Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky
a informatiky

Jednoduchý syntaktický parser príbehov z Multimediálnej čítanky

Bakalárska práca

2016 Martin Heinz

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky
a informatiky

Jednoduchý syntaktický parser príbehov z Multimediálnej čítanky

Bakalárska práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	2511 aplikovaná informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	RNDr. Marek Nagy, PhD.

Bratislava 2016 Martin Heinz

87459805

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Martin Heinz
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: aplikovaná informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Jednoduchý syntaktický parser príbehov z Multimediálnej čítanky
Simple Syntactic Parser of Stories from Multimedia Reader

Cieľ: Cieľom je vytvoriť jednoduchý syntaktický parser v jazyku JavaScript, ktorý bude využitý v aplikácii Multimediálna čítanka. Vstupom bude text, o ktorom sa predpokladá, že je viacmenej zapísaný podľa gramatických pravidiel. Vo výstupe budú identifikované slová a vety pomocou XML značiek. Vo vstupnom texte možno, v ďalšom kroku, predpokladať aj prítomnosť HTML formátovácích značiek, ktoré sa zachovajú. Očakáva sa istá univerzálnosť, časová efektívnosť a parsovanie slovenčiny i angličtiny.

Vedúci: RNDr. Marek Nagy, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 27.09.2016

Dátum schválenia: 17.10.2016 doc. RNDr. Damas Gruska, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Chcel by som sa poďakovať môjmu školiteľovi, RNDr. Marekovi Nagyovi, PhD., za svoj čas, trpezlivosť, cenné rady, pripomienky a usmernenia pri tvorbe tejto bakalárskej práce.

Čestné vyhlásenie

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne s použitím uvedených zdrojov.

V Bratislave .....

Abstrakt

HEINZ, Martin: Jednoduchý syntaktický parser príbehov z Multimediálnej čítanky [Bakalárska práca]. Fakulta matematiky, fyziky a informatiky Univerzita Komenského v Bratislave; Katedra aplikovanej informatiky. Školiteľ: RNDr. Marek Nagy, PhD. Bratislava 2017. 34 strán.

Cieľom tejto bakalárskej práce je vytvoriť syntaktický parser v jazyku JavaScript, ktorý bude využitý ako modul na parsovanie príbehov v aplikácii Multimediálna čítanka. Táto práca obsahuje popis a analýzu problematiky a parsovacieho algoritmu použitého pri tvorbe tejto aplikácie. Súčasťou práce je aj návrh a implementácia samotnej aplikácie. Táto aplikácia je navrhnutá s cieľom implementovať efektívnu metódu delenia komplexného textu obsahujúceho HTML značky na slová a vety s jednoduchým používaním a modifikovateľnosťou.

Kľúčové slová: *parser, javascriptová aplikácia, HTML, XML*

Abstract

HEINZ, Martin: Simple Syntactic Parser of Stories from Multimedia Reader [Bachelor thesis]. Faculty of Mathematics, Physics and Informatics; Department of Applied Informatics, Comenius University in Bratislava; Supervisor: RNDr. Marek Nagy, PhD. Bratislava 2017. 34 pages.

Goal of this work is to create a syntactic parser in JavaScript, which is intened to be used as a submodule for parsing of stories in Multimedia reader. This work contains description and analysis of issues and parsing algorithm used for development of this application. This work also contains design and implentation of application itself. This application is developed with goal of implementing effective method to split complex text containing HTML tags into words and sentences, while being easy to use and modify.

Keywords: *parser, javascript application, HTML, XML*

Obsah

Úvod

V dnešnej dobe je spracovávanie textu bežnou vecou, s ktorou sa stretávame veľmi často a vo veľa rôznych situáciách. Mnohé aplikácie pre svoje správne fungovanie potrebujú text rozdelený do nejakých menších celkov, napríklad viet a slov. Vo veľa prípadoch však správnosť, prípadne efektívnosť takéhoto spracovania nie je na dostatočnej úrovni a ani zďaleka sa nepribližuje presnosti, s ktorou vie takýto text spracovať človek. Takéto spracovávanie ľuďmi je však nerealistické, kvôli množstvu textu a zdĺhavosti samotného procesu.

Hlavným cieľom tejto bakalárskej práce je vytvoriť aplikáciu - syntaktický parser - ktorá bude využitá na parsovanie príbehov z Multimediálnej čítanky, ktoré bude schopná s čo najväčšou presnosťou a efektívnosťou deliť na jednotlivé slová a vety, so zachovaním pôvodného formátovania textu a s dôrazom na univerzálnosť tohoto riešenia.

Motiváciou pre vytvorenie tejto aplikácie bol fakt, že mnohé časti Multimediálnej čítanky pracujú s vetami alebo jednotlivými slovami a spoliehajú sa ich korektnosť. Či už je to výpočet syntaktickej náročnosti, generovanie otázok z viet, rôzne úlohy na čítanie s porozumením alebo analýza viet. Všetky tieto procesy sú priamo závislé od správneho rozdelenia textu a preto si myslím, že je potrebné vytvoriť aplikáciu, ktorá by toto zabezpečovala.

Práca je rozdelená do štyroch kapitol. Prvá z nich sa zaoberá cieľom, problematikou parsovania, porovnaním jednotlivých parsovacích algoritmov a analýzou algoritmu zvoleného pre túto aplikáciu. Táto kapitola tiež zahŕňa prehľad existujúcich podobných systémov a stručný náhľad na použité technológie potrebné pri implementácii výslednej aplikácie. Druhá kapitola je zameraná na návrh a samotnú implementáciu. V tejto kapitole sú predstavené jednotlivé časti aplikácie, ich funkcionalita a popis realizácie niektorých funkcií. Posledná kapitola pozostáva z testov a porovnaní efektívnosti samotného riešenia.

Záver tejto bakalárskej práce poskytuje zhodnotenie naplnenia stanovených cieľov, ako aj pohľad na ďalšie možnosti vývoja v tejto oblasti.

Kapitola 1

Východiská

1.1 Cieľ

Cieľom tejto práce je nájdenie vhodného parseru, ktorý by bol v čo najkratšom čase schopný rozdeľovania komplexného textu na vety a slová a vytvorenie aplikácie pozostávajúcej z tohoto parseru, jeho gramatiky(slovenskej aj anglickej) a slovníku a časti spracovávajúcej výstup parseru do výsledného XML dokumentu. Zároveň by gramatika aj slovník parseru mali byť ľahko modifikovateľné a mala by existovať aj možnosť pridávať ďalšie jazyky. Príklad vstupu a výstupu:

Listing 1.1: Vstup

```
1 Toto <p> je vstupný text. Obsahuje čísla, napr.: č. 1, 5.(piaty),
      alebo mená - M. Heinz a aj priamu reč... </p>
2 Eva sa <b> pýtala: - Čo ti povedali!?</b>
3 "Ahoj," povedal, "ako sa..." skočil mu do reči, "<i>ticho!</i>"
4 - Prídeš? - spýtal sa.
5 - Prídem.
```

Listing 1.2: Výstup

```
1 <p><s id="s1"><w id="w1">Toto</w><w id="w2">je</w><w id="w3">
      vstupný</w><w id="w4">text</w>.</s>
2 ["Obsahuje", "čísla", "napr.", ":", "č.", "1", ",", "5.", "(", "
      piaty", ")", ",", "alebo", "mená", "-", "M.", "Heinz", "a", "aj
      ", "priamu", "reč", "..."]</p>
3 [["Eva", "sa","<b>", "pýtala", ":", "</b>"], ["<b>", "-", "Čo", "
      ti", "povedali", "?!", "</b>"]]
4 [["Ahoj", ",", "\""],
5   ["povedal", ","],
6   ["\"","ako", "sa", "...", "\""],
7   ["skočil", "mu", "do", "reči", ","],
8   ["\"", "<i>", "ticho", "!", "</i>", "\""]]
9 [["-", "Prídeš", "?"],
10   ["-", "spýtal", "sa", "."]]
11 ["-", "Prídem", "."]
```

Poznámka: Všetko okrem prvej vety je uvedené vo forme polí kvôli čitateľnosti. Skutočný výstup by mal rovnakú štruktúru ako prvá uvedená veta.

1.2 Prehľad problematiky

1.2.1 Syntaktický parser

V tejto časti sa budem snažiť popísať, čo vlastne parsovanie je a na akom princípe pracuje, zároveň vysvetlím aj dôležité pojmy, ktoré budem v práci používať. Rozoberiem niektoré spôsoby parsovania, ich výhody a nevýhody. Tiež podrobnejšie vysvetlím vybraný algoritmus použitý na vytvorenie aplikácie.

1.2.2 Čo je to parsovanie?

Parsovanie ako pojem popisuje proces tvorby syntaktickej analýzy vety na základe vopred danej formálnej gramatiky(Context-free Grammar - CFG) a slovníku. Výsledná syntaktická analýza môže byť použitá pre ďalšiu sémantickú interpretáciu. V tejto práci sa však budeme zaoberať len syntaktickou analýzou pomocou CFG, keďže našim cieľom nie hlbšia analýza slov. [1][2]

1.2.3 Context-free grammar

Context-free grammar[2][3], skrátene CFG je typ formálnej gramatiky. Je to množina substitučných pravidiel, ktoré popisujú všetky existujúce reťazce v danom jazyku. Príklady substitučných pravidiel:

$$A \to \alpha$$

$$A \to \beta$$

Toto označuje, že môžeme vymeniť A za α alebo β

V CFG sú všetky pravidlá jedna k jednej, jedna k veľa alebo jedna k ničomu(epsilon rule). Ľavá strana pravidla je vždy neterminálny symbol. Našimi terminálnymi symbolmi budú napríklad slová, čísla alebo interpunkčné znamienka. Príklad CFG:

$$G = (N, \Sigma, R, S)$$

kde:

- N je množina neterminálnych symbolov
- Σ je množina terminálnych symbolov

- R je množina pravidiel v tvare $X \to Y_1 Y_2 \ldots Y_n$ pre $n \geq 0 \in N, Y_i \in (N \cup \Sigma)$
- S je začiatočný symbol, ktorý reprezentuje celý vstupný reťazec. S musí patriť do N

Konkrétnejší príklad - zadefinujme si nasledujúce substitúcie:

$$S \to AB$$

$$A \to a$$

$$A \to aA$$

$$B \to b$$

$$B \to bB$$

V tomto prípade A a B sú premenné a S je začiatočná premenná. a a b sú terminály. Využijeme vyššie uvedené pravidlá na nájdenie reťazcov skladajúcich sa z terminálov nasledujúcim spôsobom:

- Začneme použitím začiatočnej premennej S(inicializácia reťazca).
- Zoberieme ktorúkoľvek premennú v aktuálnom reťazci a vyberieme hociktoré pravidlo, ktoré má túto premennú na ľavej strane, následne nahradíme túto premennú pravou stranou tohoto pravidla.
- Opakujeme predchádzajúci krok, až kým sa v reťazci nenachádzajú len terminály.

Napríklad reťazec *aaaabb* môže vzniknúť z vyššie uvedených pravidiel takto:

$$S \to AB$$

$$AB \to aAB$$

$$aAB \to aAbB$$

$$aAbB \to aaAbB$$

$$aaAbB \to aaaAbB$$

$$aaaAbB \to aaaabB$$

$$aaaabB \to aaaabb$$

1.2.4 Parse Tree

Parse tree(parsovací strom) alebo strom odvodenia je strom reprezentujúci syntaktickú štruktúru reťazca podľa nejakej formálnej gramatiky. V parsovacom strome koreň re-

prezentuje začiatočný symbol, všetky vnútorné vrcholy sú označené neterminálnymi symbolmi gramatiky, potomkovia týchto vrcholov sú časťami pravej strany použitého substitučného pravidla usporiadané zľava doprava. Listami stromu sú samotné terminály. Príklad stromu na obrázku (1.2)

1.2.5 Typy parserov

Top-down parser

Názov tohto parseru vychádza zo spôsobu akým vytvára parsovací strom, teda zhora nadol. Princíp top-down parseru je možné popísať takto: Začíname so štartovacím symbolom S, pozrieme sa aký symbol sa nachádza na pravej strane úvodného pravidla (1.1), ďalej si zoberieme prvé pravidlo začínajúce týmto symbolom(Sum (1.2)) a pozrieme sa či sa prvý symbol jeho substitúcie zhoduje s tokenom v našom vstupe, ak áno, tak pokračujeme ďalším symbolom substitúcie ak nie, tak musíme vyskúšať ďalšie pravidlá(1.3)

$$S \rightarrow Sum \tag{1.1}$$
$$Sum \rightarrow Sum[+-]Product \tag{1.2}$$
$$Sum \rightarrow Product \tag{1.3}$$
$$Product \rightarrow Product[*/]Factor \tag{1.4}$$
$$Product \rightarrow Factor \tag{1.5}$$
$$Factor \rightarrow "("Sum")" \tag{1.6}$$
$$Factor \rightarrow Number \tag{1.7}$$
$$Number \rightarrow [0-9]+ \tag{1.8}$$

Príklad pre vstup: $5 + (3 * 2)$ a príklad stromu, ktorý vznikne pre tento vstup:

$S \rightarrow \bullet Sum$
$Sum \rightarrow \bullet Sum[+-]Product$
$Sum \rightarrow \bullet Product$
$Product \rightarrow \bullet Factor$
$Factor \rightarrow \bullet Number$
$Number \rightarrow "5"$
$Factor \rightarrow Number\bullet$
$Product \rightarrow Factor\bullet$
...

Fig. 1.1: Príklad pre vstup $5 + (3 * 2)$

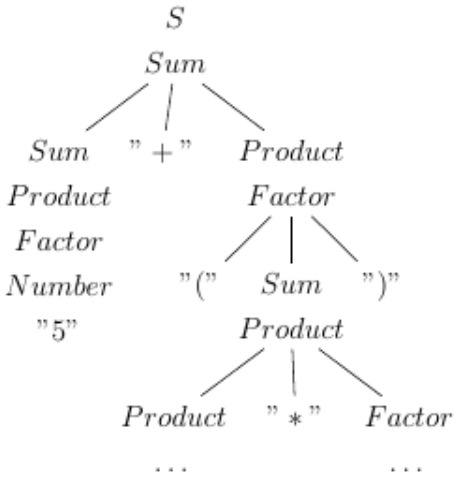

Fig. 1.2: Príklad stromu

V tomto príklade sme ale v každom kroku vybrali správne pravidlo, ktoré by nás doviedlo k výsledku, v skutočnosti by sme pri naivnom algoritme museli vyskúšať všetky možnosti.

Poznámka: Vo všetkých príkladoch budem pre ľahkú pochopiteľnosť využívať vyššie uvedenú jednoduchú aritmetiku pre CFG.

Bottom-up parser

Bottom-up parser zjavne, už podľa názvu skladá parsovací strom odspodu nahor, presnejšie od ľavého dolného rohu. Základný princíp je takýto: zoberieme si token zo vstupu a pokúsime sa nájsť zhodu na pravej strane pravidiel, ak sa nám podarí nahradiť celú pravú stranu, vymeníme túto sekvenciu symbolov za ľavú stranu použitého pravidla, toto opakujeme až kým nevytvoríme celý strom. V konečnom dôsledku získame rovnaký parsovací strom ako pri top-down parsery.

1.2.6 Chart parser

Chart parser je typ parseru vhodný pre nejednoznačné gramatiky(natural language processing). Kvôli efektívnosti využíva dynamické programovanie pre zapamätanie čiastočných hypotetických parsov. Tieto čiastočné riešenia si pamätá v tabuľke - z toho názov Chart parser. Bližší popis konkrétneho Chart parseru v ďalšej sekcii(1.2.7).

1.2.7 Earley parser

Earley parser[4] patrí medzi top-down chart parsery. Je schopný parsovať akýkoľvek Context-free grammar, teda je veľmi vhodný pre parsovanie prirodzeného jazyka. V porovnaní s obyčajnými top-down alebo bottom-up parsermi je aj vcelku efektívny - jeho výpočtová zložitosť je $O(n^3)$, kde n je dĺžka parsovaného reťazca, s tým, že pre jednoznačné gramatiky to je $O(n^2)$.

Earley items

Tak ako už bolo povedané, chart parsery si pamätajú zoznam čiastočných parsov. To isté robí aj Earley parser, prvky týchto zoznamov sa nazývajú Earley items.

$$Sum \rightarrow Sum \bullet [+-]Product \qquad (0)$$

- Číslo v zátvorkách označuje, kde tento Earley item začal - v tomto prípade je to (0), teda úplný začiatok vstupného reťazca.
- tučná bodka označuje koľko už bolo zo vstupu vyparsované, teda všetko naľavo od bodky už je hotové.

State sets

State sety sú množiny Earley item-ov. Sú uložené v dynamickom poli, ktoré môžeme nazvať jednoducho S. Toto je možná reprezentácia:

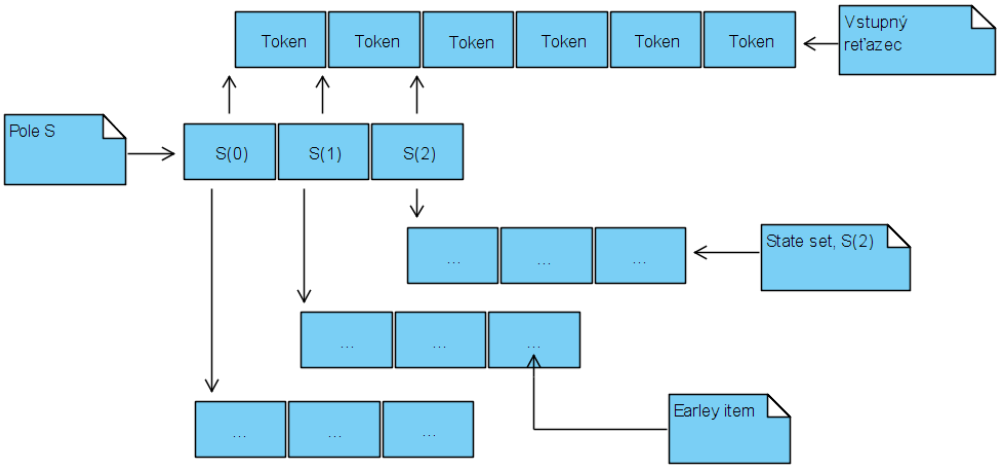

Fig. 1.3: Príklad reprezentácie

Pridávanie Earley item-ov

Existujú 3 spôsoby ako môžeme pridať nové Earley item-y do state setov, ktorý spôsob použijeme záleží na tom čo očakávame v našich pravidlách(čo sa nachádza napravo od tučnej bodky).

Prediction

Prvou z troch možností je predikcia - symbol napravo od bodky je neterminál. Pridáme pravidlo prislúchajúce k tomuto neterminálu do aktuálneho state setu.

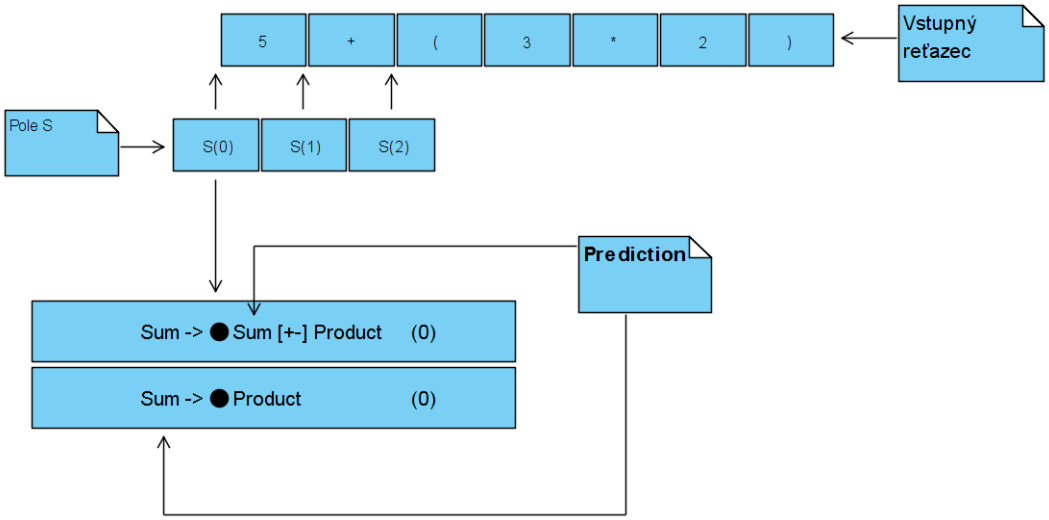

Fig. 1.4: Príklad predikcie

Scan

Druhá možnosť - symbol napravo od bodky je terminál. Skontrolujeme či sa token na vstupe zhoduje s daným terminálom, ak áno pridáme tento Earleyho item, posunutý o jeden krok do ďalšieho state setu.

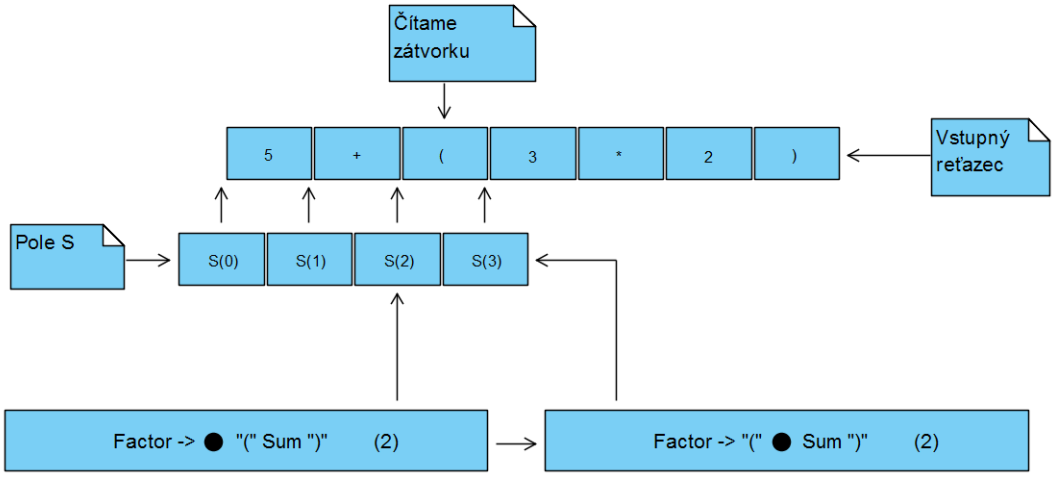

Fig. 1.5: Príklad scanu

Completion

Posledná možnosť - napravo od bodky už nie je nič. To znamená, že sme dokončili čiastočný pars. Pokúsime sa nájsť rodiča tohto item-u a pridáme ho do aktuálneho state setu, s tým že ho posunieme o krok dopredu.

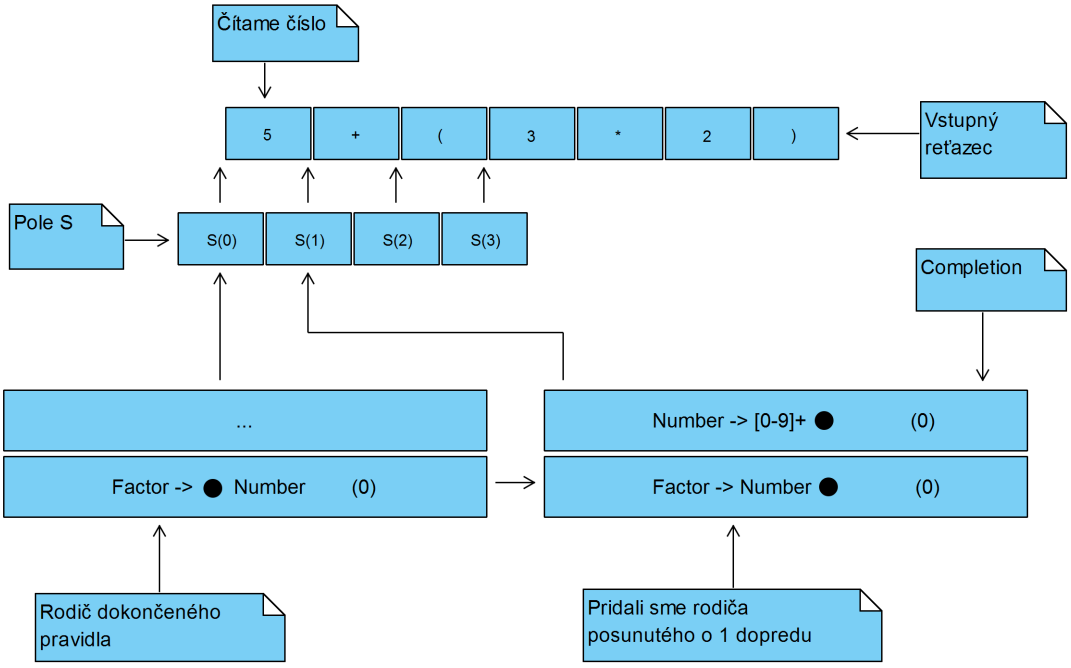

Fig. 1.6: Príklad completion

Failure modes

Existujú dve situácie kedy môže parser zlyhať:

- Nepodarí sa vyparsovať celý vstup. To sa môže stať ak je vo vstupe chyba(vstup nedáva zmysel), napríklad: "$5 + ()$"
- Podarí sa vyparsovať celý vstup, ale nevznikne kompletný pars, napríklad: "$5 *$" - toto je korektný začiatok vstupu($Product$ z našej ukážkovej gramatiky), ale na dokončenie nám chýba posledný token.

1.2.8 Porovnanie parserov

Za zmienku stoja aj ďalšie parsovacie algoritmy, ktoré však majú určité nevýhody, čím robia z Earley parseru najlepšiu voľbu pre túto aplikáciu:

- CYK algorithm: Má síce tiež výpočtovú zložitosť ako Earley, ale pamäťová zložitosť je $O(n^2)$, pričom u Earley parseru to je len $O(n)$ [5]. CYK zároveň potrebuje gramatiku v CNF, kde Earley funguje pre akúkoľvek gramatiku a tak ako už bolo spomenuté skôr, tak pre niektoré gramatiky parsuje aj s lepšou zložitosťou ako $O(n^3)$.
- LR parser: Je síce schopný parsovať vstupný reťazec v lineárnom čase, ale nie je dostatočne flexibilný pre parsovanie prirodzeného jazyka(je vhodnejší pre programovacie jazyky)
- LL parser: Je veľmi jednoduchý, nie je však schopný parsovať akýkoľvek context-free grammar.
- Valiant's algorithm: (Teoreticky)rýchlejší algoritmus než Earley algorithm použiteľný pre natural language processing. Algoritmus má výpočtovú zložitosť $O(n^{2.81})$, čo je lepšie ako Earley algorithm, ale jeho koeficienty sú také veľké, že v praxi by bol rýchlejší len pre obrovské vstupy.[6]

1.3 Prehľad existujúcich systémov

V nasledujúcej podkapitole sa budem venovať existujúcim riešeniam podobného problému. Medzi takéto riešenia môžeme zaradiť rôzne syntaktické parsery alebo knižnice určené pre natural language processing.

1.3.1 Nearly.js

Nearly.js[7] je parsovací tool kit využívajúci Earley parser napísaný v JavaScripte. Parser využíva vlastnú gramatiku a umožňuje používanie makier. Je tiež schopný parsovať nejednoznačnú gramatiku a vracia v tomto prípade všetky parsy. Nearly tiež umožňuje debugovanie a má aj určitú detekciu chýb. Ďalšou funkcionalitou je aj možnosť konvertovať gramatiku na čitateľnejšie SVG "koľajnicové"diagramy.

1.3.2 Stanford CoreNLP

Stanford CoreNLP[8] poskytuje viaceré nástroje pre natural language processing. Je napísaný v Jave, s tým, že poskytuje interface aj ďalšie programovacie jazyky. Medzi funkcionality CoreNLP patrí aj part-of-speech tagger, named entity recognizer, parser, extrakcia informácií, sentiment analysis, pattern learning a ďalšie. Toolkit tiež podporuje okrem angličtiny aj ďalšie jazyky, napríklad: čínština, francúzština, nemčina, španielčina alebo arabčina.

1.3.3 earley-parser-js

Tento parser je najjednoduchším z tu menovaných implementácií/systémov. Takisto ako Nearley.js využíva Earley algorithm. Využíva jednoduchší zápis gramatiky a jeho výstupom je parsovací strom. Táto implementácia tiež obsahuje online grafické demo (1.7), ktoré zobrazuje celý parsovací strom a je použiteľné na jednoduché testovanie gramatiky. Earley-parser-js[9] budem využívať vo výslednej aplikácii, keďže jeho jednoduchosť, minimalistickosť a ľahká modifikovateľnosť v porovnaní s inými systémami sú výhodné pre tvorbu tejto aplikácie.

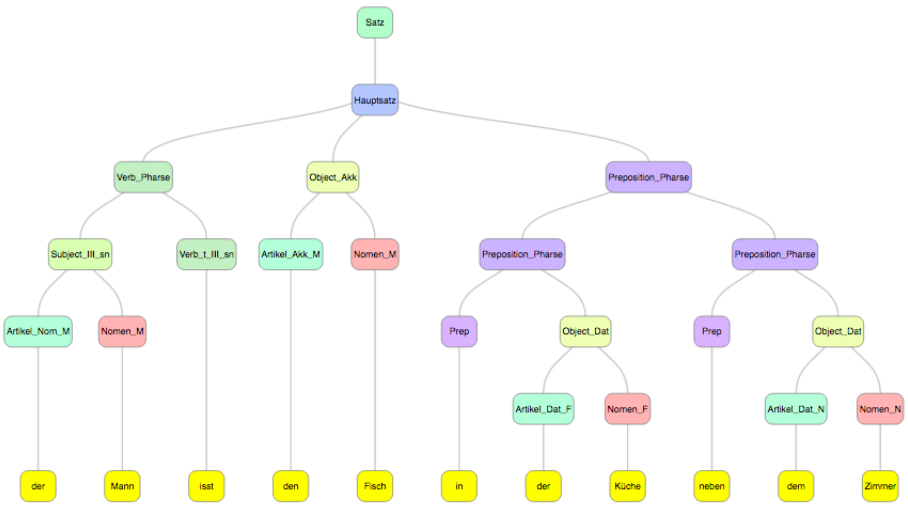

Fig. 1.7: Online Demo s použitím earley-parser-js

1.4 Prehľad technológií a nástrojov

1.4.1 Javascript

Ako programovací jazyk pre túto aplikáciu budem používať Javascript. Je to dynamický, netypovaný, interpretovaný, vyšší programovací jazyk. Je veľmi vhodný pre tvorbu webových aplikácií, keďže je podporovaný všetkými modernými webovými prehliadačmi. Jazyk tiež zahŕňa API pre prácu s textom, poľami a regulárnymi výrazmi.

Pri písaní tejto aplikácie budem využívať aktuálnu špecifikáciu ECMAScript 6(2015), ktorú využíva Javascript. Pri použití tejto špecifikácie zároveň nehrozia problémy s kompatibilitou s webovými prehliadačmi, keďže, aplikácia bude fungovať na strane servera.[10]

Node.JS

Node.js je Javascriptový runtime enviroment postavený na V8 Javascript engine od Google, vhodný pre tvorbu server-side aplikácií. Node.js využíva udalosťami riadený, asynchrónny model, ktorý ho robí veľmi rýchlym.

1.4.2 JSON formát

JSON je dátová štruktúra, respektíve formát, využívajúci pre človeka ľahko čitateľný text(štruktúru) skladajúci sa z dvojíc kľúč-hodnota. Jeho čitateľnosť ho robí veľmi výhodným napríklad pri uchovávaní gramatiky alebo slovníku pre našu aplikáciu, ktorá by mala byť dostatočne zrozumiteľná na to aby sa dala ľahko upravovať.

1.4.3 XML

XML je značkovací jazyk, ktorý je primárne určený na výmenu údajov medzi aplikáciami a na zverejňovanie dokumentov. Jazyk umožňuje opísať štruktúru dokumentu z hľadiska vecného obsahu jednotlivých častí. Sila XML je najmä v jeho hierarchickej štruktúre a pomerne jednoduchom spôsobe zápisu. Jazyk XML zároveň neobsahuje žiadne preddefinované tagy, teda užívateľ si ich definuje podľa vlastnej potreby, teda v našom prípade napríklad "*sentence*" a "*word*" tagy. XML dokument bude výstupom výslednej aplikácie.

Kapitola 2

Návrh a implementácia

Táto časť práce bude venovaná návrhu a implementácii samotnej aplikácie. V podkapitolách sa najprv pristavím pri návrhu danej časti aplikácie a následne pomocou ukážok zdrojového kódu a príkladov parsovacích stromov popíšem aj implementáciu.

2.1 Perzistentné dáta

Aplikácia nebude obsahovať vlastnú databázu, keďže bude modulom Multimediálnej čítanky, ktorá databázu má. Dáta(výstupy) aplikácie sa teda budú ukladať priamo do databázy Multimediálnej čítanky.

2.2 Vstupné a výstupné dáta

Vstupom pre aplikáciu bude reťazec v slovenskom alebo anglickom jazyku obsahujúci HTML tagy(napr.: `<b>, <i>, <span>, <p>`). Očakávaná dĺžka tohoto reťazca je maximálne okolo 1000 slov.

Výstupom bude XML reťazec, v ktorom budú tagmi označené jednotlivé slová a vety, s tým, že sa zachovajú všetky pôvodné HTML tagy zo vstupu bez toho, aby sa prekrížili s pridanými XML tagmi. Multimediálna čítanka využíva WYSIWYG editor, takže nehrozí, že by sa do aplikácie dostali nekorektné tagy(napr. neukončené alebo krížiace sa tagy). Interpunkcia sa bude nachádzať mimo slovných tagov, pokiaľ teda nie je súčasťou slova. Príklad vstupu aj výstupu je možné vidieť v listingu (1.1) a (1.2).

- formát vetného XML tagu: `<s id="s\d+"></s>`
- formát slovného XML tagu: `<w id="w\d+"></w>`

Výstupom bude takisto aj parsovací strom použitý pre vytvorenie XML výstupu. Tento strom bude vo formáte JSON, kde každý vrchol bude mať atribúty `name` a `children`. Prvý z nich bude buď menom gramatického pravidla alebo v prípade listov

stromu to bude samotné slovo/interpunkcia a druhý bude poľom ďalších vrcholov s rovnakými atribútmi.

2.3 Data-flow diagram

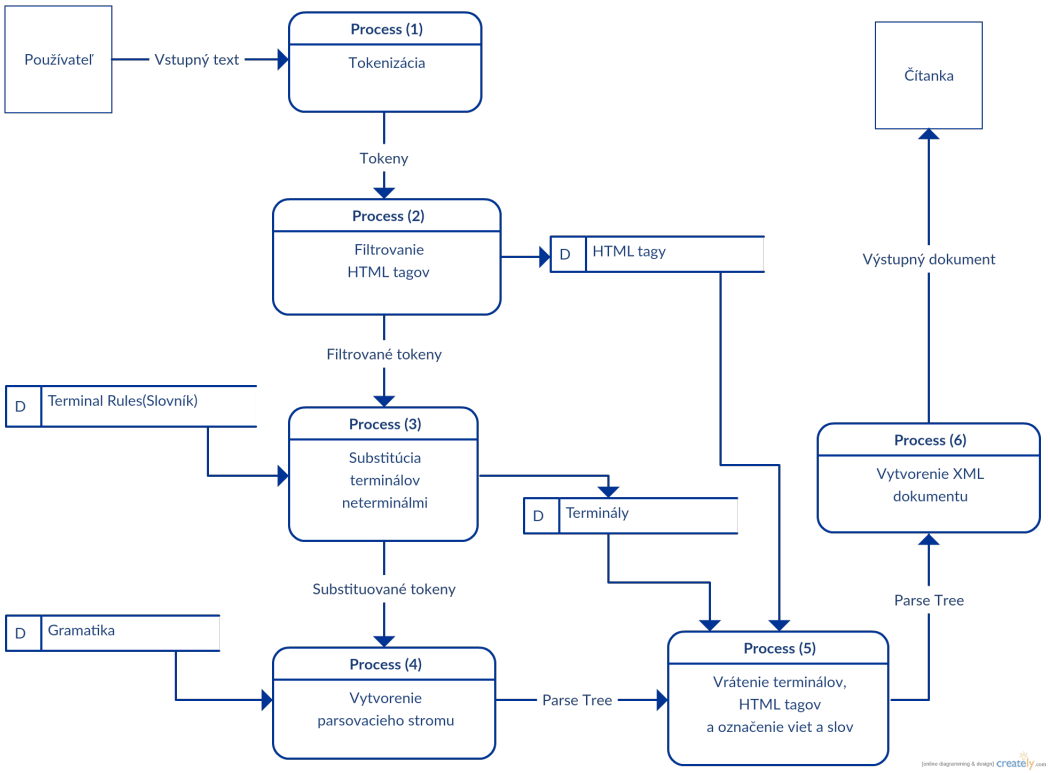

Fig. 2.1: Data-Flow diagram

Diagram(2.1) popisuje tok dát a procesy v aplikácii. Na začiatku ($Process(1)$) aplikácia dostáva vstupný text od používateľa, ktorý sa pomocou regulárnych výrazov rozdelí na jednotlivé tokeny(slová, interpunkčné znamienka a pod.).

Nasleduje proces filtrovania HTML tagov($Process(2)$), kde sa z tokenov odstránia všetky HTML tagy a v tabuľke sa zapamätajú ich pozície, aby ich bolo možné neskôr vrátiť späť.

Tieto filtrované tokeny je ďalej nutné nahradiť generickými tokenmi($Process(3)$), (napr.: "dom"$\rightarrow$ "$Word$"), s ktorými sa bude ľahšie pracovať pri tvorbe gramatiky pre parser. Pôvodné tokeny sa samozrejme uložia, aby bolo možné neskôr zrekonštruovať pôvodný text.

V tejto chvíli($Process(4)$) prichádza na rad parser, ktorý pomocou gramatiky vytvorí parsovací strom zo substituovaných tokenov.

Tento strom sa následne($Process(5)$) upravý tak, že sa do listov vrátia pôvodné slová, pomocou štruktúry stromu sa označia jednotlivé slová a vety XML tagmi a pridajú sa HTML tagy na správne miesta.

Nakoniec($Process(6)$) sa z tohto finálneho stromu vytvorí XML dokument, ktorý dostáva Multimediálna čítanka na ďalšie spracovanie. V zbytku kapitoly presnejšie popíšem jednotlivé procesy zobrazené na diagrame.

Poznámka: Výsledná aplikácia implementuje možnosť používať slovenskú aj anglickú gramatiku(a aj slovník), tu sa však budem venovať hlavne tej slovenskej a len na miestach kde to bude potrebné spomeniem zmeny/rozdiely v anglickej.

2.4 Tokenizácia

Text je nutné rozdeliť na tokeny, s ktorými sa bude dať ďalej pracovať. Tieto tokeny budú vytvorené pomocou regulárneho výrazu, ktorý rozdelí vstupný text na pole podreťazcov. Typy tokenov, ktoré vzniknú:

- nové riadky - `\n`
- Opakujúca sa interpunkcia(napr.: ...(tri bodky), !?, !!!) - `([\.!\?]{2,})`
- triedy - `([0-9]\.[A-Z])`
- iniciály - `([A-ZÉŘŔŤÝÚÍÓÁŠĎĹĽŽČ]\.)`
- dátumy a časy vo formáte DD-MM-YYYY, HH:MM:SS,XXX
  `(\d{1,2}[:-]\d{1,2}[:-]\d{1,}(,\d+)*)`
- čísla - `([+-]?\d+(\.\d+)*(,\d+)?(%|,-)?)`
- bežné slová - keďže písmená so slovenskou diakritikou nepatria do rozsahov a-z prípadne A-Z, je nutné ich takýmto spôsobom vypísať -
  `((?!-)[a-zéŕřťýúióášdľĺžčäôA-ZÉŘŔŤÝÚÍÓÁŠĎĹĽŽČ-]+)`
- interpunkčné znamienka a ďalšie znaky - `[\.,;:\"„\\(\)\?!'/\*\+%$€]`
- HTML tagy - `(<[^\d\s>]+[^>\n]*?>)`
- znamineka menšie/väčšie - `[<>]` - tieto znamienka musia byť samostatne až za HTML tagmi, aby sa nestalo, že regex nájde samostatne len znamienka namiesto celých HTML tagov
- všetko ostatné(non-whitespace) - `[\S+]`

Výsledný regex je disjunkciou jednotlivých častí vyššie.

V tejto chvíli ešte nie je možné rozlíšiť napríklad radové číslovky("10.") , toto bude riešiť gramatika, ktorá bude "spájať" jednoduchšie tokeny dohromady($Ordinal \rightarrow Num$ ".").

2.5 Filtrovanie HTML tagov

V tokenoch vytvorených v predchádzajúcom kroku sa nachádzajú HTML tagy, ktoré chceme vo výslednom dokumente, ale pri parsovaní by nám iba prekážali, preto je potrebné ich dočasne vymazať, s tým, že si ich uložíme do pomocnej tabuľky v tvare:

Listing 2.1: Tabuľka HTML tagov

```
1 HTMLTagsPositions = {
2     0:"<p>"
3     4:"<i>"
4     6:"</i>"
5     9:"<b>"
6     11:"</b>"
7     14:"</p>"
8     };
```

Kde kľúčmi sú indexy, na ktorých sa pôvodne nachádzali HTML tagy a hodnotami sú samotné tagy.

Zároveň sa z pôvodného poľa tokenov odstránia tieto tagy:

Listing 2.2: Text pred odstránením tagov

```
1 ["<p>", "Toto", "je", "jednoduchá", "<i>", "veta", "</i>", "s",
    "<b>", "html", "</b>", "tagmi", ".", "</p>"]
```

Listing 2.3: Text po odstránení tagov

```
1 ["Toto", "je", "jednoduchá", "veta", "s", "html", "tagmi", "."]
```

2.6 Substitúcia terminálov neterminálmi

Keby sme chceli parseru poslať tokeny v takom stave v akom sú teraz, tak by sme na to potrebovali kompletný slovník, pomocou ktorého by parser vedel rozlíšiť, čo s ktorým slovom spraviť a zároveň by to tým pádom aj nafúklo gramatiku, ktorá by musela riešiť veľa zbytočných alternatív. Keďže hlbšia analýza slov a viet nie je cieľom tejto práce, tak môžeme napríklad všetky slová s malými písmenami brať ako jeden typ tokenu("$WordLower$").

Na tento účel použijeme tabuľku(slovník) substitučných pravidiel, ktorou prevedieme terminály na jednotlivé typy tokenov. Táto tabuľka bude uložená v samostatnom súbore v JSON formáte. Kľúčmi atribútov v tomto JSON-e budú jednotlivé regulárne výrazy, s ktorými budeme hľadať zhodu s tokenom. Hodnotou atribútu potom budú názvy vyššie spomínaných typov tokenov.

Listing 2.4: Formát slovníku a príklad substitúcie

```
{
"^RegexKtoryMatchujeToken$": "TypTokenu",
...
"^[0-9]+$":"Num", //Čísla
"^(\d{1,2}[:-]\d{1,2}[:-]\d{1,}(,\d+)*)|([+-]?\d+(\.\d+)*(,\d+)
   ?(%|,-)?)$":"Decimal", //Desatinné čísla, dátumy a časy
"^[\.]{3,}$":"Ellipsis", //Tri(alebo viac) bodky
...
"^\S+$": "Other" // ostatné tokeny(non-whitespace)
};
```

Ako môžeme vidieť na ukážke(2.4), každý regulárny výraz začína ^(caret) a končí $(dolár), teda chceme nájsť plnú zhodu s tokenom. Pri hľadaní zhody prechádzame jednotlivé regexy až kým nenájdeme zhodu, ktorá sa vždy nájde vďaka poslednému regexu - `^\S+$`, ktorý zachytí všetko čo nenašlo zhodu skôr. Ako je už asi zjavné, tak na poradí záleží, teda nemôžeme napríklad hľadať najprv samotnú bodku - `^\.$` a až potom tri alebo viac bodiek - `^[\.]{3,}$`, lebo by sme nikdy nedostali zhodu na týchto troch bodkách.

Niektoré substitúcie v slovníku ako napríklad už spomínané tri bodky by sa mohli zdať redundantné, keďže v gramatike by sme si vedeli vytvoriť pravidlo, ktoré by rekurzívne poskladalo akýkoľvek počet opakujúcich sa bodiek. To by ale zbytočne nafukovalo gramatiku a spomaľovalo samotné parsovanie textu, preto ak je to možné tak je v takejto situácii lepšie zadefinovať takúto substitúciu tu v slovníku, než v gramatike.

Po tejto úprave pomocou slovníku môže veta vyzerať napríklad takto:

Listing 2.5: Tokeny po použití slovníku

```
tokens =
["WordStart", "WordLower", "Word", "-", "WordLower", "Decimal",
   "WordLower", "Word", "?"]
```

2.7 Vytvorenie parsovacieho stromu

Na vytvorenie parsovacie stromu bude použitý earley-parser-js, ten však potrebuje pre svoju prácu tokeny(popísané v predchádzajúcej podkapitole) a gramatiku. Gramatika bude vo forme poľa pravidiel, kde ”− >” označuje rozdelenie na ľavú a pravú stranu pravidla a ” | ” označuje alternatívu(v tomto príklade je použitá gramatika na parsovanie jednoduchej aritmetiky):

Listing 2.6: Ukážková gramatika

```
var grammar = new tinynlp.Grammar([
```

```
    'Root -> N',
    'S -> S add_sub M | M',
    'M -> M mul_div T | T',
    'N -> S lt_gt S | S',
    'T -> num | ( S )'
]);
```

A pre terminály:

Listing 2.7: Vytvorenie pravidiel pre terminály

```
grammar.terminalSymbols = function(token) {
    if ('<' === token || '>' === token) return ['lt_gt'];
    if ('+' === token || '-' === token) return ['add_sub'];
    if ('*' === token || '/' === token) return ['mul_div'];
    if ('(' === token) return ['('];
    if (')' === token) return [')'];
    // Otherwise - token considered as a number:
    return ['num'];
};
```

Pri termináloch nám prácu uľahčí to, že už máme vytvorený slovník (Listing 2.4), teda nebude nutné nijak nahrádzať tokeny, teda v našom prípade:

```
grammar.terminalSymbols = function(token) {
    return [token];
};
```

Na to, aby sme zabezpečili možnosť ľahko upravovať gramatiku, tak ju budeme mať uloženú v textovom súbore vo formáte JSON, tak ako aj pri slovníku, kde atribútom bude ľavá strana pravidla a hodnotou bude pravá strana. Vyššie uvedená gramatika(Listing 2.6) by teda vyzerala takto:

Listing 2.8: Ukážková gramatika V JSON-e

```
{
  "Root":"N",
  "S":"S add_sub M | M",
  "M":"M mul_div T | T",
  "N":"S lt_gt S | S",
  "T":"num | ( S )"
}
```

A tým pádom načítavanie gramatiky bude zjednodušené:

```
var grammar = new tinynlp.Grammar(this.loadGrammar(rulePath));
```

2.7.1 Analýza gramatiky

V tejto sekcii podrobne rozoberiem, najdôležitejšie a najkomplikovanejšie pravidlá, ktoré obsahuje gramatika pre parser a následne popíšem aj menšie optimalizácie potrebné pre efektívnosť a jednoznačnosť parsovania.

Koreňom celej gramatiky je "*Story*", z ktorého sa v prípade slovenskej gramatiky vetvia jednotlivé riadky, teda:

$$"Story \rightarrow Lines"$$

V anglickej gramatike sú to jednotlivé vety:

$$"Story \rightarrow Sentences"$$

Tento rozdiel je spôsobený tým, že v Slovenčine, na rozdiel od Angličtiny sa používa priama reč vymedzená pomlčkami a tam sa zvyknú dávať jednotlivé súvetia, respektíve prehovor jednej osoby, do samostatných riadkov, teda vedieť kde končí riadok je veľmi užitočné, napríklad:

- Prídeš? - spýtal sa Jožko.
- Prídem...
- Určite? - spýtal sa znova. Lebo si nebol istý...
- Určite!

My nevieme vopred koľko viet/riadkov bude v texte, tým pádom pravidlo pre vety/riadky musí byť rekurzívne:

$$"Sentences \rightarrow Sentences\ Sentence \mid Sentence"$$

$$"Lines \rightarrow Lines\ Line \mid Line"$$

Tieto pravidlá sa môžu zdať neprirodzene zapísané(najprv rekurzívny prípad, až potom triviálny). Dôvodom je, že tento parser má lepší výkon pri "$left-recursive$" gramatike(testy gramatiky v kapitole 3) a tiež je to potrebné pri pridávaní blokových tagov, ktoré by pri "$right-recursive$" gramatike mohli skončiť na nesprávnom mieste, kvôli tomu, že to zmení ich najbližšieho spoločného rodiča, napríklad:

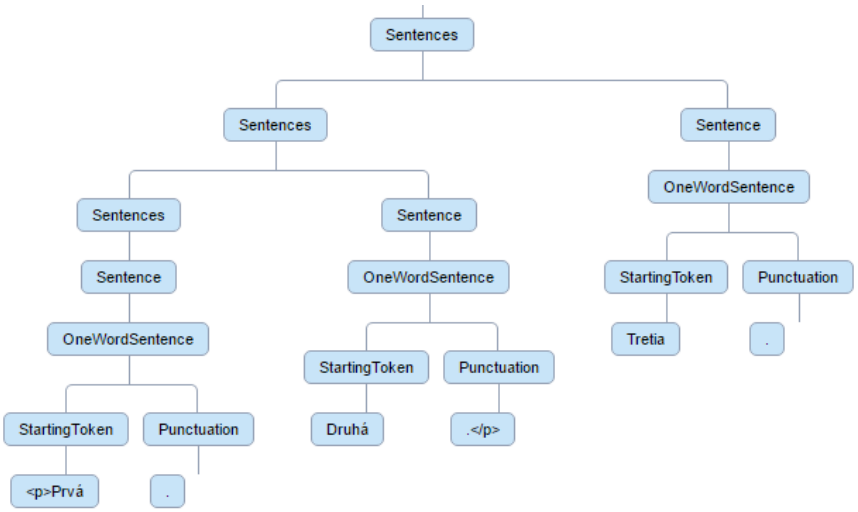

Fig. 2.2: Správny výstup pre vstup: `<p>Prvá. Druhá.</p> Tretia.`

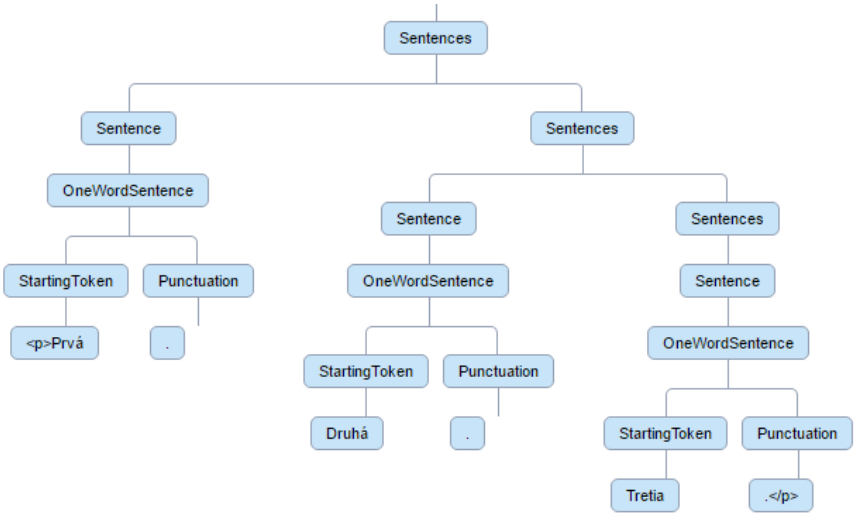

Fig. 2.3: Nesprávny výstup pri "$right - recursive$" gramatike: `<p>Prvá. Druhá. Tretia.</p>`

Poznámka: Všetky ukážky stromov sú zjednodušené kvôli čitateľnosti a kompaktnosti.

Keďže v ostatných pravidlách sa poväčšine anglická a slovenská gramatika líšia v štruktúre a nie veľmi v zmysle, tak budem popisovať iba tú slovenskú, keďže je pre nás relevantnejšia.

Ďalším pravidlom je "$Line$", ktorý rozlišuje medzi dvomi typmi viet a logicky končí novými riadkami(aspoň jedným):

$$"Line \rightarrow \ DirectSpeechDashStart\ NewLines \mid OtherSentences\ NewLines"$$

"$DirectSpeechDashStart$" je skupinou pravidiel, ktoré definujú priamu reč vymedzenú pomlčkami a "$OtherSentences$" združujú priamu reč v úvodzovkách a ostatné

jednoduché vety. Najprv sa pozrieme na jednoduché vety a následne na priame reči:

Najjednoduchšou formou vety je "*Sentence*" z nej môže vzniknúť "*OneWordSentence*" alebo "*BasicSentence*"

- "*OneWordSentence*" sa musí skladať z "*StartingToken*" a "*Punctuation*", kde prvý z nich je slovom s veľkým začiatočným písmenom, číslom alebo iným tokenom, ktorým môže začínať veta a druhým je interpunkcia, ktorá môže ukončovať vetu - napríklad: bodka, otáznik, tri bodky, ale nie čiarka.
- "*BasicSentence*" je vo forme: "*StartingToken Words Punctuation* | *Quotation*" rozdielom oproti "*OneWordSentence*" sú ďalšie slová, ktoré sú definované podobne ako "*Sentences*" alebo "*Lines*"(rekurzívne) a alternatíva v podobe vety s citátom, ktorá obsahuje navyše aj ľavé a pravé úvodzovky, ktoré však neoznačujú priamu reč, teda nie sú samostatnou vetou.

Ďalej gramatika popisuje už spomínané typy priamej reči. V Slovenčine poznáme priamu reč s úvodzovkami alebo pomlčkami. Tá s úvodzovkami má štyri formy:

1. „Priama reč`[.?!]+`“
2. Uvádzacia veta: „Priama reč`[.?!]+`“
3. „Priama reč`[,?!]+`“ uvádzacia veta`[.?!]+`
4. „Priama reč`[,?!]+`“ uvádzacia veta, „priama reč`[.?!]+`“

Poznámka: Priama reč a Uvádzacia veta tu neoznačujú jednu vetu, ale ľubovoľný počet viet, posledná z nich však musí končiť jedným zo znamienok popísaných regulárnym výrazom.

1. Prvá možnosť je definovaná pomocou troch pravidiel:"*DirectSpeechSingle*", "*DirectSpeechTwo*" a "*DirectSpeechMoreThanTwo*" Toto rozdelenie je potrebné, aby sme vedeli dať začiatočné a koncové úvodzovky do správnej vety. V prvom prípade, teda aj začiatočné aj koncové úvodzovky patria do jednej vety, v druhom prípade začiatočné do prvej a koncové do druhej a v treťom prípade začiatočné do prvej, koncové do poslednej a medzi nimi sa nachádza ľubovoľný počet "*Sentences*".

   Samotné pravidlá:

   - "$DirectSpeechSingle \rightarrow$
     $QuoteLeft\ StartingToken\ Punctuation\ QuoteRight$ |
     $QuoteLeft\ StartingToken\ Words\ Punctuation\ QuoteRight$"
   - "$DirectSpeechTwo \rightarrow DirectSpeechFirst\ DirectSpeechSecond$"

- "$DirectSpeechMoreThanTwo \rightarrow$
  $DirectSpeechFirst\ Sentences\ DirectSpeechSecond$"

"$DirectSpeechFirst$" a "$DirectSpeechSecond$" vyzerá takmer rovnako ako "$DirectSpeechSingle$", jediným rozdielom je absencia začiatočnej respektíve koncovej úvodzovky.

2. Pre vytvorenie pravidiel pre ostatné typy priamej reči potrebujeme zadefinovať ako budú vyzerať uvádzacie vety, kde v prípade uvádzacej vety na začiatku to bude: "$ISentenceStart \rightarrow StartingToken\ Words : \mid StartingToken :$"
   Za ktorou bude nasledovať jedna z priamych rečí z predchádzajúceho bodu.

   Poznámka: DS = Direct Speech, IS = Introductory Sentence

3. Tretia možnosť nás núti upraviť priamu reč aj uvádzaciu vetu z predchádzajúcich prípadov:

   - Priama reč teraz nemôže končiť bodkou, teda namiesto "$Punctuation$" použijeme "$DSPunctuation \rightarrow , \mid ? \mid ! \mid RepeatedPunctuation \mid Ellipsis$", inak táto priama reč ostáva rovnaká ako to bolo v prvom prípade.
   - Uvádzacia veta zase môže začínať aj malým písmenom, tým pádom bude vo forme: "$ISentenceEnd \rightarrow\ Words\ Punctuation$"

   Výsledné pravidlo:"$DSAndISentence \rightarrow\ DirectSpeechStart\ ISentenceEnd$"

4. Pri poslednom prípade si treba uvedomiť, že druhá a tretia časť vety sa môže ľubovoľne opakovať, napr.:

   `...uvádzacia veta,„priama reč[,?!]+“ uvádzacia veta,„priama reč[.?!]+“`

   teda druhá(opakujúca sa) časť vety potrebuje rekurzívnu definíciu. Celá veta:

   "$DSAndISentenceRepeated \rightarrow\ DirectSpeechStart\ ISAndDSRepeated$"

   Rekurzívna definícia:

   "$ISAndDSRepeated \rightarrow$
   $ISAndDSRepeated\ ISentenceMiddle\ DirectSpeechEnd \mid$
   $ISentenceMiddle\ DirectSpeechEnd$"

   kde "$ISentenceMiddle$" sa skladá z viacerých viet s tým, že prvá môže začínať malým písmenom a posledná z nich musí končiť čiarkou. Medzi nimi sa môžu nachádzať "$Sentences$"

S pomlčkami:

1. - Priama reč`[.?!]+`
2. - Priama reč`[,?!]+` - uvádzacia veta`[.?!]+`
3. Uvádzacia veta: - Priama reč`[.?!]+`“
4. - Priama reč`[,?!]+` - uvádzacia veta`[,?!]+` - priama reč`[.?!]+`

1. Tu v prvom prípade vieme vytvoriť priamu reč jednoduchším spôsobom, keďže nemusíme riešiť úvodzovky na začiatku aj na konci, ale len pomlčku na začiatku. Na tomto mieste tiež využijeme to, že náš text sa delí na riadky ukončené `\n` - teda, žiadna z viet, z ktorých sa skladá toto pravidlo nebude ukončená novým riadkom a ak sa v texte vyskytne medzi takýmito vetami nový riadok, tak sa o to postará pravidlo "$Line$".

   "$DirectSpeechDash$
   $\rightarrow$ $SentenceDashNoNewLines$ $SentencesNoNewLine$ | $SentenceDash$"

2. V druhom prípade vytvoríme pravidlo z dvoch častí:

   "$DSAndISentenceDash \rightarrow$ $DirectSpeechStartDash$ $ISentenceEndDash$"

   kde prvou časťou je priama reč("$DirectSpeechStartDash$"). Toto pravidlo je rozdelené podobne ako prvá možnosť pri priamej reči s úvodzovkami. Tu toto rozdelenie ale potrebujeme kvôli tomu, aby sme vedeli, v ktorej vete bude začiatočná pomlčka, ktorá bude v klasickej forme a ktorá nemôže končiť bodkou. Druhá časť("$ISentenceEndDash$") pravidla zase začína pomlčkou a môže mať prvé slovo s malým písmenom.

3. Ďalšia možnosť je len spojením uvádzacej vety($ISentenceStart$), ktorú sme použili pri priamej reči s úvodzovkami a priamej reči s pomlčkou z prvého bodu:

   "$ISentenceAndDSDash \rightarrow$ $ISentenceStart$ $DirectSpeechDash$"

4. Poslednou možnosťou ako poskladať priamu reč s pomlčkami je použiť priamu reč na začiatku, ktorú sme už spomínali vyššie("$DirectSpeechStartDash$") a za ňou, pridať ďalšie dve časti, kde prvá je uvádzacou vetou v podobnej forme ako tomu bolo v 2. bode, s tým rozdielom, že tu nemôže táto uvádzacia veta končiť bodkou. Poslednou časťou je priama reč, ktorá rovnako ako predchádzajúca časť môže začínať malým písmenom, nemôže však končiť čiarkou, keďže je to koniec vety:

   "$DSAndISentenceDashRepeated \rightarrow$
   $DirectSpeechStartDash$ $ISentenceMiddleDash$ $DirectSpeechEndDash$"

Tak ako už bolo spomenuté vyššie tak v Angličtine sa priama reč určená pomlčkami nepoužíva, takže v tomto ohľade je trochu jednoduchšia. V Angličtine sa ale používajú rôzne typy úvodzoviek na určenie priamej reči, tam teda musia byť využité tieto tri typy namiesto slovenských úvodzoviek:

$$InvertedComa \rightarrow \text{"'"},\ InvertedDoubleComa \rightarrow \text{"``"},\ Quote \rightarrow \text{"""}$$

Optimalizácia

Kvôli skutočnosti, že v Slovenčine môžeme skombinovať takmer hocijakú postupnosť slov a interpunkcie a aj tak vytvoriť validnú vetu, tak by pre nás bolo najjednoduchšie definovať slová vo vete ako hocijaké slová, interpunkciu a nové riadky. To ale spôsobí veľmi veľké množstvo nejednoznačností v gramatike a parser bude produkovať veľmi veľa zbytočných stromov. Tieto nejednoznačnosti primárne spôsobuje priama reč určená pomlčkami, v ktorej sa za sebou vyskytujú interpunkčné znamienka, ktoré môže obsahovať aj jednoduchá slovenská veta, v tej by sa však nemali objaviť hneď za sebou.

Listing 2.9: Nejednoznačný vstup

```
- Priama reč, - uvádzacia veta, - Priama reč...
```

Listing 2.10: Výstupné vety vo forme polí

```

[["-", "Priama", "reč", ","],
 ["-", "uvádzacia", "veta"],
 ["-", "Priama", "reč", "."]] // korektný výstup

 ["-", "Priama", "reč", ",", "-", "uvádzacia", "veta", "-", "
    Priama", "reč", "."] // jeden z nekorektných výstupov
```

Ako môžeme vidieť v listingu (2.10) parser nemusí nutne rozoznať čiarku a pomlčku ako koniec a začiatok priamej reči. Preto potrebujeme kombinácie niektorých interpunkčných znamienok vylúčiť z gramatiky. CFG však nepozná negácie, preto v tejto situácii zadefinujeme slová ako dve alternatívy. Prvou z nich je, že slovo bude samostatne a druhou, že slovo bude vo dvojici s našou nechcenou interpunkciou, to zamedzí tomu, aby sa priamo za sebou v bežných vetách vyskytla napríklad čiarka a pomlčka.

$$\text{"}MidSentencePunctuation \rightarrow , \mid ; \mid : \mid Dash \mid Ellipsis\text{"}$$

$$\text{"}WordBasic \rightarrow\ WordCapital \mid WordLower \mid Initial \mid Word \mid Number...\text{"}$$

$$\text{"}WordTokens \rightarrow\ WordBasic \mid WordBasic\ MidSentencePunctuation\text{"}$$

Táto podkapitola samozrejme neobsahuje všetky pravidlá, ktoré sa nachádzajú v gramatike, preto odporúčam pre ďalšie podrobnosti ohľadne gramatiky pozrieť si samotné zdrojové súbory gramatiky, ktoré sa nachádzajú v prílohách v zdrojových súboroch v priečinku: `parser/app/res/grammars`, prípadne využiť priloženú vizualizáciu.

2.7.2 Parsovací strom

Earley-parser-js vráti ako výstup parsovací strom, ktorý je v JSON formáte, kde atribútmi sú "*root*"(koreň) podstromu(ľavá strana pravidla, ktoré sa začína v danom vrchole), "*left*"(začiatok, respektíve index tokenu pred ktorým pravidlo začalo), "*right*"(koniec, respektíve index tokenu, za ktorým končí toto pravidlo) a "*subtrees*" (podstromy, respektíve potomkovia tohto vrcholu). Napríklad:

Listing 2.11: Parsovací strom vytvorený pomocou earley-parser-js

```
{
 ...
 "root": "SentenceSimple",
   "left": 6,
   "right": 8,
    "subtrees": [{
     "root": "OneWordSentence",
     "left": 6,
     "right": 8,
     "subtrees": [{
       "root": "StartingToken",
       "left": 6,
       "right": 7,
       "subtrees": [{
         "root": "<s id="s2"><w id="w5">Veta</w>",
         "left": 6,
         "right": 7,
         "subtrees": []
       }]
     },
{ ... }]}]}]}]}
```

2.7.3 Neúplný parsovací strom

V prípade, že by sa parseru nepodarilo získať zo vstupu úplný parsovací strom(vo vstupe je chyba), tak je earley-parser-js upravený tak, aby vrátil strom zahŕňajúci najdlhší čiastočný pars textu, ktorý začína na začiatku vstupného textu. Pre zachovanie štruktúry stromu sa ostávajúci nesparsovaný text spojí do jednej vety. Všetky ostatné tagy sa umiestnia, tak ako pri úplnom parsovacom strome.

Toto riešenie by malo pomôcť užívateľovi ľahšie si nájsť chybu vo svojom texte, keďže je zjavné, od ktorej vety už text nie je rozdelený a chyba by prirodzene mala byť

niekde na začiatku tejto vety(napríklad malé písmeno na začiatku vety).

2.8 Vrátenie terminálov

V tejto chvíli už máme k dispozícii všetko potrebné na vybudovanie výsledného XML. Ako prvé je potrebné vrátiť do stromu - ktorý nám vytvoril parser - tokeny(slová a interpunkciu), lebo ak by sme najprv pridávali všetky tagy, tak by sme zbytočne museli riešiť, ktorú časť hodnoty daného vrcholu stromu treba nahradiť.

Keďže jednotlivé listy stromu sú v tom istom poradí, ako boli tokeny, ktoré vznikli po odstránení HTML tagov, tak nám stačí prejsť všetky listy a v tomto istom poradí ich nahradiť:

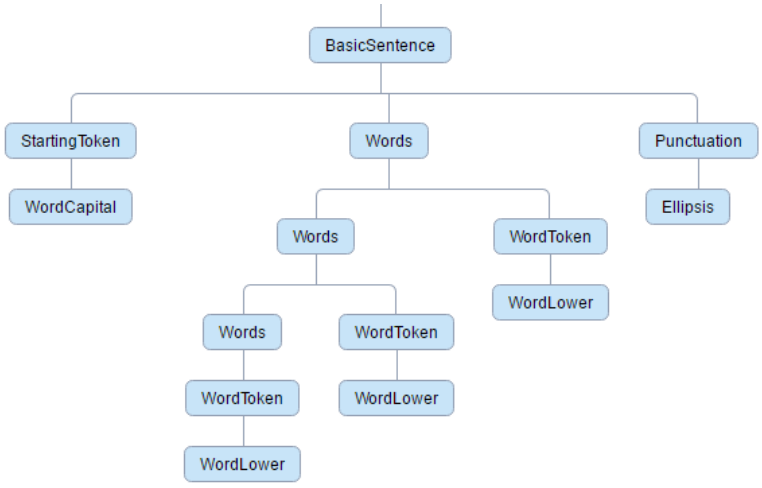

Fig. 2.4: Strom bez terminálov

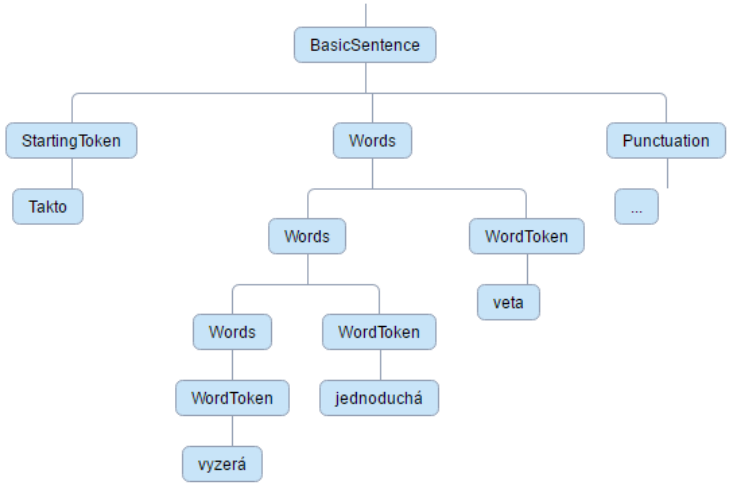

Fig. 2.5: Strom po substitúcii

2.9 Označenie slov a viet

2.9.1 Označenie slov

Pri pridávaní "*Word*" tagov je potrebné si dať pozor na tokeny, ktoré sú rozdelené do viacerých listov, napríklad: "*Ordinal* → *Num* "." ". Tieto "slová" síce nie sú v jednom vrchole, ale aj tak ich chceme označiť jedným tagom.

Vďaka štruktúre stromu vieme povedať, že to čo chceme obaliť "*Word*" tagom, sú všetky listy prislúchajúce jednému priamemu rodičovi. Teda ak sú všetci potomkovia vrcholu listy a aspoň jeden z nich nie je interpunkcia, tak prvému z týchto listov dáme otvárací tag a poslednému uzatvárací(vo väčšine prípadov je to len jeden vrchol).

Ak list obsahuje iba interpunkciu, tak ho vynechávame, lebo nás v tejto chvíli zaujímajú len slová.

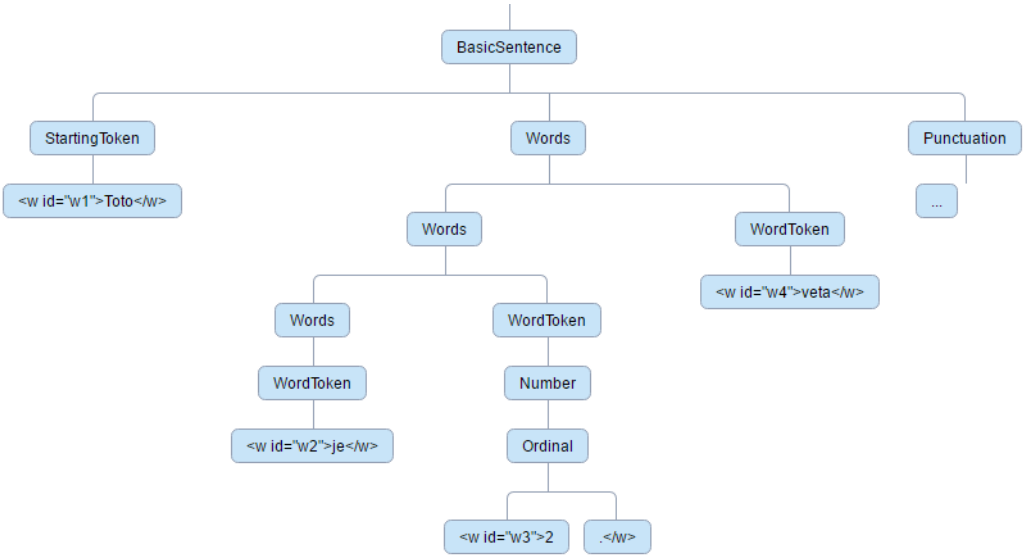

Fig. 2.6: Strom s *"Word"* tagmi

2.9.2 Označenie viet

Na to, aby sme mohli pridávať do textu `<s>` tagy, potrebujeme vedieť, ktoré pravidlá gramatiky označujú vety, napríklad:

"*BasicSentence*" : "*StartingToken Words Punctuation* | *Quotation*"

Preto v konfigurácii pre každú gramatiku je zadefinované pole "*SentenceRule*", ktoré je poľom všetkých pravidiel gramatiky, ktoré by mali mať vo výslednom texte tento tag.

```
1  "sentenceRule": ["ISentenceStart", "ISentenceEnd", "
       DirectSpeechDash", "OneWordSentence", "BasicSentence",
       "Sentence", "DirectSpeechSingle",  ...]
```

Tieto tagy, teda pridávame vždy najľavejšiemu(otvárací) a najpravejšiemu(uzatvárací) vrcholu daného podstromu, ktorého koreňom je vrchol s jedným s vyššie uvedených pravidiel.

```
addSentenceTags(tree = this.tree) {
  var counter = 1;
  function add(self, t) {
    if (t != undefined) {
      for (let node of t.subtrees) {
        if (self.config["sentenceRule"].includes(node.root)) {
          if (self.checkParent(tree, node)) {
            self.addOpeningSentenceTag(node, counter);
            self.addClosingSentenceTag(node);
            counter++;
          }
        }
        add(self, node);
      }
    }
  }
  add(this, this.tree);
}
```

Funkcia sa volá na každý vrchol parsovacieho stromu vrátane vrcholov podstromu, ktorý už bol označený `<s>` tagom, keďže vety v texte sa môžu vnárať jedna do druhej. Funkcia zároveň zabezpečuje, že všetky tagy budú v správnom poradí, teda že na vrchu bude vždy tag s najmenším poradovým číslom.

Štandardne je vytvorený `config["sentenceRule"]` tak aby sa jednotlivé vety vnárali do seba, teda dostaneme označenie `<s>` tagom pre celé skupiny viet skladajúce sa z jednotlivých uvádzacích viet a priamych rečí, tak ako aj označenie pre tieto jednotlivé uvádzacie vety a priame reči. Toto správanie sa dá veľmi ľahko zmeniť vymazaním niektorých pravidiel z `config["sentenceRule"]` a dosiahnuť tak označovanie buď len celých skupín viet alebo len ich menších častí, prípadne označovanie na základe úplne niečoho iného.

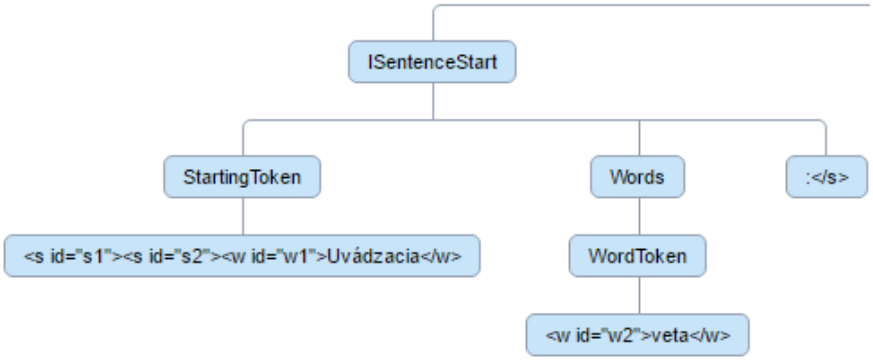

Fig. 2.7: Strom s `<s>` tagmi a vnorenou vetou(ľavá strana)

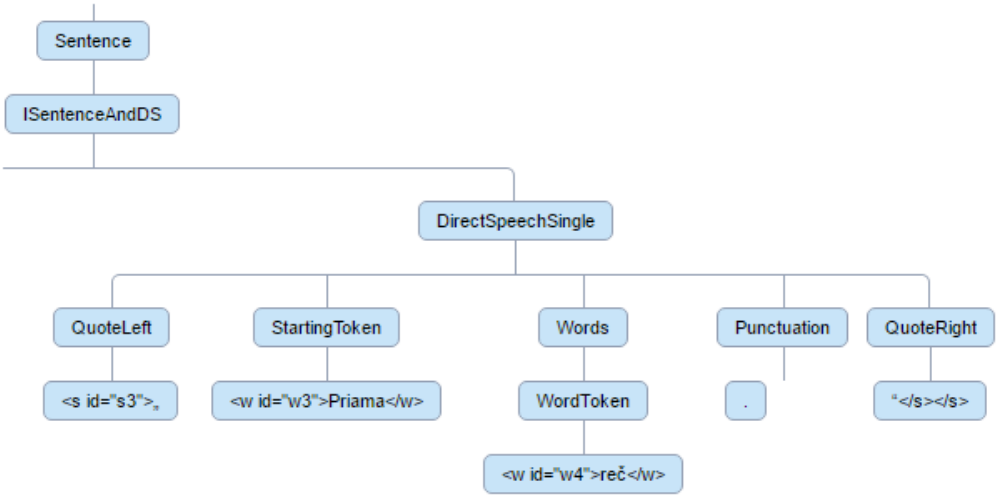

Fig. 2.8: Strom s `<s>` tagmi a vnorenou vetou(pravá strana)

2.10 Vrátenie HTML tagov

Pri HTML tagoch rozlišujeme riadkové - inline tagy(`<b>, <span>...`) a blokové tagy (`<p>, <h1-6>, <div>`) . Blokové tagy by po správnosti nemali začínať ani končiť vnútri viet, na to sa však nedá plne spoliehať. Keby sa vyskytla takáto chyba v texte, spôsobilo by to, že by sa prekrížili `<s>` tagy s tými blokovými, preto je potrebné skontrolovať a prípadne posunúť blokové tagy na začiatok, respektíve na koniec vety, v ktorej sa nachádzajú. Toto posunutie zabezpečí, že blokové tagy budú úplne navrchu a teda budú obaľovať celé vety, tým pádom sa už nemajú s čím prekrížiť.

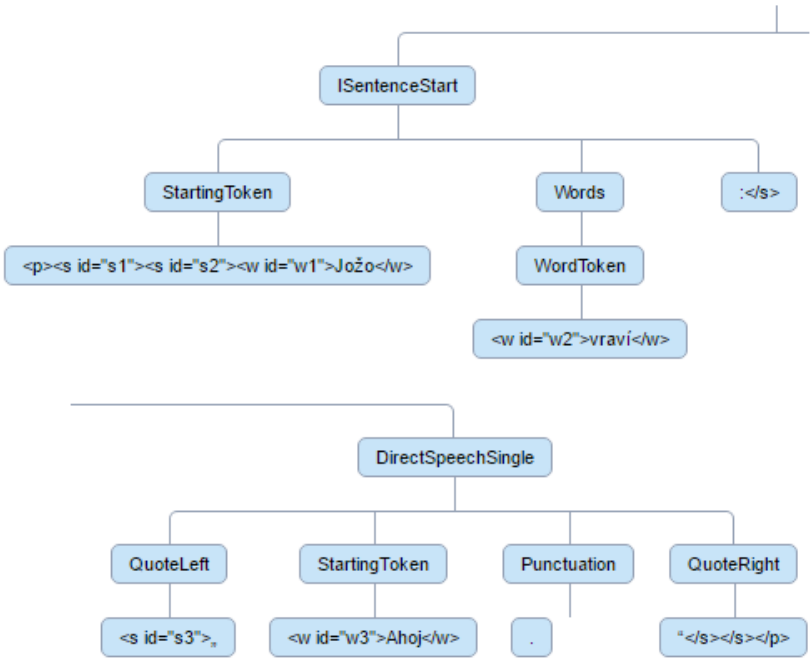

Fig. 2.9: Príklad stromu s posunutým `<p>` tagom
pre vstup: `"Jožo <p>vraví: "</p>Ahoj."`

Pri týchto tagoch sa neočakáva, že by sa vnárali jeden do druhého, keďže to Multimediálna čítanka nepotrebuje, teda na vrchu bude vždy maximálne jeden z vyššie uvedených blokových tagov.

Pokiaľ ide o riadkové tagy, tie môžu zahŕňať jedno slovo, jednu vetu alebo kľudne môžu aj začínať v strede jednej a končiť v strede druhej. Riadkové tagy, ale na rozdiel od tých blokových vieme rozdeliť na časti, tak aby sa zabránilo kríženiu. Ak teda tag prechádza cez hranice viet, tak ho na konci prvej uzatvoríme a na začiatku ďalšej zase otvoríme. Tu zároveň treba myslieť aj na to, že tieto tagy sa budú vnárať jeden do druhého a pri `<span>` tagoch to teda často budú rovnaké tagy s rôznymi atribútmi(napr. `style`), ktoré je teda potrebné uzatvárať a otvárať v správnom poradí a pri otváracích tagoch použiť prislúchajúce atribúty.

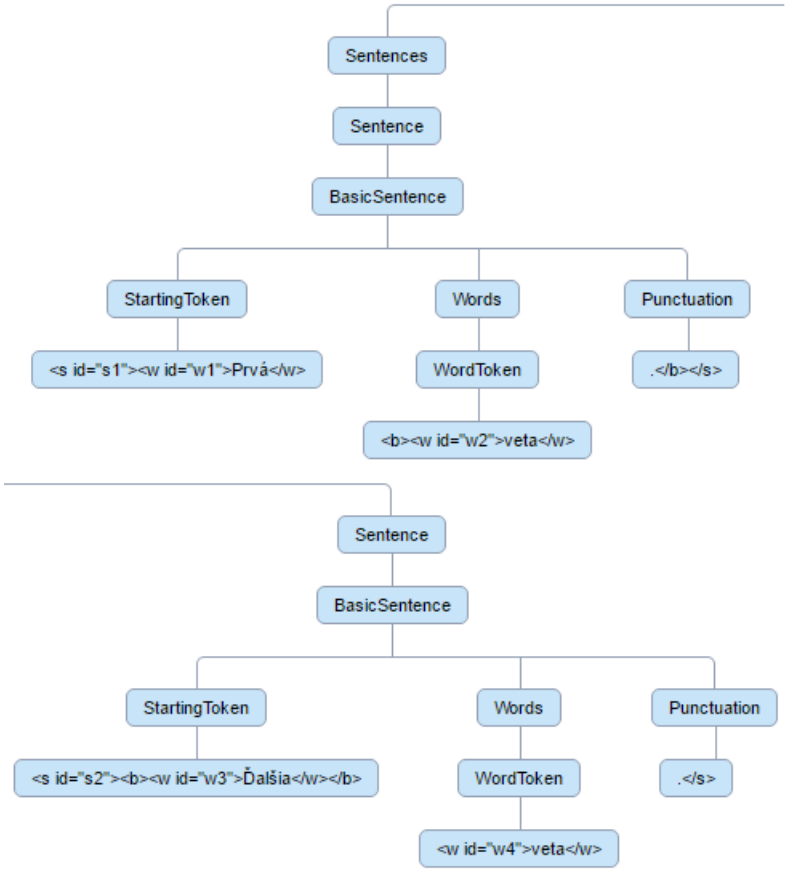

Fig. 2.10: Príklad stromu s rozdeleným `<b>` tagom pri vstupe: `"<p>Prvá <b>veta. Ďalšia</b> veta.</p>"`

2.11 Vytvorenie XML dokumentu

Po všetkých predchádzajúcich krokoch už máme v listoch stromu kompletný text so všetkými tagmi, ktoré chceme, teda nám už len stačí spojiť hodnoty jednotlivých listov dohromady a tým vznikne výsledný text.

Kapitola 3

Výkonnostné testy

V tejto kapitole sa pozrieme na výkonnosť aplikácie(parseru) a porovnáme slovenskú a anglickú gramatiku v "$left-recursive$" a "$right-recursive$" forme, pre jednoznačné aj nejednoznačné vstupy.

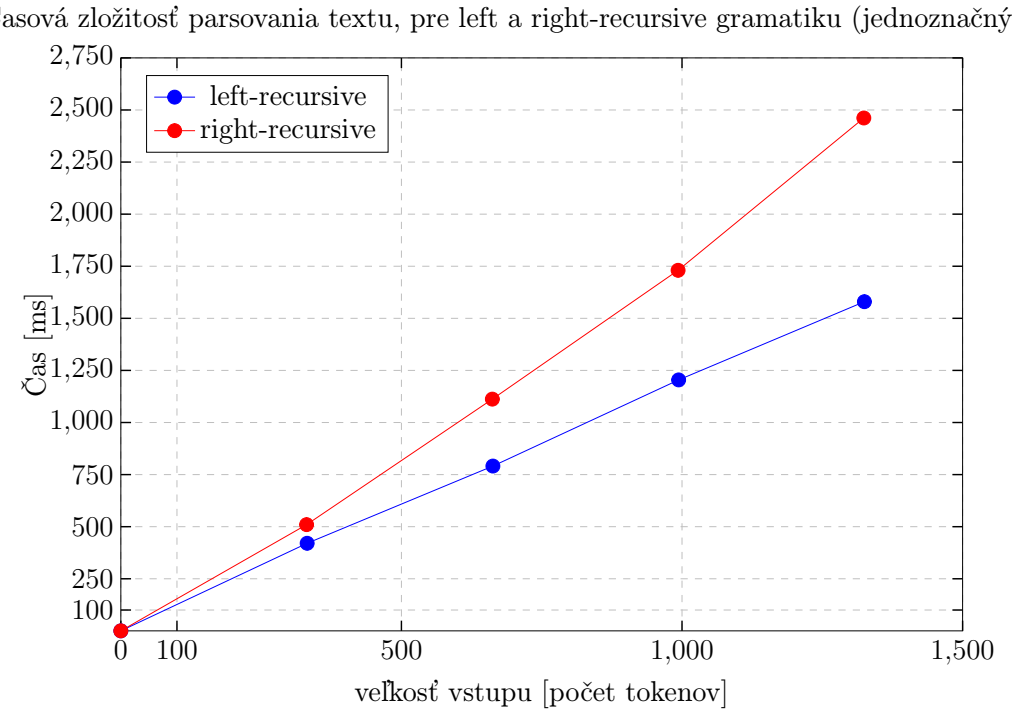

Fig. 3.1: Porovnanie *left* a *right-recurise* gramatiky

Na grafe(3.1) môžeme vidieť v akom čase je parser schopný parsovať vstupy rôznych veľkostí. Týmito vstupmi boli jednoduché vety bez priamej reči, na ktorých je najlepšie vidieť rozdiel medzi "$left$" a "$right-recursive$" gramatikou. Testované gramatiky boli rovnaké ako gramatika, ktorú využíva výsledná aplikácia, s tým, že "$right-recusive$" gramatika má všetky rekurzívne pravidlá otočené opačne. Napríklad:

$$\text{"}Lines \rightarrow Lines\ Line \mid Line\text{"} \quad \rightarrow \quad \text{"}Lines \rightarrow Line \mid Line\ Lines\text{"}$$

Je vidieť, že parser parsuje jednoznačné vstupy využívajúce slovenskú "$left-recursive$" gramatiku v lineárnom čase, a že v priebehu 1,2 sekundy zvládne vstupy okolo 1000 tokenov, čo je reálne najväčší vstup aký môže z Multimediálnej čítanky dostať.

Zároveň je zjavné, že "$right-recursive$" gramatika je oveľa pomalšia. Parser v tomto prípade parsuje približne v kvadratickom čase, čo je stále omnoho lepšie ako $O(n^3)$(general case pre Earley algorithm), ale je to zbytočné spomalenie v porovnaní s "$left-recursive$" gramatikou.

Dôvodom prečo k tomuto dochádza je, že parser v tomto prípade vykonáva veľa zbytočných krokov(*prediction* a *completion*), ktoré pre získanie parsu nepotrebuje. "$Left-recursive$" gramatika sa naopak dostáva k výsledku úplne priamočiaro[11].

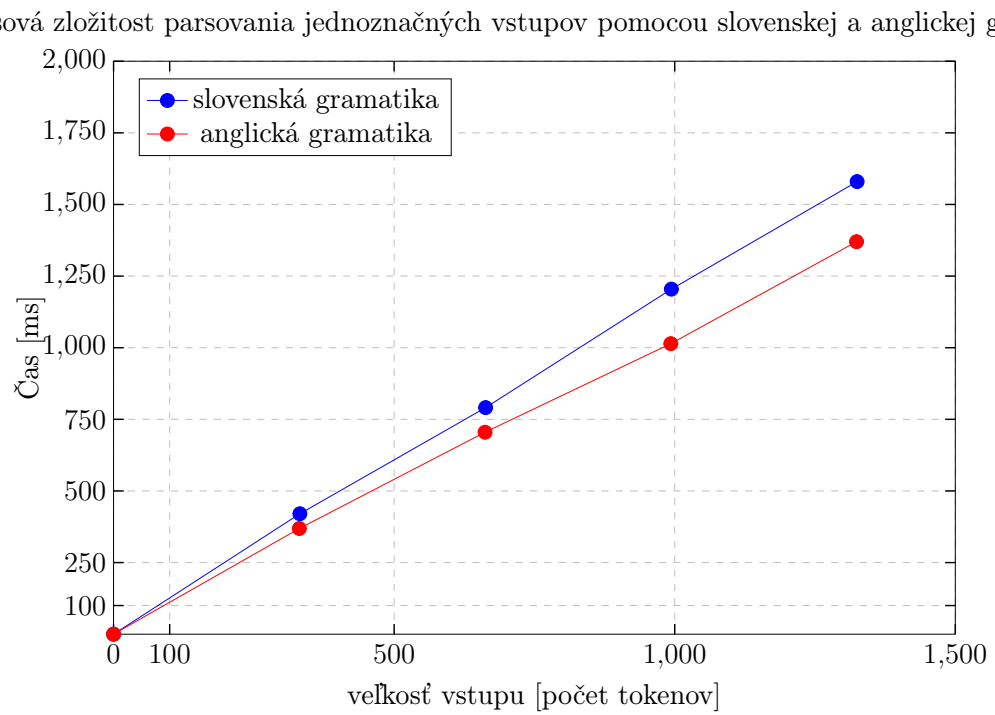

Fig. 3.2: Porovnanie slovenskej a anglickej gramatiky

Na tomto grafe(3.2) vidíme porovnanie slovenskej a anglickej gramatiky, pokiaľ ide o čas potrebný na parsovanie textu, pri použití rovnakých vstupov ako v predchádzajúcom prípade. Medzi gramatikami je v tomto ohľade minimálny rozdiel, s tým, že tá slovenská je o trochu pomalšia, z dôvodu, že obsahuje niektoré komplikovanejšie pravidlá hlavne pri priamej reči.

Pokiaľ ide o nejednoznačné vstupy, tak zložitosť sa môže pohybovať od lineárnej až po kubickú. S tým, že všetko veľmi záleží od konkrétneho vstupu, množstva nejednoznačností, ktoré sa v ňom nachádzajú a počtu úplných aj neúplných parsovacích stromov, ktoré parser vytvorí. Preto tu nebudem uvádzať graf, lebo by nemal veľkú výpovednú hodnotu.

Kapitola 4

Záver

Cieľom tejto bakalárskej práce bolo vytvoriť aplikáciu, ktorá by bola schopná sparsovať vstupný text v slovenskom alebo anglickom jazyku obsahujúci HTML a vytvoriť z neho XML dokument, v ktorom by boli označené jednotlivé vety a slová XML tagmi.

Na začiatku prebiehalo rozsiahle hľadanie rôznych parserov a ich implementácií vhodných pre parsovanie prirodzeného jazyka, spolu s oboznamovaním sa s problematikou a následnou analýzou možných implementácií s ohľadom na efektívnosť a modifikovateľnosť daného riešenia pre naše potreby.

Po zhromaždení technológií a potrebných nástrojov, prišlo na rad vytvorenie jednoduchého prototypu, schopného parsovať čistý text pomocou veľmi jednoduchej gramatiky. Nasledovalo vytvorenie plnej verzie aplikácie spolu s gramatikou, slovníkom a vizualizáciou.

Splnenie cieľu bolo úspešné a aplikácia je pripravená na používanie ako podporný modul v rámci Multimediálnej čítanky, pre ktorú bola navrhnutá, prípadne v iných situáciách, kde by bolo potrebné deliť komplexný text na vety a slová. Za úspech považujem, fakt, že aplikácia parsuje text oveľa efektívnejšie než sa pôvodne očakávalo, čo možno vidieť v kapitole 3.

Počas implementácie sa vyskytlo zopár problémov, ako napríklad nutnosť upraviť pôvodný parser, ktorý aplikácia využíva. Takisto bolo potrebné veľakrát meniť gramatiku a slovník parseru, kvôli nejednoznačnosti a veľkej flexibilite syntaxe slovenského jazyka. Všetky problémy sa však podarilo vyriešiť.

Aj napriek tomu, že všetky požiadavky zo zadania boli splnené, tak stále existuje priestor pre zlepšenie a úpravy, napríklad pokiaľ ide gramatiku, ktorú by bolo možné viac optimalizovať, prípadne ďalej rozširovať alebo použiť expresívnejšiu formu ako napríklad BNF alebo EBNF. Ďalšou možnosťou je použitie štatistického parsovania pre zlepšenie parsovania, prípadne pridanie gramatík iných jazykov okrem slovenčiny a angličtiny, čo by určite pomohlo k rozšíreniu možností využitia tejto aplikácie.

Literatúra

[1] Stephen G. Pulman, *Basic Parsing Techniques: an introductory survey.*, 1991, `https://www.cs.ox.ac.uk/files/219/parsing.pdf`, [Prístup 4.4.2017],

[2] Roxana Girju, *Introduction to Syntactic Parsing.*, 2004, `http://l2r.cs.uiuc.edu/~danr/Teaching/CS598-05/Lectures/Roxana.pdf` [Prístup 13.11.2016],

[3] John E. Hopcroft, Jeffrey D. Ullman. *Formálne jazyky a automaty. 1. vyd.*, Bratislava : Alfa, 1978, 342 s.

[4] Loup Vaillant, *Earley Parsing Explained*, `http://loup-vaillant.fr/tutorials/earley-parsing/`, [Prístup 13.11.2016].

[5] Te Li, Devi Alagappan, *A Comparison of CYK and Earley Parsing Algorithms.*, 2006, `http://staff.icar.cnr.it/ruffolo/progetti/projects/09.Parsing\%20CYK/A\%20Comparison\%20of\%20CYK\%20and\%20Earley\%20Parsing\%20Algorithms-cykeReport.pdf` [Prístup 18.11.2016],

[6] Masaru Tomita, *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*, Springer US, ISBN 978-1-4757-1885-0, 1986, 201 s.

[7] *Nearley.js* `http://nearley.js.org/` [Prístup: 13.11.2016]

[8] *Stanford CoreNLP* `http://nearley.js.org/` [Prístup: 13.11.2016]

[9] *earley-parser-js* `https://github.com/lagodiuk/earley-parser-js` [Prístup: 4.4.2017]

[10] *Javascript Documentation* `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference`, [Prístup: 4.4.2017]

[11] Dick Grune, Ceriel J.H. Jacobs, *Parsing Techniques: A Practical Guide. 2. Edition*, Springer-Verlag New York, ISBN 978-0-387-68954-8, 2008, 662 s.

Prílohy

- Digitálna príloha - CD obsahuje:
  - zdrojovými kódmi aplikácie
  - návod na inštaláciu
  - používateľská príručka
  - dokumentácia zdrojového kódu
  - tester a vizualizáciu