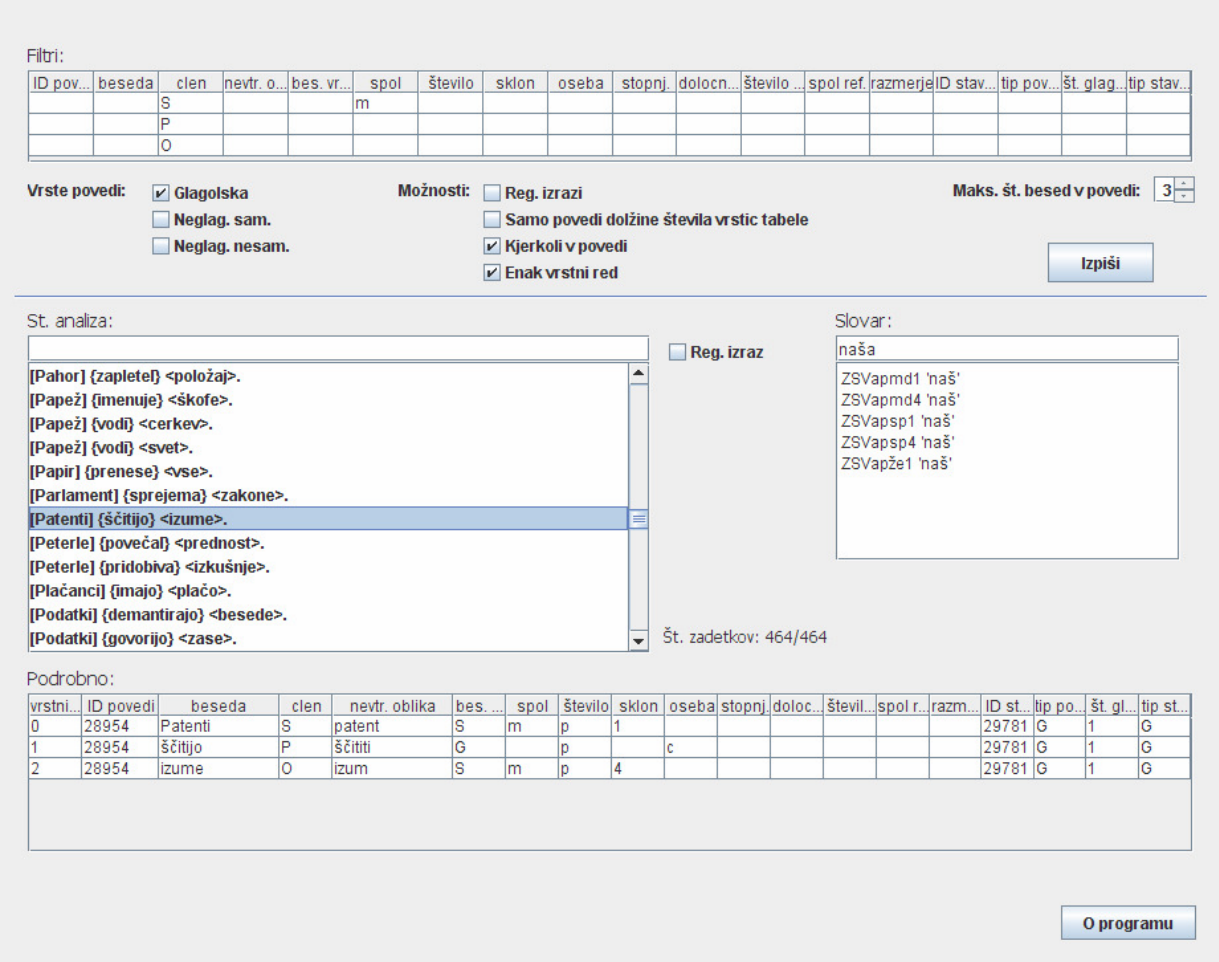
Uporabnik ob zagonu apleta ali samostojne aplikacije zagleda grafični vmesnik s Slike 3. V zgornjem delu vmesnika vnese želene filtre za iskanje po bazi analiziranih povedi. Program omogoča dva načina iskanja: med njima preklapljamo z označitvijo ali neoznačitvijo določila »Kjerkoli v povedi«. Pri prvem načinu, tj. z neoznačeno možnostjo »Kjerkoli v povedi«, vsaka vrstica filtrirne tabele predstavlja eno besedo (od treh) v povedi; prva vrstica predstavlja prvo besedo povedi, druga drugo itd. Če je celica oz. vrstica prazna, se to obravnava kot karkoli – kot nadomestni znak. Če ni izbrana možnost »Samo povedi dolžine števila vrstic tabele« in če je poved krajša od vnesenih filtrov v vrstice, program preverja ujemanje le s toliko začetnimi vrsticami, kot ima poved besed – izpiše torej tudi povedi krajše od števila vrstic tabele. Pri drugem načinu iskanju, tj. z neoznačeno možnostjo »Kjerkoli v povedi«, vsaka vrstica predstavlja kriterij za eno besedo povedi – njeno mesto v povedi ni določeno, lahko pa določimo relativni vrstni red kriterijev v filtru z označitvijo možnosti »Enak vrstni red« – program v tem primeru izbere le povedi, v katerih se pojavljajo besede v enakem vrstnem redu, kot so v vneseni filtrirni tabeli – med besedami v filtrirni tabeli pa se lahko v izpisanih povedih vrinejo tudi druge, ohranja se vrstni red vrstic. Za iskanje ni potrebno izpolniti vseh vrstic filtrirne tabele, prazne vrstice pa so lahko le za izpolnjenimi vrsticami, ne med njimi, kajti program upošteva le izpolnjene vrstice do prve prazne. Število vrstic filtrirne tabele še vedno določa število besed v povedih, po katerih iščemo. Z možnostjo »Samo povedi dolžine števila vrstic tabele« določimo, ali je to število največje število besed povedi – program potem izpisuje krajše ali enako dolge povedi – ali pa točno število besed povedi – program potem izpisuje le natanko tako dolge povedi.

Možna sta dva podnačina iskanja oz. vnosa filtrov: navadno iskanje (brez regularnih izrazov v celicah) in z regularnimi izrazi – uporabnik možnost izbere z ustrezno označitvjo okenca »Reg. izrazi«. Regularni izrazi so formalni izrazi za učinkovito identifikacijo nizov; poenostavljeno: gre za kompleksnejši sistem nadomestnih znakov. Uporabljeni so regularni izrazi sistema MySQL – v primerjavi z regularnimi izrazi sistema UNIX in Jave so nekoliko okrnjeni (dokumentacija: http://dev.mysql.com/doc/refman/5.1/en/regexp.html; http://www.regular-expressions.info/).

Primer: iskanje določenega zaimka v vseh šestih sklonih in v vseh treh osebah dobimo z vpisom regularnega izraza '[1-6]' ali '[123456]' v stolpec *sklon* in izraza '[a-c]' ali '[abc]' v stolpec *oseba*. Iskanje dveh različnih besed hkrati, npr. besed *hiša* in *stanovanje*, pa dobimo z vpisom 'hišalstanovanje' v stolpec *nevtr. oblika*. Nadomestna znaka sta:

- '*' – nadomešča poljubno število znakov, tudi prazen niz (pri navadnem iskanju je to znak '%');
- '.' – nadomešča natanko en znak (pri navadnem iskanju je to znak '_').

Pod tabelo filtrov uporabnik izbere želene tipe povedi – glagolska poved je vsaka poved, ki vsebuje glagol, neglagolska samostalniška pa vsaka, ki vsebuje vsaj en samostalnik in nobenega glagola. Uporabnik sproži iskanje s klikom na gumb **Izpiši**.



Slika 3: Program/aplet za iskanje po bazi analiziranih povedi

V osrednjem delu okna je seznam z analiziranimi povedmi – tekstovno polje nad seznamom omogoča vnos filtrov za sprotno filtriranje seznama zadetkov v pomnilniku. Iskalnik ne ločuje med malimi in velikimi črkami, možen pa je tudi vnos javanskih regularnih izrazov, pri čemer je potrebno paziti na znake '[]{}()', ki so za regularne izraze rezervirani znaki – pri iskanju z regularnimi izrazi jih je potrebno predznačiti z ubežnim znakom '\'. Iskanje z regularnimi izrazi se izbere z označitvijo okenca »Reg. izraz« poleg vnosnega polja. Iskanje uporabnik sproži s pritiskom na tipko **Enter**. V seznamu povedi so naštete povedi, ki ustrezajo iskalnim kriterijem v razdelku filtri in iskalnim kriterijem za iskanje v pomnilniku. Stavčni členi so označeni z ustreznimi oklepaji: '{}' predstavlja povedek', '[]' osebek, '<>' predmet, '//' prisl. določilo, '{{}}' pov. določilo, '|' pa ločuje glagolske stavke. S klikom na posamezno poved se na dnu ekrana v tabeli izpišuje atributi vseh besed. Možno je tudi izbiranje več povedi s pritiskom na tipko Ctrl in kliki. Na desni strani osrednjega dela je tudi za analizo povedi služivši[9] slovar – namenjen je zgolj za preverjanje uspešnosti algoritma in vsebuje le besedne oblike, ki so v korpusu.

4.2. Oblikoslovne in skladenjske oznake

Uporabljene oblikoslovne oznake so enake kot v oblikoslovnem označevalniku Primoža Jakopina (JAKOPIN 2001) – natančneje so opisane na podstrani Razlaga oblikoslovnih oznak (http://bos.zrc-sazu.si/bibliografija/o_oznake.html). Uvedene skladenjske oznake za stavčne člene pa so:

- *P* – povedek;
- *S* – osebek;
- *O* – predmet;
- *A* – prislovno določilo;
- *PD* – povedkovo določilo.

Uvedene skladenjske oznake za tipe povedi so:

- *G* – glagolska poved, tj. poved, ki vsebuje vsaj en glagolski stavek;
- *nG* – neglagolska samostalniška poved, tj. poved, ki ne vsebuje nobenega glagolskega stavka, vsebuje pa vsaj en samostalnik;

[9] Raba deležij na -ši odraža željo po ohranitvi izredno učinkovite izrazne možnosti jezika, ki žal izginja. Omenjena deležja so, neupoštevajoč kvalifikatorje SSKJ oz. pravopisa, rabljena kot slogovno nezaznamovana.

- *nnG* – neglagolska nesamostalniška poved, tj. poved, ki ne vsebuje nobenega glagolskega stavka niti nobenega samostalnika.

Uvedene skladenjske oznake za tipe stavkov so:

- *G* – glagolski stavek;
- *PS* – samostalniški pastavek;
- *PO* – nesamostalniški pastavek;

Če je stavek zanikan, ima na koncu dodan znak '-' (zaenkrat algoritem zanikane stavke prepoznava le pri glagolskih povedih).

4.3. Primeri iskanja

Navajam nekaj možnih primerov iskanja. Korpus analiziranih povedi obsega, kot že omenjeno, povedi do vključno dolžine treh besed. Primeri:

- Vse zanikane povedi tipa *osebek + vez + povedkovo določilo* izpišemo tako, da v filtru Vrste povedi označimo možnost »Glagolska«, v prvo vstico filtrirne tabele v stolpec *clen* vpišemo *S*, v drugo vrstico v isti stolpec *P*, v tretjo vrstico v isti stolpec pa *PD*. Stolpec *tip stavka* pa izpolnimo z *G-*. V filtru Možnosti označimo »Kjerkoli v povedi« in poženemo iskanje s pritiskom na gumb **Izpiši**:
    - Število zadetkov: 371;
    - Prvih deset najdenih povedi:
        - [Argument] {ni} {{slab}}.
        - [Ateist] {ni} {{nevernik}}?
        - [Bilanca] {ni} {{spodbudna}}.
        - [Bojazen] {ni} {{neutemeljena}}.
        - [Center] {ni} {{mrtev}}.
        - [Cerkev] {ni} {{nadškof}}.
        - [Čakanje] {ni} {{modro}}.
        - {{Čisto}} [tako] {ni}.
        - [Človek] {ni} {{Bog}}.
        - [Človek] {ni} {{most}}.
- Vse povedi, ki vsebujejo osebek, povedek in predmet, izpišemo tako, da v filtru Vrste povedi označimo možnost »Glagolska«, v prvo vstico filtrirne tabele v stolpec *člen* vpišemo *S*, v drugo vrstico v isti stolpec *P*, v tretjo vrstico v isti stolpec pa *O*. V filtru Možnosti označimo »Kjerkoli v povedi« in nadaljujemo iskanje z gumbom **Izpiši**;
    - Število zadetkov: 3243;

- Prvih deset najdenih povedi:
  - [Abotna] {zmaga}.
  - [Absolutiziranje] {rojeva}.
  - [Aeroplani] {so}.
  - [Ajda] {pobira}.
  - <Akcijo> {jemlje} [osebno].
  - [Alah] {nagrajuje}.
  - [Albanci] {slavijo}.
  - [Albin] {išče}.
  - [Amater] {ovadil}.
  - [Američani] {sledijo}.

- Vse povedi z zanikanim osebkom v rodilniku poiščemo tako, da v filtru Vrste povedi označimo možnost »Glagolska«, v prvo vrstico filtrirne tabele v stolpec *clen* vpišemo *S*, v stolpec *sklon* pa vpišemo *2* ter v filtru Možnosti označimo »Kjerkoli v povedi« (možnost »Enak vrstni red« naj ne bo označena) in pritisnemo na gumb Izpiši;
  - Število zadetkov: 1390;
  - Prvih deset najdenih povedi:
    - [Absolutne] garancije {ni}.
    - [Absolutne] [varnosti] {ni}.
    - [Absolutnih] [pogojev] {ni}.
    - [Absolutnih] [resnic] {ni}.
    - Ali {ni} [skeptičnosti]?
    - Ampak {nimamo} [časa]!
    - Ampak {ni} [problema].
    - Ampak {ni} [sile]!
    - Ampak [saj] {niso}!
    - [Avta] {niso} {ukradli} ...

- Vse zanikane povedi s prehodom tožilniškega predmeta v rodilnik izpišemo tako, da v filtru Vrste povedi označimo možnost »Glagolska«, v prvo vrstico filtrirne tabele v stolpec *člen* vpišemo *O*, v stolpec *sklon* vnesemo *2*, v stolpec *tip stavka* pa *G-*. V filtru Možnosti označimo »Kjerkoli v povedi« in nadaljujemo z **Izpiši**;
  - Število zadetkov: 396;
  - Prvih deset najdenih povedi:
    - <Agresije> {ni} {čutiti}.
    - <Alkohola> {ne} {strežemo}.
    - <Ambicij> {ne} {skrivam}.
    - <Avtorizacije> {ne} {zahteva}.
    - <Barabij> {ne} {zagovarjam}.
    - <Brezbrižnosti> {ne} {manjka}.

        - <Brezdelja> {ne} {prenaša}.
        - <Česar> {ne} {skriva}.
        - <Česa> {ne} {moremo}?
        - <Česa> {ne} {smem}?

- Program lahko uporabimo tudi za preučevanje vezljivosti glagolov. Vse vezave glagola *delati* z dajalnikom in tožilnikom izpišemo tako, da v filtru Vrste povedi označimo možnost »Glagolska«, v prvo vrstico filtrirne tabele v stolpec *nevtr. oblika* vpišemo *delati*, v naslednjo vrstico vpišemo v stolpec *sklon* številko *3*, v zadnjo vrstico pa vpišemo *4.* V filtru Možnosti označimo »Kjerkoli v povedi« (možnost »Enak vrstni red« naj ne bo označena) in gremo naprej z gumbom **Izpiši**;
    - Število zadetkov: 4;
    - Najdene povedi:
        - {Delam} <si> <zapiske>.
        - <Kaj> {delajo} <njegovi>?
        - <Komu> {delajo} <uslugo>?
        - <Priznanji> <časnikarjema> {Dela}.
- Program je zelo prikladen tudi za iskanje besednih zvez oziroma frazemov. Vse trdilne in nikalne pojavitve besedne zveze *imeti rad* poiščemo tako, da v filtru Vrste povedi označimo možnost »Glagolska«, v prvo vrstico filtrirne tabele v stolpec *nevtr. oblika* vpišemo *%imeti*, v naslednjo vrstico v isti stolpec pa *rad*, v filtru Možnosti pa označimo »Kjerkoli v povedi« (možnost »Enak vrstni red« naj ne bo označena) in - **Izpiši**;
    - Število zadetkov: 59;
    - Prvih deset najdenih povedi:
        - <Ameriko> {imam} rad.
        - <Časopise> {ima} rad.
        - {Imam} rad <pse>.
        - {Imeti} {se} rad.
        - Irak {imam} rad.
        - <Jih> {imaš} rad?
        - {Nimam} rad [primerjav].
        - <Oba> {imam} rad.
        - <Oboje> {imam} rad.
- <Polemike> {imam} rad. Vse pojavitve frazema *dobiti jih* izpišemo tako, da v filtru Vrste povedi označimo možnost »Glagolska«, v prvo vrstico filtrirne tabele v stolpec *nevtr. oblika* vpišemo *dobiti*, v naslednjo vrstico v isti stolpec pa *ona*, v stolpec število vpišemo *p*, v stolpec *sklon* pa vnesemo *2*. V filtru Možnosti označimo »Kjerkoli v

povedi« (možnost »Enak vrstni red« naj ne bo označena) in poženemo iskanje s pritiskom na gumb **Izpiši**.

- Število zadetkov: 4;
- Najdene povedi:
  - {Dobili} {so} .
  - {Dobite} /takoj/!
  - Kje {dobiti}?
  - {Ni} [jih] {dobil}.

5. Izvorna koda

Zaradi prevelikega obsega je navedena le izvorna koda za prvi del programa, to je del za analizo povedi.

Preostala izvorna koda je na priloženem elektronskem mediju.

5.1. Category.java

```
package stavcnaanaliza;

import java.util.regex.*;


public class Category {

    //word Category patterns
    public static final Pattern anyVerb = Pattern.compile("^G");
    public static final Pattern mainVerb = Pattern.compile("^G[ZVabc][^P]");
    public static final Pattern verbToBe = Pattern.compile("^G[PFZOR]");
    public static final Pattern verbToBeAuxilliary = Pattern.compile("^GP|^G[FZ]P");
    public static final Pattern verbToBeRelational = Pattern.compile("^GR|^G[FZ]R");
    public static final Pattern verbToBeNegated = Pattern.compile("^GZ[OPR]");
    public static final Pattern verbToBeExistenceNegated = Pattern.compile("^GZ[OP]ce");
    public static final Pattern verbToBeAndToHaveNegated = Pattern.compile("^GZ");
    public static final Pattern infinitiveOrSupineVerb = Pattern.compile("^GN[EA]");
    public static final Pattern pastParticiple = Pattern.compile("^GL");
    public static final Pattern personalVerbForm = Pattern.compile("^G[PFZORVabc]");
    public static final Pattern participle = Pattern.compile("^G[LN]");
    public static final Pattern freeVerbMorphem = Pattern.compile("^Gmp");

    public static final Pattern anyNoun = Pattern.compile("^[SI]");
    public static final Pattern possibleSubject =
Pattern.compile("^(?:[SI]|Z|P[^D][^PVZM])[\\wćčžšđĆČŽŠĐ]*1");   //check also if in
nominative
    public static final Pattern possibleNominalOrPronominalSubject =
Pattern.compile("^(?:[SI]|ZO[^P])[\\wćčžšđĆČŽŠĐ]*1");   //check also if in nominative
    public static final Pattern possibleNominalOrPronominalObject =
Pattern.compile("^(?:[SI]|ZO[^P])[\\wćčžšđĆČŽŠĐ]*[2-6]");   //check also if in nominative
    public static final Pattern possibleNegatedSubject =
Pattern.compile("^(?:[SI]|Z|P[^D][^PVZM])[\\wćčžšđĆČŽŠĐ]*2");   //check also if in
nominative
    public static final Pattern possibleNominalizedSubject =
Pattern.compile("^(?:Z|P[^D][^PVZM])[\\wćčžšđĆČŽŠĐ]*1");   //check also if in nominative
    public static final Pattern possibleNominalizedObject =
Pattern.compile("^(?:Z|P[^D][^PVZM])[\\wćčžšđĆČŽŠĐ]*[2-6]");   //check also if in
nominative
    public static final Pattern possiblePredicativePhrase =
Pattern.compile("^(?:[SI]|ZO[^P]|P[^D][^PVZM])[\\wćčžšđĆČŽŠĐ]*1");

    public static final Pattern personalPronoun = Pattern.compile("^ZO[^P]");

    public static final Pattern adverb = Pattern.compile("^A");
    public static final Pattern conjunction = Pattern.compile("^V");
    public static final Pattern preposition  = Pattern.compile("^E");
```

```
    //cmpSentType marks
    public static final String verbalCmpSent = "G";
    public static final String nonVerbalNounCmpSent = "nG";
    public static final String nonVerbalNonNounCmpSent = "nnG";

    //sentType marks
    public static final String verbalSentence = "G";  //pastavek
    public static final String nounParaSentence = "PS";  //pastavek
    public static final String otherParaSentence = "PO";
    public static final String negatedSentenceMark = "-";  //pastavek

    //artType marks
    public static final String predicateDBMark = "P";
    public static final String subjectDBMark = "S";
    public static final String objectDBMark = "O";
    public static final String adverbialDBMark = "A";
    public static final String predicatePhraseDBMark = predicateDBMark + "D";
    public static final String subjectCoreDBMark = subjectDBMark + "S";
    public static final String objectCoreDBMark = predicateDBMark + "O";


    protected String dictionaryID;
    protected String allCategoriesString;
    protected String wordType, gender, grammNo, grammCase, person, comparison,
definiteness, antecedentGrammNo, antecedentGender, relation;


    public Category(String ID, String t){
        dictionaryID = ID;
        allCategoriesString = t;

        Pattern pat = Pattern.compile("^[A-ZČŠŽ]+");
        Matcher mat = pat.matcher(t);
        if(mat.find())
            wordType = mat.group(0);
        else
            wordType = Word.nullSign;


        pat = Pattern.compile("[mžs]");
        mat = pat.matcher(t);
        if(mat.find())
            gender = mat.group(0);
        else
            gender = Word.nullSign;
        if(mat.find())
            antecedentGender = mat.group(0);
        else
            antecedentGender = Word.nullSign;

        pat = Pattern.compile("[ed]|p(?![ro])");
        mat = pat.matcher(t);
        if(mat.find())
            grammNo = mat.group(0);
        else
            grammNo = Word.nullSign;
        if(mat.find())
            antecedentGrammNo = mat.group(0);
        else
            antecedentGrammNo = Word.nullSign;
```

```
        pat = Pattern.compile("[1-6]");
        mat = pat.matcher(t);
        if(mat.find())
            grammCase = mat.group(0);
        else
            grammCase = Word.nullSign;

        pat = Pattern.compile("[abc]");
        mat = pat.matcher(t);
        if(mat.find())
            person = mat.group(0);
        else
            person = Word.nullSign;

        pat = Pattern.compile("j+");
        mat = pat.matcher(t);
        if(mat.find())
            comparison = mat.group(0);
        else
            comparison = Word.nullSign;

        pat = Pattern.compile("i");
        mat = pat.matcher(t);
        if(mat.find())
            definiteness = mat.group(0);
        else
            definiteness = Word.nullSign;

        pat = Pattern.compile("pr|po");
        mat = pat.matcher(t);
        if(mat.find())
            relation = mat.group(0);
        else
            relation = Word.nullSign;

    }

    public String catToSQLExportString(){
        StringBuffer strb = new StringBuffer();


strb.append(dictionaryID).append("\t").append(wordType).append("\t").append(gender).append
("\t").append(grammNo).append("\t").append(grammCase).append("\t").append(person).append("
\t").append(comparison).append("\t").append(definiteness).append("\t").append(antecedentGr
ammNo).append("\t").append(antecedentGender).append("\t").append(relation).append("\n");

        return strb.toString();
    }

}
```

5.2. CmpSentence.java

```
package stavcnaanaliza;
import java.util.*;
import java.util.regex.*;


public class CmpSentence {
    protected int ID, noOfVerbSent;
    protected String type;
    protected int frequency;
    protected String structure;

    protected boolean analysed = false;
    protected Corpus corpus;

    ArrayList<Word> words = new ArrayList<Word>();
    ArrayList<Element> sentences = new ArrayList<Element>();
    /*ArrayList<Element> sentences;
    ArrayList<Word> currSentVerbPhrase;*/

    CmpSentence(int id, Corpus corp, int f){
        ID = id;
        type = Word.nullSign;
        noOfVerbSent = 0;
        frequency = f;
        structure = Word.nullSign;

        corpus = corp;
    }

    public void addWord(Word w){
        words.add(w);
    }


    public void analyse() throws Exception{


        if(this.containsPossibleVerb()){       //2 tipa: S in nS
            //1.
            this.type = Category.verbalCmpSent; //glag. poved
            this.findSentences();
            this.analyzeSentences();



        }
        else{       //if (enodelen) pastavek
            if(this.containsPossibleNoun()){
                this.type = Category.nonVerbalNounCmpSent;  //samostalniška neglag. poved
            }
            else
                this.type = Category.nonVerbalNonNounCmpSent;    //nesamost. neglag. poved

            this.analyzeNonVerbalCmpSentence();
        }
        this.setWordTypesForUnanalysedWords();
        analysed = true;
```

```
    }


    public boolean containsPossibleNoun(){

       for(Word word : words){
            if(word.canBeOfType(Category.anyNoun))
                return true;
        }
        return false;
    }

    public boolean containsPossibleVerb(){

        for(Word word : words){
            if(word.canBeOfType(Category.anyVerb))
                return true;
        }
        return false;
    }


    public boolean findSentences() throws Exception{

        ArrayList<Word> currSentVerbPhrase = new ArrayList<Word>();
        int lastFoundSentBorder = -1;
        Word negativingArticle = null;
        boolean participle = false;

         for(int i = 0; i < words.size(); i++){
            Word word = words.get(i);

            if(word.canBeOfType(Category.verbToBeAndToHaveNegated))
                negativingArticle = word;

            if(word.isWord("ne")){
               negativingArticle = word;
            }
            else if(word.canBeOfType(Category.anyVerb) && !word.isWord("da")){ //if verb
                if(currSentVerbPhrase.size() > 0){ //if not the first verb in the
sentence
                    if(!checkIfPartOfTheExistingVerbPhrase(word, currSentVerbPhrase)){
                        if(word.hasAnotherPossibleWordTypeBeside(Category.anyVerb))
                           continue;

                        int supposedBorder = findSentenceBorder(lastFoundSentBorder + 1,
i);

                        if(supposedBorder != -1){      //if borderd found
                           setSentenceBorder(lastFoundSentBorder + 1, supposedBorder,
currSentVerbPhrase, negativingArticle);
                           setVerbPhrase(currSentVerbPhrase, participle);

                           lastFoundSentBorder = supposedBorder;
                        }
                        else{   //check if last verb maybe not a verb (cases like "da
bomo prišli")
                            removeLastFoundVerb(currSentVerbPhrase);
                        }
                        //new linked list for a new sentence
                        participle = false;
```

```
                        currSentVerbPhrase = new ArrayList<Word>();
                        if(negativingArticle != null)
                           negativingArticle.setWordArticle(Category.predicateDBMark);
                        negativingArticle = null;
                    }
                }
                if(word.canBeOfType(Category.participle))
                    participle = true;
                currSentVerbPhrase.add(word);
            }
            else if(word.rearPunctuationMark.contains(",") && !checkIfEnumeration(i)){
                int supposedBorder = i;
                setSentenceBorder(lastFoundSentBorder + 1, supposedBorder,
currSentVerbPhrase, negativingArticle);
                setVerbPhrase(currSentVerbPhrase, participle);
                lastFoundSentBorder = supposedBorder;
                if(negativingArticle != null && currSentVerbPhrase.size() > 0)
                    negativingArticle.setWordArticle(Category.predicateDBMark);

                participle = false;
                currSentVerbPhrase = new ArrayList<Word>();
                negativingArticle = null;
            }
         }
         //analyze the last sentence
        setSentenceBorder(lastFoundSentBorder + 1, words.size() - 1, currSentVerbPhrase,
negativingArticle);
        setVerbPhrase(currSentVerbPhrase, participle);
        if(negativingArticle != null && currSentVerbPhrase.size() > 0)
            negativingArticle.setWordArticle(Category.predicateDBMark);

        //set cmpSent noOfVerbSent
        this.noOfVerbSent = sentences.size();

        return true;

    }

    public boolean checkIfPartOfTheExistingVerbPhrase(Word word, ArrayList<Word>
verbPhrase){

        for(Word verb : verbPhrase){
            if(verb.contents.equals(word.contents)) //if verbs equal, there must be
another sentence
                return false;
            else if(!verb.canBeOfType(Category.freeVerbMorphem) && !verb.isWord("je") &&
!verb.canBeOfType(Category.infinitiveOrSupineVerb)){
                if((verb.canBeOfType(Category.mainVerb) &&
(word.canBeOfType(Category.mainVerb) || word.canBeOfType(Category.verbToBe) ||
word.isWord("bi")))
                    || (verb.canBeOfType(Category.verbToBe) &&
word.canBeOfType(Category.mainVerb))
                    || (verb.canBeOfType(Category.pastParticiple) &&
word.canBeOfType(Category.pastParticiple)))  //cases like "prišel, videl, zmagal"
                    return false;
            }
        }
        return true;
    }

    public int findSentenceBorder(int start, int limit){ //pojdi v zanki nazaj (hitreje)
```

```
        //check conjunctions
        for(int i = start; i < limit; i++){
            Word word = words.get(i);

            if(word.canBeOfType(Category.conjunction) && !word.isWord("ne")){
                if(i > 0)  //ignoriramo veznik istega stavka (potencialno prvi)
                    return i - 1;
            }
        }

        //if conj. not found, check commas etc.
        for(int i = start; i < limit; i++){
            Word word = words.get(i);

            if(word.rearPunctuationMark.matches(",|:|;|-"))
                return i;
        }
        return -1; //supposed not a verb
    }


    public boolean analyzeSentences(){

        for(Element snt : sentences){
            findNounPhrases(snt);
            findAdverbials(snt);
            //poišči prava prisl. dol. - ne iz samost.
            //find other phrases
        }

        return true;

    }

    public boolean findNounPhrases(Element snt){

        findSubject(snt);
        findPredicativePhrase(snt);

        boolean verbToBeExistenceNegated = checkIfNegatedExistenceVerbToBe(snt);
        //if subject not found
        if(snt.getSubElement(Category.subjectDBMark) == null){
            if(snt.getSubElement(Category.predicatePhraseDBMark) == null &&
verbToBeExistenceNegated)
                findNegatedSubject(snt);
            else
                findNominalizedSubject(snt);
        }

        //negated verb "ni" doesn't have objects
        if(!verbToBeExistenceNegated){
            findObjects(snt, Category.possibleNominalOrPronominalObject);
            findObjects(snt, Category.possibleNominalizedObject);
        }

        return true;

    }

    public boolean checkIfNegatedExistenceVerbToBe(Element snt){

        Element verb = snt.getSubElement(Category.predicateDBMark);
```

```
        if(snt.type.equals(Category.verbalSentence.concat(Category.negatedSentenceMark))
&&
                    ((verb.words.size() == 1 &&
verb.words.get(0).canBeOfType(Category.verbToBe)) ||
                      (verb.words.size() == 2 &&
verb.containsCategory(Category.verbToBeNegated) &&
verb.containsCategory(Category.pastParticiple))))
            return true;
        else
            return false;
    }

    public boolean findSubject(Element snt){

        mainLoop:
        for (int i = 0; i < snt.words.size(); i++){
            Word word = snt.words.get(i);
             //"ni" && samo ni kot glag.

            if(word.wordTypeAllPoss == null || word.wordType != null)
                continue;
            if(word.isWord("kot") || word.isWord("kakor")){
                i = findEndOfConjunctionPhrase(i, snt);
                continue mainLoop;
            }
            //check if a word can be a subject
            for(Lemma lem : word.wordTypeAllPoss){
                //if the word is preposition => find
               if(lem.canBeOfType(Category.preposition)){
                    i = findEndOfPrepositionPhrase(i, snt, lem);
                    continue mainLoop;
                }
               else if(lem.canBeOfType(Category.possibleNominalOrPronominalSubject)){ //if
noun (jedro) and in nominative

if(word.checkCongruenceWithVerb(snt.getSubElementWords(Category.predicateDBMark), lem)){
//preveri še ujemanje z glagolom
                        word.setWordType(lem);
                        findSubjectAntecedents(snt, lem, i);
                        return true;
                    }
                }
            }
        }

        return false;
    }


    public boolean findNegatedSubject(Element snt){

        for (int i = snt.words.size() - 1; i >= 0; i--){
            Word word = snt.words.get(i);
             //"ni" && samo ni kot glag.

            if(word.wordTypeAllPoss == null || word.wordType != null)
                continue;

            //check if a word can be a subject
            for(Lemma lem : word.wordTypeAllPoss){
               if(lem.canBeOfType(Category.possibleNegatedSubject)){ //if noun (jedro) and
```

```
in nominative
                    word.setWordType(lem);
                    findSubjectAntecedents(snt, lem, i);
                    return true;
                }
            }
        }
        return false;

    }

    public boolean findNominalizedSubject(Element snt){

        for (int i = snt.words.size() - 1; i >= 0; i--){
            Word word = snt.words.get(i);
             //"ni" && samo ni kot glag.

            if(word.wordTypeAllPoss == null || word.wordType != null)
                continue;

            //check if a word can be a subject
            for(Lemma lem : word.wordTypeAllPoss){
               if(lem.canBeOfType(Category.possibleNominalizedSubject)){ //if noun (jedro)
and in nominative

if(word.checkCongruenceWithVerb(snt.getSubElementWords(Category.predicateDBMark), lem)){
//preveri še ujemanje z glagolom
                        word.setWordType(lem);
                        findSubjectAntecedents(snt, lem, i);
                        return true;
                    }
                }
            }
        }
        return false;
    }


    public boolean findObjects(Element snt, Pattern objectCategoryPattern){

        for (int i = 0; i < snt.words.size(); i++){
            Word word = snt.words.get(i);
             //"ni" && samo ni kot glag.

            if(word.wordTypeAllPoss == null || word.wordType != null ||
word.isWord("prav"))
                continue;
            //check if a word can be a object
            for(Lemma lem : word.wordTypeAllPoss){
                if(lem.canBeOfType(objectCategoryPattern)){ //if noun (jedro) and not in
nominative
                    //preveri vezljivost
                    findObjectAntecedents(snt, i, objectCategoryPattern);
                }
            }
        }

        return false;
    }

    protected void setWordTypesForUnanalysedWords() {
```

```
        for(Word word : words){
            if(word.wordType == null){
                //supposition - the first poss. is the right one
                if(word.wordTypeAllPoss != null && word.wordTypeAllPoss.size() > 0)
                    word.wordType = word.wordTypeAllPoss.get(0);
                else
                    word.wordType = new Lemma(Word.nullSign, new Category(Word.nullSign,
Word.nullSign));
            }
        }
    }

    private void analyzeNonVerbalCmpSentence() {

         //create new sentence
        Element snt;
        String sentType;

        if(this.type.equals(Category.nonVerbalNounCmpSent))
            sentType = Category.nounParaSentence;
        else
            sentType = Category.otherParaSentence;

        snt = new Element();
        snt.ID = corpus.sentID;
        snt.type = sentType;
        sentences.add(snt);

        for(Word word : words){
            word.sentenceID = corpus.sentID;
        }
        corpus.sentID++;
    }

    private boolean checkIfEnumeration(int i) {
        return false;
    }

    private boolean checkIfOnlyVerbToBe(Element snt) {

        for(Word word : snt.getSubElementWords(Category.predicateDBMark)){
            if(!word.canBeOfType(Category.verbToBe))
                return false;
        }
        return true;
    }

    private void findAdverbials(Element snt) {
        ArrayList<Word> advWords = snt.getSubElementWords(Category.adverbialDBMark);

        if(advWords == null)
            advWords = new ArrayList<Word>();

        for (int i = 0; i < snt.words.size(); i++){
            Word word = snt.words.get(i);
             //"ni" && samo ni kot glag.

            if(word.wordTypeAllPoss == null || word.wordType != null)
                continue;
            //check if a word can be a object
            for(Lemma lem : word.wordTypeAllPoss){
                if(lem.canBeOfType(Category.adverb)){ //if noun (jedro) and not in
```

```
nominative
                    word.setWordType(lem);
                    word.setWordArticle(Category.adverbialDBMark);
                    advWords.add(word);
                }
            }
        }
        snt.addSubElement(Category.adverbialDBMark, advWords);

    }

    private int findEndOfPrepositionPhrase(int ind, Element snt, Lemma prepositionLemma) {
        int i;
        Word prepositionWord = snt.words.get(ind);
        ArrayList<Word> advWords = snt.getSubElementWords(Category.adverbialDBMark);

        if(advWords == null)
            advWords = new ArrayList<Word>();

        for (i = ind + 1; i < snt.words.size(); i++){
            Word word = snt.words.get(i);

            if(word.wordTypeAllPoss == null)
                continue;
            for(Lemma lem : word.wordTypeAllPoss){
                if(prepositionWord.canBeOfType(lem.categories.grammCase)){
                    word.setWordType(lem);
                    word.setWordArticle(Category.adverbialDBMark);
                    advWords.add(word);
                    continue;
                }
            }
            break;
        }

        //set and add preposition - set preposition the case of the last word
        if(i > ind + 1 && snt.words.get(i - 1).wordType != null)
            prepositionWord.setWordType(snt.words.get(i -
1).wordType.categories.grammCase);
        else
            prepositionWord.setWordType(prepositionLemma);
        prepositionWord.setWordArticle(Category.adverbialDBMark);
        advWords.add(prepositionWord);

        snt.addSubElement(Category.adverbialDBMark, advWords);

        return i - 1;
    }

    private int findEndOfConjunctionPhrase(int ind, Element snt) { //for conjunctions such
as "kot", "kakor"
        int i;
        String conjGrammCase = "1";
        ArrayList<Word> advWords = snt.getSubElementWords(Category.adverbialDBMark);

        if(advWords == null)
            advWords = new ArrayList<Word>();

        //set and add preposition
        Word prepositionWord = snt.words.get(ind);
        prepositionWord.setWordTypeOnePoss(Category.conjunction);
        prepositionWord.setWordArticle(Category.adverbialDBMark);
```

```
        advWords.add(prepositionWord);

        for (i = ind + 1; i < snt.words.size(); i++){
            Word word = snt.words.get(i);

            if(word.wordTypeAllPoss == null)
                continue;
            for(Lemma lem : word.wordTypeAllPoss){
                if(lem.categories.grammCase.equals(conjGrammCase)){
                    word.setWordType(lem);
                    word.setWordArticle(Category.adverbialDBMark);
                    advWords.add(word);
                    continue;
                }
            }
            break;
        }
        snt.addSubElement(Category.adverbialDBMark, advWords);

        return i - 1;
    }

    private void findPredicativePhrase(Element snt) {

        if(checkIfOnlyVerbToBe(snt)){
            ArrayList<Word> predPhraseWords = new ArrayList<Word>();

            for (int i = 0; i < snt.words.size(); i++){
                Word word = snt.words.get(i);
                 //"ni" && samo ni kot glag.

                if(word.wordTypeAllPoss == null || word.wordType != null)
                    continue;
                //check if a word can be a object
                for(Lemma lem : word.wordTypeAllPoss){
                    if(lem.canBeOfType(Category.possiblePredicativePhrase)){ //if noun
(jedro) and not in nominative

if(word.checkCongruencePredicatePhraseWithVerb(snt.getSubElementWords(Category.predicateDB
Mark), lem)){
                                word.setWordType(lem);
                                word.setWordArticle(Category.predicatePhraseDBMark);
                                predPhraseWords.add(word);
                            }
                        }
                    }
                }
                //correct verb phrase => relational verb to be
                if(predPhraseWords.size() > 0){
                    snt.addSubElement(Category.predicatePhraseDBMark, predPhraseWords);
                    ArrayList<Word> verbs = snt.getSubElementWords(Category.predicateDBMark);
                    for(int i = 0; i < verbs.size(); i++)

verbs.get(i).setWordTypeOnePossWithoutRemovingPreviousType(Category.verbToBeRelational);
                }
}
    }

    private void findSubjectAntecedents(Element snt, Lemma subjectLemma, int ind) {

        //find attributes
        int i;
```

```
        String subjectGrammCase = subjectLemma.categories.grammCase;
        Word subjectWord = snt.words.get(ind);
        ArrayList<Word> subjWords = new ArrayList<Word>();

        //add main subject part
        subjectWord.setWordType(subjectLemma);
        subjectWord.setWordArticle(Category.subjectDBMark);
        subjWords.add(subjectWord);

        for (i = ind - 1; i >= 0; i--){
            Word word = snt.words.get(i);

            if(word.wordTypeAllPoss == null)
                continue;
            else if(word.wordType != null)
                break;
            for(Lemma lem : word.wordTypeAllPoss){
                if(lem.categories.grammCase.equals(subjectGrammCase)){
                    word.setWordType(lem);
                    word.setWordArticle(Category.subjectDBMark);
                    subjWords.add(word);
                    continue;
                }
            }
            break;
        }
        snt.addSubElement(Category.subjectDBMark, subjWords);

        //set main part of the subject

snt.getSubElement(Category.subjectDBMark).addSubElement(Category.subjectCoreDBMark,
subjectWord);
    }

    private void findObjectAntecedents(Element snt, int ind, Pattern objectPattern) {

        //find attributes
        int i;
        Word objectWord = snt.words.get(ind);
        ArrayList<Word> objWords = new ArrayList<Word>();


        for (i = ind - 1; i >= 0; i--){
            Word word = snt.words.get(i);

            if(word.wordTypeAllPoss == null)
                continue;
            else if(word.wordType != null)
                break;
            for(Lemma lem : word.wordTypeAllPoss){
                if(objectWord.canBeOfType(lem.categories.grammCase)){
                    word.setWordType(lem);
                    word.setWordArticle(Category.objectDBMark);
                    objWords.add(word);
                    continue;
                }
            }
            break;
        }

        //set and add preposition - set preposition the case of the last word
        if(i < ind - 1 && snt.words.get(ind - 1).wordType != null)
```

```
            objectWord.setWordType(snt.words.get(ind - 1).wordType.categories.grammCase);
        else
            objectWord.setWordTypeOnePoss(objectPattern);

        objectWord.setWordArticle(Category.objectDBMark);
        objWords.add(objectWord);

        String objectType =
Category.objectDBMark.concat(objectWord.wordType.categories.grammCase);
        snt.addSubElement(objectType, objWords);

        //set main part of the object
        snt.getSubElement(objectType).addSubElement(Category.objectCoreDBMark,
objectWord);

    }


    private void removeLastFoundVerb(ArrayList<Word> verbPhrase) {

        for(Word verb : verbPhrase){
            //verb.setAnotherPossibleWordTypeBeside(Category.anyVerb);
            verb.wordType = null;
        }
        verbPhrase = new ArrayList<Word>();
    }


    private void setSentenceBorder(int startIndex, int finalIndex, ArrayList<Word>
verbPhrase, Word negated) throws Exception {

        //create new sentence
        Element snt = new Element();
        ArrayList<Word> sentWords = new ArrayList<Word>();
        snt.ID = corpus.sentID;

        if(negated == null) //if positive sentence
            snt.type = Category.verbalSentence;
        else
            snt.type = Category.verbalSentence.concat(Category.negatedSentenceMark);

        snt.subType = Word.nullSign;
        snt.addSubElement(Category.predicateDBMark, verbPhrase);
        sentences.add(snt);

        for(int i = startIndex; i <= finalIndex; i++){
            Word word = words.get(i);
            word.sentenceID = corpus.sentID;
            word.sentence = snt;
            word.cmpSentence = this;
            sentWords.add(word);
        }
        snt.words = sentWords;
        corpus.sentID++;

    }


    private void setVerbPhrase(ArrayList<Word> verbPhrase, boolean participleInPredicate){

        for(Word verb : verbPhrase){
           if(participleInPredicate && verb.wordTypeAllPoss != null)
```

```
                verb.setPreferredWordType(Category.verbToBeAuxilliary, Category.anyVerb);
            else
                verb.setWordTypeOnePoss(Category.anyVerb);

            verb.setWordArticle(Category.predicateDBMark);
        }

    }

    public String wordsToSQLExportString(){

        StringBuffer strb = new StringBuffer();

        for(int i = 0; i < words.size(); i++){
            Word word = words.get(i);

strb.append(word.ID).append("\t").append(i).append("\t").append(word.frontPunctuationMark)
.append("\t").append(word.contents).append("\t").append(word.rearPunctuationMark).append("
\t").append(corpus.cmpSentID).append("\t").append(word.sentenceID).append("\t").append(wor
d.artType).append("\t").append(word.wordType.neutralForm).append("\t").append(word.wordTyp
e.categories.dictionaryID).append("\n");

        }

        return strb.toString();

    }

    public String sntToSQLExportString(){

        StringBuffer strb = new StringBuffer();

        for(Element snt : sentences)

strb.append(snt.ID).append("\t").append(snt.type).append("\t").append(snt.subType).append(
"\n");

        return strb.toString();

    }

    public String cmpSentToSQLExportString(){

        StringBuffer strb = new StringBuffer();


strb.append(ID).append("\t").append(words.size()).append("\t").append(frequency).append("\
t").append(type).append("\t").append(noOfVerbSent).append("\t").append(structure).append("
\n");

        return strb.toString();

    }

}
```

5.3. Corpus.java

```
package stavcnaanaliza;

import java.io.*;
import java.util.*;
import java.util.regex.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.sax.*;
import org.xml.sax.helpers.AttributesImpl;

import java.sql.*;
import javax.swing.JTextPane;
import javax.swing.text.*;
import java.awt.*;


/**
 *
 * @author Robert
 */
public class Corpus{

     final static String corpusFileName = "korpus.txt";

     int wordID = 1, artID = 1, sentID = 1, cmpSentID = 1;

     Dictionary dictionary;
     DBConnection DBconn;

     public Corpus() throws Exception{

        int noOfCmpSent = 0;

        dictionary = new Dictionary();   //create new categories dictionary
        buildAndAnalyzeCorpus();

     }


     private void buildAndAnalyzeCorpus() throws Exception{

        BufferedReader reader = new BufferedReader(new InputStreamReader(new
FileInputStream(corpusFileName), "UTF-8"));


        //MySQL

        DBconn = new DBConnection();
        DBconn.clearDB();

        //measure insert time
        long a = System.currentTimeMillis();

        //create temp. file to insert the data into the database
        Writer fw = new OutputStreamWriter(new FileOutputStream("words-sql.txt"), "UTF-
```

```
8");
        Writer fs = new OutputStreamWriter(new FileOutputStream("snt-sql.txt"), "UTF-8");
        Writer fc = new OutputStreamWriter(new FileOutputStream("cmpsnt-sql.txt"), "UTF-
8");


        for(String line = reader.readLine(); line != null; line = reader.readLine()){
            //parse cmp sentences => cmpSentence => list of words
            String[] wordStrings = line.split ("·");
            int freq = Integer.parseInt(wordStrings[1]);
            CmpSentence cmpSent = new CmpSentence(cmpSentID, this, freq);
            wordStrings = wordStrings[0].split (" ");
            tokenizeWords(wordStrings, cmpSent);

            //analyze cmpSentence
            cmpSent.analyse();
            //put all words and word atributes into the txt file
            fw.write(cmpSent.wordsToSQLExportString());
            fs.write(cmpSent.sntToSQLExportString());
            fc.write(cmpSent.cmpSentToSQLExportString());

            cmpSentID++;
        }

        fw.flush();
        fs.flush();
        fc.flush();
        fw.close();
        fs.close();
        fc.close();
        reader.close();
        DBconn.fillDB();
        long b = System.currentTimeMillis();
        System.out.println("Čas nalaganja v bazo: "+(b-a)+" ms.");

    }

   private void tokenizeWords(String[] wordStrings,  CmpSentence cmpSen) throws Exception{

         Word word = null;

         for(String wordString : wordStrings){
              String leftPunct = "", rightPunct = "", wordStringTokenized = "";
              int i, j;

              //find punctuation marks at the beginning of the word
              for(i = 0; i < wordString.length() &&
!Character.isLetterOrDigit(wordString.charAt(i)); i++);
              leftPunct = wordString.substring(0, i);

              if(i == wordString.length()){  //if empty word - only punctuation
                  if(word != null)
                      word.rearPunctuationMark.concat(leftPunct);
                  //else //if "..." at the beginning of the word
              }
              else{

                  //find word
                  for(j = i; j < wordString.length() &&
Character.isLetterOrDigit(wordString.charAt(j)); j++);
                  wordStringTokenized = wordString.substring(i, j);
```

```
                    //find punctuation marks at the end of the word
                    rightPunct = wordString.substring(j, wordString.length());

                    //look up for a word in the dictionary
                    ArrayList<Lemma> allTypePoss =
dictionary.lemmaDict.get(wordStringTokenized);
                    word = new Word(leftPunct, wordStringTokenized, rightPunct, allTypePoss,
wordID);

                    cmpSen.addWord(word);
                    wordID++;
                }
            }

        }


    public String query(int condition){



        return null;
    }




    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

    }

}
```

5.4. Dictionary.java

```
package stavcnaanaliza;
import java.io.*;
import java.net.*;
import java.util.*;


public class Dictionary {
    final static String dictFileName = "besede_z_oznakami.txt";
    protected int omittedNoLem = 0;
    Hashtable<String, ArrayList<Lemma>> lemmaDict = new Hashtable<String,
ArrayList<Lemma>>(99000);   //the key is a neutral form of a word
    int dictionaryID = 1;


    public Dictionary() throws Exception{
         BufferedReader reader=new BufferedReader(new InputStreamReader(new
FileInputStream(dictFileName), "UTF-8"));
         String line="";
         Hashtable<String, Category> enteredCategories = new Hashtable<String,
```

```
Category>(1700);

        try{

            Writer fd = new OutputStreamWriter(new FileOutputStream("dictionary-
sql.txt"), "UTF-8");

            for( line = reader.readLine(); line != null; line = reader.readLine()){
                int k = 0, end = 0;
                String wordForm, lemma;

                for(int i = line.length() - 1; i >= 0; i--){

                    if(end == 0){
                       if(line.charAt(i) == '|')
                           end = k + 1;
                    }
                    else if(line.charAt(i) == '|'){
                       wordForm = line.substring(line.length() - k, line.length() - end);
                       lemma = line.substring(0, line.length() - k);
                       //System.out.println(wordForm);

                       String[] poss = lemma.split("[\\|;]");
                       ArrayList<Lemma> wordTypeAllPoss = new ArrayList<Lemma>();

                       for(int ct = 0; ct < poss.length - 1; ct = ct + 2){
                           Category cat;
                           if(enteredCategories.containsKey(poss[ct+1]))
                               cat = enteredCategories.get(poss[ct+1]);
                           else{
                               cat = new Category(Integer.toString(dictionaryID),
poss[ct+1]);

                               enteredCategories.put(poss[ct+1], cat);
                               fd.write(cat.catToSQLExportString());
                               dictionaryID++;
                           }
                           Lemma lem = new Lemma( poss[ct], cat);
                           wordTypeAllPoss.add(lem);
                       }

                       lemmaDict.put(wordForm, wordTypeAllPoss);
                       if(!lemma.matches(".*;I.*"))
                           lemmaDict.put(wordForm.toLowerCase(), wordTypeAllPoss);

                        break;
                    }
                    k++;
                }

            }
            fd.flush();
            fd.close();


        }catch(Exception exc){

            System.out.println("NAPAKA v vrstici: "+line);
            System.out.println("NAPAKA: " + exc);         }


    }
```

```
    public Dictionary(URL codebase) throws Exception{
         URL dictFilePath = new URL(codebase+dictFileName);
         URLConnection urlConn;
         urlConn = dictFilePath.openConnection();
       urlConn.setDoInput(true);
       urlConn.setUseCaches(false);
       BufferedReader reader=new BufferedReader(new
InputStreamReader(urlConn.getInputStream(), "UTF-8"));

         String line="";
         Hashtable<String, Category> enteredCategories = new Hashtable<String,
Category>(1700);


         for( line = reader.readLine(); line != null; line = reader.readLine()){
             int k = 0, end = 0;
             String wordForm, lemma;

             for(int i = line.length() - 1; i >= 0; i--){

                 if(end == 0){
                    if(line.charAt(i) == '|')
                        end = k + 1;
                 }
                 else if(line.charAt(i) == '|'){
                    wordForm = line.substring(line.length() - k, line.length() - end);
                    lemma = line.substring(0, line.length() - k);
                    //System.out.println(wordForm);

                    String[] poss = lemma.split("[\\|;]");
                    ArrayList<Lemma> wordTypeAllPoss = new ArrayList<Lemma>();

                    for(int ct = 0; ct < poss.length - 1; ct = ct + 2){
                        Category cat;
                        if(enteredCategories.containsKey(poss[ct+1]))
                            cat = enteredCategories.get(poss[ct+1]);
                        else{
                            cat = new Category(Integer.toString(dictionaryID),
poss[ct+1]);
                            enteredCategories.put(poss[ct+1], cat);
                            dictionaryID++;
                        }
                        Lemma lem = new Lemma( poss[ct], cat);
                        wordTypeAllPoss.add(lem);
                    }

                    lemmaDict.put(wordForm, wordTypeAllPoss);
                    if(!lemma.matches(".*;I.*"))
                        lemmaDict.put(wordForm.toLowerCase(), wordTypeAllPoss);

                     break;
                 }
                 k++;
             }

         }


    }

    public String getDictionaryString(String key) throws Exception{
```

```
        StringBuffer strb = new StringBuffer();
        ArrayList<Lemma> wordTypeAllPoss = lemmaDict.get(key);

        if(wordTypeAllPoss != null){
            for(Lemma lem : wordTypeAllPoss)
                 strb.append(lem.categories.allCategoriesString).append("
'").append(lem.neutralForm).append("'").append("\n");

            return strb.toString();
        }
        else
            return null;
    }
}
```

5.5. Element.java

```
package stavcnaanaliza;

import java.util.*;
import java.util.regex.Pattern;


public class Element {

    int ID = 0;
    String type = Word.nullSign;
    String subType = Word.nullSign;
    ArrayList<Word> words;
    Hashtable<String, Element> elements = new Hashtable<String, Element>();


    protected void addSubElement(String name, ArrayList<Word> words)
    {
        Element el = new Element();
        el.words = words;
        elements.put(name, el);
    }

    protected void addSubElement(String name, Word word)
    {
        Element el = new Element();
        ArrayList<Word> nWords = new ArrayList<Word>();
        words.add(word);
        el.words = nWords;
        elements.put(name, el);
    }

    protected Element getSubElement(String name)
    {
        return elements.get(name);
    }

    protected ArrayList<Word> getSubElementWords(String name)
    {
        if(elements.get(name) != null)
            return elements.get(name).words;
        else
```

```
            return null;
    }

    protected boolean removeSubElementWord(String name, Word word)
    {
        Element subel = elements.get(name);

        if(subel != null)
            return subel.words.remove(word);
        else
            return false;
    }

    protected boolean containsCategory(Pattern pat){

        for(Word word : words){
            if(word.canBeOfType(pat))
                return true;
        }

        return false;
    }

}
```

5.6. Lemma.java

```
package stavcnaanaliza;

import java.util.regex.Matcher;
import java.util.regex.Pattern;


public class Lemma {
    String neutralForm;
    Category categories;

    public Lemma(String n, Category cat){
        neutralForm = n;
        categories = cat;
    }

    public boolean canBeOfType(Pattern pat){

        Matcher mat = pat.matcher(categories.allCategoriesString);

        if(mat.find())
            return true;
        else
            return false;
    }
}
```

5.7. Word.java

```
package stavcnaanaliza;
import java.util.*;
import java.util.regex.*;


public class Word {

    final static String nullSign = "\\N";

    protected int ID, sentenceID;

    protected String frontPunctuationMark;
    protected String contents;
    protected String rearPunctuationMark;

    protected String artType;
    protected Lemma wordType;
    protected ArrayList<Lemma> wordTypeAllPoss;
    protected Element sentence;
    protected CmpSentence cmpSentence;


    public Word(String fp, String cont, String rp, ArrayList<Lemma> tAllPoss, int sqlID){
        frontPunctuationMark = fp;
        contents = cont;
        rearPunctuationMark = rp;

        ID = sqlID;
        sentenceID = -1;

        artType = nullSign;
        wordTypeAllPoss = tAllPoss;

        wordType = null;
        sentence = null;
    }

    public boolean isWord(String contents){

        if(this.contents.equals(contents) || this.contents.toLowerCase().equals(contents))
            return true;
        else
            return false;

    }

    public boolean canBeWord(String contents){

        if(wordTypeAllPoss != null){
            for(Lemma lem : wordTypeAllPoss){
                if(lem.neutralForm.equals(contents))
                    return true;
            }

        }
        return false;
    }
```

```
public boolean isOfType(Pattern pat){

    if(wordType == null)
        return false;

    Matcher mat = pat.matcher(wordType.categories.allCategoriesString);

    if(mat.find())
        return true;
    else
        return false;
}

public boolean canBeOfType(String categoryString){

    if(wordTypeAllPoss != null){

        for(Lemma lem : wordTypeAllPoss){
            if(lem.categories.allCategoriesString.contains(categoryString))
                return true;
        }
    }
    return false;
}

public boolean canBeOfType(Pattern pat){

    if(wordTypeAllPoss != null){

        for(Lemma lem : wordTypeAllPoss){
            Matcher mat = pat.matcher(lem.categories.allCategoriesString);

            if(mat.find())
                return true;
        }
    }
    return false;
}

public boolean setWordType(String categoryString){

    if(wordType != null)  //if the wrong lemma already found
        removePreviousWordType();


    if(wordTypeAllPoss != null){

        for(Lemma lem : wordTypeAllPoss){
            if(lem.categories.allCategoriesString.contains(categoryString)){
                wordType = lem;
                return true;
            }
        }
    }
    return false;
}

public boolean setWordType(Lemma lem){

    if(wordType != null)  //if the wrong lemma already found
        removePreviousWordType();
```

```
    wordType = lem;
    return true;
}

public boolean setWordTypeOnePoss(Pattern pat){

    if(wordType != null)  //if the right lemma already found
        removePreviousWordType();

    if(wordTypeAllPoss != null){

        for(Lemma lem : wordTypeAllPoss){
            Matcher mat = pat.matcher(lem.categories.allCategoriesString);

            if(mat.find()){
                wordType = lem;
                return true;
            }
        }
    }
    return false;
}

public boolean setWordTypeOnePossWithoutRemovingPreviousType(Pattern pat){

    if(wordTypeAllPoss != null){

        for(Lemma lem : wordTypeAllPoss){
            Matcher mat = pat.matcher(lem.categories.allCategoriesString);

            if(mat.find()){
                wordType = lem;
                return true;
            }
        }
    }
    return false;
}

 public boolean setPreferredWordType(Pattern pat1, Pattern pat2){

    if(wordType != null)  //if the right lemma already found
        removePreviousWordType();

    if(wordTypeAllPoss != null){

        for(Lemma lem : wordTypeAllPoss){
            Matcher mat = pat1.matcher(lem.categories.allCategoriesString);

            if(mat.find()){
                wordType = lem;
                return true;
            }
        }
        for(Lemma lem : wordTypeAllPoss){
            Matcher mat = pat2.matcher(lem.categories.allCategoriesString);

            if(mat.find()){
                wordType = lem;
                return true;
            }
        }
```

```
        }
        return false;
    }


    public boolean setAnotherPossibleWordTypeBeside(Pattern pat) {

        if(wordTypeAllPoss != null){

            for(Lemma lem : wordTypeAllPoss){
                Matcher mat = pat.matcher(lem.categories.allCategoriesString);

                if(!mat.find()){
                    wordType = lem;
                    return true;
                }
            }
        }
        return false;
    }

    public boolean hasAnotherPossibleWordTypeBeside(Pattern pat) {

        if(wordTypeAllPoss != null){

            for(Lemma lem : wordTypeAllPoss){
                Matcher mat = pat.matcher(lem.categories.allCategoriesString);

                if(!mat.find()){
                    return true;
                }
            }
        }
        return false;
    }


    public void setWordArticle(String artType){

        this.artType = artType;

    }

    boolean checkCongruenceWithVerb( ArrayList<Word> verbPhrase, Lemma nounLemma) {

        for(Word verbMorphem : verbPhrase){
            for(Lemma verbLemma : verbMorphem.wordTypeAllPoss){
                if(verbLemma.canBeOfType(Category.personalVerbForm)){
                    if(nounLemma.canBeOfType(Category.personalPronoun)){

if(nounLemma.categories.grammNo.equals(verbLemma.categories.grammNo) &&
nounLemma.categories.person.equals(verbLemma.categories.person)){

if(!verbMorphem.wordType.categories.person.equals(verbLemma.categories.person) ||
!verbMorphem.wordType.categories.grammNo.equals(verbLemma.categories.grammNo))
                                verbMorphem.wordType = verbLemma;
                            return true;
                        }
                    }
                    else{

if(nounLemma.categories.grammNo.equals(verbLemma.categories.grammNo) &&
```

```
verbLemma.categories.person.equals("c")){

if(!verbMorphem.wordType.categories.grammNo.equals(verbLemma.categories.grammNo))
                                verbMorphem.wordType = verbLemma;
                            return true;
                        }
                    }
                }
                else if(verbLemma.canBeOfType(Category.pastParticiple) &&
verbPhrase.size() == 1){  //if past participle - useful in cases when a verb is omitted
                    if(nounLemma.categories.grammNo.equals(verbLemma.categories.grammNo)
&& nounLemma.categories.gender.equals(verbLemma.categories.gender)){
                            verbMorphem.wordType = verbLemma;
                            return true;
                    }
                }
            }

        }
        return false;
    }

    boolean checkCongruencePredicatePhraseWithVerb( ArrayList<Word> verbPhrase, Lemma
nounLemma) {

        for(Word verbMorphem : verbPhrase){
            for(Lemma verbLemma : verbMorphem.wordTypeAllPoss){
                if(verbLemma.canBeOfType(Category.personalVerbForm)){
                    if(nounLemma.canBeOfType(Category.personalPronoun)){

if(nounLemma.categories.grammNo.equals(verbLemma.categories.grammNo) &&
nounLemma.categories.person.equals(verbLemma.categories.person)){
                            return true;
                        }
                    }
                    else{

if(nounLemma.categories.grammNo.equals(verbLemma.categories.grammNo) &&
verbLemma.categories.person.equals("c")){
                            return true;
                        }
                    }
                }
            }
        }
        return false;
    }

    private void removePreviousWordType() {

        this.sentence.removeSubElementWord(artType, this);
    }

}
```

6. Zaključek

Razvoj algoritma kljub začetnim težavam in pomanjkljivostim kaže v pravo smer. Posebno obetavna se zdi vključitev vezljivostnega slovarja, ki bi omogočila učinkovito ločevanje med predmeti in prislovnimi določili, hkrati pa bi v algoritem dodala tudi pomensko občutljivost, ki je sedaj ni. Pri nadaljnjem razvoju in uporabi algoritma za daljše povedi bi bilo tudi nujno vključiti dognanja in izkušnje Domna Marinčiča pri strojnem razčlenjevanju besedila (MARINČIČ 2008). Ozko grlo algoritma oz. tudi oblikoslovnega označevanja pa ostaja zahteva po pravopisno in slovnično pravilnem zapisu besedila, kar zožuje njegovo uporabnost na področje zbornega knjižnega jezika.

7. Viri

- BUß, Frank, 2002: Applet <–> Servlet communication. http://www.frank-buss.de/echoservlet/index.html. Stanje: 12. 10. 2009.
- *Dokumentacija podatkovnega strežnika in poizvedbenega jezika MySQL.* http://dev.mysql.com/doc/. Stanje: 12. 10. 2009.
- *Dokumentacija programskega jezika Java*. http://java.sun.com/javase/6/docs/api/. Stanje: 12. 10. 2009.
- *Dokumentacija spletnega javanskega strežnika Apache Tomcat 6.0.* http://tomcat.apache.org/tomcat-6.0-doc/index.html. Stanje: 12. 10. 2009.
- JAKOPIN, Primož, 2001: *Oblikoslovno označevanje.* http://bos.zrc-sazu.si/oblikoslovno_oznacevanje.html. Stanje: 12. 10. 2009.
- JAKOPIN, Primož, 1999: *Zgornja meja entropije pri leposlovnih besedilih v slovenskem jeziku: doktorska disertacija*. [P. Jakopin]: Ljubljana, 1999. 35–45.
- TOPORIŠIČ, Jože, 2001: *Slovenska slovnica*. Obzorja: Maribor, 2000. 469–693.
- MARINČIČ, Domen, 2008: *Strojno razčlenjevanje besedila z iskanjem stavkov in naštevanj: doktorska disertacija*. [D. Marinčič]: Ljubljana, 2008.
- ŽELE, Andreja, 2008: *Vezljivostni slovar slovenskih glagolov*. Založba ZRC SAZU: Ljubljana, 2008.

Izvleček

Računalništvo se v zadnjih letih uspešno povezuje z jezikoslovjem. Še posebej učinkovita je postala ta povezava na področju skladnje, kjer zbirke računalniško obdelanih besedil, korpusi, omogočajo hitra iskanja in preverbe jezikoslovnih hipotez. Problem korpusov pa so pogosto za računalništva neveščega uporabnika neprijazni tekstovni iskalniki in morfološko-skladenjska neoznačenost besedila. Oba problema sta za sintetične jezike, kot je tudi slovenščina, še posebno pereč problem. Cilj diplomskega dela je bilo razviti učinkovit algoritem za analizo kratke povedi v slovenščini (do treh besed) s pomočjo oblikoslovnega označevalnika Primoža Jakopina (JAKOPIN 1999) in algoritem implementirati v nekem programskem jeziku skupaj z grafičnim vmesnikom za iskanje po bazi označenih povedi. Časovna zahtevnost razvitega algoritma je reda k*O(n), pri čemer je *n* število besed v povedi, *k* pa konstantno število iteracij pri analizi besed določene povedi. Algoritem ima določene omejitve: je povsem odvisen od pravilnosti oblikoslovnega označevalnika, zaradi osredotočenosti na kratko poved ne prepozna naštevanj in prirednih besednih zvez, zaradi neupoštevanja konteksta povedi ne more povsem natančno ločiti med osebkom in povedkovim določilom (predpostavi se, da je osebek prvi stavčni člen), nadalje algoritem za vsako poved določi le eno možnost analize oz. interpretacije, in sicer prvo, zaradi svoje formalne narave algoritem tudi ne more uspešno ločevati med predmeti in prislovnimi določili. Analiza daljših povedi je sicer možna že na trenutni stopnji razvoja, ni pa še bila preizkušena.

Abstract

In recent years computing has been successfully applied to problem solving in linguistics. This statement is especially valid in the area of syntax, where computer-processed collections of texts, corpora, allow instant searching and verification of linguistic assumptions. For the users unskilled in computing, complicated user interfaces of text search engines and morpho-syntactically unmarked texts are often a problem. This is particularly acute for synthetic languages such as Slovenian. The aim of the thesis was to develop an efficient algorithm for the analysis of short sentences in Slovenian language (up to three words) using morphological tagging by Primož Jakopin (JAKOPIN 1999) and to implement the algorithm in a programming language with a graphical user interface for searching the database of syntactically and morphologically marked sentences. The time complexity of the developed algorithm is of the order k*O(n), where n is the number of words in a sentence and k is the constant number of iterations in the sentence analysis. Algorithm has certain limitations: it is entirely dependent on the effectiveness of the morphological word tagger, as it is focused on a short sentence it cannot recognize enumerations and coordinate phrases, as it does not delve into the context of the sentence the algorithm cannot exactly distinguish between subject and predicate phrase (it is assumed that the first found noun article in nominative is the subject and the second the verbal phrase), furthermore the algorithm finds only one possibility of the often many that could result from the analysis of each sentence. Moreover, the algorithm cannot, due to its formal nature, effectively distinguish between objects and adverbial phrases. The analysis of longer sentences than three words is already possible at the current stage of development, but has not been tested yet.

Izjava o avtorstvu

Študent **Robert Jakomin** izjavljam, da sem avtor tega diplomskega dela, ki sem ga napisal pod mentorstvom doc. dr. Primoža Jakopina.

V Ljubljani, 12. 10. 2009 Podpis:_____