FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Podpora tvorby a automatická oprava zjednodušeného relačního zápisu v portálu dbs.fit.cvut.cz
Student:	Martin Šach
Vedoucí:	Ing. Jiří Hunka
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2020/21

Pokyny pro vypracování

Pro předmět BI-DBS je provozován nástroj pro celkovou podporu výuky dbs.fit.cvut.cz. V rámci tohoto portálu probíhá testování studentů, přičemž podstatnou částí testování je transformace konceptuálního modelu do zjednodušeného relačního zápisu (dále jen transformace). Historicky již existuje první verze automatické opravy této transformace, avšak její stav není zcela vyhovující. Cílem této práce je realizace nového řešení včetně rozšířené funkcionality pro snadnější tvorbu zápisu (automatické našeptávání, snazší zadávání, jednodušší oprava atd.)
Postupujte v těchto krocích:
Analyzujte současný stav transformace v portálu dbs.fit.cvut.cz.
Analyzujte potřeby studentů respektivě vyučujících týkající se vypracování otázek transformace respektivě jejich tvorby.
Na základě analýzy navrhněte vhodné řešení pro obě skupiny.
Implementujte prototyp dle návrhu.
Prototyp řádně otestujte.
Opravte nalezené problémy, pokuste se nasadit Vaše řešení pro budoucí studenty a vyučující.
Zhodnoťte stav.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 19. února 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Podpora tvorby a automatická oprava zjednodušeného relačního zápisu v portálu dbs.fit.cvut.cz

Martin Šach

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Hunka

6. ledna 2020

Poděkování

Rád bych poděkoval panu Ing. Jiřímu Hunkovi, vedoucímu této práce, za jeho trpělivost, rady a odborné vedení. Rovněž bych rád poděkoval Bc. Filipovi Dolníkovi za pomoc při návrhu řešení a Bc. Oldřichovi Malcovi za spolupráci při nasazení aplikace na portál kurzu BI-DBS – Databázové systémy. Na závěr bych rád poděkoval své rodině a přátelům za jejich vytrvalou podporu v průběhu celého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 6. ledna 2020 ……

Odkaz na tuto práci

Šach, Martin. *Podpora tvorby a automatická oprava zjednodušeného relačního zápisu v portálu dbs.fit.cvut.cz*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato bakalářská práce popisuje návrh, implementaci a testování aplikace pro výukový portál kurzu Databázové systémy. Konkrétně se jedná o nástroj pro transformaci zjednodušeného relačního zápisu do vhodného datového formátu napsaný v programovacím jazyce Kotlin a textový editor umožňující snadnější zápis uživatelského vstupu, který obsahuje i vlastní našeptávač a který je napsán v JavaScriptu.

Výsledkem této práce je aplikace, která je nasazená a používaná na výukovém portálu výše zmíněného kurzu.

Klíčová slova databázové systémy, zjednodušený relační zápis, parser, LL1 analýza, syntaktická analýza, lexikální analýza

Abstract

This bachelor thesis describes design, implementation and testing of the application for education portal of Database systems course. It is a tool intended for transformation of simplified relational notation to suitable data format which was written in programming language Kotlin. This tool also includes a text editor written in JavaScript with autocomplete function which allows easier entry of user input.

The result of this thesis is application which is deployed and used on education portal of the course mentioned above.

Keywords database systems, simplified relational notation, parser, LL1 analysis, syntax analysis, lexical analysis

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Úvod

Tato práce se zabývá především návrhem a implementací aplikace pro portál `dbs.fit.cvut.cz`. Výsledkem této aplikace bude textový editor, který budou využívat studenti předmětu BI-DBS v testech. Úkolem editoru také bude přepsat vstup do vhodného datového formátu.

Hlavním přínosem této práce je zdokonalení výukového portálu právě pro předmět BI-DBS, což v neposlední řadě znamená také ušetření času a práce vyučujících. Motivací pro výběr tohoto tématu bylo to, že se autor práce podílel na vývoji výše zmíněného portálu v kurzech BI-SP1 a BI-SP2. V jistém smyslu tato práce navazuje bakalářskou práci Filipa Machaly, která je zanalyzována níže v textu.

V kapitole 1 jsou shrnuty cíle práce – hlavním cílem je dodat aplikaci, která bude zpočátku nasazena na portálu kurzu BI-DBS a následně bude součástí komplexního systému běžícího nezávisle na portálu `dbs.fit.cvut.cz`. Rovněž jsou jsou zde popsány i vedlejší cíle – a to především chytrá odezva studentům při zadávání relačního zápisu zahrnující mimo jiné i chytrý našeptávač.

Kapitola 2 mapuje předešlá řešení této problematiky včetně toho dosud aktuálního. Je zde detailní popis výhod ale také nevýhod těchto řešení.

Návrh řešení je popsán v kapitole 3. Rovněž jsou zde zavedeny i používané definice. V samotném závěru kapitoly jsou pak vysloveny základní informace o vývoji a nasazení aplikace.

Samotnou implementaci řešení problému shrnuje kapitola 4. Jsou zde popsány třídy překladače a jejich metody, komentován textový editor a také jeho propojení s překladačem.

Nasazení výsledné aplikace na portál `dbs.fit.cvut.cz` a případné problémy s tím spojené mapuje kapitola 5.

Závěrečná kapitola 6 popisuje testování celé aplikace a to včetně testování v reálném provozu studenty. Toto testování se odehrálo v rámci zápočtového testu kurzu BI-DBS.

Kapitola 1

Cíl práce

Hlavním cílem této práce je realizace nového řešení pro tvorbu a automatickou opravu zjednodušeného relačního zápisu pro předmět BI-DBS – Databázové systémy a jeho portál `dbs.fit.cvut.cz`. Dodávaná aplikace bude zpočátku nasazena přímo na portálu, avšak po dokončení komplexního systému DSM, který bude mimo jiné zajišťovat kreslení ER diagramů a srovnávat zjednodušený relační zápis přímo s diagramem, bude součástí právě tohoto systému, který poběží nezávisle na portálu. Ačkoliv v současné době již obdobné řešení existuje, jeho provedení není zcela vyhovující, neboť nezapadá do nově dodávaného systému. Rovněž také neobsahuje funkcionality, které by usnadňovaly uživatelský zápis.

Součástí práce tedy bude detailní analýza předešlého řešení, na jejímž základě bude navrhnuto a zhotoveno řešení nové obsahující mimo jiné i rozšířené funkcionality. Těmito funkcionalitami budou například snadnější tvorba zápisu, automatické našeptávání nebo průběžná kontrola syntaxových chyb. Aby se předešlo problémům při nasazení a samotném běhu aplikace, bude rovněž nutné nové řešení řádně otestovat.

Klíčovou součástí nově navrženého řešení bude transformace zjednodušeného relačního zápisu do vhodného datového formátu, který bude následně využíván při automatické opravě.

Kapitola 2

Analýza současného stavu

Tato kapitola zavádí používané pojmy a vystvětluje sktrukturu zjednodušeného relačního zápisu. Dále pak shrnuje výhody a nevýhody předešlých řešení této problematiky v rámci kurzu BI-DBS – Databázové systémy a jeho portálu `dbs.fit.cvut.com`.

2.1 Používané pojmy

- **`Kreslítko`** – aplikace v portálu `dbs.fit.cvut.cz`, která umožňuje zakreslení konceptuálního schématu,
- **`Tralex`** – aplikace v portálu `dbs.fit.cvut.cz`, která aktuálně řeší automatickou opravu zjednodušeného relačního zápisu,
- **`Nový Tralex`** – aplikace, která nahrazuje `Tralex` a jejíž zhotovení popisuje tato práce.

2.2 Zjednodušený relační zápis

Zjednodušený relační zápis umožňuje úsporně (textově) zapsat nějaký databázový konceptualní model. Převod z konceptuálního schématu na relační je jasně definován právě v kurzu BI-DBS [1].

Tedy entitní tabulka se převede do tvaru: `název_tabulky(atributy)`. Atributů může být samozřejmě více a jsou odděleny čárkou. Primární atribut (případně skupina primárních atributů) se označuje souvislým podtržením. Teoreticky tedy může nastat situace, kdy je v jedné relaci více primárních klíčů, protože nejsou podtrženy souvisle. Po domluvě s vedoucím práce bude tato situace řešena tak, že všechny skupiny primárních atributů budou sjednoceny do jedné. Pro účely portálu `dbs.fit.cvut.cz` je pak zavedeno ještě označení povinných atributů. Před tyto atributy se vkládá prefix `*`. Atributy

bez tohoto prefixu jsou brány jako nepovinné. V řešení této práce je dále zaveden i atributový prefix `@`, který značí unikátnost hodnoty tohoto atributu v databázi.

Závislosti, jako například cizí klíče, se zapisují jako tzv. referenční integrita. Ta se zapisuje ve tvaru: `název_cíle[atributy]` $\subseteq$ `název_zdroje[atributy]`. `Název_cíle` vyjadřuje entitu, ve které se vyskytuje cizí klíč (nebo skupina cizích klíčů). Podobně `název_zdroje` značí název entity, ze které je cizí klíč převzat. V případě, že je cizí klíč složen z více atributů, jsou tyto atributy odděleny čárkou, stejně jako v zápisu entity. Atributy v cílové relaci nemusejí nutně nést stejný název jako atributy ve zdrojové relaci. Záleží ale na jejich pořadí.

2.3 Předešlé verze testování na portálu dbs.fit.cvut.cz

Na portálu `dbs.fit.cvut.cz` byly v reálném provozu používané v zásadě dvě verze, které se více či méně zabývaly podobným problémem. První verze přinesla možnost testovat studenty na portálu. Aktuální verze pak vyčlenila zjednodušený relační zápis do samostatného typu testovací otázky. Obě verze popisují následující řádky.

2.3.1 První verze

Vůbec první verze testování studentů na portálu `dbs.fit.cvut.cz` byla zpracována v bakalářské práci Petra Pejši [2] v letním semestru akademického roku 2015/2016.

V této verzi ještě neexistovala samostatná odpověď typu transformace. Používala se tedy obyčejná textová odpověď. Verze podporovala mimo jiné také automatickou opravu, i když nutno říci, že značně omezenou. Pokud se studentova odpověď shodovala s referenční odpovědí učitele (byla nutná úplná shoda obou řetězců), byla tato odpověď ohodnocena maximálním možným počtem bodů. Pokud mezi studentovou a referenční odpovědí nebyla nalezena absolutní shoda, byl zde ještě druhý krok – hledání studentovy odpovědi v již ohodnocených odpovědích v databázi. Pokud byla nalezena shoda, byla odpověď ohodnocena stejným počtem bodů jako u odpovědi v databázi. Jinak následovalo přesměrování na manuální opravu, kterou prováděl učitel ručně.

Výše uvedená automatická oprava nebyla nijak zvlášť propracovaná – učitelům tolik práce neušetřila. Na druhou stranu je třeba zmínit, že automatická oprava nebyla primárním cílem práce. Dalo se tedy očekávat, že časem přijde více propracovaná verze, která se bude zabývat již jen automatickou opravou odpovědí typu transformace.

2.3.2 Aktuální verze

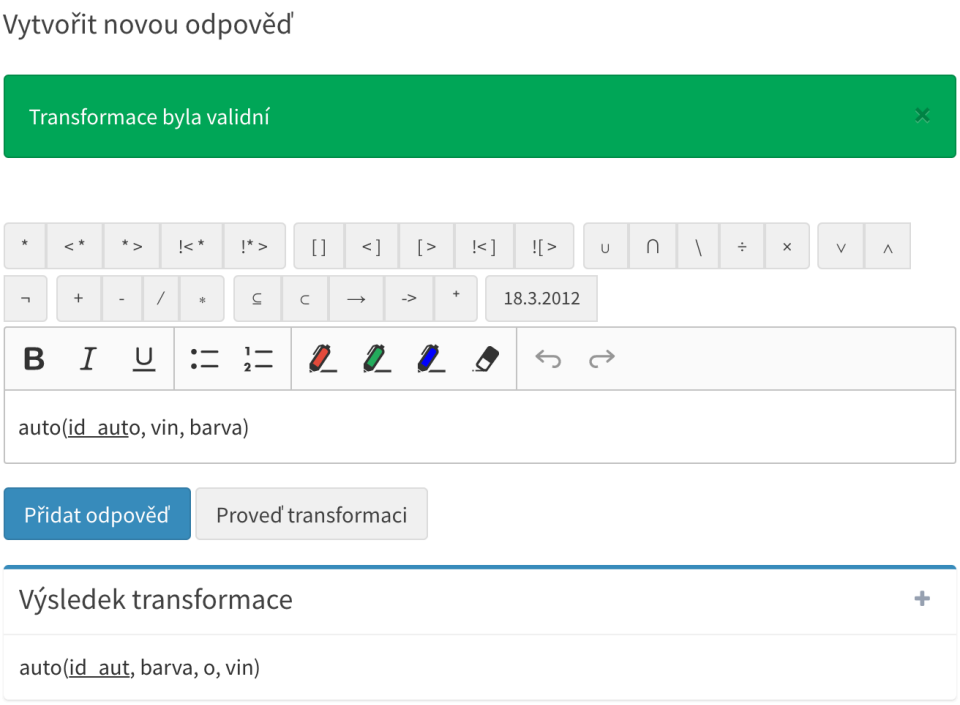
Poslední verze automatické opravy zjednodušeného relačního zápisu (dále nazývaná `Tralex`) pochází z bakalářské práce Filipa Machaly [3] z letního semestru akademického roku 2017/2018.

Tato verze dostala oproti původní verzi mnoho vylepšení. Pro začátek – transformace relačního zápisu měla nově vyhrazena speciální typ odpovědi. Student nebo učitel si mohl nechat transformovat napsaný text a následně mohl odhalit případné chyby syntaxe v zápisu. Dále mohl učitel zadat více různých odpovědí. Různých ve smyslu různých řešení problému, nikoliv například jiné pořadí entit – to je bráno jako jedno řešení.

Novinkou také bylo logické ukládání transformovaného vstupu. Vstup byl převeden do formátu JSON na jednotlivé entity a případné další důležité položky a následně bylo provedeno porovnání JSON dat od studenta a učitele (i díky tomu se dalo elegantně rozpoznat duplicitní odpovědi a srovnávat je s jednou referenční). V případě shody s alespoň jednou referenční odpovědí byla studentova odpověď ohodnocena plným počtem bodů. Pokud byla nalezena nějaká neshoda, pak musela být studentova odpověď opravena manuálně učitelem.

První nevýhodou `Tralexu` bylo, že neuměl studentům realtime (okamžitě) hlásit chyby syntaxe v zápisu. Z počátku se to nemusí zdát jako velká nevýhoda – prostě bylo jen potřeba udělat klik navíc. Nicméně i to mohlo studenty v testu zdržovat a navíc se `Tralex` pokoušel některé zápisy s chybou syntaxe doplňovat sám, což nebyla vždy správná volba. Jako příklad může posloužit situace, kdy student označoval podtržením primární klíč v nějaké entitě. Pokud totiž student podtrhl všechna písmena atributu kromě písmena posledního, pak byla transformace provedena tak, že podtržená část slova se stala primárním klíčem a poslední písmeno pak samostatným atributem. Této automatické úpravy, kterou `Tralex` provedl, si student nemusel vždy všimnout, což umocňuje fakt, že tato situace může nastat při neúmyslném nepodtržení celého slova. Tento problém je možné elegantně řešit právě realtime oznámením o syntaktické chybě, případně chybovou hláškou po kliknutí na tlačítko vyhodnocení. `Tralex` se zde zřejmě pokoušel vydedukovat pravděpodobnou myšlenku studenta, ale nutno říci, že ne vždy správně, což ve výsledku samozřejmě přidělává práci opravujícím.

Problém byl též v transformaci relací mezi entitami, tzv. referenčními integritami. Především nebylo možné tyto relaci zapisovat v libovolném pořadí. Pořadí bylo předem definováno tak, že se nejprve zapíše cílová entita (a její atributy), znak podmožiny a zdrojová entita (a její atributy). S opačným směrem si transformace poradit nedokázala. Tedy například následující vstup: `mechanik[id]` $\supseteq$ `oprava[id_mechanik]` skončil chybou. Rovněž neprošel transformací vstup `název_cíle` $\subseteq$ `název_zdroje`, který sice z databázového pohledu není smysluplný, nicméně podle obecné matematické definice podmožiny validní.



Obrázek 2.1: Transformace po nepodtržení celého primárního atributu

Rovněž nebylo studentům umožněno vložit do vstupu poznámku či komentář tak, aby transformace zůstala validní. Tato práce zavádí i zápis komentářů – a to jak jednořádkových, tak víceřádkových.

Další možnou nevýhodou pro vyučující bylo, že bylo nutné vymyslet více možných odpovědí studenta, aby se s co možná nejvyšší pravděpodobností předešlo manuální opravě. Na druhou stranu je třeba říci, že na to, aby stačila učiteli pouze jedna odpověď, `Tralex` tehdy zřejmě nebyl ani směřován. I proto se tedy přichází s novým řešením. Toto řešení by mělo odstranit výše zmíněné nedostatky. Je vyvinuto zcela nezávisle na portálu a v budoucnu bude součástí komplexní aplikace DSM.

Kapitola 3

Návrh řešení

Tato kapitola nejprve zavádí definice a následně popisuje a návrh samotného řešení. Hlavním problémem je přeložit uživatelský vstup do vhodného a validního formátu, K řešení tohoto problému autor využívá znalostní získaných v kurzu BI-AAG – Automaty a gramatiky. Částečně je také čerpáno z materiálů kurzu BI-PJP – Programovací jazyky a překladače. V závěru této kapitoly jsou také popsány základní informace týkající se samotné implementace.

3.1 Definice

Softwarový proces je množina aktivit (jejich souslednost, opakování, vstupy a výstupy) nutných k tomu, aby software vznikl. [8]

Gramatika je uspořádaná čtveřice $G = (N, T, P, S)$, kde:

- N je konečná množina neterminálních symbolů,
- T je konečná množina terminálních symbolů ($T \cap N = \emptyset$),
- P je konečná množina přepisovacích pravidel,
- $S \in N$ je počáteční symbol gramatiky. [4]

Gramatika je **bezkontextová**, jestliže každé pravidlo má tvar A $\rightarrow \alpha$, kde A je neterminální symbol a α je libovolný řetězec složený z terminálních a neterminálních symbolů. [4]

Mějme gramatiku $G = (N, T, P, S)$. **Derivační strom** je strom, který má následující vlastnosti:

- uzly derivačního stromu jsou ohodnoceny terminálními a neterminálními symboly a symbolem ϵ (pak je to jediný syn svého otcovského uzlu),

- kořen stromu je ohodnocen počátečním symbolem S,
- jestliže uzel má alespoň jednoho následovníka, je ohodnocen neterminálním symbolem,
- jestliže $n_1, n_2, ..., n_k$ jsou bezprostřední následovníci uzlu n, který je ohodnocen symbolem A, a tyto uzly jsou zleva doprava ohodnoceny symboly $A_1, A_2, ..., A_k$, pak $A \to A_1 A_2 ... A_k$ je pravidlo v P,
- koncové uzly derivačního stromu tvoří zleva doprava větnou formu nebo větu v gramatice G, která je výsledkem derivačního stromu. [4]

$\alpha \Rightarrow^k \beta$, jestliže existuje posloupnost $\alpha_0, \alpha_1, ..., \alpha_k$, pro $k \geq 0$, $k+1$ řetězců takových, že $\alpha = \alpha_0, \alpha_1 \Rightarrow \alpha_i$ pro $1 \leq i \leq k$, a $\alpha_k = \beta$. Tuto posloupnost nazveme **derivací řetězce** β **z řetězce** α, která má délku k v gramatice G. [4]

Levou derivací se pak rozumí taková derivace, při které se v každém kroku nahrazuje první neterminál zleva. Derivačnímu stromu odpovídá jediná levá derivace a naopak určité levé derivaci odpovídá jediný derivační strom [5]

Lexikální analyzátor převádí vstupní řetězec na jednotlivé tokeny. Tokeny jsou terminálními symboly bezkontextové gramatiky, která popisuje syntaxi vstupního řetězce. [6]

Je dána $G = (N, T, P, S)$. Předpokládejme, že pravidla v P jsou očíslována $1, 2, ..., |P|$. **Rozkladem větné formy** α v G je posloupnost čísel pravidel použitých v derivaci $S \Rightarrow^* \alpha$. **Levým rozkladem větné formy** α v G je pak posloupnost čísel pravidel použitých v levé derivaci $S \Rightarrow^* \alpha$. [5]

Syntaktický analyzátor vrátí pro zadanou vstupní větu její syntaktickou strukturu (též derivační strom). U bezkontextových jazyků existují dvě základní metody syntaktické analýzy: **shora dolů** a **zdola nahoru**. Obě tyto metody jsou obecně nedeterministické (nejednoznačné). [7]

Pro účely této práce je zajímavá první metoda, totiž metoda shora dolů. **Syntaktická analýza metodou shora dolů** je proces nalezení levého rozkladu dané věty v dané gramatice. [5]

Funkce $First(\alpha)$ je definovaná pro libovolný řetěz $\alpha \in (N \cup T)^*$ a je to množina všech terminálních symbolů (včetně ϵ), kterými mohou začínat řetězce generovatelné z α. [7]

Funkce $Follow(\alpha)$ je definovaná pro libovolný neterminál A a je to množina všech terminálních symbolů (včetně ϵ), které se mohou vyskytovat ve větných formách bezprostředně za symbolem A. [7]

Gramatika $G = (N, T, P, S)$ je **LL(1) gramatikou**, když pro každou dvojici přepisovacích pravidel $A \to \alpha|\beta$ platí:

- $First(\alpha) \cap First(\beta) = \emptyset$,
- jestliže $\epsilon \in First(\alpha)$, pak $Follow(\alpha) \cap First(\beta) = \emptyset$. [7]

Podstatou výše zmíněné definice je to, že při čtení vstupu je možné po přečtení každého jednoho symbolu jednoznačně určit, které pravidlo bude následovat. Neboli taková gramatika je deterministická (jednoznačná), takže nemůže nastat situace, kdy by bylo možné si po přečtení nějakého symbolu „vybrat“ z více pravidel.

3.2 Bezkontextová gramatika

Pro správné fungování celého překladače je nezbytné navrhnout detailní bezkontextovou gramatiku, která bere v úvahu všechny možné uživatelské vstupy. Nejprve jsou popsány tokeny, tedy terminální symboly této gramatiky. Oproti minulé verzi jsou zde zastoupeny i komentáře a to jak jednořádkové, tak víceřádkové. Rovněž je nově vyhrazen terminální symbol pro unikátní atributy v rámci entity, který bude reprezentován znakem „@“.

3.2.1 Terminální symboly

Nechť je množina T, která obsahuje terminální symboly bezkontextové gramatiky, definována následovně:

```
T = {
 name, text, l_paren, r_paren, l_bracket, r_bracket, comma,
 star, at, l_depend, r_depend, line_comment_start, endline,
 comment_start, comment_end, underline_start, underline_end
}
```

Vysvětlení jednotlivých terminálních symbolů (tokenů):

- `name` – název entity nebo atributu,
- `text` – komentář,
- `l_paren` – (– začátek entitních atributů (levá kulatá závorka),
- `r_paren` –) – konec entitních atributů (pravá kulatá závorka),

- `l_bracket` – [– začátek atributů závislosti (levá hranatá závorka),
- `r_bracket` –] – konec atributů závislosti (pravá hranatá závorka),
- `comma` – , – oddělovač atributů (čárka),
- `star` – * – povinný atribut (hvězdička),
- `at` – @ – unikátní atribut (zavináč),
- `l_depend` – $\subseteq$ – znak závislosti (podmnožina),
- `r_depend` – $\supseteq$ – znak závislosti (podmnožina),
- `line_comment_start` – // – jednořádkový komentář,
- `comment_start` – /* – začátek komentáře,
- `comment_end` – */ – konec komentáře,
- `underline_start` – <u> – začátek primárních atributů (HTML tag),
- `underline_end` – </u> – konec primárních atributů (HTML tag),
- `endline` – \n – konec řádku.

3.2.2 Neterminální symboly

Dále nechť je množina N, která obsahuje neterminální symboly bezkontextové gramatiky, definována následovně:

```
N = {
 START, ENTITY, RELATION, LINE_COMMENT, COMMENT,
 ATTRIBUTES, ATTR_SEP, PRIMARY_ATTR_FIRST,
 PRIMARY_ATTR, REL_PART, REL_ATTR_FIRST, REL_ATTR
}
```

Vysvětlení jednotlivých neterminálních symbolů:

- `START` – začátek čtení řádku,
- `ENTITY` – parsování entity,
- `RELATION` – parsování relace,
- `LINE_COMMENT` – parsování komentáře,
- `COMMENT` – parsování víceřádkového komentáře,
- `ATTRIBUTES` – parsování entitního atributu,

- `ATTR_SEP` – parsování oddělovače entitních atributů,
- `PRIMARY_ATTR_FIRST` – parsování prvního primárního atributu,
- `PRIMARY_ATTR` – parsování primárního atributu,
- `REL_PART` – parsování části relace,
- `REL_ATTR_FIRST` – parsování prvního relačního atributu,
- `REL_ATTR` – parsování relačního atributu.

3.2.3 Přepisovací pravidla

Množina přepisovacích, též přechodových, pravidel P nechť je definována následovně:

```
P = {
 START := ENTITY START
 | RELATION START
 | LINE_COMMENT START
 | COMMENT START
 | epsilon,

 ENTITY := name l_paren ATTRIBUTES r_paren,

 ATTRIBUTES := name ATTR_SEP
 | star name ATTR_SEP
 | at name ATTR_SEP
 | underline_start PRIMARY_ATTR_FIRST ATTR_SEP
 | epsilon,

 ATTR_SEP := comma ATTRIBUTES
 | epsilon,

 PRIMARY_ATTR_FIRST := name PRIMARY_ATTR,

 PRIMARY_ATTR := comma name PRIMARY_ATTR
 | underline_end,

 RELATION := name l_depend name
 | name r_depend name
 | REL_PART l_depend REL_PART
 | REL_PART r_depend REL_PART,

 REL_PART := name l_bracket REL_ATTR_FIRST r_bracket,
```

```
 REL_ATTR_FIRST := name REL_ATTR,

 REL_ATTR := comma name REL_ATTR
 | comma
 | epsilon,

 LINE_COMMENT := line_comment_start text endline,

 COMMENT := comment_start text comment end
}
```

3.2.4 Výsledná gramatika

Z množin sestavených v sekcích T, N a P sestavených v sekcích **3.2.1**, **3.2.2**, a **3.2.3** je pak možné sestavit výslednou bezkontextovou gramatiku v následujícím tvaru:

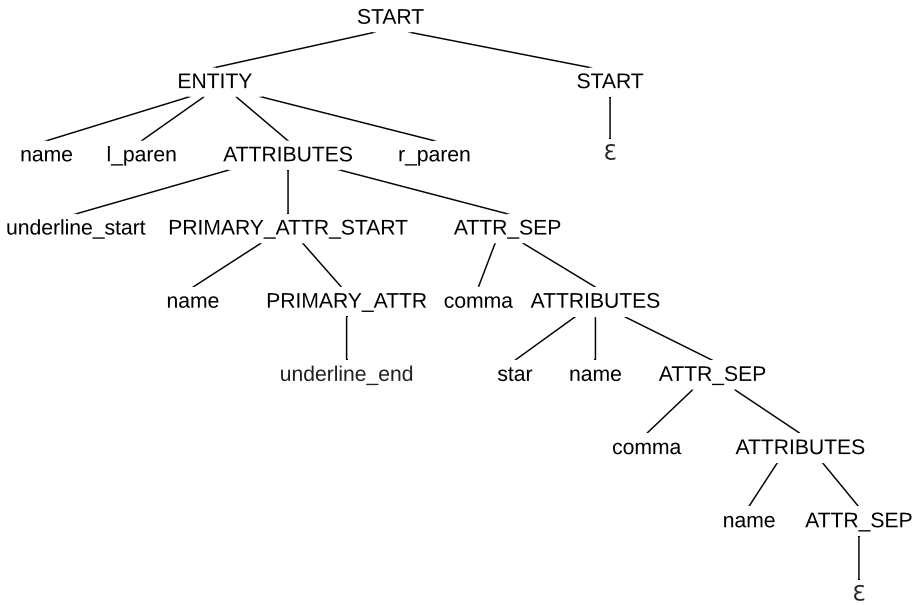
`G = (N, T, P, START).`

Gramatika G obsahuje celkem 17 terminálů (též tokenů), 12 neterminálů a 27 přechodových pravidel. Tato gramatika dokáže zpracovat jakýkoliv uživatelský vstup zadaný v jazyku zjednodušeného relačního zápisu (za předpokladu, že tento vstup bude validní a bude převeden na výše definované tokeny).

Pro příklad nechť je uvažován tento uživatelský vstup: *auto(id_auto, *vin, barva)*. Jeho přepis na tokeny a levá derivace podle gramatiky G pak vypadá následovně:

```
Tokeny:
name l_paren underline_start name underline_end comma star
name comma name r_paren

Levá derivace:
START => ENTITY START
=> name l_paren ATTRIBUTES r_paren START
=> name l_paren underline_start PRIMARY_ATTR_FIRST ATTR_SEP
r_paren START
=> name l_paren underline_start name PRIMARY_ATTR ATTR_SEP
r_paren START
=> name l_paren underline_start name underline_end ATTR_SEP
r_paren START
=> name l_paren underline_start name underline_end comma
ATTRIBUTES r_paren START
```



Obrázek 3.1: Derivační strom výše uvedeného uživatelského vstupu

```
=> name l_paren underline_start name underline_end comma
star name ATTR_SEP r_paren START
=> name l_paren underline_start name underline_end comma
star name comma ATTRIBUTES r_paren START
=> name l_paren underline_start name underline_end comma
star name comma name ATTR_SEP r_paren START
=> name l_paren underline_start name underline_end comma
star name comma name r_paren START
=> name l_paren underline_start name underline_end comma
star name comma name r_paren
```

Z množiny přechodových pravidel gramatiky plyne, že není podporován zápis referenčních integrit (cizích klíčů) s prázdnými atributy. Tedy například následující vstup není validní: *oprava[]* $\subseteq$ *mechanik[]*. Případ, kdy cílová nebo zdrojová entita neobsahuje atributy, je vyhodnocen jako chybný. Rovněž chybný je zápis, který u zdrojové entity očekává (avšak neobsahuje) atributy a u cílové entity nikoliv (nebo naopak). Příklad: *oprava* $\subseteq$ *mechanik[]*. Případ, kdy zdrojová entita obsahuje jeden či více atributů a cílová entita žádný neobsahuje ani neočekává (např. *oprava* $\subseteq$ *mechanik[id_mechanik]*) je sice teoreticky validní, ale pro účely tento práce byl označen jako chybný. Zápis, ve kterém cílová entita dědí všechny atributy z entity zdrojové (např. *oprava* $\subseteq$ *mechanik*), je brán jako validní, ačkoliv z praktického hlediska příliš

smysluplný není.

Z gramatiky G je rovněž patrné, že primární atributy nemohou končit oddělovačem (čárkou). V překladu do zjednodušeného relačního zápisu to znamená, že čárka za primárními atributy již nemůže být podtržena. Pokud podtržena bude, pak bude gramatika očekávat další primární atribut. Naproti tomu gramatika povoluje, aby za posledním atributem v entitě či referenční integritě byla čárka. Tento případ je zaveden především kvůli našeptávači, který za doplněné slovo vkládá automaticky atributový oddělovač. Lze si také všimnout toho, že v průběhu zpracovávání atributů entity je možné vícekrát zpracovávat sekvenci primárních klíčů, což znamená, že entita může obsahovat několik skupin primárních atributů (např. *oprava(datum, id_mechanik, poznamka, id_auto)*). Ve výsledném zpracování jsou pak všechny tyto skupiny primárních atributů sjednoceny do jedné. O rozlišování těchto skupin byla vedena diskuze s vedoucím práce, avšak nakonec se od této možnosti upustilo.

Popis terminálních symbolů říká, že komentáře (poznámky) jsou detekovány pro programátory dobře známými znaky:

- jednořádkový komentář musí začínat dvěma lomítky – // – a ukončuje ho znak konce řádku,
- víceřádkový komentář začíná dvojicí znaků /* a končí */.

3.3 LL(1) gramatika

Výše zmíněná bezkontextová gramatika G sice dokáže zpracovat jakýkoliv validní vstup, nicméně, jak již bylo již naznačeno v sekci **Definice**, nemusí být obecně jednoznačná (deterministická). K ověření toho, zda je gramatika G skutečně LL(1) gramatikou je třeba aplikovat funkce $First()$ a $Follow()$ na přechodová pravidla gramatiky a následně porovnat, zda výsledky odpovídají definici LL(1) gramatiky. Pro snadnější výpočet těchto funkcí a koneckonců i pro následné použití v programovacím zápisu je gramatika G upravena do tvaru $A \to \alpha B \mid \alpha \mid \epsilon$, kde $A, B \in N$ a $\alpha \in T$. Tedy tak, že každé přechodové pravidlo obsahuje buď pouze terminál (případně ϵ), nebo začíná terminálem a následuje jediný neterminál.

```
P2 = {
 START := name NAME_HELPER
 | line_comment_start LINE_COMMENT
 | comment_start COMMENT,

 NAME_HELPER := l_paren ENTITY
 | l_bracket REL_ATTR_LEFT_FIRST
 | l_depend REL_NAME_RIGHT_NO_ATTR
 | r_depend REL_NAME_RIGHT_NO_ATTR,
```

```
ENTITY := underline_start PRIMARY_ATTR_FIRST
| star ATTR_NAME
| at ATTR_NAME
| name ATTR_SEP
| r_paren START
| r_paren,

PRIMARY_ATTR := name PRIMARY_ATTR_SEP,

PRIMARY_ATTR_SEP := comma PRIMARY_ATTR
| underline_end ATTR_SEPARATOR,

ATTR_NAME := name ATTR_SEP

ATTR_SEP := comma ENTITY
| r_paren START
| r_paren,

REL_ATTR_LEFT_FIRST := name REL_ATTR_SEP_LEFT

REL_ATTR_SEP_LEFT := comma REL_ATTR_LEFT
| r_bracket OPERATOR,

REL_ATTR_LEFT := name REL_ATTR_SEP_LEFT
| r_bracket OPERATOR,

OPERATOR := depend_left REL_NAME_RIGHT
| depend_right REL_NAME_RIGHT,

REL_NAME_RIGHT := name REL_START_RIGHT,

REL_START_RIGHT := l_bracket REL_ATTR_RIGHT_FIRST,

REL_ATTR_RIGHT_FIRST := name REL_ATTR_SEP_RIGHT

REL_ATTR_SEP_RIGHT := comma REL_ATTR_RIGHT
| r_bracket START
| r_bracket,

REL_ATTR_RIGHT := name REL_ATTR_SEP_RIGHT
| r_bracket START
| r_bracket,
```

```
 REL_NAME_RIGHT_NO_ATTR := name START
 | name,

 LINE_COMMENT := text LINE_COMMENT_END,

 LINE_COMMENT_END := endline START
 | endline

 COMMENT_START := text COMMENT_END_NONTERMINAL,

 COMMENT_END_NONTERMINAL := comment_end START
 | comment_end
}
```

Pochopitelně se změnila i množina neterminálních symbolů. Nová množina bude označována jako $N2$:

```
N2 = {
 START, NAME_HELPER, ENTITY, PRIMARY_ATTR, PRIMARY_ATTR_SEP,
 ATTR_NAME, ATTR_SEP, REL_ATTR_LEFT_FIRST, REL_ATTR_SEP_LEFT,
 REL_ATTR_LEFT, OPERATOR, REL_NAME_RIGHT, REL_START_RIGHT,
 REL_ATTR_RIGHT_FIRST, REL_ATTR_SEP_RIGHT, REL_ATTR_RIGHT,
 REL_NAME_RIGHT_NO_ATTR, LINE_COMMENT, LINE_COMMENT_END,
 COMMENT_START, COMMENT_END_NONTERMINAL
}
```

Gramatika po úpravě má následující tvar: následovně:

```
G2 = (N2, T, P2, START).
```

Nyní je třeba ověřit, zda je gramatika $G2$ skutečně LL(1) gramatikou. Pro každé přechodové pravidlo je tedy nutné spočítat funkce $First()$ a $Follow()$. Spočítat funkci $First()$ je díky úpravě gramatiky jednoduché a je zřejmé, že přechodová pravidla u všech neterminálů splňují první část definice LL(1) gramatiky. Pro příklad je uveden výpočet této funkce u neterminálu $START$:

```
First(name NAME_HELPER) = {name}
First(line_comment_start LINE_COMMENT) = {line_comment_start}
First(comment_start COMMENT) = {comment_start}
```

Funkci $Follow()$ vlastně nemá smysl počítat, protože se v gramatice $G2$ neobjevuje znak ϵ. Druhou částí definice LL(1) gramatiky tedy není nutné se zabývat a gramatika $G2$ je typu LL(1).

3.4 Vývoj a nasazení

Aplikace bude vyvíjena vodopádovou metodikou [8]. A to především z toho důvodu, že zadání této práce bylo jasně specifikované a nejsou tedy očekávány žádné zásadní změny v průběhu vývoje. Při vyvíjení vodopádovou metodikou je totiž nezbytně nutné již na počátku vývoje přesně chápat, co má být výsledkem práce. To je přesně případ této práce (i proto, že zjednodušený relační zápis má jasnou syntaxi).

Pro zálohování a průběžné verzování aplikace bude používán velmi rozšířený verzovací nástroj Git [9]. Na serveru `gitlab.fit.cvut.cz` fakulta poskytuje studentům a zaměstnancům Git repozitář GitLab [10].

Vývoj nového Tralexu bude probíhat v programovacím jazyku Kotlin [11]. Kotlin je relativně mladý programovací jazyk vyvíjený softwarovou firmou JetBrains. Je kompatibilní s velice rozšířeným jazykem Java [12] a dalo by se říci, že je i jeho nástupcem. Společnost Google také umožňuje využít Kotlin k vývoji aplikací pro mobilní platformu Android. Pro účely této práce je Kotlin užitečný, protože umožňuje zkompilovat program do JavaScriptu [13], což je programovací jazyk hojně využívaný ve webových aplikacích. Vývoj textového editoru bude probíhat právě v JavaScriptu, takže propojení Tralexu a textového editoru bude bezproblémové.

Aplikace bude nasazena na portál `dbs.fit.cvut.cz` kurzu BI-DBS. Portál je naprogramován v jazyku PHP [14], konkrétně je využíván český framework Nette [15]. Výsledné nasazení na portál by mělo rovněž proběhnout bezproblémově, protože kombinace jazyků PHP a JavaScript je ve webových aplikacích velmi běžná.

Aby bylo možné dobře přenášet data mezi portálem a Tralexem, bude využíván textový formát JSON [16], který umožňuje převést prakticky jakýkoliv objekt do textové podoby. JSON řetězec se vždy nachází v bloku mezi znaky { a } (tyto znaky označují JSON objekt). Uvnitř tohoto bloku (JSON objektu) se pak vždy využívá dvojice *klíč hodnota*, kde klíč je název proměnné (v uvozovkách) a hodnota může být primitivní datový typ (řetězec, číslo, boolean hodnota), další JSON objekt nebo pole (primitivních datových typů nebo JSON objektů).

```
{
   "entities": [
      {
         "name": "auto",
         "attributes": [
            {
               "name": "id",
               "primary": true,
               "unique": false,
               "optional": false
            }
         ]
      }
   ]
}
```

Ukázky kódu 3.1: Ukázka JSON formátu

Kapitola 4

Implementace

Tato kapitola shrnuje implementaci nového Tralexu a textového editoru. Rovněž je zde ve stručnosti popsána implementace tzv. *Builderu*, který dokáže data ve formátu JSON převést zpět na zjednodušený relační zápis. Všechny zdrojové kódy jsou uloženy přímo v projektu portálu BI-DBS na fakultním GitLabu (konkrétně zde: `https://gitlab.fit.cvut.cz/dbs/tralexnew`).

4.1 Model

Prvním krokem v implementaci bylo samozřejmě sestavení modelových tříd, tedy tříd, které ponesou veškerá data vytěžená ze zjednodušeného relačního zápisu. Řešení má v zásadě dva modely – starý a nový. Starý model byl jednoduchý a splňoval přesně zadání, avšak po diskuzi s Bc. Filipem Dolníkem, vedoucím týmu, který vytváří DSM modul, byla aplikace převedena na nový model. Nový model je rozsáhlejší, ale zajišťuje bezproblémovou komunikaci s dalšími částmi DSM pruduktu (jako je například `Kreslítko`).

Moderní jazyk Kotlin nabízí kromě běžných tříd známých například z Javy také tzv. datové třídy. Datová třída umožňuje ve velmi úsporném zápisu definovat třídu, jejíž primárním účelem je držet data. V ukázce níže je zobrazeno srovnání datové třídy v Javě a Kotlinu. Třída zapsaná v Javě navíc neobsahuje metody *equals*() a *hashCode*(). Za všimnutí také stojí to, že v proměnná *id* je v obou třídách tzv. immutable, tedy že je není možné ji po inicializaci modifikovat. V Javě to znamená, že tato proměnná není veřejná a třída neobsahuje setter, přes který by tuto proměnnou bylo možné změnit. V Kotlinu se tato vlastnost zapisuje klíčovým slovem *val*. Naopak mutable proměnná se pak označuje slovem *var* a k proměnným se přistupuje přímo přes jejich názvy.

```
// Java class
public class JavaEntity {
   private int id;
   private String name;

   public JavaEntity(int id, String name) {
      this.id = id;
      this.name = name;
   }

   public int getId() {
      return this.id;
   }

   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      return this.name;
   }
}

// Kotlin class
data class KotlinEntity(val id: int, var name: String)
```

Ukázky kódu 4.1: Porovnání data class v Javě a Kotlinu

4.1.1 Starý model

Starý model obsahuje čtyři základní třídy: *Attribute, Note, Entity, Relation*. Data class *Attribute* drží informaci o jménu daného atributu. Třída *Note* pro změnu obsahuje jedinou proměnnou s *message* – zde jsou uchovávány komentáře, které byly nalezeny ve vstupu a případně také chyby při překladu. *Entity* je třída, do které jsou ukládány informace o entitě. Tedy její název a seznamy primárních, povinných, unikátních a volitelných atributů. A konečně třída *Relation*, která obsahuje instance zdrojové a cílové entity. Tyto dvě proměnné mohou nabývat i null hodnot, protože může přijít i vstup, který obsahuje v referenční integritě entitu, která nebyla řádně definována. Dále jsou tu samozřejmě seznamy atributů zdrojové a cílové entity a metoda *swapSides*(), která je používána v případě, že je referenční integrita na vstupu zadána tak, že nejdříve je čtena entita zdrojová (symbol $\supseteq$). Výše zmíněné třídy samozřejmě obsahují také metody *serialize*(), respektive *deserialize*(), které slouží k exportu dat do JSON formátu, respektive k importu z toho formátu. Na ukázce

níže je metoda *serialize*() třídy *Entity*. Volaná metoda *Helper.arrayToJson*() přijímá parametrem pole objektů, v tomto případě objektů typu *Attribute*. Následně se na každý objekt volá metoda *serialize*() a metoda vrací řetězec obsahující JSON pole serializovaných objektů.

```
fun serialize(): String {
  var r = "{"
  r += "\"name\": \"$name\","
  r += "\"primary\": ${Helper.arrayToJson(primary)},"
  r += "\"required\": ${Helper.arrayToJson(required)},"
  r += "\"unique\": ${Helper.arrayToJson(unique)},"
  r += "\"secondary\": ${Helper.arrayToJson(secondary)}"
  r += "}"
  return r
}
```

Ukázky kódu 4.2: Serializace třídy Entity ze starého modelu

4.1.2 Nový model

Jak již bylo zmíněno, hlavní rozdíl mezi starým a novým modelem je ten, že nový model je robustnější. Z toho důvodu jsou popsány jen základní odlišnosti – datové třídy a případné atributy, které tato práce nevyužívá, zmíněny nejsou. Nový model spravují vývojáři z DSM týmu. Serializace a deserializace je řešena zcela odděleně od datových tříd. Serializace vychází z rozhraní *ExportService*, které obsahuje jedinou metodu *export*(). Podobně deserializace vychází z rozhraní *ImportService* a obsahuje jedinou metodu *import*(). Základním stavebním kamenem nového modelu je třída *Canvas*, která obsahuje seznamy entit, relací a poznámek. Změna, nutno říct, že k lepšímu, nastává v ukládání atributů nějaké entity. V novém modelu je totiž informace o tom, zda je atribut primární, povinný, unikátní či volitelný uložena přímo ve třídě *Attribute*. Seznam atributů je pak uložen v instanci třídy *Entity*. Hlavní třídy *Attribute*, *Entity*, *Note* a *Relation* obsahují jako instanční proměnné také instance datových tříd. Typickým příkladem může být název entity či atributu. Proměnná *name* ve třídě *Attribute* očekává instanci datové třídy *AttributeName*, která v konstruktoru očekává řetězec s názvem atributu.

Při nasazování nového modelu byly také nalezeny dva zásadní problémy, které vyžadovaly jeho úpravu. První problém byl ten, že třída *RelationEnd*, která nese informaci o zdrojové či cílové entitě referenční integrity, vůbec neudržovala informaci o relačních atributech. Třída *Entity* sice obsahuje seznam instancí třídy *RelationEnd*, což ale není dostačující. Problematické to bylo i z toho důvodu, že se vlastně nedala uložit informace o chybném zápisu referenční integrity. Například vstup *oprava[id_mechanik, datum]* $\subseteq$ *mechanik[id]*

je sice validní z pohledu gramatiky (student ho může reálně použít), avšak je chybný. Tento problém byl vyřešen přidáním seznamu relačních atributů do třídy *RelationEnd*.

Druhý problém nastal v situaci, kdy byla v referenční integritě použita entita, která ještě nebyla řádně definována. Kupříkladu byla použita referenční integrita napsaná výše, ale dosud nebyla definována entita *oprava*, tedy vstup neobsahoval podřetězec *oprava(…)*. Pokud tato situace nastala, tak pokus o serializaci do formátu JSON skončil pádem aplikace. Proto byla do třídy *Entity* přidána členská proměnná *implicit* s defaultní hodnotou $false$. V případě, že instance entity dosud neexistovala, byla vytvořena a její proměnná *implicit* byla nastavena na hodnotu *true*.

4.1.3 Další třídy

Dalšími modelovými třídami, které nový Tralex obsahuje, jsou výčtové typy (enum třídy) *TerminalSymbol*, *NonTerminalSymbol* a *ErrorType*. Třída *TerminalSymbol* obsahuje, jak již název napovídá, terminální symboly LL(1) gramatiky definované v sekci **3.3**. Podobně také třída *NonTerminalSymbol* obsahuje neterminální symboly definované LL(1) gramatiky. Výčtový typ *ErrorType* definuje všechny možné chyby, které mohou nastat při průchodu LL(1) gramatikou. Například:

```
MISSING_ENTITY_NAME, MISSING_LEFT_PAREN_OR_BRACKET,
MISSING_PRIMARY_ATTRIBUTE_NAME, MISSING_ATTRIBUTE_NAME.
```

Pro úplnost je zmíněna i datová třída *Token*, která drží informaci o tokenech při průchodu zpracovávání vstupu. Konkrétně konstruktor této třídy očekává 4 hodnoty:

- *type* – výčtový typ typu *TerminalSymbol*, případně hodnota *null* (využívána kupříkladu při detekování konce vstupu),
- *value* – hodnota tokenu, pokud je nějaká očekávána, jinak prázdný řetězec (například název entity či atributu),
- *line* – řádek vstupu, na kterém se token nachází (řádky jsou číslovány od 1),
- *position* – pozice na řádku, na které token začíná (číslováno od 0).

4.2 Třída Escaper

Účelem třídy *Escaper* je připravit vstup na průchod LL(1) gramatikou, což především znamená odstranit, případně vhodně upravit HTML tagy, které přijdou na vstupu. Třída má dvě veřejné metody: $containsOnlyHtmlTags()$

a *escape*(). Prvně jmenovaná metoda je spíše pomocná a má za úkol odhalit, zda vstup, který je dán parametrem, obsahuje pouze HTML tagy nebo jestli je dokonce prázdný. Tato metoda je využívána především na portálu `dbs.fit.cvut.cz` a pokud vrátí *true*, tak je student, případně učitel, upozorněn, že zadal prázdný vstup.

```
fun containsOnlyHtmlTags(input: String): Boolean {
  var escapedInput = input
  escapedInput = escapedInput.replace("&lt;", "<")
  escapedInput = escapedInput.replace("&gt;", ">")
  var position = 0
  var inTag = false
  while (position != escapedInput.length) {
    when (escapedInput[position]) {
      '<' -> inTag = true
      '>' -> inTag = false
      else -> {
        if (!inTag) return false
      }
    }
    position++
  }
  return true
}
```

Ukázky kódu 4.3: Metoda containsOnlyHtmlTags() třídy Escaper

Druhá jmenovaná metoda *escape*() je volána vždy před rozparsováním vstupu. Jejím úkolem je odstranit ze vstupu HTML tagy, případně je nahradit či vhodně redukovat. To znamená, že metoda prochází postupně řetězec, který obdržela parametrem. Pokud na nějaké pozici vstupu začíná HTML tag (neboli znak na této pozici je „<“), tak se rozlišuje několik případů:

- tag <div …> – nahrazen znakem „\n“, protože contenteditable div [17], který je využíván v textovém editoru, uzavírá jednotlivé řádky do párového tagu ,
- tag <br …> – nahrazen mezerou, aby se oddělilo poslední slovo na aktuálním řádku od prvního na řádku následujícím,
- tag <u …> – redukován na tag „<u>“ pro případ, že se v tagu nachází nějaký atribut (např. style),
- tag </u …> – redukován na tag „</u>“ ze stejného důvodu jako tag <u>,
- jakýkoliv jiný HTML tag je odstraněn.

4.3 Třída Tokenizer

Třída *Tokenizer* dostane v konstruktoru řetězec, který už byl upraven třídou *Escaper*, a má v zásadě jediný úkol – totiž číst postupně vstup a vracet vždy aktuální token. K tomu slouží veřejná metoda bez parametrů *nextToken*(), která vrací instanci třídy *Token* (případně null, pokud nastane konec řetězce).

Před vracením samotného tokenu je nutné přeskočit znaky, které nejsou tokeny. K tomu slouží metoda *readWhileWhiteSpace*(), která zajišťuje přeskočení „bílých“ znaků, tedy v tomto případě znaků mezery („ “) a nového řádku („\n“). Tato metoda také zajišťuje správné počítání řádků (když nastane znak nového řádku, tak se hodnota proměnné *line* zvýší o 1).

Následně se rozlišují případy podle toho, který znak (případně sekvence znaků) je aktuálně čten. Například pokud je aktuální znak „/“ a následující znak rovněž, pak je jasné, že tato dvojice znaků tvoří terminální symbol `line_comment_start`. Je-li aktuálním terminálním symbolem některý z dvojice `name` nebo `text`, pak je jasné, že je třeba zaznamenat také hodnotu tohoto terminálního symbolu. V ostatních případech to nutné není, protože hodnotu terminálu jednoznačně určuje jeho typ (například u terminálu `l_paren` je zřejmé, že jeho hodnota je „(“).

Pokud je aktuálně zpracovávaný terminál typu `name`, pak se také ověřuje, zda je hodnota terminálu validní. Každý název entity či atributu musí splňovat následující:

- začíná písmenem,
- v těle obsahuje písmeno, číslo, případně spojovník („-“) či podtržítko („_“),
- končí buď písmenem nebo číslem.

Tuto kontrolu demonstruje ukázka metody *isNameAllowedChar*() (že je první znak písmeno, je ověřováno ještě před vstupem do této metody).

```
private fun isNameAllowedChar(): Boolean {
  if (isLetter(input[position])
      || isNumber(input[position]))
    return true
  if ((input[position] == '-'|| input[position] == '_')
      && position + 1 < length &&
      (isLetter(input[position + 1])
       || isNumber(input[position + 1]))) {
      return true
    }
  return false
}
```

Ukázky kódu 4.4: Metoda isNameAllowedChar() třídy Tokenizer

4.4 Třída Grammar

Tato třída vlastně reprezentuje LL(1) gramatiku. Instance této třídy si v proměnné *currentState* drží aktuální neterminální symbol (nějaká hodnota výčtového typu *NonTerminalSymbol*). Výchozí hodnota této proměnné je pochopitelně neterminál `START`. Třída *Grammar* má jedinou metodu *next*(), která jako argument očekává výčtový typ *TerminalSymbol*. Naopak vrací výčtový typ *ErrorType*, případně null, pokud nenastala chyba. Při zavolání této metody se v závislosti na aktuální hodnotě v proměnné v *currentState* rozhodne, které terminály jsou validní. Neboli zda v neterminálu *currentState* existuje přechodové pravidlo začínající terminálem z parametru. Pokud je terminál předaný parametrem validní, pak je *currentState* upravena na následující neterminál a funkce vrací null. V opačném případě zůstává proměnná *currentState* nezměněna a vrací se příslušná hodnota výčtového typu *ErrorType*.

```
when (currentState) {
  ...
  NonTerminalSymbol.START -> {
    nextState = when (terminal) {
      TerminalSymbol.NAME ->
        NonTerminalSymbol.NAME_HELPER
      TerminalSymbol.LINE_COMMENT_START ->
        NonTerminalSymbol.LINE_COMMENT
      TerminalSymbol.COMMENT_START ->
        NonTerminalSymbol.COMMENT_START
      else -> {
        errorType = ErrorType.MISSING_ENTITY_NAME
        null
      }
    }
  }
  ...
}
```

Ukázky kódu 4.5: Ukázka části metody next() třídy Grammar

4.5 Třída Parser

Třída *Parser* je hlavní třídou celé aplikace. Kromě toho, že obsahuje instance výše zmíněných tříd *Escaper*, *Tokenizer* a *Grammar*, implementuje také klíčové metody *parse*() a *checkInput*(). Instance této třídy vytvářena „zvenku“, podobně tak jsou volány i její metody. To znamená, že tato třída je využívána v JavaScriptu (mimo jiné v textovém editoru). Viz sekce **Kompilace do JavaScriptu**.

4.5.1 Metoda *parse()*

Metoda *parse*() má jediný úkol – zpracovat a exportovat (serializovat) vstup, tedy zjednodušený relační zápis, do JSON formátu. To znamená, že tato metoda prochází vstup – postupně odebírá tokeny prostřednictvím metody *nextToken*() třídy *Tokenizer* – a pomocí instance třídy *Grammar* kontroluje, zda pro aktuální token existuje přechodové pravidlo (neboli zda je syntaxe vstupu v pořádku). Pokud metoda *next*() třídy *Grammar* nevrátí null (tedy došlo k chybě při překladu), tak je aktuální neterminál této třídy (uložen v proměnné *currentState*) změněn na počáteční neterminál `START`, aby byla gramatika připravena na další průchod. Chyba syntaxe je totiž při zpracovávání vstupu přeskočena a pokračuje se od nejbližšího možného terminálu. To znamená, že pokud dojde k chybě v přechodových pravidlech neterminálu `START` či neterminálu `NAME_HELPER`, třída *Tokenizer* odebírá neterminály dokud nenastane konec řádku. Jakmile konec řádku nastane, pokračuje se ve zpracovávání vstupu a je vytvořena poznámka (instance třídy *Note*) o tom, že na tomto místě došlo k chybě překladu. Tato poznámka je pak pochopitelně přidána do výsledného JSONu. V případě, že nastane chyba při zpracovávání entitních atributů, je zbytek entitních atributů přeskočen (tedy jsou odebírány terminály dokud nenastane konec entity – terminál `r_paren`). U relace je to velmi podobné s tím rozdílem, že pokud nastane chyba, tak jsou odebírány tokeny dokud nenastane druhý terminál `r_bracket`.

Zpracování vstupu je v zásadě rozděleno na tři podproblémy: zpracování entity, zpracování relace a zpracování poznámky (komentáře). Zpracování poznámky je nejjednodušší podproblém. Třída *Tokenizer* totiž rozlišuje terminály `name` a `text`. V případě poznámky tedy stačí vzít hodnotu tokenu `text` a uložit ji (informace jsou brány z instance třídy *Token*, kterou vrací metoda *nextToken*() třídy *Tokenizer*).

Vráti-li *Tokenizer* terminál `name`, rozhodne se podle následujícího terminálu, který ze zbylých dvou podproblémů nastal. Pokud je následující token typu `l_paren`, tak je jasné, že je třeba rozparsovat entitu. Naopak je-li následující token typu `l_bracket`, `depend_left` či `depend_right`, nastává problém zpracování relace. Výhodou při řešení těchto podproblémů je právě dobře navržená gramatika, která vše usnadňuje. Když při parsování entity přijde znak povinného respektive unikátního atributu (terminály `star` a `at`), změní se hodnota příznaku *required* respektive *unique*. Podle hodnot těchto příznaků se pak posílají hodnoty do konstruktoru třídy *Attribute*. U primárních atributů žádný příznak není potřeba, protože informaci o tom, že je zpracováván primární atribut, nese přímo gramatika (neterminál `PRIMARY_ATTR`).

Při parsování relace je nutné uložit cílovou a zdrojovou entitu ve správném pořadí. Výchozí způsob ukládání je v pořadí cílová entita – zdrojová entita (znak $\subseteq$, terminál `depend_left`). V případě, že *Tokenizer* vrátí terminál `depend_right`, je třeba prohodit cílovou a zdrojovou entitu a jejich případné atributy.

Jakmile nastane konec vstupu, je vytvořena instance třídy *Canvas*, která v konstruktoru očekává seznam entit, relací a poznámek. Tato instance je následně vyexportována do JSONu a metoda vrátí tento vyexportovaný řetězec.

4.5.2 Metoda *checkInput()*

Primárním účelem této metody je realtime (okamžitá) kontrola syntaxe vstupního řetězce. Tedy tato metoda, na rozdíl od metody *parse*(), vstup vůbec nezpracovává, ale pouze prochází gramatikou, která je reprezentována třídou *Grammar*. Pokud nastane situace, kdy metoda *nextToken*() vrátí null, tak se podle aktuálního neterminálu uloženého v proměnné *currentState* instance třídy *Grammar* vrátí chybová hláška. Chybová hláška obsahuje také řádek a pozici na řádku, kde k chybě došlo. V případě, kdy metoda *nextToken*() vrátí null (tedy nastane konec vstupu), jsou jako číslo řádku a pozice brány poslední hodnoty z třídy *Tokenizer*. V opačném případě se volá funkce *next*() na instanci třídy *Grammar*. Pokud funkce vrátí null, tak je vše v pořádku a pokračuje se v průchodu gramatikou. Jinak se vrácena chybová hláška (nějaká hodnota výčtového typu *ErrorType*).

Kromě syntaxe tato metoda kontroluje také to, zda je jméno entity či atributu (terminál typu `name`) ve slovníku očekávaných hodnot. Viz **následující sekce**. Příklady chybových hlášek, které vrací metoda *checkInput*():

```
Missing '(' or '[' - line: 2 column: 7,
Missing ',' - line: 2 column: 16,
Unknown word: testovaci_slovo - line: 2 column: 17.
```

4.5.3 Další metody a funkcionality

Třída *Parser* očekává v konstruktoru řetězec klíčových slov ve formátu JSON. Konkrétně pole možných názvů entit pod klíčem „entities“ a pole možných názvů atributů pod klíčem „attributes“. Tento JSON je při inicializaci instance rozparsován a uložen do datové třídy *AutocompleteWords*, která obsahuje jako proměnné dva unikátní seznamy řetězců – *entities* a *attributes*. Tyto seznamy jsou využívány k našeptávání v textovém editoru, proto pro ně existují gettery ve třídě *Parser*. Jak již bylo popsáno výše, metoda *checkInput*() tyto seznamy klíčových slov prochází a pokud se nějaké slovo v seznamu nenachází, je na to student (případně učitel) upozorněn.

4.6 Třída Builder

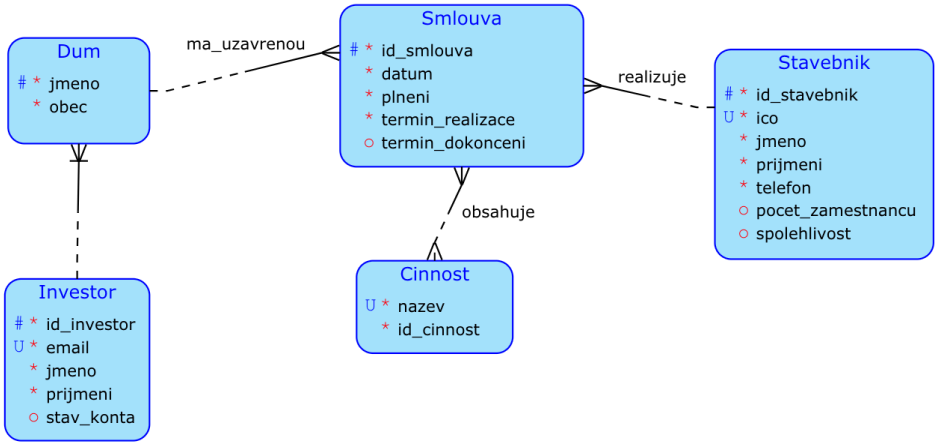
Tato třída je podobně jako třída *Parser* veřejná, což znamená, že je volána přímo z JavaScriptu. Má dvě veřejné metody: *buildKeywordsJson*() a *buildHtmlFromJson*().

Prvně jmenovaná metoda dělá to, že převede JSON, který vrací `Kreslítko`, na JSON, který přijímá třída *Parser*. Klíčová slova entit a atributů, které kontroluje *Parser* a které našeptává textový editor, totiž pocházejí přímo ze zadaní otázky typu transformace v portálu `dbs.fit.cvut.cz`. A právě otázky typu transformace mají zadání téměř výhradně tvořeno diagramem, který je vytvořen prostřednictvím `Kreslítka`. V případě, že tomu tak není, tedy JSON v parametru je prázdný nebo nemá očekávaný formát, je vrácena výjimka. Tato metoda je tedy volána v try-catch bloku. Pokud je výjimka odchycena, tak je do konstruktoru třídy *Parser* posílán prázdný řetězec. Prázdný řetězec označuje, že se nepodařilo načíst klíčová slova. Nebudou tedy kontrolována v metodě *checkInput*() a nebudou ani našeptávána v textovém editoru. Do seznamu klíčových slov entit se nepřidávají pouze jejich názvy, ale rovněž také popisky referenčních integrit, které jsou pro `Kreslítko` typické. Například entitní klíčová slova z obrázku níže budou mimo jiné obsahovat i `ma_uzavrenou`, `realizuje` a `obsahuje`. Rozšířeny jsou i klíčová slova atributů. Jelikož primární atributy obvykle začínají prefixem „id_“, mohl by našeptávač studentům neúmyslně radit, které atributy podtrhnout. Proto jsou do klíčových slov atributů přidány všechny atributy s tímto prefixem i bez něj.

Metoda *buildHtmlFromJson*() přijímá parametrem JSON řetězec, který vrací metoda *parse*() třídy *Parser*. A jejím úkolem je vytvořit a vrátit HTML řetězec, který zobrazí strukturu zjednodušeného relačního zápisu. Dalo by se říci, že tato metoda dělá přesně opačný proces v porovnání se zmiňovanou metodou *parse*(). Výstup metody *buildHtmlFromJson*() je jistým způsobem standardizován, což znamená, že bude vypadat vždy stejně – pořadí entit a jejich atributů, relací a jejich atributů a komentářů bude vždy při zavolání této metody na stejný vstup stejné. Tato vlastnost je zmíněna především proto, že pořadí objektů a jejich elementů ve formátu JSON není obecně definováno (pořadí JSON polí ano) [18]. Pořadí v generovaném HTML řetězci je následující (všechny skupiny jsou řazeny abecedně s výjimkou atributů v relaci):

1. entity,
    a) primární atributy,
    b) povinné atributy,
    c) unikátní atributy,
    d) volitelné atributy,
2. relace (atributy relací nejsou řazeny, protože záleží na jejich pořadí),
3. poznámky.

Jednotlivé řádky výstupu jsou oddělovány HTML tagem
 a celý HTML řetězec je uzavřen do tagu . Následující příklad demonstruje, jak vypadá zjednodušený relační zápis po zpracování metodou *parse*() třídy



Obrázek 4.1: Příklad zadaní vytvořeného Kreslítkem

Parser a následném vygenerování metodou *buildHtmlFromJson*() třídy *Builder* (HTML tagy jsou pro zachování přehlednosti vynechány):

```
Vstup:
  oprava(vin, datum_opravy)
  // komentar
  auto(pohon, *vin, @spz)
  oprava[vin] ⊆ auto[vin]
Výstup:
  auto(*vin, @spz, pohon)
  oprava(datum_opravy, vin)
  oprava[vin] ⊆ auto[vin]
  // komentar
```

4.7 Kompilace do JavaScriptu

Jak již bylo zmíněno v sekci **3.4**, je možné zkompilovat kód napsaný v Kotlinu do JavaScriptu. V tomto projektu je k tomu využíván buildovací nástroj Gradle [19], který tuto kompilaci automatizuje. Buildovací skript je pak napsán v jazyku Apache Groovy [20], který je hojně používán pro psaní buildovacích skriptů v jazyce Java. Kompilace vygeneruje několik souborů, ze kterých jsou dva užitečné pro účely této práce. V jednom souboru je samotný jazyk Kotlin vygenerovaný do své javascriptové podoby a ve druhém jsou v zdrojové kódy aplikace (rovněž v javascriptovém formátu). Aby bylo možné vytvořit instance tříd v JavaScriptu, je nutné tyto dva vygenerované soubory vložit do

HTML kódu stránky, kde budou třídy používány. Ke třídám se vcelku logicky přistupuje přes jmenné prostory (namespace).

Aby měly vygenerované metody univerzální názvy (název metody vygenerované do JavaScriptu typicky končí číslovkou), je v této práci využívána anotace `@JsName`, kterou Kotlin nabízí. Tuto anotaci stačí přidat nad název jakékoliv veřejné metody a do závorky pak přidat řetězec s názvem, který má být dostupný v JavaScriptu. Například u metody *parse*() třídy *Parser* je tato anotace: `@JsName("parseInput")`. Následující ukázka předvádí vytvoření instance třídy *Parser* a volání její metody *parse*() v JavaScriptu:

```
<script src="kotlin.js"></script>
<script src="new_tralex.js"></script>
<script>
  let parser = new cz.cvut.fit.dsm.tralex.Parser("");
  parser.parseInput("");
</script>
```

Ukázky kódu 4.6: Ukázka použití třídy Parser v JavaScriptu

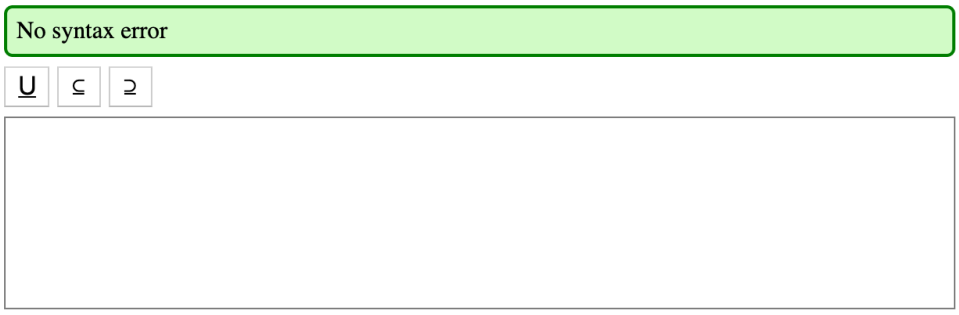
4.8 Textový editor

Textový editor je klíčovou součástí nového řešení aplikace Tralex. V zadání této práce jsou poměrně jasně specifikovány funkcionality, které jsou od něj očekávány – konkrétně automatické našeptávání a snadnější zadávání. Výsledná aplikace bude fungovat na vzájemném propojení třídy *Parser* a právě textového editoru, který je napsán v JavaScriptu. Editor je tvořen HTML divem, který má vlastnost contenteditable [17], tedy, že je možné do tohoto divu psát. Využití párového tagu <textarea> bohužel nepřipadalo v úvahu a to navzdory tomu, že práce s textem v tomto tagu je o mnoho jednodušší než v tagu . Hlavním důvodem, proč textareu nelze použít, je, že tento tag podporuje pouze prostý (plain) text. To znamená, že by v editoru nebylo možné podtrhnout primární atributy – což je ovšem nedílná součást syntaxe zjednodušeného relačního zápisu.

V editoru jsou hojně používané tzv. event listenery [21], které umožňují detekovat a reagovat na nějakou interakci od uživatele. Například event listener typu „click“ detekuje kliknutí uživatele na nějaký HTML element. Event listener „keyup“ detekuje stisknutí klávesy v contenteditable divu, tedy primárně buď změnu obsahu nebo posun kurzoru.

Textový editor je v zásadě rozdělen na následující tři části:

- oznamovací panel – zde se zobrazují zprávy o chybách syntaxe, případně upozornění o použití neznámých slov,



Obrázek 4.2: Ukázka textového editoru

- tlačítka – konkrétně tlačítko na podtržení označeného textu a tlačítka na vložení symbolu $\subseteq$ respektive $\supseteq$,
- našeptávač, který je po spuštění aplikace skryt.

4.8.1 Oznamovací panel

Oznamovací panel reaguje na změnu obsahu v contenteditable divu. Po detekování této události je volána metoda *checkInput*() třídy *Parser* a je zde zobrazen její výstup. To znamená, že se zde zobrazují tři typy informací. Uživatelský vstup je buď v pořádku, pak má panel na pozadí zelenou barvu. Druhou možností je, že byl zadán vstup s chybnou syntaxí. V tomto případě se zobrazí informace o tom, jaký token je na které pozici očekáván s červenou barvu na pozadí. Třetí možností je varování (oranžová barva), že vstup obsahuje jedno nebo více slov neznámých slov – neznámých v tom smyslu, že nejsou v seznamu klíčových slov. Pak se zobrazí informace o všech těchto slovech – která to jsou a kde se nacházejí.

4.8.2 Tlačítka

Jak již bylo zmíněno, tlačítka jsou v editoru tři a kliknutí na ně je logicky detekováno pomocí event listeneru typu „click“. To, že tlačítka vykonají své úkoly (tedy podtržení textu a vložení symbolu), zajišťuje chytrá javascriptová funkce *execCommand*() [22]. Ta podle pozice kurzoru vykoná zadanou operaci (např. *document.execCommand('underline')*). Po stisknutí jakékoliv tlačítka je pomocí funkce *dispatchEvent*() [23] vyvolána událost, která detekuje stisk klávesy v textovém editoru (z toho důvodu, že byl upraven vstup a je tedy třeba zkontrolovat jeho syntaxi pomocí metody *checkInput*() třídy *Parser*).

4.8.3 Našeptávač

Našeptávač umožňuje snadnější, respektive pohodlnější zadávání zjednodušeného relačního zápisu a to jak pro studenty, tak pro učitele. Seznamy klíčových slov jsou dodávány ze třídy *Parser*. Tyto seznamy jsou dva – názvy entit a názvy atributů.

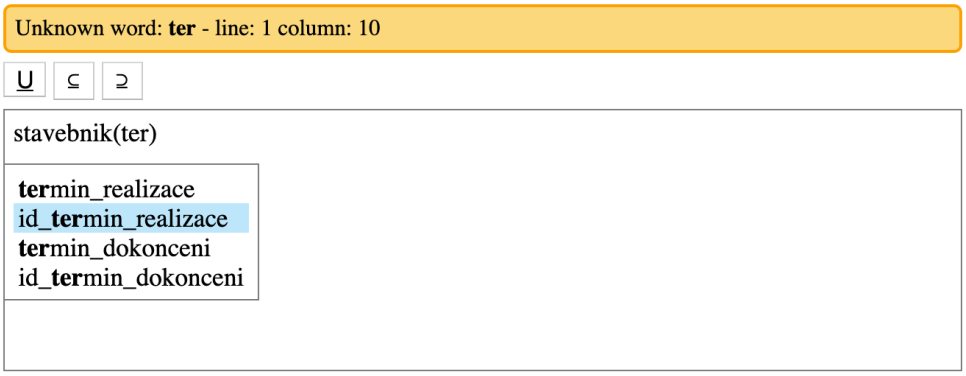
Textový editor, respektive jeho event listener reaguje na stisknutí klávesy. Uvnitř tohoto listeneru je, mimo jiné, volána funkce $getCurrentWordInfo()$, která očekává dva parametry – řetězec z textového editoru a aktuální pozici kurzoru. Vrací pak strukturu, která obsahuje informace o slově na pozici kurzoru. Konkrétně:

- `word` – samotné slovo,
- `in_attributes` – přepínač, který určuje, zda se mají aktuálně našeptávat entity nebo atributy,
- `in_relation_attributes` – přepínač, který detekuje, zda je slovo atributem relace či nikoliv (význam toho přepínače je vysvětlen níže),
- `in_underline_tag` – tato boolean hodnota nese informaci o tom, zda se slovo nachází v párovém HTML tagu <underline>, pokud ano, tlačítko na podtržení textu se zvýrazní,
- `caret_position` – pozice kurzoru v tomto slově.

Po zjištění informací o aktuálním slově jsou nalezena slova k našeptávání (v závislosti na hodnotě přepínače `in_attributes` jsou vyhledávána buď v seznamu entit, nebo v seznamu atributů). Tyto slova musejí obsahovat jako podřetězec aktuální slovo. Například je-li aktuální slovo „mech“, tak našeptávána mohou být tato slova (za předpokladu, že jsou v seznamu klíčových slov): „id_mechanik“ nebo „automechanik“.

Je-li seznam slov k našeptávání neprázdný, našeptávač se zobrazí. Jeho ovládání je velmi intuitivní – mezi slovy je možné přepínat šipkami nahoru a dolů, slovo se vloží do textového editoru a příslušnou pozicí po té, co na něj uživatel buď klikne nebo stihne klávesu Enter. Po kliknutí mimo našeptávač, případně po stisknutí klávesy Escape se našeptávač zavře. K detekování stisku klávesy je opět použit event listener. Pokud je našeptávač zobrazený, tak se pomocí funkce $preventDefault()$ [24] potlačí událost, která by nastala za běžných okolností. Místo toho se, v závislosti na klávese, která byla stisknuta, provede událost s našeptávačem. Například v případě stisku klávesy Escape je výchozí událostí to, že z textového editoru zmizí kurzor. Nebo pokud by uživatel pomocí šipek nahoru a dolu vybíral slovo k našeptání, tak by se bez použití funkce $preventDefault()$ mohla měnit i pozice kurzoru v editoru.

Samotné vložení slova do textového editoru je řešeno tak, že se pomocí aktuálního slova na pozici kurzoru a slova, které bylo vybráno k vložení, vypočtou délky prefixu a suffixu. Tedy kolik písmen chybí před a za aktuálním



Obrázek 4.3: Ukázka našeptávače

slovem. Za použití těchto dvou hodnot, vybraného slova a pozice kurzoru v aktuálním slově funkce $insertTextAtCursorPosition()$ patřičně upraví aktuální slovo v textovém editoru na zvolené slovo. Je-li navíc vkládané slovo atribut, automaticky se za něj vloží čárka a mezera tak, aby bylo možné hned psát další atribut (gramatika poslední čárku nediskutuje). Po diskuzi s vedoucím práce bylo doplňování čárky a mezery zrušeno v atributech relace z toho důvodu, že víceatributová relace není v kurzu BI-DBS tak běžným jevem. Proto byl zaveden výše zmíněný přepínač `in_relation_attributes`.

Kapitola 5

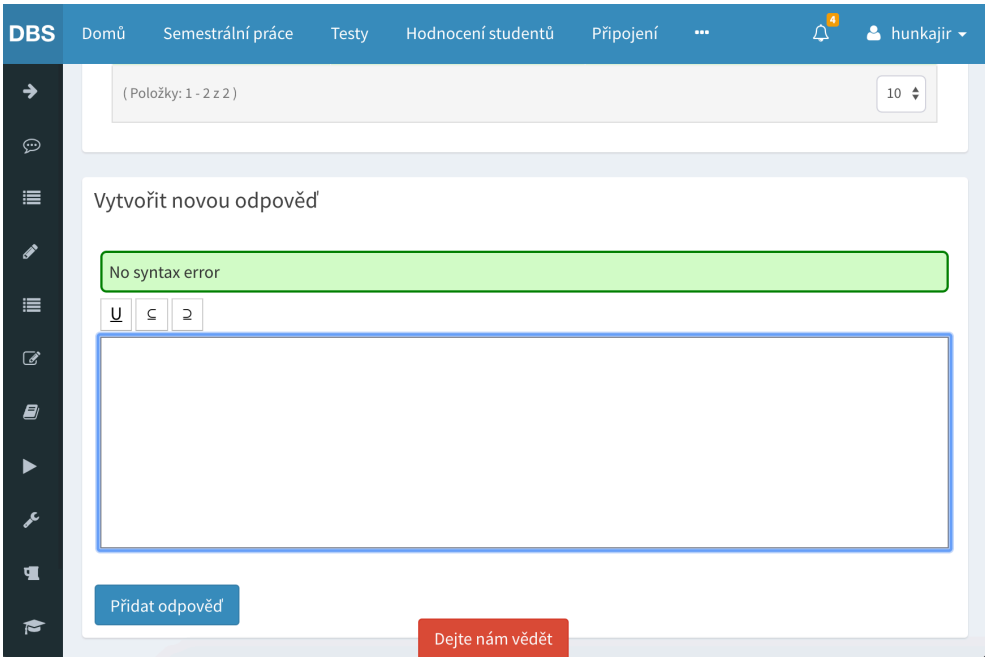
Nasazení na portál dbs.fit.cvut.cz

V této kapitole je popsáno, jak probíhalo nasazení aplikace na portál kurzu BI-DBS a jsou diskutovány případné problémy s tím spojené.

Jak již bylo zmíněno, portál `dbs.fit.cvut.cz` je napsán v programovacím jazyce PHP, konkrétně je využíván framework Nette. Šablony jsou pak tvořeny nástrojem Latte [25], který je součástí tohoto frameworku. Pro samotný textový editor je vytvořena právě Latte šablona *TralexNewEditor.latte*, do které je vložen potřebný HTML kód (obsahující contenteditable div, tlačítka atd.) a javascriptová knihovna textového editoru. Tato šablona je pak vkládána na místa, kde byl původní Tralex. Tyto místa jsou v zásadě tři: vytvoření a editace odpovědí u otázek typu transformace z pohledu učitele a pak samotný test z pohledu studenta. Na všech těchto místech je potřeba v podstatě udělat to samé. To znamená vložit latte šablonu textového editoru a vytvořit zde potřebné instance tříd z nového Tralexu, které bude používat textový editor. Některé tyto metody budou rovněž používány při ukládání uživatelského vstupu do databáze.

Nově bylo také potřeba do každé latte šablony, která v sobě má vloženou šablonu textového editoru, vložit zadání – přesněji řečeno jeho obsah. Jak totiž bylo zmíněno výše v práci, zadání u otázek typu transformace je téměř výlučně tvořeno diagramem, který je vytvořen pomocí `Kreslítka`. A právě z tohoto diagramu jsou pomocí metody *buildKeywordsJson*() třídy *Builder* „vytěžena“ slova k našeptávání, která jsou pak posílána do konstruktoru třídy *Parser*.

K ukládání dat je využívána funkce $.*ajax*() javascriptové knihovny jQuery [26], protože celý nový Tralex je ve výsledku tvořen JavaScriptem. Před samotným uložením je provedena kontrola, zda je vstup neprázdný a validní. Neprázdnost vstupu je kontrolována metodou *containsOnlyHtmlTags*() třídy *Escaper*. Ta vrací hodnotu *true* pokud je vstup úplně prázdný, případně pokud



Obrázek 5.1: Nový Tralex v BI-DBS portálu

obsahuje pouze HTML tagy (např.
). Učiteli není umožněno uložit prázdnou či nevalidní odpověď. Studentovi ano, ale je na to upozorněn varovnou hláškou. K tomuto bodu se váže také způsob ukládání studentovy odpovědi. Ta je ukládána jako prostý HTML řetězec. Převedena na JSON formát bude až při opravě, kterou provádí učitel.

Při testování nasazené aplikace vedoucím práce byl odhalen problém s našeptáváním v textovém editoru. Respektive se získáním slov k našeptávání ze zadání – tedy z diagramu vytvořeného `Kreslítkem`. Příčinou tohoto problému byl fakt, že testovací otázka v portálu `dbs.fit.cvut.cz` může mít více zadání. Problém byl vyřešen jednoduše tak, že zadání, ze kterého jsou brána slova k našeptávání, musí být tvořeno diagramem. Pokud takové zadání u otázky není, pak je seznam klíčových slov prázdný a našeptávání fungovat nebude.

Aby učitelé mohli opravovat odpovědi studentů, bylo samozřejmě nutné na BI-DBS portál nasadit také *Builder*, který dokáže převést JSON na zjednodušený relační zápis. Při zobrazování studentovy odpovědi jsou podle jejího obsahu rozlišovány následující případy:

- odpověď je prázdná (nebo obsahuje pouze HTML tagy) – *Builder* není využíván, učitel je pouze informován o prázdné odpovědi,
- odpověď je nevalidní – metoda *checkInput*() třídy *Parser* vrátila chybu a odpověď je zobrazena tak, jak ji uložil student (tedy je zobrazen HTML kód z textového editoru uložený v databázi u studentovy odpovědi),

- odpověď je validní – rozparsována na JSON formát metodou *parse*() třídy *Parser* a následně zbuildována metodou *buildHtmlFromJson*() třídy *Builder* a zobrazena.

Kapitola 6

Testování

Tato kapitola popisuje, jak probíhalo testování celé aplikace. Je zde rovněž popsáno testování reálnými uživateli – studenty.

Testování aplikace probíhalo prakticky po celou dobu vývoje. Pravděpodobně nejdůležitější byla správná funkčnost třídy *Grammar*, která rozpoznává validitu vstupu. Nejen tuto třídu, ale i gramatiku, bylo třeba při vývoji několikrát upravit. Textový editor, respektive jeho našeptávač, byl vcelku pochopitelně testován převážně ručně.

K testování před nasazením byla autorovi práce poskytnuta historická data (odpovědi studentů) z portálu `dbs.fit.cvut.cz`. Na těchto datech byly testovány především metody *checkInput*() a *parse*() a třídy *Parser*, které ověřují validitu vstupu, respektive exportují zjednodušený relační zápis do JSON formátu. Dále byla testována metoda *buildHtmlFromJson*() třídy *Builder*, která má za úkol transformovat zjednodušený relační zápis z JSON formátu do čitelného HTML (a to včetně seřazení dat – viz **třída Builder**). Přibližně u stovky vstupů byla provedena ruční kontrola funkčnosti *Builderu* (tedy správné seřazení dat, podtržení primárních atributů atd.). Při testování byly použity pozitivní i negativní testy, které ověřují, jestli funguje co fungovat má a nefunguje co fungovat nemá [27]. Příkladem negativního testu může být pokus o rozparsování zjednodušeného relačního zápisu s neplatným názvem entity nebo atributu, který by měl skončit chybou (viz **třída Tokenizer**).

6.1 Testování po nasazení na portál

Po nasazení nového Tralexu na portál `dbs.fit.cvut.cz`, které je popsáno v **páté kapitole**, bylo nutné všechny funkcionality řádně otestovat, aby (například v průběhu testu psaného studenty) nenastal nějaký neočekávaný problém. Bylo tedy třeba otestovat odpovědi u otázek typu transformace – jejich vytvoření a editace z pohledu učitele a vyplnění z pohledu studenta. Jak již bylo zmíněno, nastal zde problém v situaci, kdy měla otázka přiřazených více za-

dání, avšak byl záhy opraven. Otestovat bylo třeba také zobrazení odpovědí v opravě. Zde nastala chyba v hromadné opravě, kdy byla ve všech odpovědích zobrazena stejná hodnota. Konkrétně byl problém v tom, že v JavaScriptu docházelo k vícenásobné definici proměnné, protože bylo více odpovědí na jedné HTML stránce. Toto bylo vyřešeno jednoduchým uzavřením kódu do bloku.

Dále pak vyučujícím při opravě nevyhovovala různá velikost písmen v odpovědi. Velikost písmen v textu se totiž původně neřešila (zůstávala shodná se zápisem studenta). To mohlo být využíváno například při zápisu víceslovného názvu tzv. velbloudí notací (tedy způsobu zápisu, kdy jednotlivá slova nejsou oddělena mezerou, ale začínají velkým písmenem). Nicméně při opravě z pohledu učitele se různá velikost písmen ukázala jako nevyhovující, takže metoda *buildHtmlFromJson*() třídy *Builder* byla upravena a nově zobrazuje všechna písmena jako malá.

6.2 Testování v reálném provozu

Dne 27. 11. 2019 proběhl v kurzu BI-DBS zápočtový test, ve kterém byl nový Tralex otestován přímo studenty. Při zápočtovém testu nenastal žádný zásadní problém, ale byly objeveny a následně opraveny tři menší chyby.

Prvním problémem bylo, že kontrola klíčových slov metodou *checkInput*() třídy *Parser* nebrala v úvahu velikost písmen. V seznamech klíčových slov entit a atributů jsou všechny názvy uloženy s malými písmeny. Za předpokladu, že v seznamu názvu entit bylo slovo „smlouva“, a student do textového editoru zapsal toto slovo s velkým počátečním písmenem („Smlouva“), metoda *checkInput*() ho vyhodnotila jako neznámé.

Další problém se objevil v seznamu klíčových slov atributů. V případě, že zadání otázky (tvořeno `Kreslítkem`) obsahovalo popisek u nějaké referenční integrity, tak tento popisek sice byl v seznamu entit, ale chyběl v seznamu atributů. Byl tedy přidán s prefixem „id_“ i bez něj.

Třetí problém nastal v situaci, kdy byl zobrazen našeptávač, student posunul kurzorem (například vlevo) a následně vložil slovo označené v našeptávači (za předpokladu, že se kurzor nacházel stále v tom stejném slově). V případě, kdy byl zobrazen našeptávač, totiž byla chyba ve výpočtu pozice, na kterou se měl vložit suffix našeptávaného slova. Tato chyba se projevila právě až po posunutí kurzoru v rámci stejného slova.

O tom, že výše zmíněné chyby, nalezené v reálném provozu, nebyly nijak závažné, může svědčit fakt, že jejich nalezení i opravení bylo otázkou několika minut.

Po dopsání testu byli studenti ještě vyzváni k tomu, aby anonymně ohodnotili, jak se jim pracovalo s novým Tralexem. Test psalo celkem 24 studentů a hodnocení se zúčastnili všichni. Byly použity známky jako ve škole (A – nejlepší, F – nejhorší) a výsledky znázorňuje následující tabulka:

Hodnocení	A	B	C	D	E	F
Počet studentů	16	7	0	0	0	1

Tabulka 6.1: Hodnocení aplikace studenty

Následně byl studentům odeslán e-mail s anonymním dotazníkem, do kterého mohli vyplnit konkrétní případy toho, co jim vadilo nebo co naopak ocenili. Tohoto „pokračování“ se zúčastnila pouze hrstka studentů – konkrétně dorazily pouze dva vyplněné dotazníky. Vesměs byly popisovány výše zmíněné problémy a na žádné další se nepřišlo. Výsledky dotazníku jsou zobrazeny a okomentovány v **příloze A**.

Závěr

Tato práce se zabývá především zdokonalením předmětu BI-DBS – Databázové systémy, respektive jeho výukového portálu `dbs.fit.cvut.cz`. Konkrétně se zdokonalení týká zjednodušeného relačního zápisu, který je nezbytnou součástí tohoto předmětu. V práci jsou postupně popsána předešlá řešení této problematiky, následuje návrh a implementace. V závěru práce je zmapováno nasazení výsledné aplikace na BI-DBS portál a také její otestování studenty v zápočtovém testu.

Hlavním cílem práce bylo dodat nové řešení transformace zjednodušeného relačního zápisu (což zahrnuje i eliminaci nevýhod vyskytujících se v předešlých řešeních) a to včetně funkcionalit pro snadnější zadávání, mezi které patří i našeptávání. Dále pak pokusit se aplikaci nasadit a otestovat v reálném provozu. Výsledkem práce je textový editor, který dokáže našeptávat slova přímo ze zadaní, které dodává portál `dbs.fit.cvut.cz`. Tento editor rovněž celý vstup převede do vhodného, dále použitelného, formátu. Aplikaci se podařilo nasadit na portál a otestovat v reálném provozu. Cíle práce tedy byly splněny a aplikace již plní svůj účel.

Výhledem do budoucna je jednoznačně propojení aplikace s modulem DSM (respektive integrace aplikace do tohoto modulu). Toto propojení by mělo zahrnovat i automatickou opravu otázek zjednodušeného relačního zápisu. Jak totiž bylo zmíněno, zadání u otázek typu transformace je tvořeno diagramem, který byl zakreslen v `Kreslítku`. A právě aplikace vytvořená v této práci transformuje uživatelský vstup na stejný datový formát jako využívá `Kreslítko`. Zkompletováním nového Tralexu a DSM modulu by pak měla být oprava otázek typu transformace zcela automatizovaná.

Literatura

[1] VALENTA, Michal. *Transformace konceptuálního modelu na relační* [online] [cit. 2019-04-17]. Dostupné z: `https://courses.fit.cvut.cz/BI-DBS/materials/slides/hand-les05-transformace.pdf` [Soubor přístupný po přihlášení do sítě ČVUT].

[2] PEJŠA, Petr. *Systém pro podporu testování studentů v BI-DBS*. Praha, 2016. Dostupné také z: `https://dspace.cvut.cz/handle/10467/65093`. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, Katedra softwarového inženýrství informatiky. Vedoucí práce Jiří Hunka.

[3] MACHALA, Filip. *Automatická oprava zjednodušeného relačního zápisu*. Praha, 2018. Dostupné také z: `https://dspace.cvut.cz/handle/10467/76661`. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, Katedra softwarového inženýrství informatiky. Vedoucí práce Jiří Hunka.

[4] HOLUB, Jan. *Základní pojmy* [online] [cit. 2019-04-17]. Dostupné z: `https://courses.fit.cvut.cz/BI-AAG/lectures/bi-aag-01-zakladni_pojmy.pdf` [Soubor přístupný po přihlášení do sítě ČVUT]

[5] HOLUB, Jan. *Zásobníkové automaty* [online] [cit. 2019-04-17]. Dostupné z: `https://courses.fit.cvut.cz/BI-AAG/lectures/bi-aag-08-zasobnikove_automaty.pdf` [Soubor přístupný po přihlášení do sítě ČVUT]

[6] JANOUŠEK, Jan. *Programovací jazyky a překladače: Lexikální analýza* [online] [cit. 2019-04-17]. Dostupné z: `https://courses.fit.cvut.cz/BI-PJP/media/lectures/02/pjp-prednaska2.pdf` [Soubor přístupný po přihlášení do sítě ČVUT]

[7] JANOUŠEK, Jan. *Programovací jazyky a překladače: Syntaktická analýza* [online] [cit. 2019-11-17]. Dostupné z: `https://courses.fit.cvut.cz/BI-PJP/media/lectures/03/pjp-prednaska3.pdf` [Soubor přístupný po přihlášení do sítě ČVUT]

[8] HLAVATÝ, Martin. *Softwarový proces* [online] [cit. 2019-11-20]. Dostupné z: `https://moodle-vyuka.cvut.cz/pluginfile.php/171216/course/section/28533/1_SoftwareProcess.pdf` [Soubor přístupný po přihlášení do sítě ČVUT]

[9] SOFTWARE FREEDOM CONSERVANCY. *Git Docs* [online] [cit. 2019-11-20]. Dostupné z: `https://git-scm.com/about/`

[10] GITLAB. *GitLab Docs* [online] [cit. 2019-11-30]. Dostupné z: `https://docs.gitlab.com/ee/`

[11] JETBRAINS. *Kotlin Docs* [online] [cit. 2019-11-20]. Dostupné z: `https://kotlinlang.org/docs/reference/`

[12] ORACLE. *Java Docs* [online] [cit. 2019-11-20]. Dostupné z: `https://docs.oracle.com/en/java/`

[13] MOZILLA. *JavaScript Docs* [online] [cit. 2019-11-20]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript`

[14] THE PHP GROUP. *PHP Docs* [online] [cit. 2019-11-20]. Dostupné z: `https://www.php.net/docs.php`

[15] NETTE FOUNDATION. *Nette Docs* [online] [cit. 2019-11-20]. Dostupné z: `https://doc.nette.org/en/3.0/`

[16] MOZILLA. *JSON* [online] [cit. 2019-11-21]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON`

[17] MOZILLA. *Making content editable* [online] [cit. 2019-11-21]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Editable_content`

[18] BRAY, Tim. *The JavaScript Object Notation (JSON) Data Interchange Format* [online] [cit. 2019-11-26]. Dostupné z: `http://www.rfc-editor.org/rfc/rfc7159.txt`

[19] GRADLE. *Gradle User Manual* [online] [cit. 2019-11-26]. Dostupné z: `https://docs.gradle.org/current/userguide/userguide.html`

[20] APACHE GROOVY PROJECT. *Groovy documentation* [online] [cit. 2019-11-26]. Dostupné z: `https://groovy-lang.org/documentation.html`

[21] MOZILLA. *EventListener* [online] [cit. 2019-11-27]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/EventListener`

[22] MOZILLA. *Document.execCommand* [online] [cit. 2019-11-27]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/Document/execCommand`

[23] MOZILLA. *EventTarget.dispatchEvent()* [online] [cit. 2019-11-28]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/dispatchEvent`

[24] MOZILLA. *Event.preventDefault()* [online] [cit. 2019-11-28]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault`

[25] NETTE FOUNDATION. *Latte Docs* [online] [cit. 2019-11-30]. Dostupné z: `https://latte.nette.org/en/guide`

[26] THE JQUERY FOUNDATION. *jQuery API Documentation* [online] [cit. 2019-11-30]. Dostupné z: `https://api.jquery.com`

[27] *Testování* [online] [cit. 2019-12-1]. Dostupné z: `https://moodle.fit.cvut.cz/pluginfile.php/172/course/section/11326/06_Testing.pdf` [Soubor přístupný po přihlášení do sítě ČVUT]

Příloha A

Výsledky dotazníku

Dotazník zjišťoval poznatky studentů (kladné či záporné) k novému Tralexu a jeho textovému editoru. Níže jsou shrnuty odpovědi dvou respondentů, kteří tento dotazník zodpověděli. Odpovědi jsou citovány, případně drobně přeformulovány. Doplněn je také komentář autora práce.

A.1 Respondent 1

- „Našeptávač našeptává i atributy z jiných entit“ – to je správně v souladu se zadáním práce. Důvod je ten, že našeptávač nemá za úkol radit studentům, ale pouze jim usnadnit zápis.
- „Našeptávač doplňuje čárku i za poslední atribut“ – doplňování čárek za atributy probíhá pouze uvnitř entitních atributů, nikoliv relačních, protože v relaci je více atributů spíše ojedinělý případ. V entitních atributech je čárka doplňována vždy a to jednoduše proto, že není možné určit, který atribut bude poslední. Navíc poslední čárka v entitních atributech je při zpracovávání vstupu ignorována.
- Možnost využívat tabulátor pro vložení označeného slova – zajímavý nápad, který by mohl být v budoucnu zrealizován (aktuálně je možné vložit slovo pomocí tlačítka Enter a také kliknutím myši).

A.2 Respondent 2

- Chybějící názvy referenčních integrit v seznamu našeptávaných atributů – opraveno po zápočtovém testu.
- Problém s našeptáváním při posunutí kurzoru – rovněž opraveno po zápočtovém testu.

Příloha **B**

Seznam použitých zkratek

ER diagram – Entity relationship diagram

JSON – JavaScript Object Notation

HTML – Hypertext Markup Language

DSM – Database Schema Modeller

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `BP_Sach_Martin_2020.pdf` ..... text práce ve formátu PDF