Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Rozšírenie prototypu sémantického stroja transparentnej intenzionálnej logiky

Diplomová práca

2021 **Bc. Branislav Bednár**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Rozšírenie prototypu sémantického stroja transparentnej intenzionálnej logiky

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	9.2.1. Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	Ing. Zuzana Bilanová, PhD.

Košice 2021 **Bc. Branislav Bednár**

Abstrakt v SJ

Cieľom diplomovej práce je rozšírenie prototypu sémantického stroja používajúceho transparentnú intenzionálnu logiku. Prototyp sémantického stroja predstavuje konzolovú aplikáciu, ktorá dokáže pomocou externého syntaktického analyzátora, Stanford CoreNLP, analyzovať a formulovať význam viet prirodzeného jazyka. Prototyp obsahuje implementáciu typovej analýzy pre základné a niektoré rozšírené typy TIL, a taktiež tvorbu konštrukcií významu vety.

Pre úspešné rozšírenie je potrebné aplikovanie ďalších funkcionalít, ktoré transparentná intenzionálna logika ponúka a ich zapracovanie do typovej analýzy a tvorby konštrukcie. Taktiež je nevyhnutné technické vylepšenie prototypu pre potreby budúceho používania. To znamená zmenu konzolovej aplikácie na webovú službu, pokrytie implementácie testami, modularizáciu aplikácie do modulov podľa funkcionality.

Výsledok daných rozšírení predstavuje sémantický stroj so schopnosťou pokryť väčšie množstvo viet prirodzeného jazyka, nielen jednoduchých ale aj zložitejších. Taktiež výsledkom daných rozšírení bude plnohodnotný produkt pripravený na používanie.

Kľúčové slová v SJ

Flask, logická analýza prirodzeného jazyka, programovanie, Python, sémantická analýza, sémantický stroj

Abstrakt v AJ

The diploma thesis aims to extend the prototype of the semantic machine, which uses transparent intensional logic. The prototype of the semantic machine is a console application, which can analyze and form the meaning of the natural language sentences by an external syntax analyzer, Stanford CoreNLP. The prototype covers the implementation of type analysis for some basic and also extended TIL types and the creation of sentence meaning construction.

For successful extending of prototype, there is a need to apply other functionalities which are offered by transparent intensional logic and also their implementation within type analysis and construction creation. Technical improvements

are also needed for future usage. It means a change of the console application to web service, covering implementation with tests, modularize application to logic modules by its functionality.

The result of certain extensions represents a semantic machine with the ability to cover more natural language sentences, not only simple ones but also more complicated ones. Also, the result of this thesis should be the product ready to use.

Kľúčové slová v AJ

Flask, logical analysis of natural language, programming, Python, semantic analysis, semantic machine

Bibliografická citácia

BEDNÁR, Branislav. *Rozšírenie prototypu sémantického stroja transparentnej intenzionálnej logiky*. Košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky, 2021. 55s. Vedúci práce: Ing. Zuzana Bilanová, PhD.

61137

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra počítačov a informatiky

ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **Informatika**
Študijný program: **Informatika**

Názov práce:

Rozšírenie prototypu sémantického stroja transparentnej intenzionálnej logiky

Extension of semantic machine prototype of transparent intensional logic

Študent: **Bc. Branislav Bednár**
Školiteľ: **Ing. Zuzana Bilanová, PhD.**
Školiace pracovisko: **Katedra počítačov a informatiky**
Konzultant práce:
Pracovisko konzultanta:

Pokyny na vypracovanie diplomovej práce:

1. Naštudovanie možností transparentnej intenzionálnej logiky ako nástroja pre automatizované spracovanie viet prirodzeného jazyka.
2. Analýza súčasného stavu sémantického stroja z hľadiska možnosti rozšíriť jeho implementáciu tak, aby bol schopný zväčšiť množinu analyzovaných viet prirodzeného jazyka. Identifikácia nedostatkov pôvodného riešenia.
3. Návrh algoritmu umožňujúceho spracovanie vybraných rozšírení sémantického stroja.
4. Implementácia vybraných rozšírení danej logiky do sémantického stroja a otestovanie použiteľnosti vytvoreného riešenia.
5. Spracovanie dokumentácie podľa pokynov vedúceho práce.

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 23.04.2021
Dátum zadania diplomovej práce: 30.10.2020

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -12-

prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som záverečnú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry.

Košice, 23.4.2021 .....

Vlastnoručný podpis

Poďakovanie

Rád by som v prvom rade poďakoval svojej rodine za podporu počas nielen univerzitného štúdia, svojej snúbenici, ktorá mi bola neodmysliteľnou oporou a priateľom, s ktorými som prežil svoje univerzitné životné obdobie. Vďaka patrí pedagógom, ktorých som mal možnosť počas štúdia stretnúť, ktorí robili svoju prácu najlepšie ako len vedeli a snažili sa nám odovzdať čo naviac vedomostí a skúseností.

Veľká vďaka patrí Ing. Zuzane Bilanovej, PhD., za príkladné vedenie počas spracovávania bakalárskej a následne aj prezentovanej diplomovej práce. Ďakujem za jej čas, za ochotu kedykoľvek o práci diskutovať, prípadne pomôcť, a za odbornosť a profesionalitu pri vykonávaní svojej práce.

Obsah

1 Úvod **1**

2 Formulácia úlohy **3**

3 Analýza logického aparátu **4**
3.1 Transparentná intenzionálna logika 4
3.1.1 Typy 5
3.1.2 Konštrukcie 7
3.1.3 Použitie 8

4 Analýza súčasného stavu **11**
4.1 Technologické riešenie 12
4.2 Schopnosti aktuálneho riešenia 12
4.3 Aktuálne algoritmy pre tvorbu konštrukcií 14
4.3.1 Všeobecný algoritmus tvorby konštrukcie 15
4.3.2 Algoritmus konštruovania vlastnosti objektu 15
4.3.3 Algoritmus konštruovania konjunkcie 16
4.3.4 Algoritmus konštruovania vlastnosti hodnoty 17

5 Návrh rozšírení aktuálneho riešenia **21**
5.1 Návrh technického rozšírenia aktuálneho riešenia 21
5.1.1 Modularizácia 21
5.1.2 Práca s výstupom syntaktickej analýzy 22
5.1.3 Testovanie sémantického stroja 23
5.2 Návrh funkčného rozšírenia aktuálneho riešenia 24
5.2.1 Premenné 24
5.2.2 Kvantifikátory 25

5.2.3 Implikácia 25

6 Rozšírenie sémantického stroja 26
6.1 Testovanie 26
6.1.1 Unitest 27
6.1.2 Pytest 27
6.2 CI/CD 28
6.3 Tvorba webovej služby 29
6.3.1 Výber webového aplikačného rámca 31
6.3.2 Používanie webovej služby 31
6.4 Spracovanie konjunkcie 33
6.4.1 Algoritmus konštruovania konjunkcie 34
6.5 Spracovanie implikácie 37
6.5.1 Algoritmus pre identifikáciu implikácie vo vete 37
6.5.2 Algoritmus konštruovania implikácie 39
6.6 Spracovanie premenných a kvantifikátorov 41
6.6.1 Rozšírenie typovej analýzy 41
6.6.2 Algoritmus konštruovania premenných a kvantifikátorov . . 43
6.7 Riešenie funkčných nedostatkov 45

7 Vyhodnotenie 48
7.1 Funkčné rozšírenia 48
7.2 Technické rozšírenia 49
7.3 Nedostatky riešenia 50
7.4 Možnosti budúceho rozšírenia 50

8 Záver 52

Literatúra 53

Zoznam skratiek 56

Zoznam príloh 57

A Používateľská príručka 59
A.1 Štruktúra projektu 59
A.2 Systémové požiadavky 59

A.3 Konfigurácia prostredia 60
A.4 Spustenie webovej služby 61
A.5 Používanie webovej služby 62
A.6 Negatívne scenáre 64

B Systémová príručka **67**
B.1 Systémové požiadavky 67
B.2 Štruktúra projektu 67
B.3 Implementácia 69
B.3.1 semantic_machine_module.py 69
B.3.2 corenlp_module.py 69
B.3.3 type_analysis_module.py 70
B.3.4 construction_creation_module.py 70
B.3.5 exceptions_module.py 71
B.3.6 Ostatné 72
B.4 Testovanie 72

C Príklady spracovaných viet **74**

Zoznam obrázkov

3.1 Použitie transparentnej intenzionálnej logiky na logickú analýzu vety prirodzeného jazyka [14] 10

4.1 Potrebné kroky pre implementáciu prototypu 12
4.2 Spustenie servera Stanford CoreNLP 13
4.3 Súčasný stav implementácie sémantického stroja transaprentnej intezionálnej logiky 14
4.4 Ukážka identifikácie závislostí z výstupu syntaktickej analýzy . . . 15
4.5 Ukážka identifikácie slov z výstupu syntaktickej analýzy 16
4.6 Algoritmus pre tvorbu konštrukcie 17
4.7 Algoritmus pre konštruovanie vlastnosti objektu 18
4.8 Algoritmus pre konštruovanie konjunkcie 19
4.9 Algoritmus pre konštruovanie vlastnosti hodnoty 20

5.1 Návrh modularizácie aplikácie 22
5.2 Práca s JSON objektom 23

6.1 Výstup statickej analýzy kódu 29
6.2 Výstup úlohy, ktorá spúšťa jednotkové testy 30
6.3 Pokrytie kódu testami 30
6.4 Používanie sémantického stroja ako webovej služby 32
6.5 Použitie dotazového parametra *output=json* 33
6.6 Základné závislosti z výstupu syntaktickej analýzy 34
6.7 Výstup syntaktickej analýzy vety obsahujúcej konjunkciu 35
6.8 Rozšírené závislosti z výstupu syntaktickej analýzy 35
6.9 Nový algoritmus pre spracovanie konjunkcií 36
6.10 Nový algoritmus pre tvorbu konštrukcie 36

6.11 Syntaktická analýza viet obsahujúcich implikáciu 38
6.12 Výstup syntaktickej analýzy vety obsahujúcej implikáciu 39
6.13 Algoritmus konštruovania implikácie 40
6.14 Syntaktická analýza viet s potrebou použitia premennej 42
6.15 Výstup syntaktickej analýzy vety s potrebou použitia premennej . . 43
6.16 Všeobecný algoritmus pre konštruovanie rozšírených typov 44

A.1 Štruktúra projektu webovej služby 60
A.2 Spustenie servera Stanford CoreNLP 61
A.3 Spustenie webovej služby sémantického stroja 62
A.4 Prevolanie informačného koncového bodu 62
A.5 Používanie sémantického stroja ako webovej služby 63
A.6 Použitie dotazového parametra *output=json* 64
A.7 Nevalidný dotazový parameter 65
A.8 Nepovolený znak v tele požiadavky 65

B.1 Štruktúra projektu webovej služby 68

Zoznam tabuliek

3.1 Často používané extenzionálne operátory v TIL [13] 8
3.2 Typy konštrukcií v TIL 9

Zoznam zdrojových kódov

1 Príklad použitia knižnice *unittest* 27
2 Príklad použitia knižnice *pytest* 28

1 Úvod

Logická analýza prirodzeného jazyka predstavuje v súčasnosti odbor, ktorý sa znova stáva pre informatikov, logikov a matematikov lákavým z pohľadu riešenia, respektíve optimalizovania riešenia dobre známych problémov. To je dôsledkom toho, že v dnešnej dobe existujú rôzne platformy a aplikácie, ktoré aktívne pracujú s prirodzeným jazykom. Poznáme napríklad aplikácie, ktoré robia korektúru textu, aplikácie, ktoré ponúkajú návrhy na zlepšenie nami napísaného textu, či už gramatické alebo obsahové a taktiež sú tu aplikácie, ktoré analyzujú text za účelom analytického spracovania dát. Všetky vymenované druhy platforiem sa zaoberajú spracovaním prirodzeného jazyka vo veľkom, vďaka možnostiam, ktoré ponúka správne použitie umelej inteligencie alebo strojového učenia. Tieto dve odvetvia sa do záujmového rámca širšej skupiny informatikov dostali len pred nedávnom, v dobe, kedy prestalo byť problémom zadovážiť a používať výpočtový výkon potrebný na tieto výpočtovo náročné úlohy.

Aby bolo spracovanie viet prirodzeného jazyka predmetné a správne, je potrebné na začiatku tohto procesu vykonať logickú analýzu viet prirodzeného jazyka, čo bolo aj predmetom mojej bakalárskej práce. Daná analýza je v rámci viet vykonávaná na dvoch základných úrovniach a to syntaktickej a sémantickej.

V danej práci bolo primárnym cieľom dôkladné naštudovanie domény problematiky logickej analýzy prirodzeného jazyka, naštudovanie existujúcich princípov a systémov zaoberajúcich sa touto problematikou, porovnanie a naštudovanie logických aparátov vhodných na logickú analýzu prirodzeného jazyka a následné vytvorenie prototypu sémantického stroja s použitím vhodného logického aparátu.

Vytvorenie prototypu po vyriešení viacerých implementačne náročných úloh, ako napríklad vykonanie typovej analýzy vety prirodzeného jazyka, implementovanie tvorby konštrukcie významu vety, bolo úspešné. Fungujúci prototyp síce

veľmi dobre demonštroval ciele bakalárskej práce, no stále ostali otvorené výzvy v podobe možností, ktoré zvolený logický aparát ponúka, a ktoré by umožnili spracovanie viacerých druhov viet prirodzeného jazyka. Otvorené sú aj technické možnosti, pomocou ktorých by bolo možné spraviť sémanticky stroj ešte lepším a ľahšie použiteľným. Preto uvažovanie o výbere témy pre diplomovú prácu netrvalo dlho a teda rozhodol som sa v rámci nej implementovať rozšírenie sémantického stroja, ktoré by naplno využilo potenciál transparentnej intenzionálnej logiky a stalo sa tak nápomocným, či už v rámci komerčného alebo akademického využitia.

2 Formulácia úlohy

Úspešné rozšírenie prototypu sémantického stroja transparentnej inenzionálnej logiky vyžaduje naštudovanie možností transparentnej intenzionálnej logiky ako nástroja pre automatizované spracovanie viet prirodzeného jazyka.

V ďalších krokoch je nevyhnutné analyzovanie súčasného stavu prototypu sémantického stroja z hľadiska možnosti rozšírenia jeho implementácie tak, aby bol schopný zväčšiť množinu analyzovaných viet prirodzeného jazyka a taktiež identifikovanie nedostatkov pôvodného riešenia. Táto dôkladná analýza umožní navrhnúť algoritmy umožňujúce spracovanie vybraných rozšírení sémantického stroja, ktoré budú v ďalšom štádiu implementované. Dôležitou časťou implementácie vybraných rozšírení je taktiež testovanie použiteľnosti vytvoreného riešenia.

Poslednou dôležitou časťou diplomovej práce je spracovanie dokumentácie nového riešenia podľa pokynov vedúceho práce.

3 Analýza logického aparátu

Slovenská a česká, prípadne ešte pred niekoľkými dekádami československá akademická obec, zaznamenáva množstvo vedeckých úspechov nielen doma, ale vo veľkej miere aj celosvetovo. Poznáme dnes preto mnoho úspešných akademikov, ktorí sa vďaka kvalite svojej práce podieľajú na riešení najväčších globálnych výziev. Hovoríme o odvetviach, akými sú okrem počítačových vied aj zdravotníctvo, letectvo a podobne.

V dvadsiatom storočí bol jedným z najzvučnejších mien československej akademickej obce Pavel Tichý [1]. Rodák z Brna, sa mohol pýšiť pôsobením na rôznych európskych či svetových univerzitách, medzi ktoré patria napríklad: Karlova univerzita, University of Exter (Spojené kráľovstvo), University of Dunedin, University of Otago (oba Austrália) a University of Pittsburgh.

Pavel Tichý sa zaslúžil o vytvorenie dôležitého základu pre veľmi úspešný a rozsiahly výskumný program v Československu so zameraním na sémantiku a logiku prostredníctvom svojho najznámejšieho diela, ktorým bola transparentná intenzionálna logika.

Prvý krát boli prístupy k sémantike v rámci logiky spomínané v prvej polovici dvadsiateho storočia [1]. Výrazným sa stal až prístup, ktorý nastolil Richard Montague v sedemdesiatych rokoch [2]. Tento prístup má v porovnaní s Tichého intenzionálnym prístupom zhodný základ, no odborná verejnosť sa zhoduje na tom, že prístup Pavla Tichého k sémantike je oveľa elegantnejší, precíznejší a menej obmedzujúci.

3.1 Transparentná intenzionálna logika

Transparentná intenzionálna logika predstavuje logický aparát vyššieho rádu založený na Fregeho objektovej sémantike. Tá sa venuje vyjadreniu významu slov

vo vete definovaním vzťahov medzi výrazmi [3].

V prirodzenom jazyku je zmysel výrazu explikovaný ako abstraktná procedúra, ktorá nesie názov konštrukcia. Konštrukcia vety prirodzeného jazyka predstavuje štruktúrovaný objekt, ktorý je zložený z podkonštrukcií jednotlivých výrazov v danej vete. Primárne použitie *TIL* je na vykonávanie logickej analýzy prirodzeného jazyka, ktorej začiatky siahajú do konca 19. storočia.

Ako vlastne vznikla potreba vykonania analýzy prirodzeného jazyka? Ak uvážime umelé jazyky, tie problém s logickou analýzou nemajú. Či už hovoríme o umelých jazykoch ako sú napríklad matematika, fyzika, prípadne nejaký programovací jazyk, tie majú svoju sémantiku striktne danú pravidlami a definíciami. Na druhej strane, prirodzený jazyk vznikal živelne na základe potrieb ľudí. To znamená, že najprv vznikal jazyk a pravidlá vznikali dodatočne. Z toho dôvodu je nám dnes umožnené jazykom vyjadriť akúkoľvek myšlienku, úvahu, názor, ktoré môžu mať vzhľadom na ďalšie rôzne faktory úplne odlišnú sémantiku. Ak chceme sémantiku vety prirodzeného jazyka dostatočne dobre vyjadriť, je potrebné vykonanie logickej analýzy. Tá nám pomôže s odstránením sémantických nejednoznačností a taktiež význam vety bude jednoznačnejšie identifikovaný za pomoci lepších nástrojov analýzy.

Uplatnenie transparentnej intenzionálnej logiky nie je obmedzené iba na použitie v rámci prirodzeného jazyka. Je môžné jej použitie v rôznych ďalších oblastiach akými sú napríklad: analytická filozofia [4] prípadne HIT [5], počítačové vedy [6] atď. Možnosti jej uplatnenia sú široké vďaka viacerým parametrom, ktorými sú precízne a priame zovšeobecnenie základných ideí, možnosť softvérovej implementácie vďaka použitiu lambda kalkulu [7], možnosť riešenia veľkého počtu problémov jedným logickým aparátom.

Transparentná intenzionálna logika nie je totožná s Tichého teóriou typov [8]. Je možné skonštatovať, že transparentná intenzionálna logika predstavuje časť Tichého teórie typov, ktorá obsahuje indivíduá, možné svety, pravdivostné hodnoty a časové okamihy [7].

3.1.1 Typy

Teória typov predstavuje vhodný formálny logický jazyk pre formalizáciu matematiky a jej pridružených vedných disciplín. Primárnym cieľom však pri jej vzniku bolo, aby sa zabránilo nejasnostiam a paradoxom v logických systémoch.

Konkrétne v tomto prípade ide o teóriu typov, ktorú deklaroval významný americký matematik Alonzo Church v roku 1940, je teda možné hovoriť o Churchovej teórii typov [9]. Pomocou Churchovej teórie typov je možné reprezentovať znalosť informačných systémov, ktoré využívajú umelú inteligenciu ako napríklad systémy na overovanie matematických dôkazov, čo z nej robí dôležitý nástroj v rámci formálnej sémantiky. Churchova teória typov aj napriek tomu, že je jednoduchšia no zároveň viac všeobecná ako Russelova teória typov, tiež patrí medzi logiky vyššieho rádu [10]. Na základe vyššie uvážených výhod, typy v transparentnej intenzionálnej logike vychádzajú práve z Churchovej teórie typov.

Atomické typy [11] môžu byť nasledujúce:

Indivíduum - Nie je nutné si pod indivíduom predstavovať prvok skupiny(triedy) dediaci všetky jej vlastnosti. Konkrétnemu indivíduu tak prislúchajú vlastnosti, ktoré ho v rámci výrazu popisujú. Taktiež je možné konštatovať, že množina individuí je zdieľaná všetkými možnými svetmi [12]. Indivíduum je reprezentované symbolom ι.

Časový moment - predstavuje vlastnosť kontinuity v čiastočnej závislosti na určitej entite. Spomínanou entitou môže byť v tomto prípade napríklad *indivíduum*. Časový moment je reprezentovaný symbolom τ.

Pravdivostná hodnota - sada dvoch prvkov, ktoré predstavujú pravdivostnú hodnotu. Tento typ je možné používať v kombinácii so štandardnými logickými operáciami konjunkciou, disjunkciou, implikáciou a negáciou. Operácie môžu byť následne reprezentované ako objekty typu $o o$(pre negáciu, respektíve unárne logické operátory) alebo $o o o$(pre konjunkciu, disjunkciu, implikáciu, respektíve binárne logické operátory). Typ pravdivostnej hodnoty je reprezentovaný symbolom o.

Možný svet - predstavuje typ, ktorý hovorí o možnosti vzájomného vzťahu medzi indivíduom a jeho vlastnosťou vzhľadom na čas a svet. Typ možný svet je reprezentovaný symbolom ω.

Každý prvok vety, nad ktorou je vykonávaná logická analýza práve prostredníctvom TIL, musí mať priradený typ. Bežným prípadom je, že množina atomických typov nemusí stačiť na zanalyzovanie vety prirodzeného jazyka. V takom

prípade je objektu priradený niektorý z odvodených typov [13], ktorými môžu byť napríklad:

Propozícia má definovanú pravdivostnú hodnotu na základe sveta a času. Príkladom na takýto typ môže byť veta *„Matej je ostrihaný."*. Tento typ je reprezentovaný symbolom $((o\tau)\omega)$.

Intenzionálna rola predstavuje funkciu, ktorá neprináleží konkrétnej osobe, indivíduu. V závislosti na svete a čase sa *indivíduum* v tejto funkcii líši. Príkladom pre tento typ je veta *„Víťaz Maratónu mieru."*. Typ je reprezentovaný ako $((\iota\tau)\omega)$.

Vlastnosť vyjadruje intenzionálnu črtu, ktorú *indivíduum* v danom svete a čase má alebo nemá. Následne je možné *vlastnosť* vyjadriť triedou *indivíduí*, ktoré danou vlastnosťou disponujú v konkrétnom svete a čase. Príkladom je slovné spojenie *„nebyť unavený"*. Typ je reprezentovaný ako $(((o\iota)\tau)\omega)$.

Chronológia zobrazuje to, ako sa v čase mení ξ-objekt, kde ξ reprezentuje ľubovoľný typ. Príkladom môže byť slovo *„predvčerom"* a typ je reprezentovaný ako $(\xi\tau)$.

Intenzia vyjadruje závislosť ξ-objektu na možnom svete a čase. Dôsledkom aplikovania intenzie je konštrukcia, ktorá sa nazýva intenzionálny zostup. Typ intenzia môže byť reprezentovaný napríklad ako $((\xi\tau)\omega)$. Na druhej strane, ak ξ nepredstavuje intenziu, v takom prípade ide o extenziu a v tom prípade čas a svet nemá vplyv na stav objektu. Najlepším príkladom na extenziu sú všetky matematické symboly, operácie a konštanty.

V komplexnejších konštrukciách TIL je možný aj výskyt logických operátorov. Najčastejšie používané sa nachádzajú v tabuľke 3.1.

3.1.2 Konštrukcie

Konštrukcia predstavuje abstraktnú procedúru, ktorá popisuje, ako získať štruktúru myšlienky vety prirodzeného jazyka. Danou procedúrou je následne možné interpretovať význam vety a jej jednotlivých lexikálnych jednotiek. Typy konštrukcií popísané v tabuľke 3.2 je možné aplikovať rekurzívne, čo umožní vytvorenie komplexnejších konštrukcií.

Operácia	Typ	Notácia	Popis
$\wedge, \vee, \Rightarrow$, ...	(ooo)	${}^0A \wedge^0 B$	Logické operátory. Sú zvyčajne zapísané v infixnej forme namiesto prefixnej, ktorá je typická pre λ-kalkul.
$\neg$	(oo)	$[\neg^0 A]$	Negácia
$\subset_\xi$	$(o(o\xi)(o\xi))$	$[{}^0A \subset \xi\ {}^0B]$	Vzťah podmnožiny medzi triedami objektov typu ξ.
$\Rightarrow$	$(o\pi\pi)$	$[{}^0A \Rightarrow^0 B]$	Vzťah implikácie medzi dvoma propozíciami.
Π^ξ	$(o(o\xi))$	$\forall x...$	Univerzálny kvalifikátor zapísaný v štandardnej notácii.
Σ^ξ	$(o(o\xi))$	$\exists x...$	Extenzionálny kvantifikátor zapísaný v štandardnej forme.
I^ξ	$(\xi(o\xi))$	γ x..	Singularizátor hovorí o jedinom x, ktoré ...; zapísané v krátkej notácií.

Tabuľka 3.1: Často používané extenzionálne operátory v TIL [13]

3.1.3 Použitie

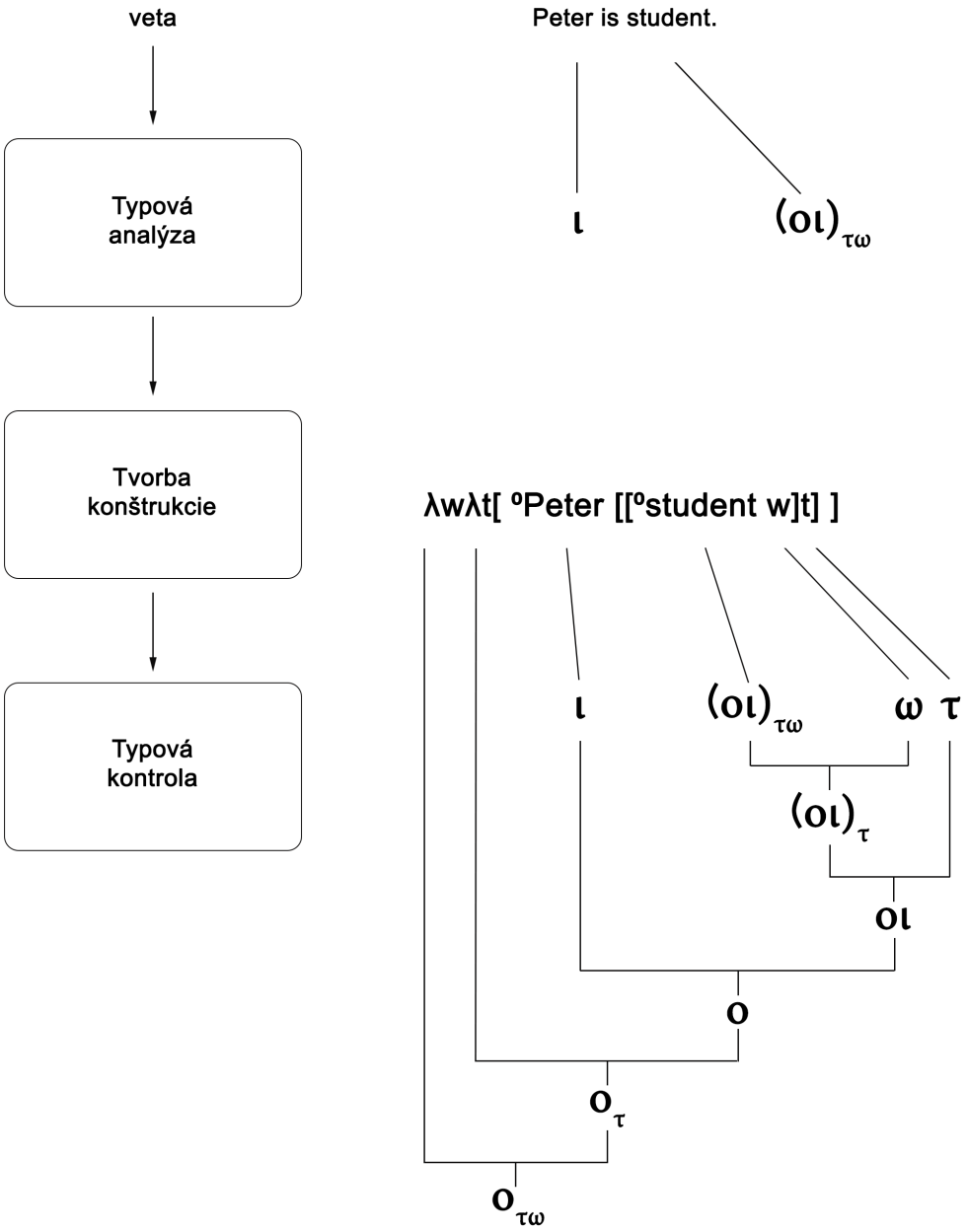
Spôsob, akým pristupuje transparentná intenzionálna logika k logickej analýze vety prirodzeného jazyka, je demonštrovaný na obrázku 3.1.

V prvom kroku sa každej lexikálnej jednotke priradí typ. Následne je vytvorená konštrukcia, reprezentujúca význam vety. Posledný krok predstavuje spätné overenie správnosti vytvorenej konštrukcie.

Po predstavení Tichého myšlienok, ktoré reflektoval do tvorby TIL, je možné tieto teoretické poznatky využiť pri návrhu a následne implementácii funkčných rozšírení prototypu sémantického stroja.

Typ	Popis
Premenná	Predstavuje atomický typ konštrukcie, ktorý tvorí základ všetkých ostatných konštrukcií. Takéto konštrukcie sú reprezentované jednoducho znakmi ako napríklad x,y,z.
Trivializácia	Predstavuje najjednoduchší typ konštrukcie, ktorého cieľom je zobrať objekt a vytvoriť konštrukciu z jeho hodnoty. Ak teda máme objekt X, potom jeho konštrukcia bude vyzerať nasledovne: ${}^0X$.
Kompozícia	Je možné vysvetliť ako aplikáciu funkcie na jej argument. Uvažujme teda funkciu s argumentom x. Konštrukcia pre takúto funkciu by vyzerala následovne: $[Fx]$.
Uzáver	Je možné vysvetliť ako abstrakciu funkcie. Ak teda máme funkciu s argumentom x, ktorú na tento parameter aplikujeme, potom uzáverom môžme označiť samotnú funkciu. Takýto uzáver je možné v konštrukcii vyjadriť nasledovnou reprezentáciou: $\lambda x.Fx$.
Prevedenie	Predstavuje vykonanie konštrukcie. Majme teda konštrukciu X, jej prevedenie bude reprezentované následovne: ${}^1X$.
Dvojité prevedenie	Predstavuje situáciu, keď konštrukcia vytvára ďalšiu konštrukciu. Teda ak máme konštrukciu X, potom dvojité prevedenie bude reprezentované ako ${}^2X$.

Tabuľka 3.2: Typy konštrukcií v TIL



Obr. 3.1: Použitie transparentnej intenzionálnej logiky na logickú analýzu vety prirodzeného jazyka [14]

4 Analýza súčasného stavu

Súčasné riešenie sémantického stroja predstavuje funkčnú aplikáciu na spracovanie viet prirodzeného jazyka. Daný sémantický stroj obsahuje dva základné aspekty, vďaka ktorým môžme povedať, že zvolený prístup pri implementácii sémantického stroja je nový a jedinečný.

Prvým dôležitým aspektom je použitie syntaktického analyzátora tretej strany. Konkrétne ide o riešenie vyvinuté na akademickej pôde v Stanforde [15]. Stanford CoreNLP predstavuje stabilné riešenie, ktoré sa stretlo s veľkým úspechom v kruhoch zaoberajúcich sa logickou analýzou prirodzeného jazyka. Dostatočným dôkazom toho je vyhľadávanie projektov na githube, ktoré po zadaní klúčových slov *stanford CoreNLP* do vyhľadávania, vráti približne 400 repozitárov, ktoré tento produkt používajú alebo nejakým spôsobom upravujú/rozširujú. Vďaka jeho skvelým produktovým vlastnostiam je možné jeho použite v kombinácii s rôznymi programovacími jazykmi.

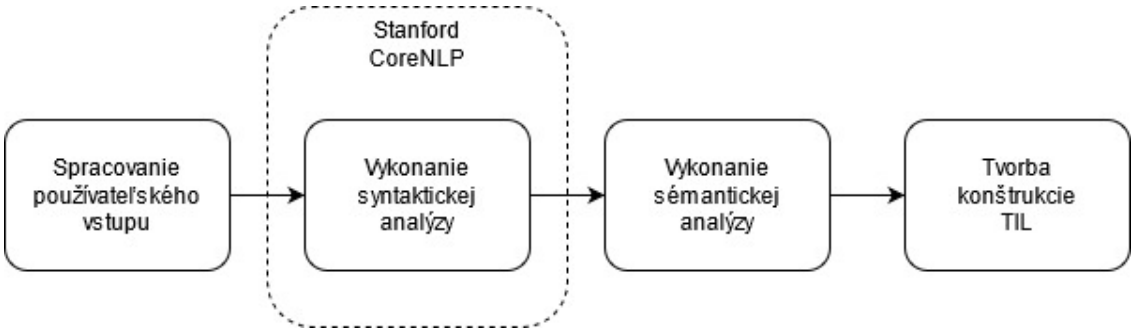
Druhým dôležitým aspektom zvoleného prístupu je použitie transparentnej intenzionálnej logiky na vykonávanie sémantickej analýzy. Táto logika nepredstavuje najnovšie riešenie, no vďaka jej univerzálnosti predstavuje vhodnú logiku na použitie, keďže napríklad oproti logike, ktorú deklaroval Richard Montague, nie je viazaná na konkrétny prirodzený jazyk. Samozrejme, široké možnosti TIL neostali nepovšimnuté, a preto už existuje niekoľko implementácií logickej analýzy prirodzeného jazyka, ktoré využívajú práve tento logický aparát [13]. Najznámejšia vznikla v krajine jej tvorcu Pavla Tichého, a to konkrétne na Masarykovej univerzite. Ide konkrétne o implementáciu, ktorá nesie názov *Normálový translačný algoritmus* a vznikla na začiatku milénia. Táto implementácia TIL je orientovaná výhradné na vety v českom jazyku, ktorý sa komplexnosťou rovná slovenčine. V rámci normálového translačného algoritmu sa autorom podarilo obsiahnuť väčšinu českého jazyka [13].

4.1 Technologické riešenie

Pre jednoduchosť rozhrania, ktoré Stanford CoreNLP ponúka pre programovací jazyk Python, bol pre implementáciu zvolený práve tento programovací jazyk. Prvotne bol v rámci našej implementácie syntaktický analyzátor Stanford CoreNLP testovaný s Javou a Pytonom. Použitie lokálneho servera CoreNLP v kombinácii s Javou bolo náročné na zdroje, hlavným problémom bola veľkosť Java Heapu. Tá by mohla byť problematická, ak by sa naša aplikácia používala napríklad pri výučbe, kde by nemuseli byť dostatočne výkonné stroje. Takiež nastavenie prostredia pre vývoj bolo značne jednoduchšie pre programovací jazyk Pyhon. Jednoduchosť použitia aplikácie bola ďalším a základným faktorom, ktorý rozhodol v prospech použitia programovacieho jazyka Python pre implementáciu sémantického stroja.

4.2 Schopnosti aktuálneho riešenia

Na začiatku vývoja sémantického stroja boli vytýčené základné body pre úspešnú implementáciu riešenia. Potrebné bolo spracovať používateľský vstup, obsahujúci vety prirodzeného jazyka. Následne bolo potrebné vykonanie syntaktickej a sémantickej analýzy a posledný krok predstavovalo vytvorenie konštrukcie TIL ako je možné vidieť na obrázku 4.1.



Obr. 4.1: Potrebné kroky pre implementáciu prototypu

Vzhľadom na to, že používateľské rozhranie nemá vplyv na správnosť alebo funkcionalitu popisovaného riešenia, v počiatočnej fáze vývoja bolo rozhodnuté, že konzolová aplikácia bude pre prototyp vhodnou voľbou. Tento prístup taktiež uľahčí nastavovanie vývojového prostredia pre prípadných kontributorov, keďže

nie je nutné disponovať žiadnymi knižnicami na prácu s grafickým rozhraním. To zároveň poskytuje možnosť zlepšenia, prípadne prepracovania do budúcna.

Aplikácia tretej strany, ktorá vykonáva v rámci sémantického stoja syntaktickú analýzu je implementovaná ako server, ktorý je pre správnu funkcionalitu sémantické stroja potrebné spustiť lokálne. Po úspešnom spustení cez príkazový riadok je používateľ informovaný o spustení alebo prípadnom probléme prostredníctvom výpisu, ako je možné vidieť na obrázku 4.2.

```
D:\School\TUKE\Master_thesis\stanford-corenlp-full-2018-10-05>java -mx1g -cp "*" edu.stanford
.nlp.pipeline.StanfordCoreNLPServer -port 9000 -timeout 15000
[main] INFO CoreNLP - --- StanfordCoreNLPServer#main() called ---
[main] INFO CoreNLP - setting default constituency parser
[main] INFO CoreNLP - warning: cannot find edu/stanford/nlp/models/srparser/englishSR.ser.gz
[main] INFO CoreNLP - using: edu/stanford/nlp/models/lexparser/englishPCFG.ser.gz instead
[main] INFO CoreNLP - to use shift reduce parser download English models jar from:
[main] INFO CoreNLP - http://stanfordnlp.github.io/CoreNLP/download.html
[main] INFO CoreNLP -     Threads: 8
[main] INFO CoreNLP - Starting server...
[main] INFO CoreNLP - StanfordCoreNLPServer listening at /0:0:0:0:0:0:0:0:9000
```

Obr. 4.2: Spustenie servera Stanford CoreNLP

Po úspešnom spustení servera je možné spustenie samotnej aplikácie sémantického stroja. Po spustení, a následnom úspešnom spracovaní používateľského vstupu, ktorým sú vety prirodzeného jazyka, sú tieto vety odoslané na lokálny server CoreNLP, ktorý vykoná syntaktickú analýzu. Následne server odpovie štruktúrou v nami zvolenom formáte a táto štruktúra popisuje vzťahy medzi jednotlivými slovami a ich ďalšie vlastnosti. Nad touto štruktúrou sa začne pre jednotlivé vety vykonávať prvý krok sémantickej analýzy, typová analýza. Aktuálne riešenie pozná základné typy TIL, ktoré dokáže jednotlivým lexikálnym jednotkám priradiť. Vzhľadom na to, že pri tvorbe prototypu išlo o prvý sémantický stroj takéhoto typu, vývoj bol zameraný na spracovanie bežne používaných viet v prirodzenom jazyku. Z toho dôvodu boli implementované aj niektoré rozšírené typy, aby bola množina možných vstupov čo najväčšia.

Vytvorenie konštrukcie významu vety prirodzeného jazyka, ktorá bola zadaná na vstup, predstavuje druhý a posledný krok sémantickej analýzy. Celý proces v rámci aktuálneho riešenia je možné vidieť na obrázku 4.3.

```
************************************************************
*     Semantic machine of Transparent intensional logic    *
*                                                          *
*                    Branislav Bednar                      *
*                         2019                             *
************************************************************

Enter sentece or sentences for analyzing: Joey is not good actor and
 liar. Rachel's father is good doctor. Ross is terrible musician.
*** Construction ***
λwλt[[¬[[[^0actor w]t] [[^0good w]t]] ^0Joey] ∧ [¬[[^0liar w]t] ^0Joey]]
λwλt[[[[^0doctor w]t] [[^0good w]t]]λwλt[^0Rachel [[^0father w]t]]]
λwλt[[[[^0musician w]t] [[^0terrible w]t]] ^0Ross]
```

Obr. 4.3: Súčasný stav implementácie sémantického stroja transaprentnej intezionálnej logiky

4.3 Aktuálne algoritmy pre tvorbu konštrukcií

Základ pre tvorbu konštrukcií tvoria dáta, ktoré reprezentujú výstup typovej analýzy. Tie sú rozdelené do dvoch základných množín, a to základné a rozšírené typy. Tvorba konštrukcií rozšírených typov však nie je identická. Pomyselne ich je možné taktiež rozdeliť do dvoch skupín na základe toho, či sú závislé od iných typov.

Pred tým ako budú jednotlivé algoritmy aktuálneho riešenia demonštrované, je potrebné porozumieť výstupu syntaktickej analýzy, s ktorým tieto algoritmy pracujú. Výstup analýzy sa skladá z viacerých časti, no pre aktuálne riešenie sú najzásadnejšie nasledujúce dve.

Prvá časť definuje závislosti medzi slovami vo vete, ako je možné vidieť na obrázku 4.4. Tieto závislosti [16] sú v riešení dôležité ako pri určovaní typov počas typovej analýzy, tak pri konštruovaní výslednej konštrukcie vety.

Druhá časť výstupu syntaktickej analýzy, ktorá je pre aktuálne riešenie potrebná, je definícia jednotlivých slov vo vete. Tú je možné vidieť na obrázku 4.5. Použitie definícií slov je použité iba v rámci typovej analýzy, kde v niektorých prípadoch rozhoduje aj slovný druh, ktorý je možné v tomto výstupe vidieť ako

atribút POS.

Konštruovanie typov zobrazené na obrázkoch 4.7, 4.8 a 4.9 bolo potrebné vykonať prostredníctvom rozšírených algoritmov, keďže pri konštruovaní daných typov môže nastať viacero situácii. Ďalšie typy, ktoré je možné počas typovej analýzy získať, sú konštruované jednoducho pridaním do konštrukcie.


```
{
   "dep":"ROOT",
   "governor":0,
   "governorGloss":"ROOT",
   "dependent":3,
   "dependentGloss":"singer"
}

{
   "dep":"cop",
   "governor":3,
   "governorGloss":"singer",
   "dependent":2,
   "dependentGloss":"is"
}

{
   "dep":"nsubj",
   "governor":3,
   "governorGloss":"singer",
   "dependent":1,
   "dependentGloss":"David"
}

{
   "dep":"punct",
   "governor":3,
   "governorGloss":"singer",
   "dependent":4,
   "dependentGloss":"."
}
```


Obr. 4.4: Ukážka identifikácie závislostí z výstupu syntaktickej analýzy

4.3.1 Všeobecný algoritmus tvorby konštrukcie

Všeobecný algoritmus pre tvorbu konštrukcií je možné vidieť na obrázku 4.6. Na ňom je možné vidieť, že prvým krokom je nájdenie základného, respektíve hlavného prvku (slova) celej vety, ktorý bol určený syntaktickým analyzátorom pri vykonávaní analýzy syntaxe vety. Následne sú spracovávané jednotlivé rozšírené typy, ktoré postupne vytvárajú konštrukciu vety, ktorá predstavuje výstup aplikácie.

4.3.2 Algoritmus konštruovania vlastnosti objektu

Jednotlivé algoritmy priamo pracujú aj s dátami, ktoré boli výstupom syntaktickej analýzy vykonávanej pomocou Stanford CoreNLP. Ako je možné vidieť aj v algoritme na obrázku 4.7, ktorý konštruuje vlastnosť objektu, sú prechádzané všetky závislosti medzi jednotlivými slovami vo vete.

```
{
    "index": 1,
    "word": "David",
    "originalText": "David",
    "lemma": "David",
    "characterOffsetBegin": 0,
    "characterOffsetEnd": 5,
    "pos": "NNP",
    "before": "",
    "after": " "
}
{
    "index": 2,
    "word": "is",
    "originalText": "is",
    "lemma": "be",
    "characterOffsetBegin": 6,
    "characterOffsetEnd": 8,
    "pos": "VBZ",
    "before": " ",
    "after": " "
}
{
    "index": 3,
    "word": "singer",
    "originalText": "singer",
    "lemma": "singer",
    "characterOffsetBegin": 9,
    "characterOffsetEnd": 15,
    "pos": "NN",
    "before": " ",
    "after": ""
}
{
    "index": 4,
    "word": ".",
    "originalText": ".",
    "lemma": ".",
    "characterOffsetBegin": 15,
    "characterOffsetEnd": 16,
    "pos": ".",
    "before": "",
    "after": ""
}
```

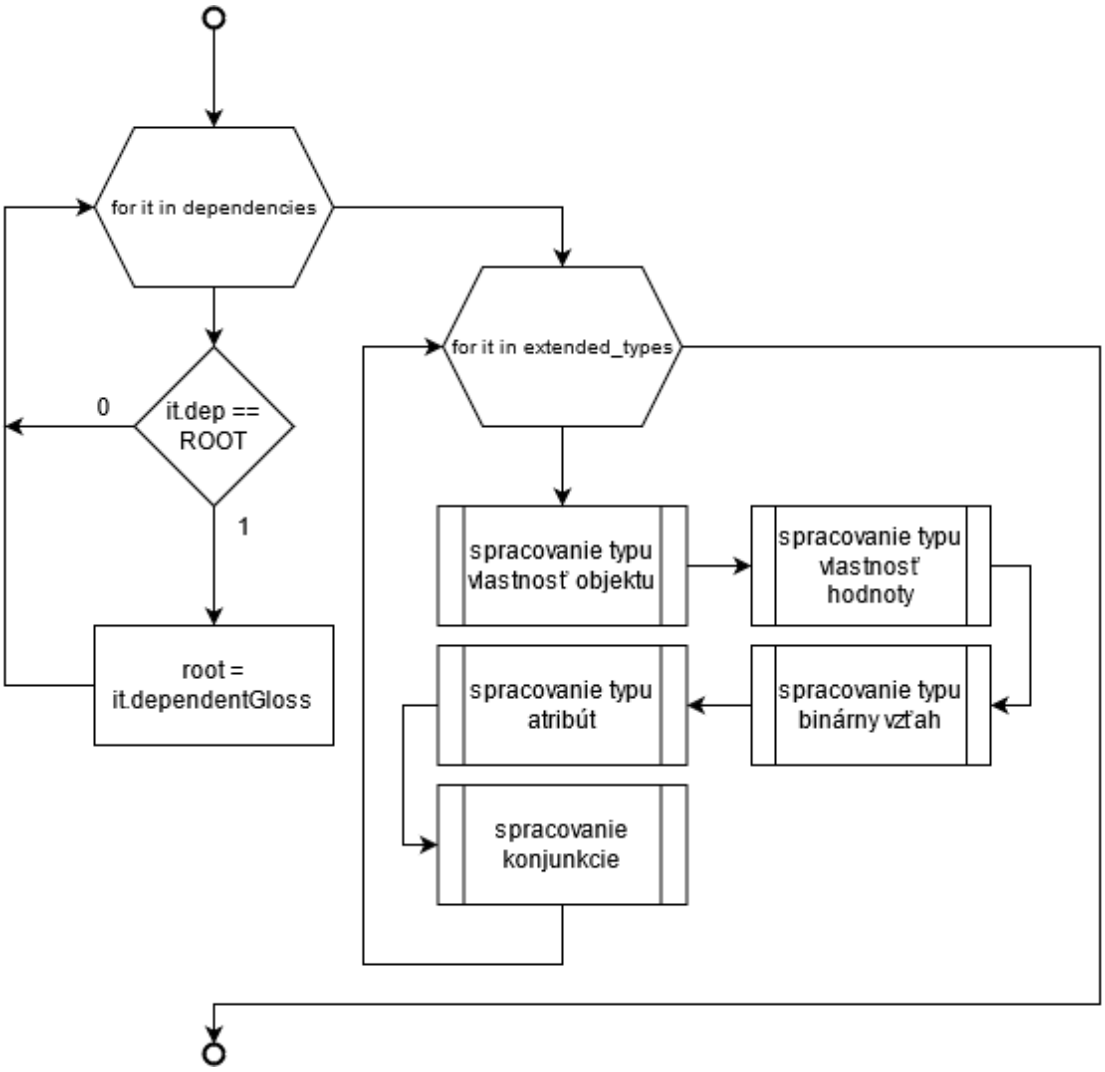

Obr. 4.5: Ukážka identifikácie slov z výstupu syntaktickej analýzy

Následne, ak sa nájde závislosť, ktorá spĺňa podmienky, že je modifikátorom prídavného mena alebo príslovky, a jej nezávislou zložkou je *root*, bude konštruovaná intenzionálna vlastnosť. K tej je následne vyhľadaný závislý objekt, ktorým bude vytvorená konštrukcia rozšírená.

4.3.3 Algoritmus konštruovania konjunkcie

Aktuálna implementácia konštruovania konjunkcie je uľahčená tým, že samotný syntaktický analyzátor pozná, čo je to konjunkcia. Teda v prípade tohto algoritmu, ako je možné vidieť na obrázku 4.8, je prvým krokom prejdenie všetkých závislostí zo syntaktickej analýzy a nájdenie závislosti *CONJ*, pre ktorú platí, že jej nezávislú zložku predstavuje *root*.

Ak sme takýto prvok našli, nastavíme ho ako *root*, čo zabezpečí, že sa bude analyzovať časť vety za konjunkciou. V ďalšom kroku je potrebné nájdenie logic-



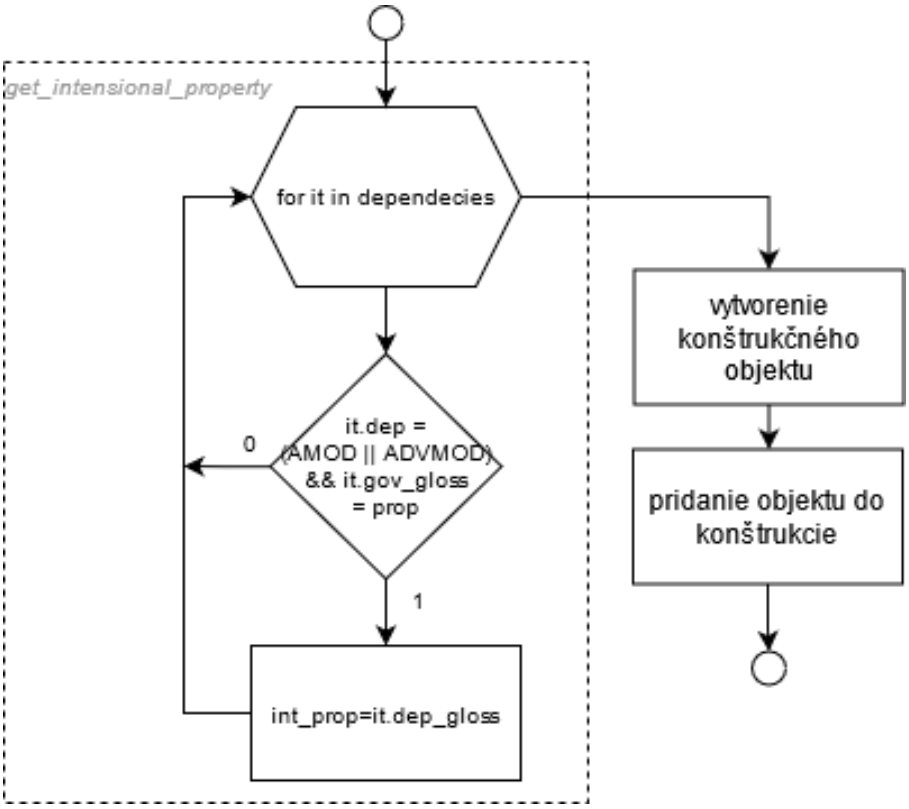
Obr. 4.6: Algoritmus pre tvorbu konštrukcie

kého operátora medzi závislosťami tak, že nájdeme závislosť koordinačnej spojky CC. Takýto operátor je následne pridaný do konštrukcie.

4.3.4 Algoritmus konštruovania vlastnosti hodnoty

Algoritmus na skonštruovanie vlastnosti hodnoty je možné označiť za najzložitejší. Prvým krokom je tak ako v predošlom algoritme nájdenie intenzionálnej vlastnosti vo vete.

Následne je konštruovaný element, ktorý obsahuje *root* a taktiež intenzionálnu vlastnosť. V ďalšom kroku sú znova preverované všetky závislosti zo syntaktickej analýzy. Ak je nájdená taká, ktorá predstavuje vetný člen predmet a jej nezávislou



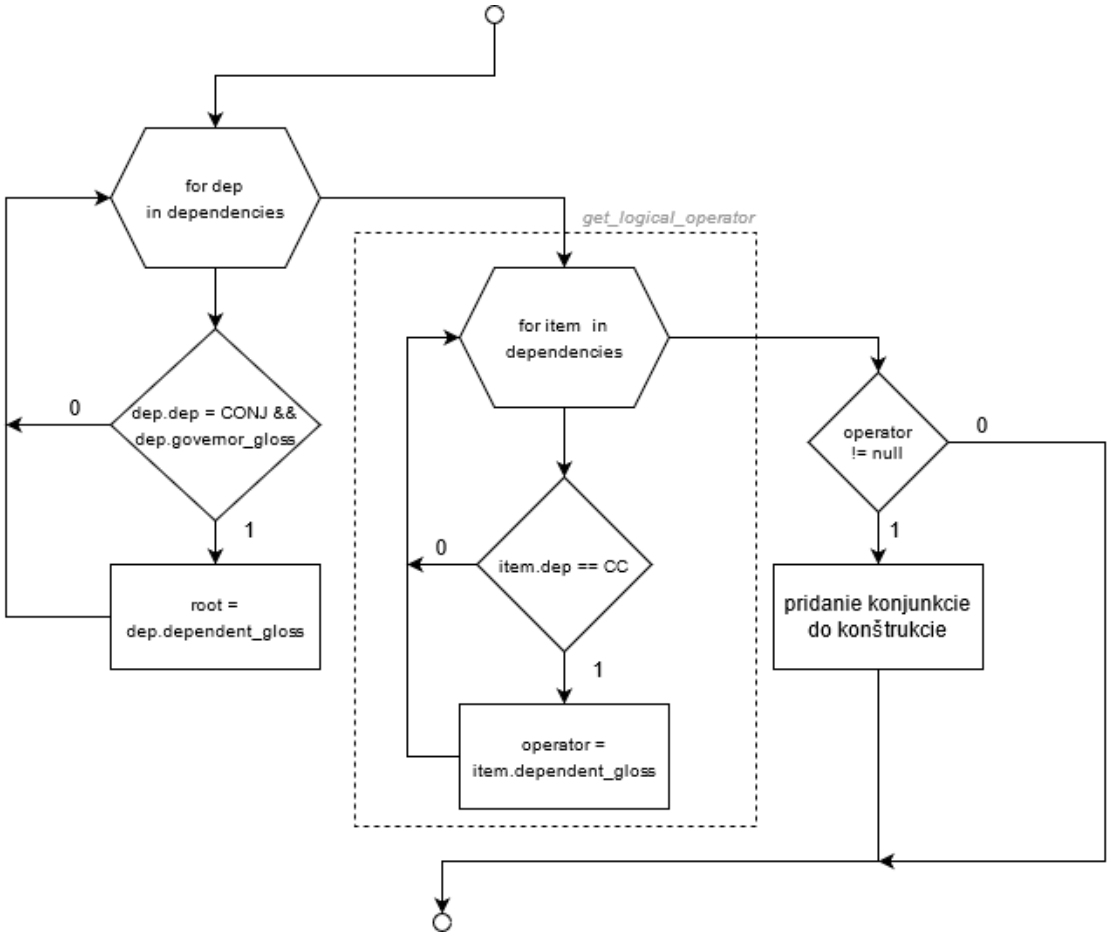
Obr. 4.7: Algoritmus pre konštruovanie vlastnosti objektu

zložkou je *root* ide o vlastnosť hodnoty. Táto vlastnosť je následne skonštruovaná. Po tomto kroku ostáva skonštruovať samostatnú hodnotu, kde nastavajú dva potenciálne scenáre.

Prvý, ktorý je na obrázku 4.9 znázornený ako metóda na spracovanie hodnoty, pokrýva prípad, kedy je hodnota vyjadrená neurčitým tvarom napr. *počet planét/count_of_planets*. V prípade druhom, kedy je hodnota vyjadrená štandardne - číselne, je skonštruovaná vlastnosť rozšírená o túto hodnotu.

Algoritmy pre konštruovanie rozšíreného typu atribút a rozšíreného typu binárny vzťah sú jednoduchšie ako algoritmy pre skonštruovanie spomenutých rozšírených typov alebo konjunkcie. V danom prípade je vytvorená konštrukcia priamo pre typ nájdený počas typovej analýzy. Tieto rozšírené typy nevyžadujú ďalšiu potrebu overovania.

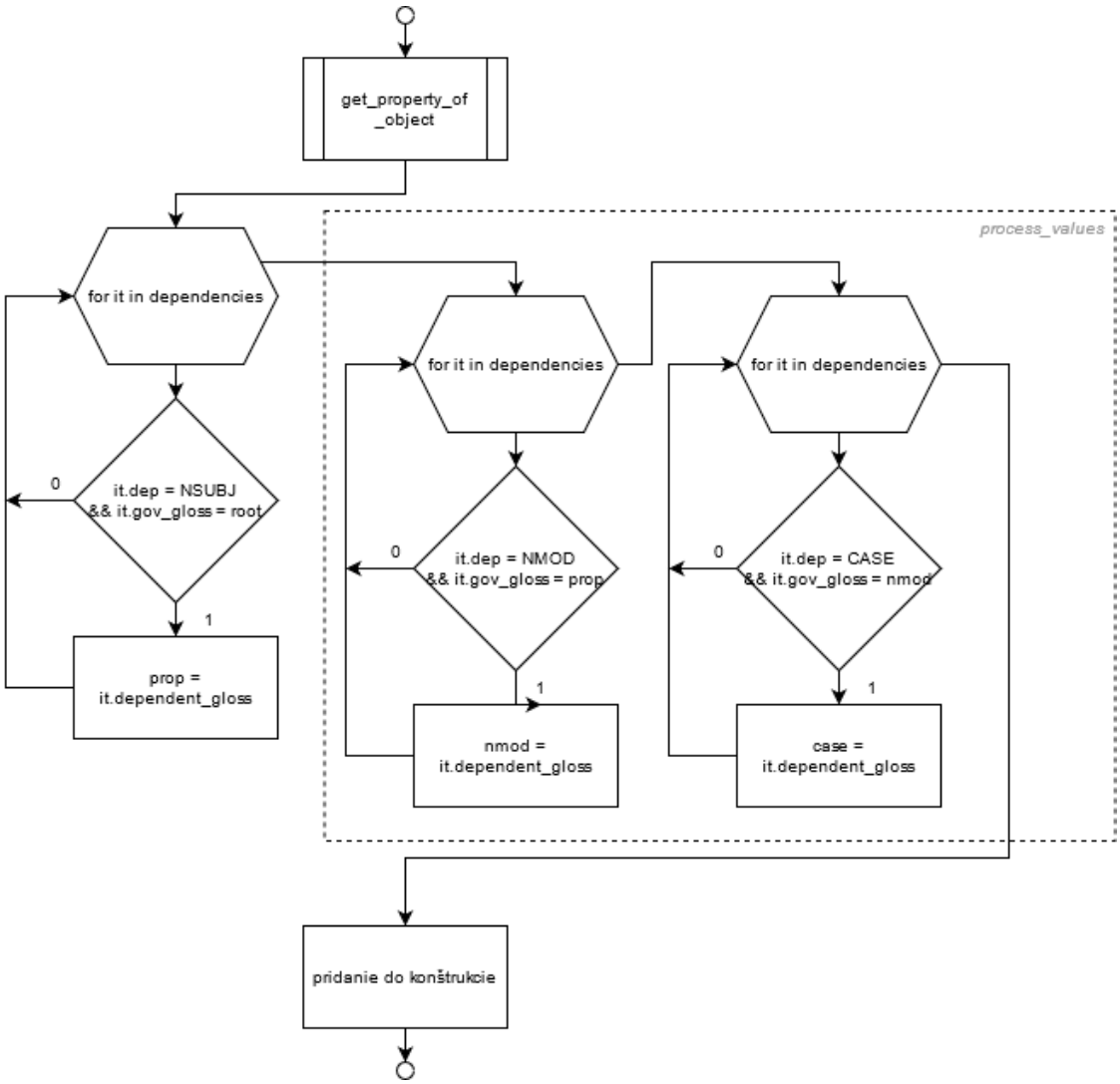
Aktuálne používané algoritmy vykonávajú svoje úlohy v rámci sémantickej analýzy viet prirodzeného jazyka korektne, nevýhodou je však veľká konkrétnosť pri algoritmoch zabezpečujúcich konštruovanie jednotlivých typov. Daná konkrétnosť konštruovania jednotlivých typov zvyšuje ako komplexnosť algoritmov,



Obr. 4.8: Algoritmus pre konštruovanie konjunkcie

tak aj komplexnosť samotného riešenia.

Lepším prístupom v rámci budúcich rozšírení, je zovšeobecnenie časti týchto algoritmov. Hlavným dôvodom je to, aby pridanie spracovania novej funkcionality transparentnej intenzionálnej logiky zo sebou neprinášalo potrebu zásadných zmien v rámci sémantického stroja, ale aby bolo jednoduché túto funkcionalitu pridať.



Obr. 4.9: Algoritmus pre konštruovanie vlastnosti hodnoty

5 Návrh rozšírení aktuálneho riešenia

Pre úspešné rozšírenie prototypu sémantického stroja je nutné navrhnúť a implementovať rozšírenia v dvoch základných rovinách. Prvou je technická rovina, ktorá sa sústredí skôr na samotnú implementáciu a vylepšenia v rámci nej. Druhou je rovina funkčná, ktorá zastrešuje funkcie a možnosti sémantického stroja.

5.1 Návrh technického rozšírenia aktuálneho riešenia

Aktuálne je používanie aplikácie sémantického stroja vskutku jednoduché. Používateľ pomocou príkazového riadku spustí inštanciu lokálneho servera, ktorý spracuje syntaktickú analýzu pre vety prirodzeného jazyka zo vstupu. Ak je server spustený, následne je možné spustiť python skript aplikácie, či už prostredníctvom príkazového riadku alebo vhodného vývojového nástroja. Aplikácia následné inštruuje používateľa k ďalším krokom, ktoré je potrebné vykonať.

5.1.1 Modularizácia

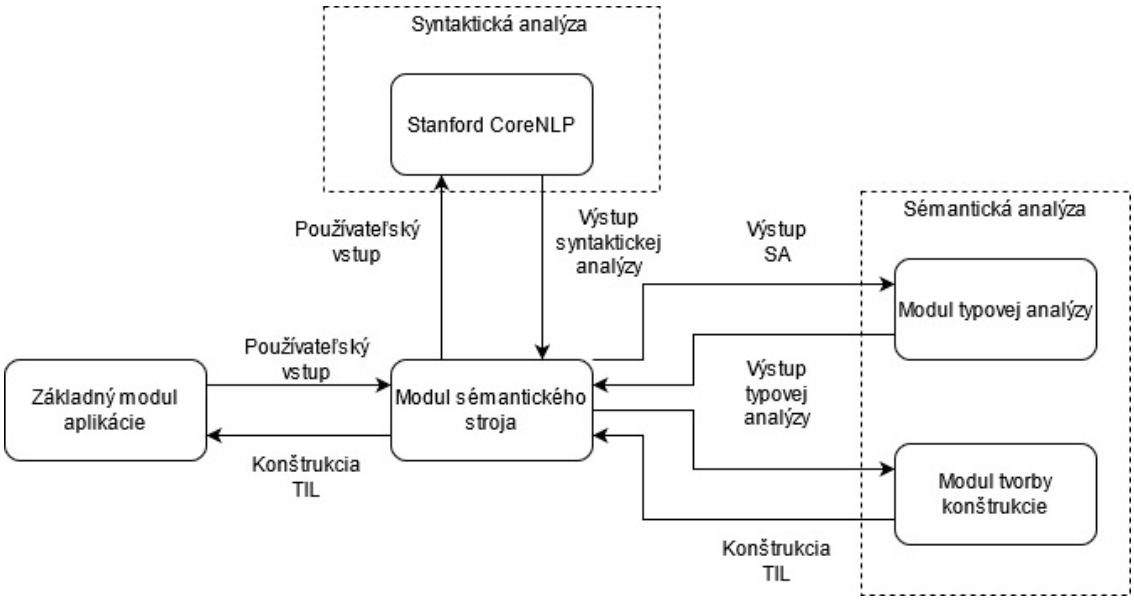
Ak sa neberie do úvahy, že sémantický stroj funguje správne, je možné konštatovať, že spomenutý prístup nie je úplne najlepší. To sa však prejavilo až pri návrhu a úvahách o rozšírení sémantického stroja v rámci tejto práce. Jeden skript, ktorý obsahuje celú implementáciu je dlhý, miestami nečitateľný a porušuje základné princípy softvérového inžinierstva ako napríklad opakovanie kódu, nejednoznačné názvy premenných a podobne [17].

Transparentná intezionálna logika, ktorá bola použitá na spracovanie viet prirodzeného jazyka však svojim postupom načrtáva návrh vhodnej modularizácie softvérového riešenia. Prvým krokom pri použití transparentnej intenzionálnej

logiky je vykonanie typovej analýzy. Túto funkciu teda bude vhodné zastrešiť v rámci jedného modulu. Vďaka tomu sa zvýši prehľadnosť a čitateľnosť kódu, ktorý sa týka výlučne typovej analýzy. Taktiež v prípade potreby rozširovania alebo úpravy implementácie typovej analýzy v budúcnosti bude potrebné zmenu implementovať iba v danom module.

Následným krokom pre úspešné vykonanie sémantickej analýzy zvolenou logikou je vytvorenie konštrukcie pre vetu v prirodzenom jazyku. To znova umožňuje rozdeliť implementáciu a pridať ďalší modul, ktorý zastreší celú implementáciu vytvárania konštrukcie, čo uľahčí prípadné rozšírenie implementácie tvorby konštrukcie. Možné je aj uvažovať o vytvorení samostatných aplikácií zo spomenutých modulov, ktoré by medzi sebou komunikovali a bolo by ich možné prípadne sprístupniť formou koncových bodov.

Rozdelenie zdrojového kódu aplikácie do modulov, ako môžeme vidieť na obrázku 5.1, taktiež prispeje k uľahčeniu hľadania chýb v jednotlivých krokoch procesu spracovania vety prirodzeného jazyka.



Obr. 5.1: Návrh modularizácie aplikácie

5.1.2 Práca s výstupom syntaktickej analýzy

Ďalší problém predstavuje práca s výstupom syntaktického analyzátora. Jazyk Python v kombinácii s knižnicou *JSON* ponúka možnosť veľmi jednoducho a ele-

gantne pracovať s elementami dátovej štruktúry *JSON*, ktorou je reprezentovaný výstup syntaktickej analýzy pre vety zadanej používateľom.

Nevýhodu však predstavuje spôsob, ktorým to bolo použité v aktuálnom riešení. V rámci typovej analýzy sa pracuje priamo s JSON objektom a jeho atribútmi. Takýto prístup síce fungoval, no prináša v rámci implementácie viacero rizík. Prvým, skôr kozmetickým rizikom je to, že v skripte aplikácie to pridáva na neprehľadnosti kódu ako je možné vidieť na obrázku 5.2.

Čo je však horšie, pri práci s atribútmi JSON vývojové prostredie neupozorní na možné chyby v implementácii, akou môže byť napríklad prístup k neexistujúcemu elementu štruktúry. V takom prípade môže nastať pád aplikácie, čo je pre nás nežiadúce. Odstránenie tohto problému môže byť dosiahnuté implementovaním mapovania *JSON* štruktúry do objektu na začiatku sémantickej analýzy. Takýto objekt nám zabezpečí jednoduchšiu a bezpečnejšiu prácu s výstupom syntaktickej analýzy. Taktiež práca s objektami typov transparentnej intenzionálnej logiky nám umožní enkapsulovať explicitné kontroly, ktoré sú vykonávané nad daným typom.

```
for dependency in sentence['basicDependencies']:
    tv = dependency['governorGloss']
    negation = get_negation(dep, tv)
```

Obr. 5.2: Práca s JSON objektom

5.1.3 Testovanie sémantického stroja

Veľmi dôležitý proces počas vývoja softvéru predstavuje testovanie. Ak hovoríme o testovaní Python aplikácie, akou je aj tá naša, poznáme viacero základných typov testovania aplikácií.

Prvým je *jednotkové testovanie*, v rámci ktorého sa testujú dielčie časti aplikácie ako napríklad konkrétne metódy. Takéto testovanie predstavuje testovanie v izolácii, čo znamená že použitie externých služieb je nahradené konkrétnymi očakávanými dátami. Ďalším typom je *integračné testovanie*, ktoré testuje vzájomné fungovanie spolupracujúcich metód. Stále ide o testovanie v izolácii, aby sa prišlo na chyby v rámci aplikácie. Následne poznáme *systémové testovanie*, ktoré sa vykonáva po úspešnom vykonaní predchádzajúcich dvoch typov testov [18]. Tento typ testu testuje systémové celky ako napríklad moduly. Najjednoduchším typom

je používateľské testovanie, kde sa overuje funkčnosť na priamo spustenej aplikácii.

V prípade vývoja prototypu sémantického stroja boli vykonávané iba jednoduché používateľské testy. Tie boli už ku koncu vývoja aplikácie pomerne pracné a zakaždým zaberali viac a viac času. Ich počet sa teda pri implementácii rozšírení bude znova rapídne navyšovať, čo by znamenalo veľa manuálnej práce. Preto bude ideálne, ak pri alebo dokonca pred vývojom nejakého nového rozšírenia bude taktiež implementovaný aj test, ktorý bude hneď funkčnosť overovať.

Potreba implementovania automatických testov je taktiež zvýšená vzhľadom na refaktorizáciu kódu v súvislosti so spomenutými nedostatkami aktuálneho riešenia. Zvolením správneho prístupu testovania je možné efektívne sledovať a zabezpečiť, aby nedošlo pri refaktorizácii kódu aktuálneho riešenia k nesprávnej oprave, ktorá spôsobí nesprávnu funkčnosť sémantického stroja. Tým sa pokryjú aj už existujúce funkcionality.

5.2 Návrh funkčného rozšírenia aktuálneho riešenia

Cieľom tejto práce je rozšírenie aktuálnej implementácie. Možnosti na rozšírenie aktuálneho riešenia je možné nájsť ako v prvej časti sémantickej analýzy - typovej analýze, tak aj v rámci druhej časti, tvorbe konštrukcií. Pre potrebu možnosti spracovania komplexnejších viet prirodzeného jazyka, ale aj viet iných jazykov, aký predstavujú napríklad matematické výrazy, je potrebné rozšírenie množiny typov, ktoré aplikácia pozná. Čo sa tvorby konštrukcie týka, tu je možné implementovanie metód transparentnej intenzionálnej logiky na tvorbu konštrukcií. To môže viesť k prípadom, kedy sémantika jednej vety bude determinovaná rôznymi konštrukciami.

5.2.1 Premenné

Premenná v rámci TIL predstavuje základný spôsob tvorby objektu pre konkrétny výraz - lexikálnu jednotku vety [19]. Je možné povedať, že premenná je samostatnou plnohodnotnou konštrukciou, ktorá vytvára nový objekt v závislosti na jeho valuácii [20].

S výskytom premenných vo vetách prirodzeného jazyka je možné sa stretnúť

primárne vo vetách, kde nie je jednoznačný podmet alebo kde podmet nepredstavuje konkrétnu osobu. Taktiež môže ísť o vety, kde je podmet neurčitý.

5.2.2 Kvantifikátory

Pojem kvatifikovaný výraz je možné poznať vo viacerých rovinách. Takýto výraz poznáme napríklad v predikátovej logike, kde hovorí o tom, aký je počet objektov takých, pre ktoré platí, že majú rovnaké vlastnosti.

Nie len v predikátovej logike, ale aj v hovorenej reči sa kvantifikovaný výraz nachádza pomerne často, bez toho aby sme si to uvedomovali. Takýto výraz v bežnej komunikácii označuje frázu, kde je očakávaný objekt danej triedy, no namiesto toho fráza hovorí o počte viacerých takýchto objektov [21].

V rámci transparentnej intenzionálnej logiky je kvantifikovaný výraz označovaný ako $(o(o\iota))$. Tichého logika však v prípade týchto výrazov umožňuje dve rôzne notácie. V prvom prípade uvažujeme o členoch triedy, na ktorú odkazujeme pomocou kvantifikovaného výrazu, ako o indivíduách. Druhý prípad uvažuje o členoch triedy ako o skupine, čo sa v rámci konštrukcie prejaví tak, že kvantifikovaný výraz predstavuje argument pre sloveso spojené s týmto výrazom.

5.2.3 Implikácia

Implikácia tak, ako ju poznáme z matematiky prípadne logiky, má svoje využitie aj v prirodzenom jazyku a teda taktiež pri tvorbe konštrukcií TIL. Vo všeobecnosti operácia implikácie obsahuje dva argumenty. Vo výraze *a implikuje b* sú týmito argumentami *a*,*b*. Argument *a* označuje premisu výrazu a argument *b* označuje jeho záver [22].

Implikácia v spojení s TIL sa bude týkať prípadov viet, ktoré obsahujú určitú formu podmienenia. Takou vetou môže byť napríklad *„Ak si študent končiaceho ročníka, tak tvojou úlohou je napísať diplomovú prácu."*. Aktuálne riešenie takúto vetu spracovať nedokáže. Potrebnou pre úspešne spracovanie viet obsahujúcich implikáciu je zmena vykonávania typovej analýzy a taktiež zohľadnenie implikácie pri tvorbe konštrukcie TIL. Úspešné implementovanie spracovania implikácie vo vetách prirodzeného jazyka by značne rozšírilo možnosti sémantického stroja pri spracovaní viet prirodzeného jazyka.

6 Rozšírenie sémantického stroja

Na základe výziev v podobe rozšírení stanovených v analýze, budú do sémantického stroja implementované dané rozšírenia. Tieto rozšírenia v prípade úspešnej implementácie zabezpečia presnejšiu funkcionalitu sémantického stroja a taktiež umožnia formalizovať význam širšej množiny viet. Pred implementáciou rozšírení je potrebné zabezpečiť, aby pôvodná funkcionalita ostala neporušená.

6.1 Testovanie

Testovanie v softvérovom inžinierstve predstavuje nespočetne veľa krát rozoberanú a pomerne komplexnú tému. Každý systém alebo aplikácia bola otestovaná minimálne raz. Takýto test predstavuje samostatné prvé spustenie aplikácie. Tento častý prístup však nie je úplne správny.

Testovanie by malo byť v ideálnom prípade súčasťou aplikácie od jej prvotných vývojových fáz a taktiež následne počas celej jej evolúcie. Testovanie počas vývoja dokáže odhaliť množstvo chýb, na ktoré by prišiel až používateľ, na druhej strane testovanie počas evolúcie aplikácie napomáha tomu, aby implementácia nových zmien nezapríčinila chyby, prípadne nefunkčnosť aplikácie.

Pre úspešné rozširovanie našej aplikácie je potrebné doplniť testy natoľko, aby nenastal vyššie spomínaný prípad, kde by mohli nové funkcionality ovplyvniť, prípadne znefunkčniť aplikáciu. Pre úspešné pokrytie aplikácie budú implementované *jednotkové* testy.

Na ich implementovanie je potrebné zvoliť správny testovací aplikačný rámec. Pre testovanie aplikácií, ktoré sú implementované v programovacom jazyku Python, sa najčastejšie využívajú dve knižnice a to *unittest* a *pytest*.

6.1.1 Unitest

Táto testovacia knižnica je v programovacom jazyku Python obsiahnutá od jeho verzie 2.1. Na implementáciu testov pomocou tejto knižnice je potrebné splnenie niekoľkých podmienok:

- vytvorenie tried, ktoré budú obsahovať testovacie metódy,
- použitie *TestCase* triedy na overovanie výsledkov.

```
class TestSemanticMachine(unittest.TestCase):

        def test_semantic_analysis_ok(self):
            self.assertEqual(
                provide_semantic_analysis(synt_analysis_output),
                'λwλt[[[^0singer w]t] ^0Barney]')


        def test_semantic_analysis_nok(self):
            self.assertNotEqual(
                provide_semantic_analysis(synt_analysis_output),
                'λwλt[[[^0singer w]t] Barney]')


if __name__ == '__main__':
    unittest.main()
```

Zdrojový kód 1: Príklad použitia knižnice *unittest*

6.1.2 Pytest

Druhou známou a často používanou testovacou knižnicou je knižnica *pytest*. Podporuje aj spúšťanie testov, ktoré boli napísané pomocou knižnice *unittest*. Predstavuje však výhodu v tom, že nie je potrebné vytváranie testovacej triedy. Testy teda predstavujú postupnosť metód, ktorých názvy musia začínať prefixom **test_**.

Medzi ďalšie výhody použitia tejto knižnice patrí:

- podpora vstavaných *assert* výrazov,
- prehľadnejší a informačnejší výpis v prípade neúspešného testu,
- množstvo podporovaných rozšírení,
- parametrizované testy.

```
def test_semantic_analysis_ok():
    assert provide_semantic_analysis(synt_analysis_output) ==
        'λwλt[[[^0singer w]t] ^0Barney]'


def test_semantic_analysis_nok():
    assert provide_semantic_analysis(synt_analysis_output) !=
        'λwλt[[[^0singer w]t] Barney]'
```

Zdrojový kód 2: Príklad použitia knižnice *pytest*

Ako je možné vidieť na ukážkach zdrojového kódu 1 a 2, použitím knižnice *pytest* je možné vytvorenie rovnakých testov pomocou menšieho množstva kódu. Taktiež boli popísané výhody použitia tejto knižnice, čo pre potreby nášho použitia hovorí v jej prospech. Vzhľadom na tieto dôvody bude použitá pri implementovaní jednotkových testov v rámci našej aplikácie.

6.2 CI/CD

S príchodom agilného prístupu bolo potrebné priniesť taktiež metódy a postupy, ktoré budú uľahčovať vývoj agilného systému. Tieto metódy majú za úlohu overovanie správnosti implementácie nových zmien vzhľadom na predchádzajúcu implementáciu a taktiež doručovanie jednotlivých verzií systému, či už priamo zákazníkovi alebo interne, napríklad na interné servre z dôvodu testovania. Vďaka týmto potrebám vznikol v roku 2000 pojem CICD, ktorý môžme voľne preložiť ako nepretržitá integrácia(CI) a nepretržité doručovanie(CD) [23].

Vo všeobecnosti nepretržitá integrácia predstavuje proces, kedy členovia tímu pravidelne zapracovávajú zmeny. Adaptácia týchto zmien spúšťa automatický build, testy a validáciu, vďaka čomu je možné odhalenie novo vzniknutých chýb.

Nepretržité doručovanie predstavuje druhú časť automatického procesu, kedy je nová verzia systému nasadená na server alebo doručená zákazníkovi, so všetkým potrebným pre dané prostredie.

Prvým dôležitým krokom pre správne použitie CICD, je vytvorenie súboru *requirements.txt* v koreňovom adresári projektu. Tento súbor obsahuje názvy všetkých knižníc a balíčkov, použitých v rámci aplikácie spolu s ich verziami. Tie sa pri spustení automatických úloh nainštalujú, aby bolo možné spustenie aplikácie v danom prostredí.

V rámci našej implementácie pri každej požiadavke na zmenu, je automaticky spustená séria úloh. Prvou je statická analýza kódu, ktorá je vykonávaná pomocou knižnice *pylint*. Výstupom statickej analýzy je číslo, ktoré hodnotí kvalitu kódu na základe rôznych aspektov, ako napríklad dĺžka výrazov, výskyt nepoužívaných premenných alebo importov, duplicitné metódy, opakujúci sa kód a rôzne ďalšie. Výstup statickej analýzy kódu pre našu aplikáciu je možné vidieť na obrázku 6.1.

```
$ pylint -d C0301,C0114 semantic_machine_of_til_v2/*.py
-----------------------------------
Your code has been rated at 10.00/10
Cleaning up file based variables
Job succeeded
```

Obr. 6.1: Výstup statickej analýzy kódu

Ďalšou automaticky spúšťanou úlohou je spustenie jednotkových testov, vďaka ktorému sa pri požiadavke na zmenu overí, či funkcionalita ostala správne zachovaná. Výsledok tejto úlohy je možné vidieť na obrázku 6.2.

V rámci tejto úlohy je taktiež analyzované aj pokrytie kódu testami, čo je možné vidieť na obrázku 6.3.

Po úspešnom vykonaní všetkých úloh je možné adaptovať nové zmeny do aplikácie.

6.3 Tvorba webovej služby

Primárnym cieľom prototypu sémantického stroja bolo demonštrovanie použitia logického aparátu transparentnej intenzionálnej logiky s nezávislým syntaktic-

```
$ cd semantic_machine_of_til_v2/tests/;pytest -v
============================= test session starts ==============================
platform linux -- Python 3.8.7, pytest-5.4.1, py-1.10.0, pluggy-0.13.1 -- /usr/local/bin/python
cachedir: .pytest_cache
rootdir: /builds/bb627ko/master-thesis-semantic-machine-2.0/semantic_machine_of_til_v2/tests
plugins: cov-2.10.1
collecting ... collected 14 items
test_examples_for_thesis.py::TestSemanticMachine::test_semantic_analysis_nok SKIPPED [  7%]
test_examples_for_thesis.py::TestSemanticMachine::test_semantic_analysis_ok SKIPPED [ 14%]
test_semantic_machine.py::test_simple SKIPPED                            [ 21%]
test_semantic_machine.py::test_another_foo SKIPPED                       [ 28%]
test_semantic_machine.py::test_third_another_foo SKIPPED                 [ 35%]
test_semantic_machine_units.py::test_semantic_analysis PASSED            [ 42%]
test_semantic_machine_units.py::test_get_truth_values PASSED             [ 50%]
test_semantic_machine_units.py::test_get_individuals PASSED              [ 57%]
test_semantic_machine_units.py::test_get_intensional_roles PASSED        [ 64%]
test_semantic_machine_units.py::test_get_property_of_object PASSED       [ 71%]
test_semantic_machine_units.py::test_get_binary_relations PASSED         [ 78%]
test_semantic_machine_units.py::test_get_values PASSED                   [ 85%]
test_semantic_machine_units.py::test_get_attributes PASSED               [ 92%]
test_semantic_machine_units.py::test_get_property_of_value PASSED        [100%]
======================== 9 passed, 5 skipped in 0.66s ========================
Cleaning up file based variables
Job succeeded
```

Obr. 6.2: Výstup úlohy, ktorá spúšťa jednotkové testy

```
---------- coverage: platform linux, python 3.8.9-final-0 ----------
Name                                                                 Stmts   Miss  Cover
----------------------------------------------------------------------------------------
semantic_machine_of_til_v2/modules/__init__.py                           0      0   100%
semantic_machine_of_til_v2/modules/construction_creation_module.py     276      8    97%
semantic_machine_of_til_v2/modules/corenlp_constants.py                 24      0   100%
semantic_machine_of_til_v2/modules/corenlp_module.py                    78      4    95%
semantic_machine_of_til_v2/modules/exceptions_module.py                 40     26    35%
semantic_machine_of_til_v2/modules/predicates.py                        44      2    95%
semantic_machine_of_til_v2/modules/semantic_machine_module.py           52     12    77%
semantic_machine_of_til_v2/modules/type_analysis_module.py             439     23    95%
----------------------------------------------------------------------------------------
TOTAL                                                                  953     75    92%
```

Obr. 6.3: Pokrytie kódu testami

kým analyzátorom. Pre tieto ciele bolo postačujúce vytvorenie konzolovej aplikácie, ktorá funkčnosť prototypu dostatočne demonštrovala.

Po dokončení prototypu bola ako jedna z možných budúcich vylepšení vy-

slovená myšlienka zmeny sémantického stroja na webovú službu, ktorá by bola použiteľná ďalšími aplikáciami prostredníctvom API. Vďaka tomu by bolo možné sémantický stroj považovať za použiteľný produkt.

6.3.1 Výber webového aplikačného rámca

Pri zmene konzolovej aplikácie vytvorenej v programovacom jazyku Python na webovú službu sa núka viacero možností. Veľkosť komunity okolo jazyka Python spôsobila pri príchode webových aplikácií príchod množstva nových aplikačných rámcov, využívajúcich práve tento programovací jazyk [24]. Vďaka tomu je možné si vyberať z veľkého množstva a podľa rôznych kritérií.

Za hlavnú kategóriu môžeme považovať plnohodnotné aplikačné rámce. Medzi takéto môžeme zaradiť napríklad Django, Web2py, Pyramid alebo CubicWeb. Tieto aplikačné rámce je možné použiť pre rôzne druhy webových aplikácií, no primárne sú určené na zastrešenie komplexných projektov [25]. Ich hlavnými výhodami je množstvo funkcionality, ktorú obsahujú, ako napríklad práca s databázou, autentifikačné mechanizmy, podpora webových serverov, dátové zabezpečenie a iné.

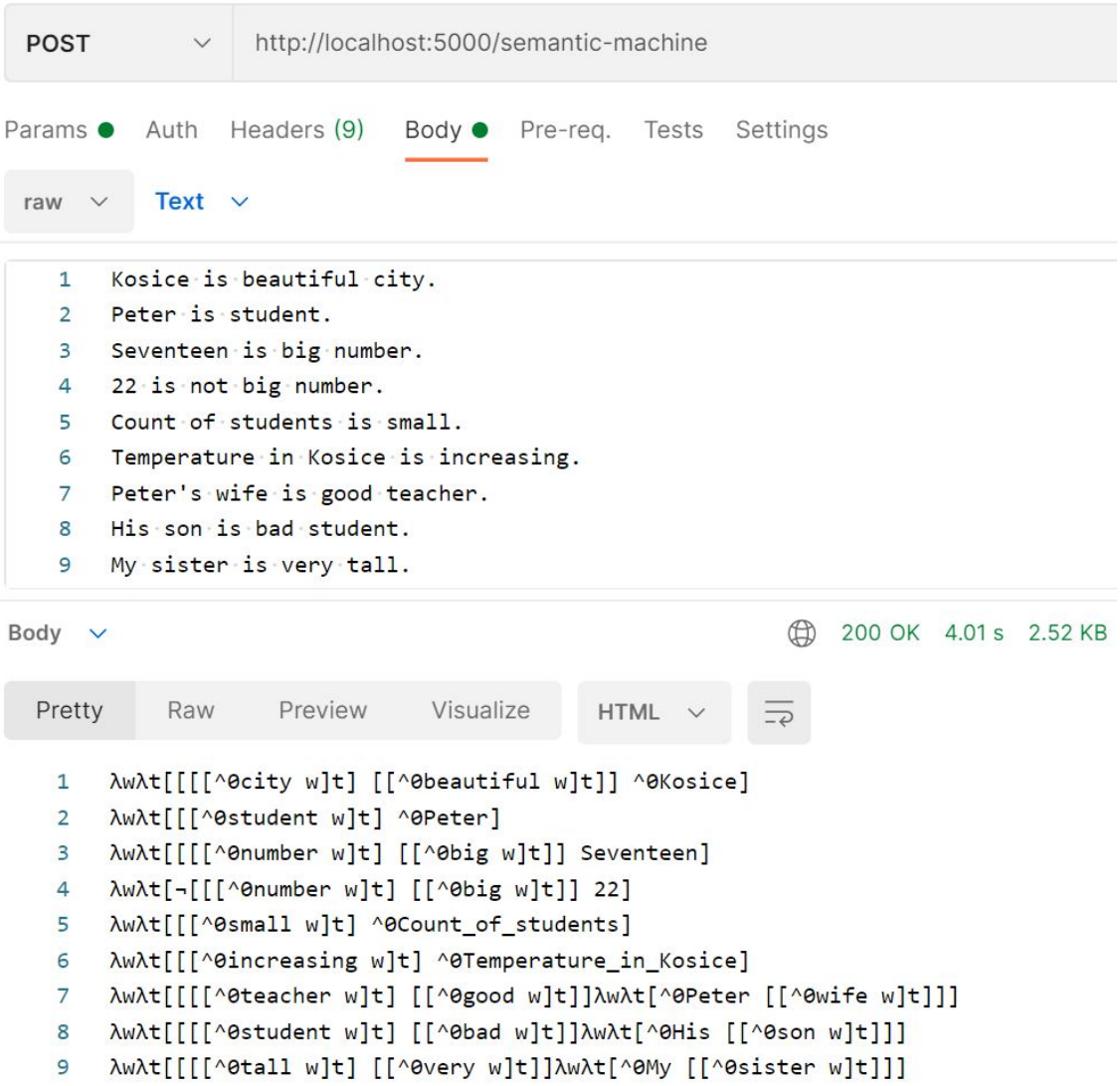
Na druhej strane poznáme aplikačné rámce s pomenovaním mikro. Tieto primárne slúžia na zastrešenie malých a stredných webových aplikácii a služieb. Medzi takéto aplikačné rámce patria napríklad Flask, CherryPy, Falcon, Bottle a iné. Výhodou týchto aplikačných rámcov je, že na ich použite je zväčša potrebná minimálna konfigurácia a minimálne množstvo kódu oproti prechádzajúcim menovaným.

Na základe osobných skúseností, na tvorbu webovej služby sémantického stroja bol vybraný aplikačný rámec Flask. Flask patriaci do skupiny mikro aplikačných rámcov bol navrhnutý tak, aby spĺňal základné potreby pre tvorbu webových aplikácií/služieb a taktiež aby bol schopný spolupráce s ďalším rozširujúcimi aplikáciami alebo aplikačnými rámcami.

6.3.2 Používanie webovej služby

Úspešné rozšírenie sémantického stroja o webový aplikačný rámec Flask, umožní používanie sémantického stroja ako webovej služby. Ideálom pre náš prípad použitia je komunikácia s webovou službou prostredníctvom restového rozhrania.

Vzhľadom na to, že zo sémantického stroja dáta nielen získavame, ale aj doň dáta posielame, je potrebné komunikovať so službou pomocou metódy POST, kde telo budú tvoriť vety prirodzeného jazyka. Názorné použitie sémantického stroja ako webovej služby je možné vidieť na obrázku 6.4.

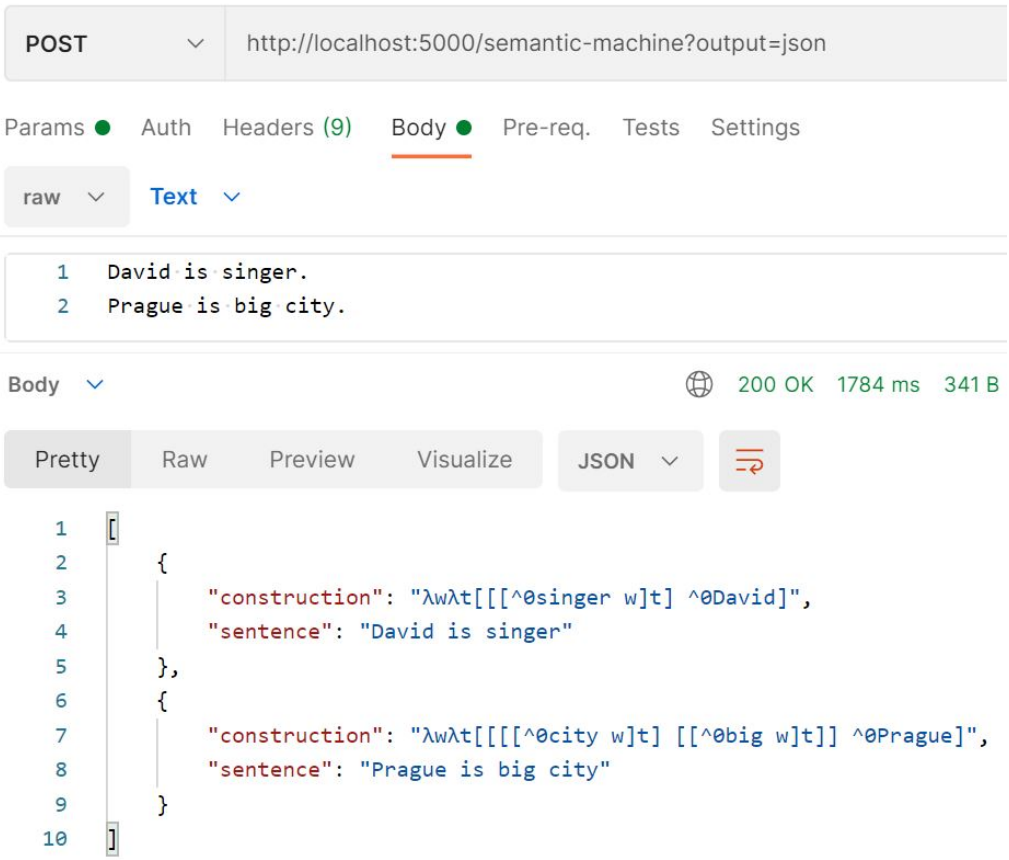


Obr. 6.4: Používanie sémantického stroja ako webovej služby

Pri implementovaní webovej služby bolo zohľadnené potenciálne budúce využitie tejto služby inou aplikáciou. Z toho dôvodu bola do sémantického stroja pridaná funkcionalita, ktorá vráti výsledok sémantickej analýzy vo forme zrozumiteľnejšej pre konzumenta webovej služby, konkrétne vo forme JSON.

Túto funkcionalitu je možné docieliť použitím dotazového parametra *output*. Ten môže mať dve hodnoty a to konkrétne *text* a *json*. Predvolené správanie webo-

vej služby je rovnaké ako pri použití parametra *output=text*. Naopak, pri použití parametra *output=json*, webová služba vráti odpoveď vo forme, ktorú je možné vidieť na obrázku 6.5.



Obr. 6.5: Použitie dotazového parametra *output=json*

6.4 Spracovanie konjunkcie

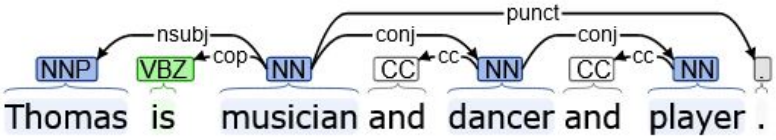
Existujúce riešenie okrem iného obsahovalo aj spracovanie konjunkcie vo vetách prirodzeného jazyka pomocou algoritmu, ktorý je možné vidieť na obrázku 4.8. Daný algoritmus hľadal konjunkciu medzi rozšírenými typmi TIL, teda napríklad medzi vlastnosťou objektu a binárnym vzťahom. Takýto prístup sa javil a fungoval správne, no nedokázal sa vysporiadať s vetami, kde bola konjunkcia mnohonásobná a vzťahovala sa len na jeden konkrétny typ.

Z toho dôvodu bolo zapríčinené, že význam vety *„Thomas is musician and dancer and player."*, bol skonštruovaný následovne:

$$\lambda w \lambda t\ [[[^0 musician\ w]t]\ ^0 Thomas] \wedge [[[\ ^0 dancer\ w]t]\ ^0 Thomas \wedge [\,]$$

To predstavuje nesprávnu konštrukciu, keďže chýba v poradí druhá konjunkcia *Thomas-player*. Prvým krokom pre docielenie správnej funkcionality bolo použitie rozšírených závislostí zo syntaktickej analýzy.

Obrázok 6.6 zobrazuje základné závislosti, ktoré boli používané v doterajšom riešení. Ako je možné na obrázku vidieť, pri typovej analýze bolo potrebné nájsť 3 vlastnosti objektu a to konkrétne *Thomas-musician*, *Thomas-dancer*, *Thomas-player*. To samozrejme nefungovalo správne, keďže sa konjunkcia hľadala podľa závislosti CONJ vzhľadom na prvú nájdenú vlastnosť objektu a to *Thomas-musician*.



Obr. 6.6: Základné závislosti z výstupu syntaktickej analýzy

Použitie rozšírených závislosti tento problém vyriešilo. Ako je možné vidieť na obrázku 6.7, ktorý reprezentuje syntaktickú analýzu získanú z CoreNLP a taktiež na obrázku 6.8, v tomto prípade sú už všetky konjunkcie v závislosti s vlastnosťou objektu *Thomas-musician*.

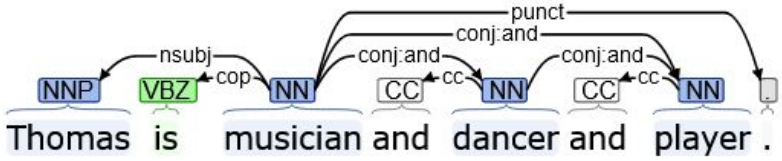
6.4.1 Algoritmus konštruovania konjunkcie

Druhým krokom pre správne konštruovanie viet obsahujúcich konjunkciu bola úprava algoritmu konštruovania významu vety. Najdôležitejšou potrebou pre nový algoritmus bolo zovšeobecnenie hľadania konjunkcie pre akýkoľvek typ. Obrázok 6.9 zobrazuje, ako je táto požiadavka realizovaná. Ak sa vrátime k príkladu vety *„Thomas is musician and dancer and player."*, tak ak máme vlastnosť objektu *Thomas-musician*, *musician* predstavuje slovo, pre ktoré je konjunkcia vo vete hľadaná. V rozšírených závislostiach sa hľadá závislosť taká, ktorá predstavuje konjunkciu a zároveň jej nezávislou zložkou je slovo *musician*. Pri úspešnom nájdení závislá zložka závislosti následne predstavuje konjunkciu, čo vedie k vytvoreniu objektu konjunkcie, ktorý je následné pridaný do konštrukcie.

Aby sa predišlo možným problémom s duplicitným konštruovaním jednotlivých typov, napríklad aj pri konštruovaní konjunkcie, každý nájdený typ obsahuje atribút *already_constructed*, ktorého iniciálna hodnota je *false*, no v prípade ak

```
{
  "dep":"ROOT",
  "governor":0,
  "governorGloss":"ROOT",
  "dependent":3,
  "dependentGloss":"musician"
}
{
  "dep":"nsubj",
  "governor":3,
  "governorGloss":"musician",
  "dependent":1,
  "dependentGloss":"Thomas"
}
{
  "dep":"cop",
  "governor":3,
  "governorGloss":"musician",
  "dependent":2,
  "dependentGloss":"is"
}
{
  "dep":"cc",
  "governor":3,
  "governorGloss":"musician",
  "dependent":4,
  "dependentGloss":"and"
}
{
  "dep":"conj:and",
  "governor":3,
  "governorGloss":"musician",
  "dependent":5,
  "dependentGloss":"dancer"
}
{
  "dep":"cc",
  "governor":5,
  "governorGloss":"dancer",
  "dependent":6,
  "dependentGloss":"and"
}
{
  "dep":"conj:and",
  "governor":3,
  "governorGloss":"musician",
  "dependent":7,
  "dependentGloss":"player"
}
{
  "dep":"conj:and",
  "governor":5,
  "governorGloss":"dancer",
  "dependent":7,
  "dependentGloss":"player"
}
{
  "dep":"punct",
  "governor":3,
  "governorGloss":"musician",
  "dependent":8,
  "dependentGloss":"."
}
```


Obr. 6.7: Výstup syntaktickej analýzy vety obsahujúcej konjunkciu

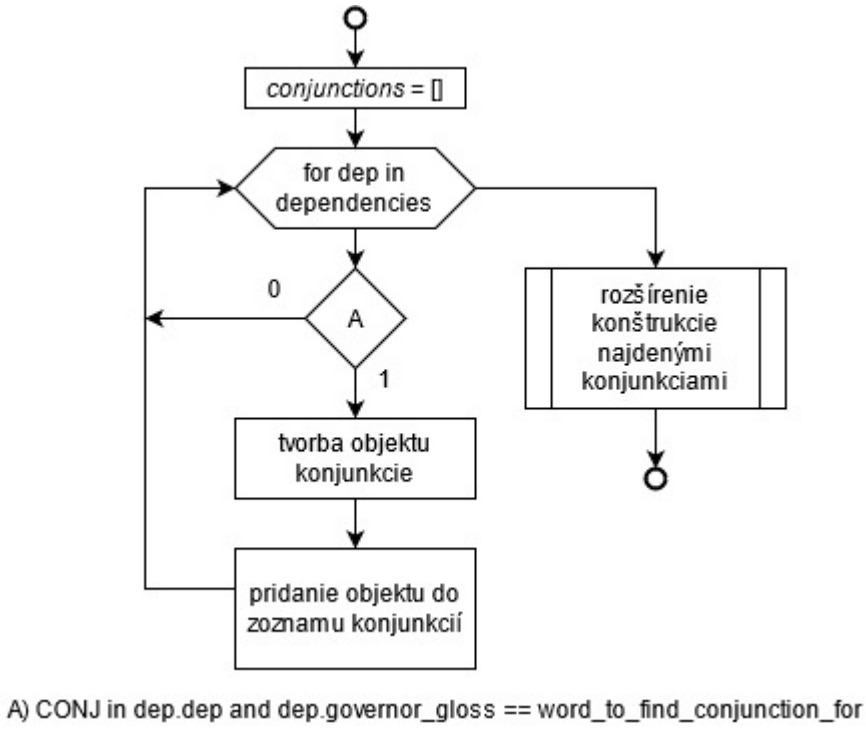


Obr. 6.8: Rozšírené závislosti z výstupu syntaktickej analýzy

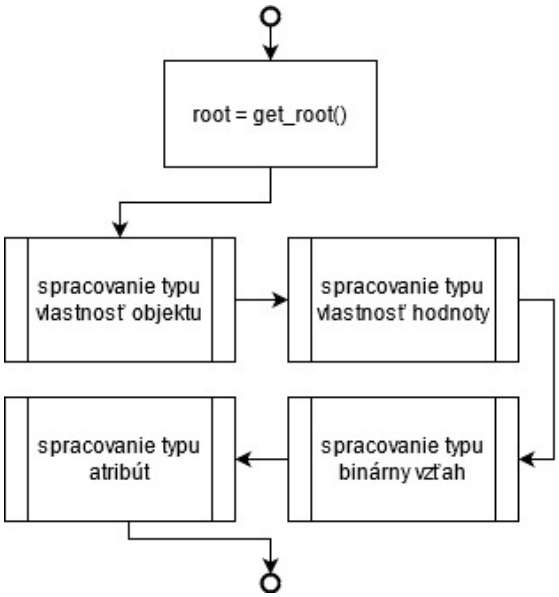
bol konkrétny typ, napríklad *Thomas-musician*, skonštruovaný je tomuto atribútu priradená hodnota *true*.

Zovšeobecnenie algoritmu pre hľadanie konjunkcií umožnilo zjednodušenie a vylepšenie algoritmu na tvorbu konštrukcie významu oproti pôvodnému, ktorý bol zobrazený na obrázku 4.6. Nový algoritmus na tvorbu konštrukcií je zobrazený na obrázku 6.10. Táto zmena následne ponúkla možnosť zjednodušenia aj samotných algoritmov na konštruovanie jednotlivých typov TIL.

Zovšeobecnený nový algoritmus pre konštruovanie jednotlivých typov je zobrazený na obrázku 6.16. Ten ako môžeme vidieť, je oproti predchádzajúcim algoritmom, ktoré je možné vidieť na obrázkoch 4.7, 4.8 a 4.9, oveľa jednoduchší čítateľnejší, menej komplexný a bez množstva prebytočných cyklov.



Obr. 6.9: Nový algoritmus pre spracovanie konjunkcií



Obr. 6.10: Nový algoritmus pre tvorbu konštrukcie

Po úspešnej úprave všetkých potrebných algoritmov je možné vidieť, že význam analyzovanej vety *„Thomas is musician and dancer and player."* je skonštruovaný správne:

$$\lambda w \lambda t\ [[[^0 musician\ w]t]\ ^0 Thomas \wedge [[\ ^0 dancer\ w]t]\ ^0 Thomas \wedge [[^0 player\ w]t]\ ^0 Thomas]$$

Úprava taktiež zabezpečila rozšírenie množiny viet, ktoré je možné spracovať sémantickým strojom. Medzi takéto vety patria napríklad:

Jim is student and musician and dancer and sportsman.
Michael is good student and Mathew is not gorgeous musician.
Thomas is student and 315 is big number.
193 is big number and 17 is prime.

6.5 Spracovanie implikácie

Ďalšiu dôležitú časť prirodzeného jazyka, ktorej spracovanie rozšíri množinu viet použiteľnú pre sémantický stroj, predstavuje implikácia. Pri používaní jazyka si výskyt implikácie takej, akú poznáme napríklad z logiky, neuvedomujeme. Jej používanie je pre nás prirodzené, čo znamená, že bez premýšľania formulujeme vety a vetné konštrukcie, ktoré majú tvar práve implikácie. Dôvod, prečo predstavuje dôležitú súčasť jazyka je prostý. Tvar implikácie poskytuje možnosť formulácie zložitejších viet a súvetí. V prípade slovenského jazyka je možné sa s jej použitím najčastejšie stretnúť v priraďovacích súvetiach. V anglickom jazyku je použitie implikácie najčastejšie v podmienených vetách, ktoré poznáme pod názvom kondicionály.

6.5.1 Algoritmus pre identifikáciu implikácie vo vete

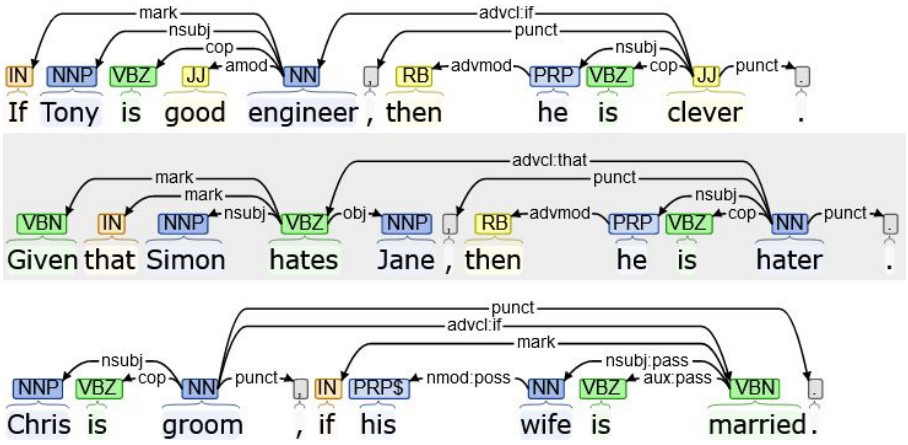
Existuje viacero tvarov viet obsahujúcich implikáciu. Základný a najzreteľnejší prípad predstavujú vety v tvare *„If Tony is good engineer, then he is clever."*. Okrem nich však môžeme nájsť implikáciu v ďalších vetách ako napríklad *„Unless Monica is smiling, Chandler is joker."*. Menej zriedkavým prípadom môže byť aj veta v tvare *„Given that Simon hates Jane, then he is hater."*. Pre takéto vety však neexistuje jasná a jednoznačná logická analýza, čo vytvára potrebu pre tvorbu a implementáciu algoritmu na ich spracovanie.

Rovnako ako aj pri spracovaní konjunkcie vo vetách prirodzeného jazyka, taktiež aj pri spracovaní implikácie bolo potrebné začať úpravou algoritmov zabez-

pečujúcich typovú analýzu. Ako je možné vidieť na obrázku 6.11, zásadnými pre identifikovanie implikácie vo vetách budú závislosti ADVCL a MARK.

Uvažujme vetu *„If Tony is good engineer, then he is clever."*. V rámci prvého kroku spracovania tejto vety, typovej analýzy, je potrebné nájsť dva typy, vlastnosti objektu, a to konkrétne *Tony-engineer* a *Tony-clever*. Pri aktuálnej implementácii typovej analýzy bude úspešne nájdená prvá vlastnosť objektu *Tony-engineer*. V tomto momente však nastáva potreba rozšírenia algoritmu o nájdenie implikácie k tejto vlastnosti.

Pre úspešné nájdenie implikácie je potrebné vo výstupe syntaktickej analýzy zobrazenom na obrázku 6.12, nájsť závislosť ADVCL, ktorej závislú zložku predstavuje vlastnosť *engineer*. Ak je takáto závislosť nájdená, jej nezávislá zložka predstavuje implikáciu k pôvodnej vlastnosti objektu.



Obr. 6.11: Syntaktická analýza viet obsahujúcich implikáciu

Na obrázku 6.12 je možné vidieť, že výstup syntaktickej analýzy vlastnosť *clever* spája so slovom *he* a nie s indivíduom *Tony*, ako je pre nás potrebné. Tento problém po dôkladnej analýze ponúkol riešenie v podobe explicitnej kontroly možnosti, kedy by nájdená implikácia bola v závislosti NSUBJ s iným indivíduom. Takáto kontrola nám zabezpečuje správne skonštruovanie vety *„If Tony is good engineer, then he is clever."* a taktiež aj vety ako napríklad *„If Jane is singer, then Monica is dancer."*, kde implikovaná vlastnosť *dancer*, je v závislosti s indivíduom *Monica*.

Správne vytvorená konštrukcia prezentovaných viet môže vyzerať následovne:

$$\lambda w \lambda t\ [\Rightarrow [[[[^0 engineer\ w]t]\ [[^0 good\ w]t]\ ^0 Tony]\ [[[^0 clever\ w]t]\ ^0 Tony]]$$

$$\lambda w \lambda t\ [\Rightarrow [[[[^{0}singer\ w]t]\ ^{0}Jane]\ [[[^{0}dancer\ w]t]\ ^{0}Monica]]$$


```
{
  "dep":"ROOT",
  "governor":0,
  "governorGloss":"ROOT",
  "dependent":10,
  "dependentGloss":"clever"
}

{
  "dep":"mark",
  "governor":5,
  "governorGloss":"engineer",
  "dependent":1,
  "dependentGloss":"If"
}

{
  "dep":"nsubj",
  "governor":5,
  "governorGloss":"engineer",
  "dependent":2,
  "dependentGloss":"Tony"
}

{
  "dep":"cop",
  "governor":5,
  "governorGloss":"engineer",
  "dependent":3,
  "dependentGloss":"is"
}

{
  "dep":"amod",
  "governor":5,
  "governorGloss":"engineer",
  "dependent":4,
  "dependentGloss":"good"
}

{
  "dep":"advcl:if",
  "governor":10,
  "governorGloss":"clever",
  "dependent":5,
  "dependentGloss":"engineer"
}

{
  "dep":"punct",
  "governor":10,
  "governorGloss":"clever",
  "dependent":6,
  "dependentGloss":","
}

{
  "dep":"advmod",
  "governor":8,
  "governorGloss":"he",
  "dependent":7,
  "dependentGloss":"then"
}

{
  "dep":"nsubj",
  "governor":10,
  "governorGloss":"clever",
  "dependent":8,
  "dependentGloss":"he"
}

{
  "dep":"cop",
  "governor":10,
  "governorGloss":"clever",
  "dependent":9,
  "dependentGloss":"is"
}

{
  "dep":"punct",
  "governor":10,
  "governorGloss":"clever",
  "dependent":11,
  "dependentGloss":"."
}
```

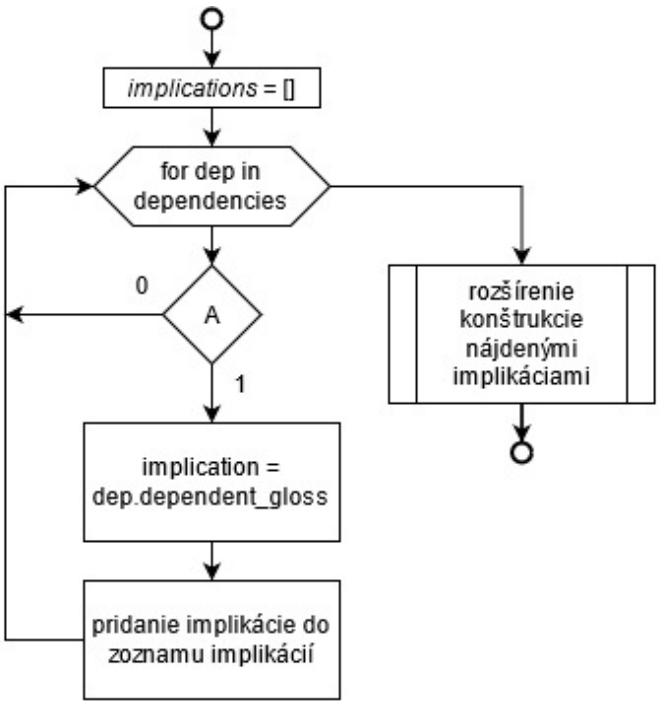

Obr. 6.12: Výstup syntaktickej analýzy vety obsahujúcej implikáciu

6.5.2 Algoritmus konštruovania implikácie

Úspešné identifikovanie implikácie vo vete prirodzeného jazyka počas vykonávania typovej analýzy naznačuje presun k druhému kroku, ktorý je potrebný pre úspešné spracovanie implikácie, a to k jej konštruovaniu. Vzhľadom na zmenu všeobecného algoritmu pre tvorbu konštrukcie, ktorý je možné vidieť na obrázku 6.10, algoritmus pre skonštruovanie významu vety obsahujúcej implikáciu musí byť všeobecný, bez ohľadu na typ, pre ktorý bola implikácia vyhľadávaná.

Výskyt implikácie vo vete však spôsobí dôležitú zmenu. Koreňom vety, v tomto prípade nebude vlastnosť *engineer*, ale vlastnosť *clever*, čo je taktiež možné vidieť na obrázku 6.12. Táto zmena spôsobí, že počas konštruovania významu vety, bude ako prvá konštruovaná vlastnosť objektu *Tony-clever*.

Počas konštruovania objektu *Tony-clever* sa vykonáva overenie, či pre daný typ je možné vo vete nájsť implikáciu. Podobne ako aj pri algoritme, ktorý detegoval implikáciu vo vete počas typovej analýzy, aj tu je základom úspešného spracovania nájdenie závislosti ADVCL, ako je možné vidieť na obrázku 6.13.



Obr. 6.13: Algoritmus konštruovania implikácie

Zmena však nastáva pri overovaní. Keďže poradie konštruovania objektov je opačné voči tomu ako boli nájdené počas typovej analýzy, v tomto prípade sa overuje, či nezávislá zložka závislosti ADVCL je *clever*. V kladnom prípade teda závislá zložka predstavuje implikáciu, ktorá sa následne uloží do zoznamu nájdených implikácií. Nájdené implikácie potom slúžia na prehľadanie všetkých typov nájdených pri typovej kontrole. Ak niektorý z typov obsahuje nájdenú implikáciu, tento typ je vo vzťahu implikácie k typu *Tony-clever*. Po tomto zistení je implikovaný typ pridaný do konštrukcie významu vety.

Implementácia spracovania implikácie v rámci nášho sémantického stroja zabezpečila rozšírenie množiny viet, ktoré dokáže spracovať, o vety obsahujúce implikáciu v rôznych formách. Príkladmi takýchto viet sú:

If Tony is good engineer, then he is clever.

If Samuel is teacher, Matthew is student.
If Jane is singer, then Monica is dancer.
If 1053 is big number, then 17 is even number.
If Peter is student, then 8 is big number.
Given that Simon hates Jane, then he is hater.
Chris is groom, if his wife is married.
Matthew is student, if Samuel is teacher.
Unless Monica is smiling, Chandler is joker.
If Tony is singer, Matthew is dancer and Peter is technician.
If Andrew is musical student, he will be musician.
If Tom hates Gwen, he would be hater.

6.6 Spracovanie premenných a kvantifikátorov

Všetky vety, ktoré je doposiaľ možné sémantickým strojom spracovať, disponujú jednou spoločnou vlastnosťou. Každá veta obsahuje typ indivíduum, respektíve má konkrétne vyjadrený podmet, na ktorom závisí zvyšná časť vety. Existuje však mnoho viet takých, ktoré priamo podmet, prípadne indivíduum neobsahujú, no obsahujú napríklad kvantifikovaný výraz. Takýmito vetami môžu byť napríklad:

Are some employees professors?
Thomas loves all girls.
If some programmer is smart, he is clever.

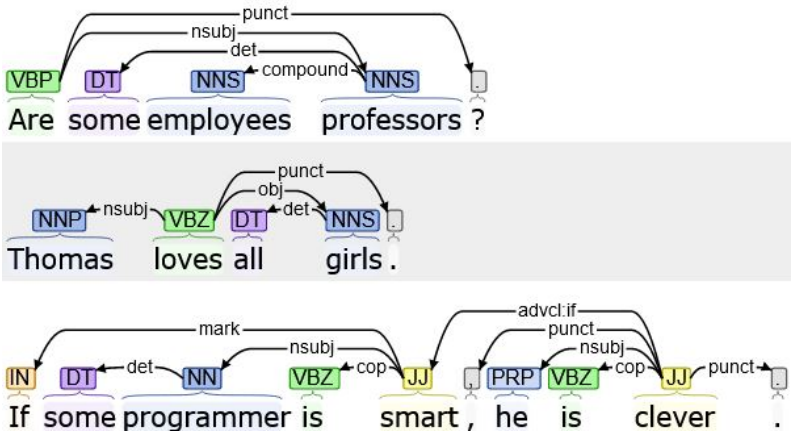
6.6.1 Rozšírenie typovej analýzy

Typová analýza doterajšieho riešenia predstavuje algoritmus, ktorý pri spracovávaní vety v prvom kroku hľadá základné typy TIL, teda indivíduá a pravdivostné hodnoty. V druhom kroku sú vyhľadávané rozšírené typy TIL, pomocou už nájdených základných typov. Samozrejme aj v doterajšom riešení je možné spracovanie viet, ktoré neobsahujú indivíduum. Príkladom takýchto viet sú okrem iných aj nasledujúce vety:

Table is small and wooden.
My brother is very slim.

Aj keď sa v daných vetách typ indivíduum nenachádza, je možné v nich počas vykonávania typovej analýzy nájsť typ pravdivostné hodnoty, napríklad pre prvú vetu ide o slová *small* a *wooden*. Pomocou nich je následne možné vo vete nájsť vlastnosť objektu, kde je objekt jednoznačne daný pre aktuálny čas a svet, teda pre túto vetu ide o objekt *table*.

Oproti tomu, vety obsahujúce kvantifikátor ako napríklad *all, some* označujú v danom svete všeobecnú entitu danej triedy a toto zovšeobecnenie je potrebné zohľadniť ako pri typovej analýze, tak pri konštruovaní sémantiky vety.



Obr. 6.14: Syntaktická analýza viet s potrebou použitia premennej

Podľa syntaktickej analýzy zobrazenej na obrázku 6.14 je možné usúdiť, že v prípade, kedy sa vo vete nenašlo žiadne indivíduum je potrebné sa pri hľadaní rozšírených typov zamerať na závislosť DET, ktorá súvisí práve s kvantifikátormi.

Uvažujme tretiu vetu, *„If some programmer is smart, he is clever."*. Pri typovej analýze nie je nájdené žiadne indivíduum iba dve pravdivostné hodnoty *smart* a *clever*. Na rad prichádza hľadanie rozšírených typov, pre analyzovanú vetu ide konkrétne o hľadanie vlastnosti objektu, kde je potrebné doterajšiu implementáciu rozšíriť o prípad, kedy sa vo vete indivíduá nenachádzajú.

Pre nájdené pravdivostné hodnoty je potrebné nájsť závislosť NSUBJ, kde nezávislou zložkou bude práve niektorá z pravdivostných hodnôt. Po nájdení takejto závislosti napríklad pre pravdivostnú hodnotu *smart* je možné na obrázku 6.15 vidieť, že závislou zložkou tejto závislosti je slovo *programmer*. Následne je potrebné overiť, že existuje vo vete závislosť DET taká, že jej nezávislou zložkou je slovo *programmer*. V prípade, že aj táto predispozícia je splnená, je možné skonštatovať, že sa podarilo nájsť vlastnosť objektu *programmer-smart*. Analogické spraco-

```
{
    "dep": "ROOT",
    "governor": 0,
    "governorGloss": "ROOT",
    "dependent": 9,
    "dependentGloss": "clever"
}
{
    "dep": "nsubj",
    "governor": 5,
    "governorGloss": "smart",
    "dependent": 3,
    "dependentGloss": "programmer"
}
{
    "dep": "punct",
    "governor": 9,
    "governorGloss": "clever",
    "dependent": 6,
    "dependentGloss": ","
}
{
    "dep": "mark",
    "governor": 5,
    "governorGloss": "smart",
    "dependent": 1,
    "dependentGloss": "If"
}
{
    "dep": "cop",
    "governor": 5,
    "governorGloss": "smart",
    "dependent": 4,
    "dependentGloss": "is"
}
{
    "dep": "nsubj",
    "governor": 9,
    "governorGloss": "clever",
    "dependent": 7,
    "dependentGloss": "he"
}
{
    "dep": "punct",
    "governor": 9,
    "governorGloss": "clever",
    "dependent": 10,
    "dependentGloss": "."
}
{
    "dep": "det",
    "governor": 3,
    "governorGloss": "programmer",
    "dependent": 2,
    "dependentGloss": "some"
}
{
    "dep": "advcl:if",
    "governor": 9,
    "governorGloss": "clever",
    "dependent": 5,
    "dependentGloss": "smart"
}
{
    "dep": "cop",
    "governor": 9,
    "governorGloss": "clever",
    "dependent": 8,
    "dependentGloss": "is"
}
```


Obr. 6.15: Výstup syntaktickej analýzy vety s potrebou použitia premennej

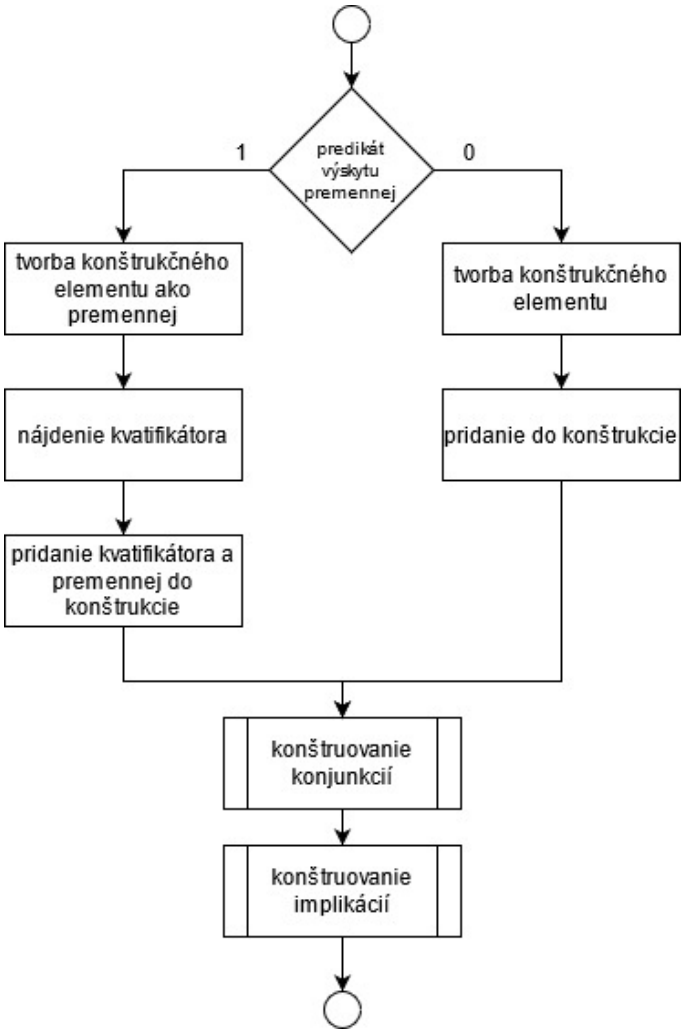
vanie následné prebieha pre zvyšok vety, čo má za následok úspešné vykonanie typovej analýzy, počas ktorej boli nájdené dve vlastnosti objektu *programmer-smart* a *programmer-clever*.

6.6.2 Algoritmus konštruovania premenných a kvantifikátorov

Pre úplné spracovanie premenných a kvantifikátorov sémantickým strojom je potrebné vytvorenie algoritmu, ktorý premenné zohľadní aj pri konštruovaní jednotlivých typov TIL, nezávisle od daného typu.

Majme dve vlastnosti objektu získané pri analýze vety *„If some programmer is smart, he is clever."*. Pri konštruovaní jednotlivých vlastností, je v prvom rade potrebné zistiť, či veta obsahuje kvantifikátor a teda, či je potrebné použitie premennej vo výslednej konštrukcii, ako je možné vidieť na obrázku 6.16. V rámci tohto overenia sa overujú dve základné podmienky. Prvou podmienkou pre úspešné identifikovanie potreby použitia premennej vo vete je overenie prípadu, či veta

obsahuje závislosť DET. Druhý prípad predstavuje overenie výskytu slov kto (*who*), ktorý (*which*), vďaka čomu algoritmus zastrešuje aj otázky.



Obr. 6.16: Všeobecný algoritmus pre konštruovanie rozšírených typov

V prípade nesplnenia týchto podmienok konštruovanie tejto vybranej vlastnosti objektu prebehne štandardným spôsobom.

Splnenie oboch týchto podmienok spôsobí vytvorenie konštrukčného elementu s použitím premennej, čo sa od štandardného konštruovania vlastnosti objektu, prípadne ďalších typov líši tým, že takýto element bude v konštrukcii zobrazený s premennou, ktorá ho reprezentuje. Je teda možné konštatovať, že pre vlastnosť objektu *clever-programmer* sa vytvoria dva konštrukčné objekty a to konkrétne $[[^0programmer\ w]t]\ x$ a $[[^0clever\ w]t]\ x$.

V ďalšom kroku je vyhodnocovaný kvantifikátor nachádzajúci sa vo vete. Toto

vyhodnotenie slúži na použitie správneho kvantifikátora vo výslednej konštrukcii a určenie, či je nutné použiť existenčný alebo všeobecný kvatnifikátor. Pre analyzovanú vetu sa správne vyhodnotí kvantifikátor existenčný ${}^0\exists\lambda x$, ktorý bude následne použitý vo výslednej konštrukcii.

Následne je konštrukcia významu vety rozšírená o nájdené premenné a kvantifikátor. Spracovanie každého typu v ďalších krokoch pokračuje hľadaním konjunkcií a implikácií. Výsledná konštrukcia pre analyzovanú vetu bude po spracovaní implikácie, premenných a kvatifikátorov vyzerať nasledovne.

$$\lambda w \lambda t\ [\Rightarrow\ [[[[{}^0\mathit{smart}\ w]t]\ x \wedge [[{}^0\mathit{programmer}\ w]t]\ x]]\ [{}^0\exists\lambda x\ [[[{}^0\mathit{clever}\ w]t]\ x \wedge [[{}^0\mathit{programmer}\ w]t]\ x]]]$$

Implementácia spracovania premenných a kvantifikátorov sémantickým strojom rozširuje množinu spracovávaných viet, vetami ako sú napríklad:

Are some employees professors?
All singers are good musicians.
Which professors are dancers.
Some student is programmer and good dancer.
If some programmer is smart, he is clever.
Taylor hates some workers.
Some girls hates Rachel and loves Phoeobe.
Some number is prime.
If some engineer is smart, he is clever.

Spracovanie premenných a kvantifikátorov sémantickým strojom zastrešuje aj otázky, ako je možné vidieť v ukážkových vetách. Počas implementácie však bola zistená nekonzistentnosť syntaktickej analýzy v prípade, že sa vo vete nachádzal otáznik. Syntaktická analýza sa v takom prípade dosť líšila a pravdepodobne ide o nedostatok syntaktického analyzátora. Z tohto dôvodu je potrebné pri zadávaní otázok použiť bodku namiesto otáznika.

6.7 Riešenie funkčných nedostatkov

Funkčné nedostatky predstavujú neoddeliteľnú súčasť mnohých aplikácií a aplikácia prototypu sémantického stroja nie je výnimkou. Počas implementácie no-

vých rozšírení bolo nájdených niekoľko závažných nedostatkov, ktoré spôsobovali nesprávne konštruovanie významu vety prirodzeného jazyka.

Majme vetu *„His son is bad worker."*. Pri spracovaní tejto vety sú počas typovej analýzy nájdené korektne tri typy. Konkrétne ide o atribút *His son*, vlastnosť objektu *worker* a intenzionálnu vlastnosť *bad*. Problém v pôvodnej verzii bol nájdený v algoritme konštruovania typu atribút. V prípade, že veta obsahovala intenzionálnu vlastnosť, konštrukcia vyzerala nasledovne:

$$\lambda w \lambda t[[[[^0worker\ w]t]\ [[^0bad\ w]t]]\ ^0son[[^0worker\ w]t]\lambda w \lambda t[^0His\ [[^0son\ w]t]]]$$

Samozrejme takáto konštrukcia pre danú vetu nie je správna. Po úprave algoritmu na konštruovanie typu atribút, je daná veta spracovaná správne.

$$\lambda w \lambda t[[[[^0worker\ w]t]\ [[^0bad\ w]t]]\lambda w \lambda t[^0His\ [[^0son\ w]t]]]$$

Ďalší odhalený nedostatok mal taktiež súvis so spracovaním intenzionálnej vlastnosti, no tentokrát pri konštruovaní typu vlastnosť objektu. V prípade, že veta obsahovala dve rovnaké intenzionálne vlastnosti, vo finálnej konštrukcii sa nachádzala iba jedna. Konštrukcia významu vety *„Ted is good architect and good singer"* teda vyzerala následovne:

$$\lambda w \lambda t[[[[[^0architect\ w]t]\ [[^0good\ w]t]]\ ^0Ted\ ] \wedge [[[\ ^0singer\ w]t]\ ^0Ted]]$$

Úprava algoritmu pre konštruovanie typu vlastnosť objektu vyriešila tento problém, čo znamená, že po úprave sú aj vety takéhoto druhu spracované správne.

$$\lambda w \lambda t[[[[[^0architect\ w]t]\ [[^0good\ w]t]]\ ^0Ted\ ] \wedge [[[\ ^0singer\ w]t]\ [[^0good\ w]t]]\ ^0Ted]]$$

Posledný odhalený nedostatok sa vyskytoval pri konštruovaní typu hodnota. V prípade konštruovania vety *„If Theodore is player, then 8 is big number."* vyzerala konštrukcia významu vety následovne:

$$\lambda w \lambda t[\Rightarrow[[[^0player\ w]t]\ ^0Theodore]\ [[[[^0number\ w]t]\ [[^0big\ w]t]]\ ^08]$$

Ako je možné vidieť, chybu v tomto prípade predstavuje trivializovanie hodnoty *8*, keďže význam tejto hodnoty je bezohľadu na svet a čas nemenný. Pre vyriešenie tohto problému bolo potrebné upravenie všeobecnej triedy konštrukčného elementu, ktorá je používaná pre konštruovanie jednotlivých typov TIL.

Konkrétne je overované, či je hodnota vyjadrená číslom v tvare číslice alebo slova, ako napríklad *seventeen*.

Po zapracovaní tejto úpravy do sémantického stroja je konštrukcia významu danej vety vytvorená správne:

$$\lambda w \lambda t[\Rightarrow[[[^0 \textit{player}\ w]t]\ ^0\textit{Theodore}]\ [[[[^0 \textit{number}\ w]t]\ [[^0 \textit{big}\ w]t]]\ 8]$$

7 Vyhodnotenie

Predmetom diplomovej práce bolo rozšírenie prototypu sémantického stroja transparentnej intenzionálnej logiky. Samotný prototyp bezohľadu na svoju správnu funkcionalitu ponúkal množstvo priestoru na zlepšenie, ktorým mohlo byť zlepšenie po funkčnej alebo technologickej stránke, prípadne pridanie novej funkcionality.

7.1 Funkčné rozšírenia

Ak uvážime funkčnú stránku prototypu, prototyp bol schopný spracovať určitú množinu jednoduchých viet prirodzeného jazyka. Aby bolo možné pokladať rozšírenie prototypu za úspešné, bolo nutné, aby funkčne dokázal sémantický stroj pokryť väčšie množstvo viet prirodzeného jazyka a spracovať sémantickú analýzu nielen viet jednoduchých ale aj komplexnejších, ktoré sú v bežnej komunikácii používané. Pre túto potrebu bolo nutné opätovné, dôkladnejšie naštudovanie transparentnej intenzionálnej logiky a zváženie zapracovania niektorých z možností, ktoré tento logický aparát ponúka.

Najzaujímavejšími prvkami TIL pre rozšírenie nášho prototypu bolo spracovanie implikácie, premenných a kvantifikátorov. Vďaka zapracovaniu týchto komplexnejších prvkov TIL do sémantického stroja, je možné spracovávať napríklad vety obsahujúce podmienky v rôznych tvaroch, vety ktoré neobsahujú menný podmet - konkrétnu osobu alebo vec, prípadné rôzne variácie týchto typov viet, s vetami, ktoré dokázal spracovať aj prototyp.

Pri implementácii nových funkčných vylepšení sa prejavil problém so spracovávaním konjunkcií v rámci prototypu. Ten dokázal spracovať len veľmi malé množstvo viet obsahujúcich konjunkciu, konkrétne dokázal nájsť konjunkciu iba medzi rôznymi typmi TIL a v niektorých prípadoch dokonca nesprávne. Pri náj-

dení tohoto problému a analyzovaní možnej opravy, z ktorej by bolo možné benefitovať aj pri implementácií nových funkcionalít, bol v rámci sémantického stroja implementačne úplne zmenený prístup k sémantickej analýze. Pri tejto zmene boli taktiež zmenené algoritmy zastrešujúce typovú analýzu a taktiež aj samotné algoritmy pre tvorbu konštrukcie TIL. Aj táto zmena zabezpečila možnosť spracovania ďalších typov viet, ktoré prototyp spracovať nedokázal, konkrétne ide o rôzne variácie viet s konjunkciami.

7.2 Technické rozšírenia

Zdrojový kód prototypu predstavoval jednoduchý jedno-súborový skript, ktorého veľkosť spôsobovala nečitateľnosť a zhoršovala udržateľnosť samotnej aplikácie. Pre zjednodušenie rozširovania prototypu bol tento skript rozdelený do modulov, ktoré pokrývali jednotlivé funkcionality sémantického stroja. Taktiež bola vykonaná refaktorizácia zdrojového kódu.

Každé rozširovanie existujúceho systému vyžaduje garanciu zachovania aktuálnej funkcionality počas implementovania nových rozšírení. Takáto požiadavka sa stala technickou výzvou aj v rámci tejto diplomovej práce. Implementácia prototypu nebola doposiaľ pripravená na aplikovanie zásadných zmien, čo bolo potrebné zmeniť, ešte pred samotným funkčným rozšírením.

Z tohto dôvodu bolo potrebné implementovanie mechanizmov, ktoré by pri zmenách implementácie sémantického stroja overovali jeho pôvodnú funkcionalitu. Bolo teda potrebné vybrať vhodnú technológiu a metodológiu na testovanie aplikácie. Analýza dostupných možností v rámci danej problematiky poukázala na metódu jednotkových testov ako na najúčinnejšiu pre naše riešenie. Implementácia jednotkových testov pomocou knižnice Pytest umožnila priebežné manuálne kontrolovanie zachovania pôvodnej funkcionality.

K zdokonaleniu testovania pôvodnej funkcionality chýbalo už iba automatizovanie testovania pri každej požiadavke na pridanie funkcionality. Systém na správu verzií, ktorý bol počas vývoja sémantického stroja používaný, disponuje možnosťami CICD, vďaka ktorým sa pri správnom nakonfigurovaní podarilo zautomatizovať spúšťanie jednotkových testov, kontrolu kvality kódu a taktiež aj kontrolu percentuálneho pokrytia kódu testami.

Najzásadnejším technickým rozšírením, ktoré má dopad na budúcu použiteľ-

nosť sémantického stroja, je zmena z konzolovej aplikácie na webovú službu dostupnú cez restové rozhranie. Vďaka tejto zmene je aplikácia sémantického stroja hneď po nasadení na server pripravená na používanie, či už používateľmi alebo ďalšími aplikáciami prostredníctvom API. Vďaka tejto zmene a taktiež zmenám už spomenutým je možné výsledné riešenie tejto diplomovej práce viac nepovažovať za prototyp, ale za plnohodnotný produkt pripravený na používanie, prípadne ďalšie rozširovanie.

7.3 Nedostatky riešenia

Počas rozširovania prototypu bolo nájdených mnoho funkčných aj technických nedostatkov. Tie sa však vo väčšine podarilo úspešne vyriešiť. Stále však ostalo zopár nedostatkov. Ako pred tým prototyp, tak aj aktuálne riešenie používa syntaktický analyzátor Stanford CoreNLP v stabilnej verzii 3.9.2. Pri implementovaní niektorých rozšírení boli pozorované zásadné rozdiely vo výstupe syntaktickej analýzy, ktorú sémantický stroj využíva. Problematickou je napríklad syntaktická analýza pri analyzovaní otázok, kedy pri použití otáznika sú zjavne problematicky určené závislosti medzi slovami, čo následne predstavuje problém pri typovej analýze. Ak sa v otázke otáznik nepoužije, veta je syntaktickým analyzátorom zanalyzovaná správne.

Nebolo pokladané za správne typovú analýzu ohýbať pre takýto prípad. Výhodnejším by do budúcna mohlo byť povýšenie používanej verzie syntaktického analyzátora na najaktuálnejšiu verziu 4.2.0.

7.4 Možnosti budúceho rozšírenia

Po úspešnom implementovaní potrebných zmien predstavuje sémantický stroj plnohodnotný produkt, ktorý je možné začať plnohodnotne používať ihneď. Pre úspešné používanie je potrebné vykonať nasadenie aplikácie a taktiež syntaktického analyzátora na server.

V tomto prípade je priestor pre budúce vylepšenie. Pre zjednodušenie procesu prípravy aplikácie na používanie sa ponúka viacero možností. Prvou možnosťou by mohlo byť pridanie úlohy do skriptu pre CICD, ktorá by nasadzovala webovú službu na aplikačný server, kde by bola služba dostupná verejne prípadne v rámci

domény.

Najideálnejšou možnosťou by však mohlo byť vytvorenie docker obrazu, ktorý by mohol obsahovať potrebné nastavenia prostredia, aplikáciu syntaktického analyzátora a taktiež aplikáciu sémantického stroja. Použitie takéhoto obrazu by bolo jednoduchšie, stačilo by nasadiť jeden obraz ktorý, by spustil obe aplikácie.

8 Záver

Cieľom diplomovej práce bolo rozšírenie prototypu sémantického stroja, ktorý ako východiskový logický aparát využíva transparentnú intenzionálnu logiku. Splnenie tohto cieľa znamenalo analyzovanie súčastného stavu prototypu po technickej a funkčnej stránke a naštudovanie logického aparátu a jeho možností pre funkčné rozšírenie prototypu, ktoré by umožnilo pokrytie väčšej množiny viet prirodzeného jazyka.

Po splnení týchto dvoch základných požiadaviek bolo potrebné navrhnúť jednotlivé kroky funkčného rozšírenia prototypu s dôrazom na ponechanie, prípadne vylepšenie pôvodnej funkcionality a taktiež navrhnúť technické rozšírenia, ktoré by v budúcnosti zabezpečili jednoduchšie rozširovanie sémantického stroja a taktiež urobili z prototypu produkt, pripravený na okamžité používanie.

Implementácia navrhnutých rozšírení po funkčnej stránke, znamenala vytvorenie nových algoritmov pre spracovanie konjunkcie, implikácie, premenných a kvantifikátorov. Tieto nové rozšírenia bolo potrebné zapracovať ako do typovej analýzy tak aj do tvorby konštrukcií TIL v rámci sémantického stroja. Plnenie týchto cieľov poukázalo na niektoré nedostatky prototypu, ktoré boli v rámci implementácie nových rozšírení odstránené.

Po úspešnom implementovaní spomenutých rozšírení, bolo potrebné vykonanie posledného kroku a teda vytvorenie webovej služby, ktorá by poskytovala funkcionalitu sémantického stroja. Táto posledná požiadavka bola realizovaná za pomoci aplikačného rámca Flask a jej splnenie umožnilo zmenu prototypu, na plnohodnotný produkt.

Vzhľadom na zložitosť a množstvo rozšírení, ktoré boli v rámci prezentovanej práce implementované, je možné považovať stanovené ciele za splnené. Práca taktiež poukazuje na ďalšie možnosti vylepšenia sémantického stroja, prípadne na odstránenie nedostatkov.

Literatúra

1. HOLSTER, Andrew. *An Introduction to Pavel Tichy and Transparent Intensional Logic.* 2003. Dostupné tiež z: http://philsci-archive.pitt.edu/1479/.
2. KALISH, D.; MONTAGUE, R.; MAR, G. *Logic: Techniques of Formal Reasoning*. Harcourt, Brace & World, 1964. ISBN 9780155511804. Dostupné tiež z: https://books.google.sk/books?id=VzoNAQAAMAAJ.
3. SPEAKS, Jeff. *An introduction to Freagean semantics* [https://www3.nd.edu/~jspeaks/courses/2007-8/93914/_HANDOUTS/fregeanism-introduction.pdf]. 2008. (Accessed on 12/13/2020).
4. PEZLAR, Ivo. On Two Notions of Computation in Transparent Intensional Logic. *Axiomathes*. 2018, roč. 29, č. 2, s. 189–205. Dostupné z DOI: 10.1007/s10516-018-9401-7.
5. DUZI, Marie. Logical Foundations of Conceptual Modelling Using HIT Data Model. In: 2000, s. 65–80.
6. CIPRICH, Nikola; DUZI, Marie; KOŠINÁR, Michael. The TIL-script language. In: 2008, zv. 190, s. 166–179. Dostupné z DOI: 10.3233/978-1-58603-957-8-166.
7. RACLAVSKÝ, Jiří. *A light introduction to transparent intensional logic and its application in semantics*. Department of Philosophy, Masaryk University, Brno, 2014.
8. TICHY, P. *The Foundations of Frege's Logic*. De Gruyter, 2012. Grundlagen der Kommunikation und Kognition / Foundations of Communication and Cognition. ISBN 9783110849264. Dostupné tiež z: https://books.google.sk/books?id=4d8hAAAAQBAJ.

9. FITTING, Melvin. Intensional Logic. In: ZALTA, Edward N. (ed.). *The Stanford Encyclopedia of Philosophy* [https://plato.stanford.edu/archives/spr2020/entries/logic-intensional/]. Spring 2020. Metaphysics Research Lab, Stanford University, 2020.
10. BENZMÜLLER, Christoph; ANDREWS, Peter. Church's Type Theory. In: ZALTA, Edward N. (ed.). *The Stanford Encyclopedia of Philosophy* [https://plato.stanford.edu/archives/sum2019/entries/type-theory-church/]. Summer 2019. Metaphysics Research Lab, Stanford University, 2019.
11. HORÁK, Aleš. TYPES IN TRANSPARENT INTENSIONAL LOGIC AND EASEL–A COMPARISON. 2004.
12. Questions and Logical Analysis of Natural Language: the Case of Transparent Intensional Logic. 2005, roč. 2004, č. 185-188.
13. HORÁK, Aleš. *The Normal Translation Algorithm in Transparent Intensional Logic for Czech*. 2001. Diz. pr. Faculty of Informatics, Masaryk University, Brno.
14. BEDNÁR, Branislav. *Sémantický stroj transparentnej intenzionálnej logiky*. 2019. Bakalárska práca. Fakulta elektrotechniky a informatiky, Technická univerzita v Košiciach.
15. MANNING, Christopher D.; SURDEANU, Mihai; BAUER, John; FINKEL, Jenny; BETHARD, Steven J.; MCCLOSKY, David. The Stanford CoreNLP Natural Language Processing Toolkit. In: *Association for Computational Linguistics (ACL) System Demonstrations*. 2014, s. 55–60. Dostupné tiež z: http://www.aclweb.org/anthology/P/P14/P14-5010.
16. MARNEFFE, Marie-Catherine de; MANNING, Christopher D. *Stanford typed dependencies manual* [online]. 2008-09 [cit. 2014-03-25]. Dostupné z: http://nlp.stanford.edu/software/dependencies_manual.pdf.
17. MARTIN, Robert C.; COPLIEN, James O. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, 2009. ISBN 9780132350884 0132350882.
18. YOUNG, David. Software Testing Overview. *White Paper*. 2015.
19. RACLAVSKY, Jiri. On Partiality and Tichý's Transparent Intensional Logic. *Magyar Filozofiai Szemle*. 2010, roč. 54, č. 4, s. 120–128.

20. DUŽÍ, Marie. Property Modifiers and Intensional Essentialism. *Computación y Sistemas*. 2017, roč. 21, č. 4. Dostupné tiež z: http://www.cys.cic.ipn.mx/ojs/index.php/CyS/article/view/2811.

21. PETERS, Stanley; WESTERSTÅHL, Dag. *Quantifiers in Language and Logic*. 2006. ISBN 978-0-19-929125-0. Dostupné z DOI: 10.1093/acprof:oso/9780199291267.001.0001.

22. Notes on mathematical logic. 2010. Dostupné tiež z: https://www.cs.yale.edu/homes/aspnes/pinewiki/attachments/MathematicalLogic/mathematical-logic.pdf.

23. ARACHCHI, S A I B; PERERA, Indika. Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. In: 2018. Dostupné z DOI: 10.1109/MERCon.2018.8421965.

24. ASLAM, Fankar; MOHAMMED, Hawa; LOKHANDE, Prashant. Efficient Way Of Web Development Using Python And Flask. *International Journal of Advanced Research in Computer Science*. 2015, roč. 6.

25. YEGULALP, Serdar. *Review: 13 Python web frameworks compared* [https://www.infoworld.com/article/3105502/review-13-python-web-frameworks-compared.html]. 2018.

Zoznam skratiek

ADVCL Modifikátor príslovkovej vety.

API Aplikačné programovacie rozhranie.

CC Koordinátor.

CICD Postupná integrácia a postupné doručovanie.

CONJ Konjunkcia.

DET Determinant.

GB Giga Bajt.

HIT Homogénny, Integrovaný, Typovo-orientovaný.

JSON JavaScript object notation.

MARK Marker.

NSUBJ Nominálny predmet.

POS Slovný druh.

TIL Transparentná intenzionálna logika.

Zoznam príloh

Príloha A Používateľská príručka,

Príloha B Systémová príručka,

Príloha C Príklady spracovaných viet

Príloha D CD médium – záverečná práca v elektronickej podobe

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Rozšírenie prototypu sémantického stroja transparentnej intenzionálnej logiky

Diplomová práca

Príloha A: Používateľská príručka

Študijný program:	Informatika
Študijný odbor:	9.2.1. Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	Ing. Zuzana Bilanová, PhD.

Košice 2021 **Bc. Branislav Bednár**

A Používateľská príručka

Používateľská príručka k webovej službe sémantického stroja, používajúceho transparentnú intenzionálnu logiku. Webová služba spracováva vety prirodzeného jazyka a vykonáva nad nimi syntaktickú a sémantickú analýzu. Po úspešnom analyzovaní vety je konzumentovi služby vrátený sformulovaný význam danej vety pomocou konštrukcie TIL.

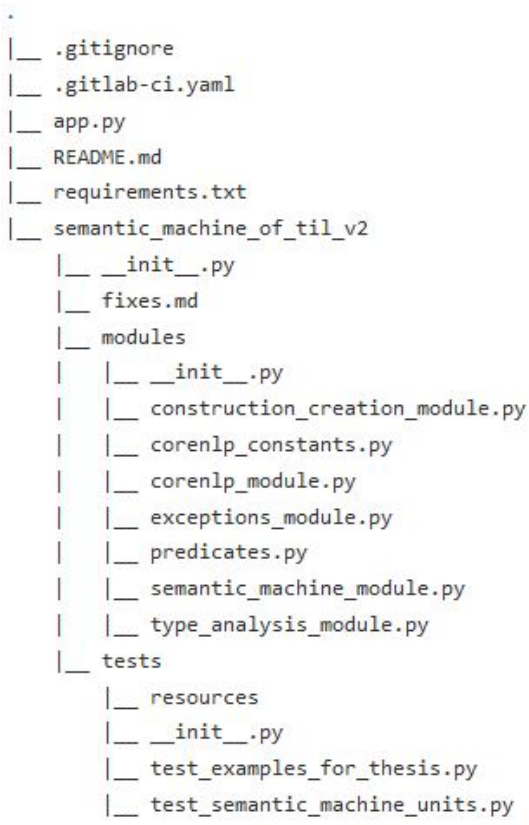
A.1 Štruktúra projektu

Na obrázku A.1, je možné vidieť štruktúru projektu webovej služby sémantického stroja.

A.2 Systémové požiadavky

Riešenie webovej služby sémantického stroja bolo navrhnuté so zreteľom na jednoduchosť používania a taktiež na nenáročnosť systémových požiadaviek, v prípade používania v rámci akademickej pôdy prípadne výučbového procesu. Na úspešné spustenie a používanie webovej služby, je teda potrebné splnenie piatich nevyhnutných systémových požiadaviek:

- Java vo verzii minimálne 1.8,
- Python vo verzii minimálne 3.7.0,
- Pip vo verzii minimálne 20.3.3,
- Stanford CoreNLP vo verzii 3.9.2,
- voľná kapacita na disku 1GB.



Obr. A.1: Štruktúra projektu webovej služby

Java v danej verzii je pre webovú službu nevyhnutná z dôvodu spustenia servera syntaktického analyzátora Stanford CoreNLP. Ten je možné v potrebnej verzii stiahnuť na nasledujúcej adrese:

https://stanfordnlp.github.io/CoreNLP/history.html

Inštalácia Pythonu je potrebná pre samotné spustenie a používanie webovej služby a taktiež je nutná inštalácia balíčkového inštalátora *Pip*, vďaka ktorému je možné stiahnuť všetky balíčky v správnych verziách potrebné pre beh a používanie webovej služby.

Poslednou požiadavkou je voľný diskový priestor o veľkosti 1GB, čo je potrebné hlavne z dôvodu používania lokálneho servera Stanford CoreNLP.

A.3 Konfigurácia prostredia

Balíčkový inštalátor *Pip* zabezpečí správnu konfiguráciu prostredia pre spustenie a beh aplikácie. Ako je možné vidieť na obrázku A.1, koreňový adresár projektu

obsahuje súbor `requirements.txt`. Daný súbor obsahuje všetky balíky spolu s verziami, ktoré sú pre webovú službu potrebné. Tieto balíky je potrebné nainštalovať zadaním nasledujúceho príkazu do príkazového riadka v koreňovom adresári projektu:

```
$ pip install -r requirements.txt
```

A.4 Spustenie webovej služby

Prvým krokom spustenia webovej služby je spustenie syntaktického analyzátora Stanford CoreNLP v potrebnej verzii. Ten je potrebné spustiť v koreňovom adresári stiahnutého analyzátora `/stanford-corenlp-full-2018-10-05` príkazom:

```
$ java -mx1g -cp "*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer
-port 9000 -timeout 15000
```

Obsah príkazového riadka po úspešnom spustení syntaktického analyzátora je možné vidieť na obrázku A.2.

```
D:\School\TUKE\Master_thesis\stanford-corenlp-full-2018-10-05>java -mx1g -cp "*" edu.stanford
.nlp.pipeline.StanfordCoreNLPServer -port 9000 -timeout 15000
[main] INFO CoreNLP - --- StanfordCoreNLPServer#main() called ---
[main] INFO CoreNLP - setting default constituency parser
[main] INFO CoreNLP - warning: cannot find edu/stanford/nlp/models/srparser/englishSR.ser.gz
[main] INFO CoreNLP - using: edu/stanford/nlp/models/lexparser/englishPCFG.ser.gz instead
[main] INFO CoreNLP - to use shift reduce parser download English models jar from:
[main] INFO CoreNLP - http://stanfordnlp.github.io/CoreNLP/download.html
[main] INFO CoreNLP -     Threads: 8
[main] INFO CoreNLP - Starting server...
[main] INFO CoreNLP - StanfordCoreNLPServer listening at /0:0:0:0:0:0:0:0:9000
```

Obr. A.2: Spustenie servera Stanford CoreNLP

Po úspešnom spustení syntaktického analyzátora je možné spustenie samotnej webovej služby sémantického stroja. Spustenie je možné vykonať pomocou príkazového riadka v koreňovom adresári projektu, zadaním príkazu:

```
$ py app.py
```

Obsah príkazového riadka po úspešnom spustení webovej služby sémantického stroja je možné vidieť na obrázku A.3

```
D:\School\TUKE\Master_thesis_impl\master-thesis-semantic-machine-2.0>py app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

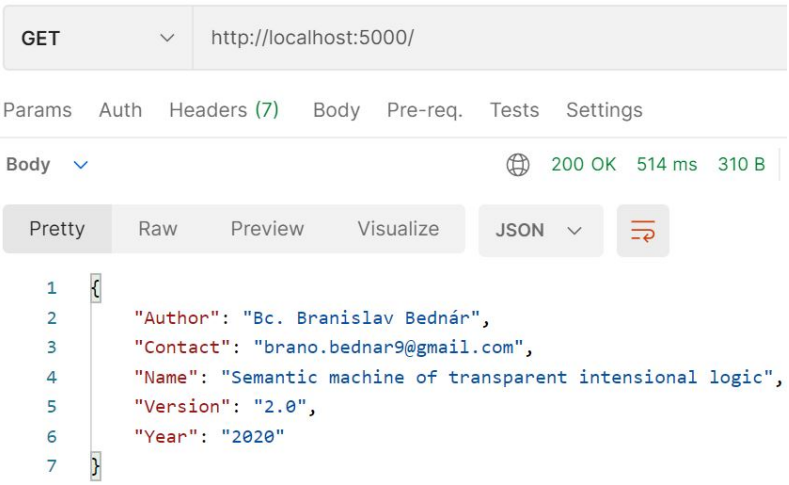
Obr. A.3: Spustenie webovej služby sémantického stroja

A.5 Používanie webovej služby

Používanie webovej služby je možné pomocou restového rozhrania. Služba obsahuje dva koncové body. Pre prehľadnosť je na demonštrovanie používania sémantického stroja použitá aplikácia Postman.

```
GET http://localhost:5000/
POST http://localhost:5000/semantic-machine
```

Prvý koncový bod je informačný, pomocou ktorého vieme overiť, že webová služba je spustená a taktiež obsahuje základné informácie o službe, ako je možné vidieť na obrázku A.4.

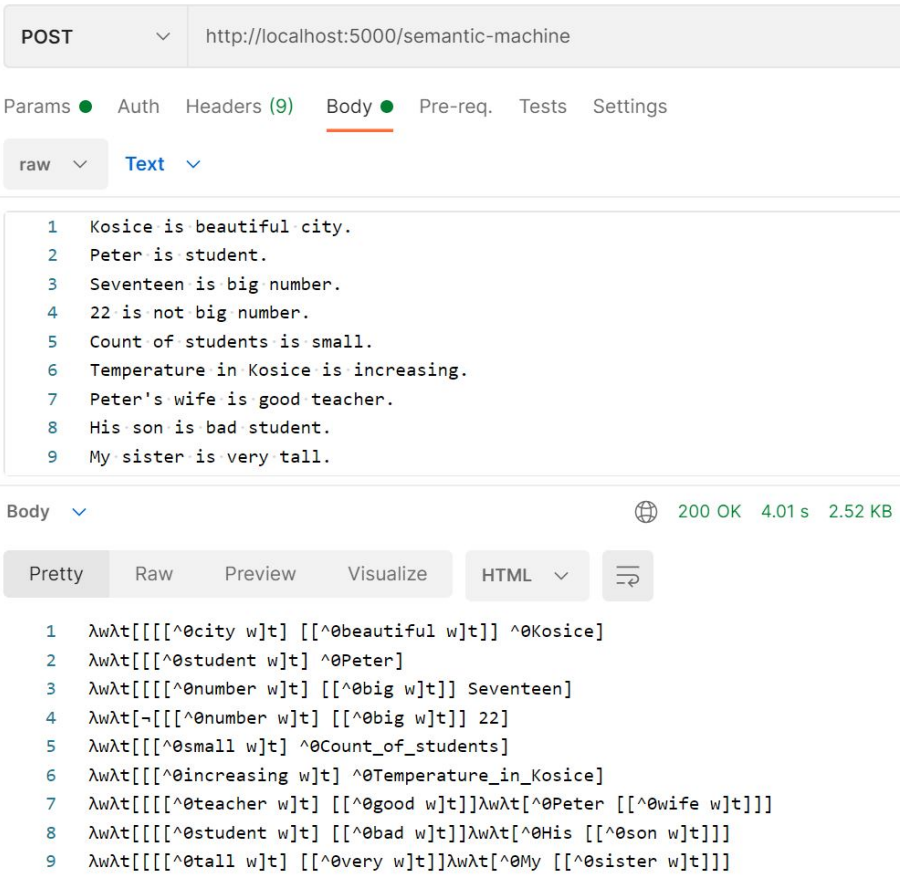


Obr. A.4: Prevolanie informačného koncového bodu

Alternatívne je možné použiť aj príkazový riadok:

```
$ curl -X GET http://localhost:5000 | python -m json.tool
```

Druhý koncový bod slúži na samotné používanie sémantického stroja. Základné použitie tohto koncového bodu je možné vidieť na obrázku A.5, kde do tela požiadavky vložíme vety prirodzeného jazyka.



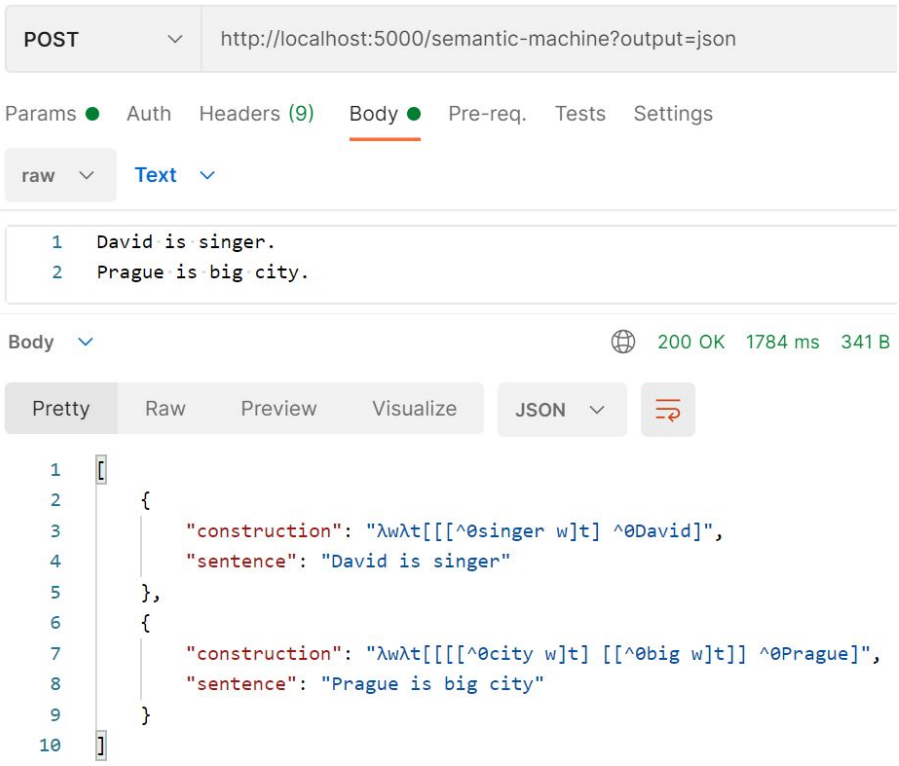
Obr. A.5: Používanie sémantického stroja ako webovej služby

Webová služba ponúka možnosť vrátenia odpovede vo formáte JSON, pre prípad, že by s ňou spolupracovala iná aplikácia. To je možné zabezpečiť dotazovým parametrom *output=json*, čo je možné vidieť na obrázku A.6.

V oboch prípadoch je znova možné použiť sémantický stroj pomocou príkazového riadka.

```
$ curl -X POST -H "Content-Type: text/plain" --data "David is singer."
    http://localhost:5000/semantic-machine

$ curl -X POST -H "Content-Type: text/plain" --data "David is singer."
    http://localhost:5000/semantic-machine?output=json
```



Obr. A.6: Použitie dotazového parametra *output=json*

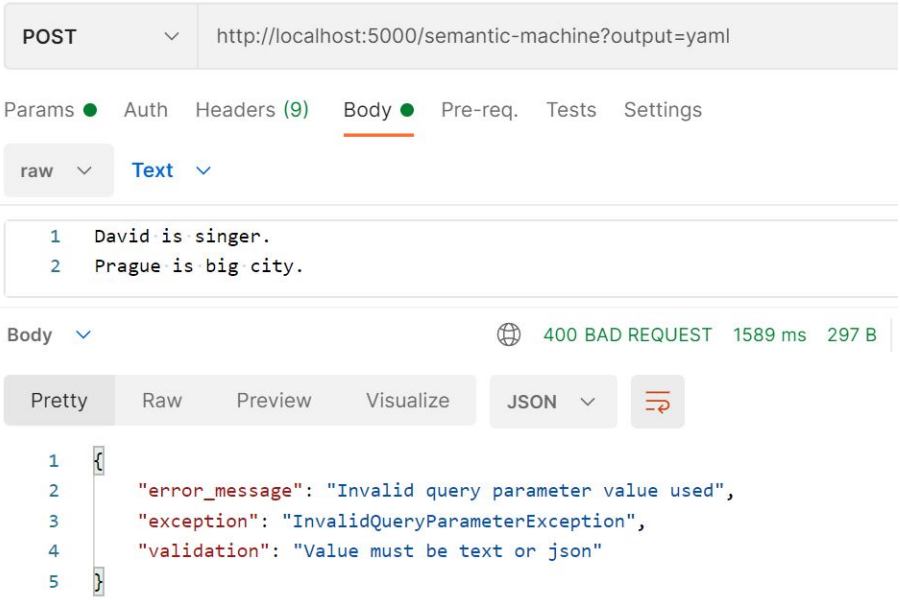
A.6 Negatívne scenáre

Každá webová služba potrebuje reagovať na chyby korektnou odpoveďou, ktorá konzumentovi napovie, či ide o chybu v požiadavke alebo v službe samotnej. Webová služba sémantického stroja taktiež zastrešuje aj takéto negatívne scenáre.

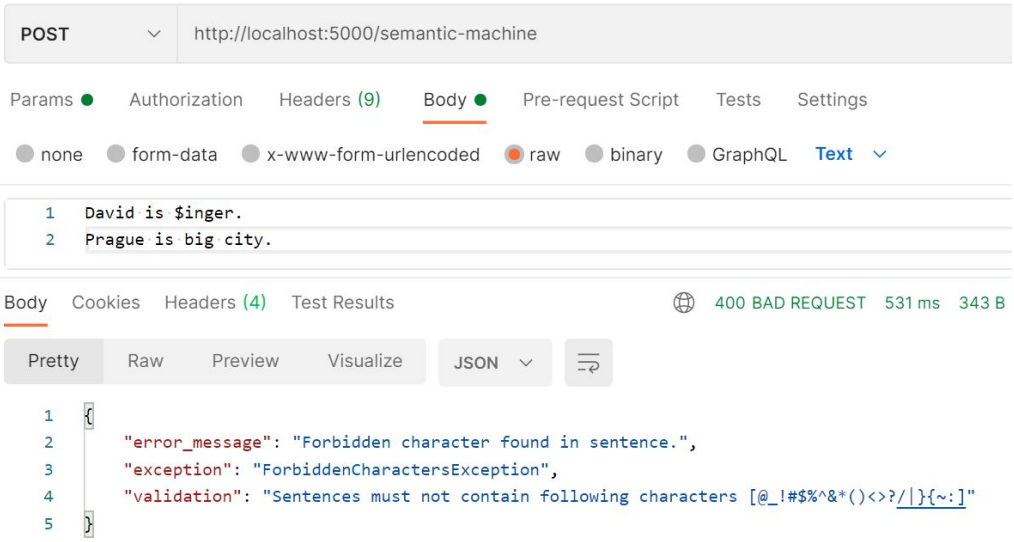
Vo všeobecnosti výnimky v rámci prezentovanej webovej služby obsahujú chybovú správu, ktorá popisuje aká chyba nastala, výnimku, ktorá bola sémantickým strojom vyhodená a validáciu, ktorá bola neúspešná.

Na obrázku A.7 je možné vidieť odpoveď webovej služby v prípade, že bol zadaný nevalidný dotazový parameter.

Ďalším negatívnym scenárom je situácia, kedy telo používateľskej požiadavky obsahuje nepovolený znak. Kontrola nepovolených znakov má primárne bezpečnostný charakter, aby telo požiadavky neobsahovalo napríklad škodlivý skript, ktorý by mohol napadnúť, prípadne prevziať kontrolu nad hostiteľským systémom. Vysporiadanie sa s takýmto scenárom je možné vidieť na obrázku A.8.



Obr. A.7: Nevalidný dotazový parameter



Obr. A.8: Nepovolený znak v tele požiadavky

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Rozšírenie prototypu sémantického stroja transparentnej intenzionálnej logiky

Diplomová práca

Príloha B: Systémová príručka

Študijný program:	Informatika
Študijný odbor:	9.2.1. Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	Ing. Zuzana Bilanová, PhD.

Košice 2021 **Bc. Branislav Bednár**

B Systémová príručka

Systémová príručka k webovej službe sémantického stroja, používajúceho transparentnú intenzionálnu logiku. Webová služba spracováva vety prirodzeného jazyka a vykonáva nad nim syntaktickú a sémantickú analýzu. Po úspešnom analyzovaní vety je konzumentovi služby vrátený sformulovaný význam danej vety pomocou konštrukcie TIL.

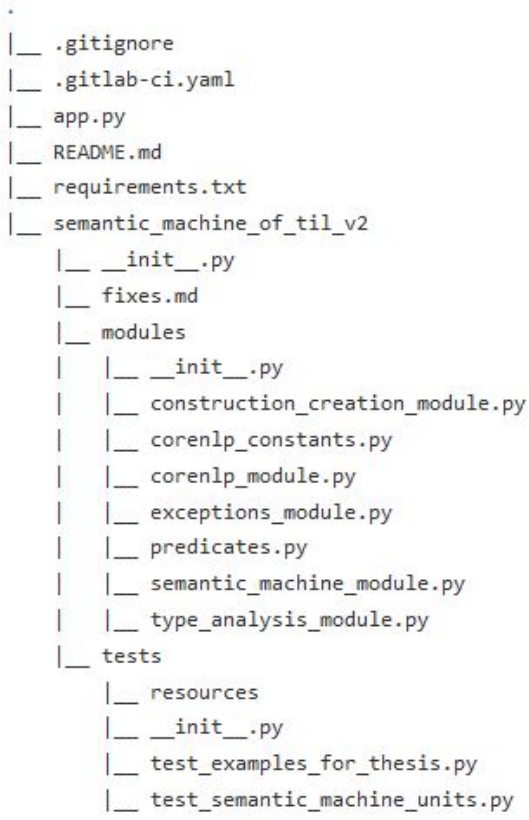
B.1 Systémové požiadavky

Riešenie webovej služby sémantického stroja bolo navrhnuté so zreteľom na jednoduchosť používania a taktiež na nenáročnosť systémových požiadaviek, Pre potreby rozširovania alebo spustenia a používania webovej služby, je potrebné splnenie piatich nevyhnutných systémových požiadaviek

- Java vo verzii minimálne 1.8,
- Python vo verzii minimálne 3.7.0,
- Pip vo verzii minimálne 20.3.3
- Stanford CoreNLP vo verzii 3.9.2,
- voľná kapacita na disku 1GB.

B.2 Štruktúra projektu

Na obrázku B.1, je možné vidieť štruktúru projektu webovej služby sémantického stroja. Koreňový adresár obsahuje tieto dôležité súbory:



Obr. B.1: Štruktúra projektu webovej služby

- app.py - základný modul, ktorý inicializuje webovú službu sémantického stroja,
- README.md - súbor obsahujúci základné informácie o projekte,
- requirements.txt - súbor obsahujúci zoznam balíčkov s konkrétnymi verziami, ktoré sú v projekte využívané.

Projekt následne obsahuje základný balíček `\semantic_machine_of_til_v2`, ktorý je rozdelený na ďalšie dva balíčky.

Prvý balíček obsahuje moduly aplikácie, ktoré sú logicky členené podľa svojej funkcionality, a taktiež sa tu nachádzajú pomocné súbory využívané v týchto moduloch. Ide o balíček `\semantic_machine_of_til_v2\modules`.

Balíček `\semantic_machine_of_til_v2\tests` zaisťuje testovanie v rámci aplikácie. Obsahuje jednotkové testy, a taktiež zdrojové súbory k testom, ktoré sú vo forme textových súborov. Vďaka tomu, že projekt disponuje samostatným balíč-

kom, ktorý zastrešuje testovanie, je možné tento balíček využiť na automatizované testovanie.

B.3 Implementácia

Pre jednoduchosť budúceho rozširovania, prípadne úprav sémantického stroja [14], bola implementácia prototypu rozdelená do logických modulov, ktoré zastrešujú jeho jednotlivé funkcionality

B.3.1 semantic_machine_module.py

Modul zastrešujúci celú funkcionalitu sémantického stroja, obsahuje nasledujúce metódy:

- `def execute_semantic_machine(sentence_for_parsing)` - metóda, ktorá overí používateľský vstup a následne inicialiuje ďalšie kroky spracovávania vety prirodzeného jazyka, syntaktickú a sémantickú analýzu,
- `def provide_syntactic_analysis(sentence_for_parsing)` - metóda, ktorá inicializuje spojenie so syntaktickým analyzátorom, následne doň odošle vetu prirodzeného jazyka, pre ktorú je potrebné vykonať analýzu, a vráti výstup syntaktickej analýzy,
- `def provide_semantic_analysis(core_nlp_response)` - metóda, ktorá v prvom kroku vykoná typovú analýzu pre vetu prirodzeného jazyka a následne vytvorí konštrukciu významu tejto vety.

B.3.2 corenlp_module.py

Modul, ktorého funkciou je mapovanie výsledku syntaktickej analýzy do objektov, čo umožňuje prechádzať rôznym problémom. Objekty, do ktorých je mapovaný výstup syntaktickej analýzy:

- `class Dependency` - trieda, do ktorej sú mapované závislosti medzi jednotlivými lexikálnymi jednotkami vo vete,
- `class Sentence` - trieda, do ktorej je mapovaná syntaktická analýza pre jednu vetu,

- `class Token` - trieda, do ktorej sú mapované jednotlivé lexikálne jednotky.

Mapovanie do týchto tried zabezpečujú nasledujúce metódy:

- `def json_mapper(core_nlp_data)`,
- `def parse_dependencies(basic_dependencies)`,
- `def parse_tokens(tokens)`.

B.3.3 type_analysis_module.py

Zoznam väčšiny metód využitých v rámci modulu typovej analýzy bol popísaný v systémovej príručke prototypu [14]. Oproti prototypu však pribudli nasledovné metódy, ktoré zabezpečujú novú funkcionalitu:

- `def find_conj_for_pop()` - metóda slúžiaca na nájdenie konjunkcie vo vete nezávisle od typov,
- `def find_impl_for_pop()` - metóda slúžiaca na nájdenie implikácie vo vete nezávisle od typov,
- `def find_quantified_pop()` - metóda slúžiaca na nájdenie kvantifikovaných výrazov,
- `def find_pop_without_individuals()` - metóda slúžiaca na nájdenie typu vlastnosť objektu vo vete, kde sa nenachádza žiadne indivíduum.

B.3.4 construction_creation_module.py

Rovnako ako pri module pokrývajúcom typovú analýzu aj v tomto prípade bola väčšina metód popísaná už v systémovej príručke prototypu [14]. Pribudlo však niekoľko nových tried, ktoré zabezpečujú novú funkcionalitu, prípadne zefektivňujú tvorbu konštrukcie.

- `class ConstructionElement` - všeobecná trieda predstavujúca element konštrukcie a zabezpečujúca jeho správne skonštruovanie, napríklad pridanie trivializácie, intenzionalizáciu elementu a podobne,

- `class Conjunction` - všeobecná trieda predstavujúca konjunkciu, ktorá môže byť v rámci transparentnej intezionálnej logiky vo forme *and*, prípadne *or*.

Taktiež pre potreby rozšírenia sémantického stroja vznikli nové metódy.

- `def construct_needed_type()` - metóda zabezpečujúca konštruovanie potrebného typu transparentnej intenzionálnej logiky,
- `def find_quantifier()` - metóda zabezpečujúca nájdenie kvantifikátora vo vete,
- `def find_variables()` - metóda zabezpečujúca nájdenie premennej vo vete a pridanie nájdenej premennej do konštrukcie vety,
- `def find_conjunction()` - metóda zabezpečujúca nájdenie konjunkcie vo vete,
- `def construct_conjunctions()` - všeobecná metóda zabezpečujúca konštruovanie konjunkcie bez ohľadu na typ, ktorý je s konjunkciou spojený,
- `def update_construction_w_conjunctions()` - metóda zabezpečujúca pridanie skonštruovanej konjunkcie do doterajšej konštrukcie významu vety,
- `def find_implications()` - metóda zabezpečujúca nájdenie implikácie vo vete,
- `def construct_implications()` - všeobecná metóda zabezpečujúca konštruovanie implikácie bez ohľadu na typ, ktorý je s implikáciou spojený,
- `def update_construction_with_implications()` - metóda zabezpečujúca pridanie skonštruovanej implikácie do doterajšej konštrukcie významu vety.

B.3.5 exceptions_module.py

Modul obsahujúci definície výnimiek, ktoré môžu byť počas behu sémantického stroja identifikované.

- `class ForbiddenCharacters(Exception)` - výnimka je identifikovaná v prípade, kedy používateľský vstup obsahuje niektorý z nepovolených znakov,
- `class InvalidQueryParameter(Exception)` - výnimka je identifikovaná v prípade, kedy hodnota dostupného dotazovaného parametra je nevalidná.

B.3.6 Ostatné

V rámci implementácie sémantického stroja existujú dva dodatočné pomocné moduly, prvým je `corenlp_constants.py`, ktorý obsahuje konštanty súvisiace s výstupom syntaktickej analýzy. Druhým pomocným modulom je `predicates.py`, ktorý obsahuje predikáty, teda boolovské metódy so zložitými podmienkami, ktoré by v rámci štandardných modulov znižovali čitateľnosť zdrojového kódu.

B.4 Testovanie

Pre potreby automatizovaného testovania bol vytvorený samostatný balík s názvom `\semantic_machine_of_til_v2\tests`. V ňom je možné nájsť okrem modulov aj adresár `\resources`, ktorý obsahuje vstupné dáta pre testy, teda výstupy syntaktickej analýzy, ktoré sú následne využívané na jednotkové testovanie funkcionality sémantického stroja.

Balík taktiež obsahuje dva samostatné moduly. Prvý z dvoch testovacích modulov je `test_examples_for_thesis.py`. Ten je iba ukážkový pre potreby analýzy testovacích knižníc v rámci hlavného textu diplomovej práce. Poukazuje na rozdiel pri použití knižnice *Pytest* a *Unittest*.

Druhým modulom je `test_semantic_machine_units.py`. Tento modul obsahuje jednotkové testy, ktoré overujú správnosť vykonania typovej analýza pre jednotlivé typy transparentnej intezionálnej logiky a taktiež obsahuje test na overenie správneho vytvorenia konštrukcií viet prirodzeného jazyka.

Pre spustenie všetkých testov je potrebné v balíku `\tests` spustiť:

```
$ py -m pytest -v
```

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Rozšírenie prototypu sémantického stroja transparentnej intenzionálnej logiky

Diplomová práca

Príloha C: Príklady spracovaných viet

Študijný program:	Informatika
Študijný odbor:	9.2.1. Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	Ing. Zuzana Bilanová, PhD.

Košice 2021 **Bc. Branislav Bednár**

C Príklady spracovaných viet

If Tony is good engineer, then he is clever.
$\lambda w \lambda t\ [\Rightarrow [[[[^0 engineer\ w]t]\ [[^0 good\ w]t]\ ^0 Tony]\ [[[^0 clever\ w]t]\ ^0 Tony]]$

If Samuel is teacher, Matthew is student.
$\lambda w \lambda t\ [\Rightarrow [[[^0 teacher\ w]t]\ ^0 Samuel\ [[[^0 student\ w]t]\ ^0 Matthew]]$

If Jane is singer, then Monica is dancer.
$\lambda w \lambda t\ [\Rightarrow [[[^0 singer\ w]t]\ ^0 Jane\ [[[^0 dancer\ w]t]\ ^0 Monica]]$

If Theodore is player, then 8 is big number.
$\lambda w \lambda t\ [\Rightarrow [[[^0 player\ w]t]\ ^0 Theodore]\ [[[[^0 number\ w]t]\ [[^0 big\ w]t]\ 8]]$

Given that Simon hates Jane, then he is hater.
$\lambda w \lambda t\ [\Rightarrow [[[^0 hates\ w]t]\ ^0 Simon\ ^0 Jane]\ [[[^0 hater\ w]t]\ ^0 Simon]]$

Chris is groom, if his wife is married.
$\lambda w \lambda t\ [\Rightarrow [[[^0 married\ w]t] \lambda w \lambda t [^0 his\ [[^0 wife\ w]t]]]\ [[[^0 groom\ w]t]\ ^0 Chris]]$

Unless Monica is smiling, Chandler is joker.
$\lambda w \lambda t\ [\Rightarrow [[[^0 smiling\ w]t]\ ^0 Monica]\ [[[^0 joker\ w]t]\ ^0 Chandler]]$

If Tony is singer, Matthew is dancer and Andrew is student.
$\lambda w \lambda t\ [\Rightarrow [[[^0 singer\ w]t]\ ^0 Tony]\ [[[^0 dancer\ w]t]\ ^0 Matthew \wedge [[^0 student\ w]t]\ ^0 Andrew]]$

If Andrew is musical student, he will be musician.
$\lambda w \lambda t\ [\Rightarrow [[[^0 student\ w]t]\ [[^0 musical\ w]t]]\ ^0 Andrew]\ [[[^0 musician\ w]t]\ ^0 Andrew]]$

Howard is student and musician and dancer and sportsman.
$\lambda w \lambda t [[[^0 student\ w]t]\ ^0 Howard \wedge [[^0 musician\ w]t]\ ^0 Howard \wedge [[^0 sportsman\ w]t]\ ^0 Howard]$

Simon is good student and Mathew is not gorgeous musician.
$\lambda w \lambda t[[[[^0 student\ w]t]\ [[^0 good\ w]t]]\ ^0 Simon \wedge \neg[[[^0 musician\ w]t]\ [[^0 gorgeous\ w]t]]\ ^0 Matthew]$

Thomas is student and 315 is big number.
$\lambda w \lambda t[[[^0 student\ w]t]\ ^0 Thomas \wedge [[[^0 number\ w]t]\ [[^0 big\ w]t]]\ 315.$

193 is big number and 17 is prime.
$\lambda w \lambda t[[[[^0 number\ w]t]\ [[^0 big\ w]t]]\ 193 \wedge [[^0 prime\ w]t]\ 17]$

Some student is programmer and good dancer.
$\lambda w \lambda t[^0 \exists \lambda x\ [[[^0 programmer\ w]t]\ x \wedge [[^0 student\ w]t]\ x] \wedge [[[[^0 dancer\ w]t]\ x\ [^0 good\ w]t] \wedge [[^0 student\ w]t]\ x]]$

Are some employees professors?
$\lambda w \lambda t[^0 \exists \lambda x\ [[[^0 professors\ w]t]\ x \wedge [[^0\ employees\ w]t]\ x]]$

All students are good musicians.
$\lambda w \lambda t[^0 \forall \lambda x\ [[[[^0 musicians\ w]t]\ x\ [[^0 good\ w]t]\ x] \wedge [[^0 students\ w]t]\ x]]$

Which professors are dancers
$\lambda w \lambda t[^0 \exists \lambda x\ [[[^0 dancers\ w]t]\ x \wedge [[^0 professors\ w]t]\ x]]$

Some student is programmer and good dancer.
$\lambda w \lambda t[^0 \exists \lambda x\ [[[^0 programmer\ w]t]\ x \wedge [[^0 student\ w]t] \wedge [[[[^0 dancer\ w]t]\ x\ [[^0 good\ w]t] \wedge [[^0 student\ w]t]\ x]]$

If some student is engineer, he is clever.
$\lambda w \lambda t\ [\Rightarrow[[[[^0 engineer\ w]t]\ x \wedge [[^0 student\ w]t]\ x]]\ [^0 \exists \lambda x\ [[[^0 clever\ w]t]\ x \wedge [[^0 sudent\ w]t]\ x]]]$

Thomas hate some student.
$\lambda w \lambda t[^0 \exists \lambda x\ [[^0 hate\ w]t]\ ^0 Thomas\ [[^0 student\ w]t]\ x]$

Some student hates Rachel and loves Phoeobe.
$\lambda w \lambda t[^0 \exists \lambda x\ [[^0 hates\ w]t]\ [[^0 student\ w]t]\ x\ ^0 Rachel \wedge [[^0 loves\ w]t]\ [[^0 student\ w]t]\ x$

^{0}*Phoeobe*]

Some number is prime.
$\lambda w \lambda t[^0\exists \lambda x\ [[[^0prime\ w]t]\ x \wedge [[^0number\ w]t]\ x]]$

If some engineer is smart, he is clever.
$\lambda w \lambda t\ [\Rightarrow [[[[^0smart\ w]t]\ x \wedge [[^0engineer\ w]t]\ x]]\ [[^0\exists \lambda x\ [[[^0clever\ w]t]\ x \wedge [[^0engineer\ w]t]\ x]]]$

His son is bad student.
$\lambda w \lambda t[^0\exists \lambda x\ [[[[^0student\ w]t]\ [[^0bad\ w]t]]\lambda w \lambda t[^0His\ [[^0son\ w]t]]]$

Andrew is not good student and good player.
$\lambda w \lambda t[\neg[[[^0student\ w]t]\ [[^0good\ w]t]]\ ^0Andrew \wedge [[[^0player\ w]t]\ [[^0good\ w]t]\ ^0Andrew]$

Emily is student and musician and dancer.
$\lambda w \lambda t[[[^0student\ w]t]\ ^0Emily \wedge [[^0musician\ w]t]\ ^0Emily \wedge [[^0dancer\ w]t]\ ^0Emily]$