Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# Automatizácia kontroly formalizačných cvičení v logike prvého rádu

Bakalárska práca

2021
Samantha Gombárová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# Automatizácia kontroly formalizačných cvičení v logike prvého rádu

Bakalárska práca

| | |
|---|---|
| Študijný program: | Informatika |
| Študijný odbor: | Informatika |
| Školiace pracovisko: | Katedra aplikovanej informatiky |
| Školiteľ: | Mgr. Ján Kľuka, PhD. |

Bratislava, 2021
Samantha Gombárová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky



# ZADANIE ZÁVEREČNEJ PRÁCE

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

**Názov:** Automatizácia kontroly formalizačných cvičení v logike prvého rádu
*Automated checking of formalisation exercises in first-order logic*

**Anotácia:** Formalizácia patrí k najnáročnejším témam pri výučbe logiky. Jej cieľom je čo najpresnejšie zachytiť význam neformálneho tvrdenia v prirodzenom jazyku pomocou obmedzených prostriedkov formálneho jazyka. Výučba formalizácie sa podobá výučbe cudzieho jazyka. Na jej zvládnutie je potrebné spraviť množstvo cvičení a dostať na ne spätnú väzbu. Ideálne je, keď v prípade nesprávneho či neadekvátneho riešenia spätná väzba objasní, prečo študentovo riešenie nie je vhodné – napríklad formou kontraprík ladu. Poskytovať adekvátnu a promptnú spätnú väzbu v rozsahu, aký by niektorí študenti potrebovali, často nie je v silách učiteľov.

Hoci vo všeobecnosti formalizácia nemá jednoznačné riešenie, cvičenia na túto tému sa spravidla volia tak, aby boli čo najjednoznačnejšie a kombinovali známe štandardné idiómy. Navyše sú zvyčajne výsledkom formalizácie relatívne jednoduché formuly. Relatívna jednoduchosť a jednoznačnosť riešení poskytuje príležitosť využiť na kontrolu a poskytovanie spätnej väzby existujúce dokazovače pre logiku prvého rádu. V práci na túto tému by sme tento prístup chceli bližšie preskúmať, implementovať a otestovať pri výučbe.

**Cieľ:**
- Vytvoriť webovú službu na overovanie správnosti formalizácie voči vhodne uloženému očakávanému riešeniu alebo riešeniam a generovanie kontraprík ladov neočakávaných riešení.
- Doplniť službu vhodným používateľským rozhraním pre študentov aj učiteľov.
- Integrovať rozhranie do interaktívneho pracovného hárka vytvoreného v predchádzajúcej bakalárskej práci.

**Literatúra:**
Barker-Plummer, D., Barwise, J., Etchemendy, J. et al.: Language, Proof and Logic. Second Edition. Stanford: CSLI, 2010.
Barker-Plummer, D., Dale, R., Cox, R., Etchemendy, J. Automated Assessment in the Internet Classroom. In: Proceedings of the AAAI Fall Symposium on Education Informatics, Arlington, VA. AAAI 2008.
Perikos, I., Grivokostopoulou, F., Hatzilygeroudis, I. Automatic marking of NL to FOL conversions. In: Uskov, V. (ed.). Procs. Computers and Advanced Technology in Education (CATE 2012). ACTA Press, 2020.
Kniha, N. Interaktívny pracovný hárok pre výučbu logiky pre informatikov. Bakalárska práca. Bratislava: Univerzita Komenského, 2020.


Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

McCune, W. Prover9 and Mace4. [online] http://www.cs.unm.edu/~mccune/Prover9, 2005-2010.

**Kľúčové slová:** nástroje pre logiku, logika prvého rádu, automatické dokazovanie tvrdení, vyhodnocovanie cvičení, webová aplikácia

**Vedúci:** Mgr. Ján Kľuka, PhD.
**Katedra:** FMFI.KAI - Katedra aplikovanej informatiky
**Vedúci katedry:** prof. Ing. Igor Farkaš, Dr.

**Dátum zadania:** 01.09.2020

**Dátum schválenia:** 24.09.2020 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

**Poďakovanie:** Chcela by som sa poďakovať školiteľovi mojej bakalárskej práce, pánovi Mgr. Jánovi Kľukovi, PhD., za jeho odborné rady, ochotu a trpezlivosť.

# Abstrakt

Táto bakalárska práca sa zaoberá vytvorením klient-server webovej aplikácie, ktorá umožňuje študentom riešiť formalizačné cvičenia. Študent si následne môže svoje riešenia nechať automaticky vyhodnotiť, čím získa cennú spätnú väzbu. Zároveň bude študentovi objasnené, v čom spravil chyby. Aplikácia bola vytvorená v programovacom jazyku JavaScript, pričom na strane klienta bola použitá knižnica React a na strane servera Express.js. Vyhodnocovanie cvičení bolo realizované s pomocou automatického dokazovača Vampire.

**Kľúčové slová:** formalizačné cvičenia, klient-server webová aplikácia, automatizované vyhodnotenie cvičenia

# Abstract

This bachelor thesis deals with the development of a client-server web application that allows students to solve formalization exercises. Students can then have their solutions automatically checked, thus gaining valuable feedback. At the same time, student will find out where he made mistakes. The application was developed in programming language JavaScript, using React library on the client side and Express.js on the server side of the application. Checking of exercises was carried out with the help of the Vampire automated theorem prover.

**Keywords:** formalization exercises, client-server web application, automated checking of exercises

# Obsah

# Úvod

Dôležitosť schopnosti formálne zapísať tvrdenie v matematike a iných príbuzných disciplínach pravdepodobne netreba zdôrazňovať. Preto je za účelom nadobudnutia tejto zručnosti pre študentov potrebné venovať formalizácii (prekladu tvrdenia z prirodzeného jazyka do niektorého formálneho jazyka) dostatočne veľa času. Častokrát je pre jej pochopenie dôležité vyriešiť veľké množstvo cvičení a úloh, vďaka ktorým si študent na formálny jazyk zvykne a efektívnejšie si formalizáciu tvrdení osvojí.

Avšak pre študenta je často potrebné mať možnosť si správnosť riešenia nejakým rýchlym a spoľahlivým spôsobom overiť. Ak bude študentovi navyše objasnené, v čom spravil chyby, nadobudne ešte lepšiu predstavu o tom, na aké špeciálne prípady si pri formalizovaní tvrdení musí dávať pozor, aby podobnú chybu nespravil znova.

Cieľom tejto bakalárskej práce je vytvoriť webovú aplikáciu, ktorá by toto overovanie automatizovala a študentom by bola k dispozícii 24 hodín denne 7 dní v týždni. Študenti by tak pomocou tejto aplikácie získali promptnú spätnú väzbu na ich riešenia formalizačných úloh a bolo by im zároveň objasnené, v čom spravili chyby.

Podobné systémy už existujú, ale sú zamerané inak. Grade Grinder ([2]) je uzavretý systém na vyhodnocovanie úloh z učebníc, ktorých zakúpením k nemu študent získa prístup, a neposkytuje študentom detailnejšiu spätnú väzbu, iba hodnotenie. Systém Perikosa a kol. ([7]) kontroluje riešenia porovnaním so vzorovým na základe syntaktických kritérií, nie významu.

Text tejto bakalárskej práce je rozdelený na kapitoly. Kapitola 1 poskytuje ucelený pohľad na teoretickú stránku veci. Vysvetľuje základné pojmy, s ktorými budeme ďalej v texte pracovať. V kapitole 2 sú zhrnuté základné požiadavky na našu aplikáciu a správanie, ktoré od nej očakávame. Kapitola 3 nám zase priblíži návrh aplikácie, pre ktorý sme sa rozhodli. V kapitole 4 sme opísali najdôležitejšie technológie, ktoré sme pri vytváraní aplikácie použili. Kapitola 5 pojednáva o dokazovači Vampire, ktorý nám umožnil rýchle a spoľahlivé overovanie správnosti riešenia. V kapitole 6 si povieme niečo o samotnej implementácii a organizácii zdrojového kódu.

# Kapitola 1

# Jazyk logiky prvého rádu a formalizácia

V prvej kapitole definujeme a zhrnieme základné pojmy, ktoré budeme v texte tejto práce používať. Vzhľadom na to, že v rôznych odborných textoch sa spôsoby, ktorými sú tieto pojmy definované, môžu v menšom rozsahu líšiť, považujeme za dôležité, aby bol čitateľ vopred oboznámený s takým znením definícií, na základe ktorého bola budovaná naša webová aplikácia na vyhodnocovanie úloh z formalizácie. Vychádzame predovšetkým z definícií uvedených v prednáškach [5] a v [10].

Pod pojmom formalizácie rozumieme preklad tvrdenia z prirodzeného jazyka do niektorého formálneho jazyka, v našom prípade konkrétne do jazyka logiky prvého rádu. Jednotlivé formálne tvrdenia nazývame *formuly*. V tejto kapitole ukážeme spôsob, akým formuly v prvorádovej logike vytvárame, a uvedieme aj niekoľko príkladov formalizácie jednoduchých tvrdení.

## 1.1 Jazyk logiky prvého rádu

Jazyk logiky prvého rádu sa skladá z troch hlavných skupín symbolov. Prvou skupinou sú logické symboly, druhou mimologické symboly a treťou sú symboly pre premenné. Okrem týchto troch zložiek jazyk obsahuje aj pomocné symboly, ako sú napríklad zátvorky, avšak ich používanie nie je vždy striktne stanovené a mimo prípadov, v ktorých je použitie zátvoriek nevyhnutné za účelom určenia správneho poradia operácií, môžeme ich použitie považovať za vec vkusu.

*Logické symboly* zahŕňajú výrokovologické spojky pre základné logické operácie, ktorými sú negácia, konjunkcia, disjunkcia, implikácia a ekvivalencia (v danom poradí ich budeme označovať $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Longleftrightarrow$ ). Ďalej medzi logické symboly patria aj

symboly pre všeobecný a existenčný kvantifikátor (budeme ich označovať $\forall$, $\exists$). Tiež sem zahrnieme aj špeciálny predikátový symbol rovnosti ($=$).

*Mimologické symboly* sú časťou jazyka, ktorú musíme zadefinovať vopred v rámci každej formalizačnej úlohy samostatne. Ide o symboly, ktoré v závislosti od svojho typu označujú nejaký konkrétny objekt, dávajú mu nejakú vlastnosť, určujú vzťah medzi viacerými objektami alebo skupine objektov priraďujú niektorý iný objekt. Množina mimologických symbolov je zjednotením troch vzájomne disjunktných množín:

(i) množina symbolov pre konštanty $\mathcal{C}$

(ii) množina predikátových symbolov $\mathcal{P}$

(iii) množina funkčných symbolov $\mathcal{F}$

*Symboly pre konštanty* pomenúvajú konkrétne individuálne objekty. Môže ísť o istú analógiu mien, ale netreba zabúdať na to, že v našom prípade symbol pre konštantu vždy označuje práve jeden unikátny objekt (a to aj napriek tomu, že v reálnom svete môže existovať viacero objektov s rovnakým menom).

*Predikátové symboly* vyjadrujú nejakú vlastnosť objektov alebo vzťah medzi viacerými objektami. Každý predikátový symbol má priradené nenulové prirodzené číslo, ktoré určuje počet jeho argumentov.

*Funkčné symboly* vyjadrujú jednoznačné priradenie jedného objektu iným objektom na základe nejakého vzťahu. Každý funkčný symbol má priradené prirodzené číslo, ktoré určuje počet jeho argumentov.

*Symbolov pre premenné* je nekonečne, ale spočítateľne veľa. Množina symbolov pre premenné musí byť disjunktná s množinou mimologických symbolov.

## 1.2 Interpretácia jazyka

Ak chceme preložiť nejaké tvrdenie z prirodzeného jazyka do jazyka logiky prvého rádu, je potrebné jednotlivým mimologickým symbolom priradiť rozumný význam.

V [5] je definovaný pojem *štruktúra* pre jazyk ako dvojica $(D, i)$, kde $D$ označuje neprázdnu množinu objektov (zvykne sa označovať ako *doména* alebo *univerzum*) a $i$ je zobrazenie, ktoré:

(i) každému symbolu pre konštantu z množiny $\mathcal{C}$ priradí konkrétny objekt z univerza $D$

(ii) každému predikátovému symbolu $p$ z množiny $\mathcal{P}$ s počtom argumentov $m$ priradí $m$-árnu reláciu na množine $D$ (čiže musí platiť $i(p) \subseteq D^m$)

(iii) každému funkčnému symbolu $f$ z množiny $\mathcal{F}$ s počtom argumentov $n$ priradí zobrazenie $f : D^n \to D$

Pojem štruktúry ilustrujeme na jednoduchých príkladoch.

Majme množinu symbolov pre konštanty $\mathcal{C} = \{ n \mid n \in \mathbb{N} \wedge 1 \leq n \leq 4 \}$. Ďalej majme množinu predikátových symbolov $\mathcal{P} = \{$ `Dvojnásobok` $\}$, kde symbol `Dvojnásobok` je binárny predikátový symbol. Uvažujme prázdnu množinu funkčných symbolov.

Definujeme nejakú štruktúru pre tento jazyk. Uvažujme doménu, ktorá je v princípe zhodná s množinou $\mathcal{C}$ a platí $i(n) = n$ pre každú konštantu $n$. Ak chceme, aby v tejto štruktúre platilo, že dvojnásobkom čísla 1 je číslo 2 a dvojnásobkom čísla 2 je číslo 4, musíme obraz predikátového symbolu `Dvojnásobok` v zobrazení $i$ definovať ako

$$i(\texttt{ Dvojnásobok }) = \{ \ (1,2), \ (2,4) \ \}$$

Uvedieme ešte jeden príklad štruktúry. Tentoraz majme nekonečnú množinu konštánt $\mathcal{C} = \mathbb{N}$, pričom doména je tiež zhodná s $\mathbb{N}$ a platí $i(n) = n$ pre každé prirodzené číslo. Nech je množina predikátových symbolov prázdna. Majme množinu funkčných symbolov $\mathcal{F} = \{$ `Zdvojnásob` $\}$, pričom

$$i(\texttt{ Zdvojnásob }) = z$$

kde $z : \mathbb{N} \to \mathbb{N}$ je funkcia definovaná ako $z(n) = 2 \cdot n$ pre každé $n \in D$.

## 1.3 Formuly

Na základe jazyka, ktorý sme si definovali v predchádzajúcich podkapitolách, môžeme vytvárať *formuly* - tvrdenia, ktorým dokážeme priradiť pravdivostnú hodnotu. Pri tvorbe formúl budeme využívať výhradne symboly z jazyka, ktorý sme si definovali.

Základným stavebným prvkom každej formuly sú *atomické formuly*. Z hľadiska formúl ide o nedeliteľný celok, ktorý je sám formulou. Atomické formuly vyjadrujú tie najjednoduchšie vety v prirodzenom jazyku. V našom prípade teda atomickou formulou môže byť výlučne iba predikátový symbol s požadovaným počtom argumentov alebo predikát rovnosti vytvorený pomocou špeciálneho symbolu rovnosti. Argumentami predikátového symbolu môžu byť jedine termy - matematické výrazy. Medzi termy patria symboly pre premenné, symboly pre konštanty a ďalšie termy vytvorené induktívne z iných termov aplikovaním funkčných symbolov.

Uvedieme niekoľko príkladov atomických formúl. Uvažujme jazyk, ktorý je podobný jazyku v predchádzajúcej podkapitole, teda s množinou konštánt $\mathcal{C} = \mathbb{N}$, predikátovým symbolom `Dvojnásobok` a funkčným symbolom `Zdvojnásob`. Majme štruktúru, kde sú konštanty a funkčný symbol interpretované rovnako, ako v poslednom príklade v predchádzajúcej podkapitole, a predikátový symbol `Dvojnásobok` definujeme v zobrazení $i$ ako $\{ (n, 2 \cdot n) \mid n \in \mathcal{C} \}$.

Predikátový symbol `Dvojnásobok` teda vyjadruje jednoduchú vetu v prirodzenom jazyku - ak napíšeme tvrdenie `Dvojnásobok`$(x, y)$, znamená to, že tvrdíme, že dvojnásobkom čísla $x$ je číslo $y$. Ak teda použijeme aj predikát rovnosti, vieme vytvoriť napríklad nasledovné atomické formuly:

(a) `Dvojnásobok`$(2, 4)$

(b) `Dvojnásobok`$(4, 9)$

(c) $2 =$ `Zdvojnásob(3)`

(d) `Dvojnásobok`$(4,$ `Zdvojnásob`$(4))$

Ako môžeme vidieť, atomické formuly $(a)$ a $(b)$ vznikli tak, že za argumenty predikátu `Dvojnásobok` sme dosadili termy, v tomto prípade iba jednoduché konštanty. V prípade atomickej formuly $(c)$ sme použili predikát rovnosti, pričom jeho prvý argument je konštanta a druhý je term, ktorý vznikol aplikovaním funkčného symbolu `Zdvojnásob` na konštantu 3. Atomická formula $(d)$ vznikla z predikátového symbolu `Dvojnásobok` dosadením konštanty 4 a funkčného symbolu `Zdvojnásob` aplikovaného na konštantu 4. Každej z týchto atomických formúl vieme jednoznačne priradiť pravdivostnú hodnotu na základe štruktúry, ktorú sme definovali. Zjavne sú formuly $(a)$ a $(d)$ v nami uvažovanej štruktúre pravdivé a formuly $(b)$ a $(c)$ naopak nepravdivé.

Zložitejšie formuly ďalej vytvárame z atomických formúl iba pomocou logických spojok a kvantifikátorov. Zvykne sa používať nasledovná definícia formuly:

(i) každá atomická formula je formula

(ii) ak je $A$ formula, potom aj $\neg A$ je formula

(iii) ak sú $A, B$ formuly, potom aj $A \wedge B,\ A \vee B,\ A \Rightarrow B,\ A \iff B$ sú formuly

(iv) ak je $A$ formula a $x$ je premenná, potom aj $(\forall x)(A(x)),\ (\exists x)(A(x))$ sú formuly

(v) nič iné nie je formulou

Formuly teda vieme zostrojiť z atomických formúl konečným počtom použití pravidiel (ii), (iii) a (iv).

Uvedieme jednoduchý príklad. Ak by sme chceli v jazyku logiky prvého rádu vyjadriť tvrdenie

„Ak sa číslo rovná svojmu dvojnásobku, tak sa rovná nule. “

formalizovali by sme ho ako

$$(\forall x)(x = \texttt{Zdvojnásob}(x) \;\Rightarrow\; x = 0)$$

## 1.4 Ekvivalencia formúl

Majme tri predikátové symboly `student`, `znamka` a `hodnotenie` s nasledovnými významami:

1. $\texttt{student}(x)$ - $x$ je študent
2. $\texttt{znamka}(x)$ - $x$ je známka
3. $\texttt{hodnotenie}(x, y)$ - $x$ je hodnotený známkou $y$

Majme nasledovné tvrdenie v prirodzenom jazyku:

„Každý študent je hodnotený nejakou známkou. “

Vieme ho formalizovať minimálne dvoma spôsobmi:

$$(\forall x)(\texttt{student}(x) \;\Rightarrow\; (\exists y)(\texttt{znamka}(y) \;\wedge\; \texttt{hodnotenie}(x, y)))$$

$$\neg(\exists x)(\texttt{student}(x) \;\wedge\; (\forall y)(\neg\texttt{znamka}(y) \;\vee\; \neg\texttt{hodnotenie}(x, y)))$$

Obe tieto formuly vyjadrujú to isté tvrdenie, a ak platí jedna formula, musí automaticky platiť aj druhá. Formuly sú *ekvivalentné*. Uvedieme aj presnejšiu definíciu.

Formula $A$ je *platná* (označujeme $\vDash A$), ak je pravdivá v každej štruktúre.

Hovoríme, že dve formuly $A$ a $B$ sú *ekvivalentné*, ak je formula $A \iff B$ platná ($\vDash (A \iff B)$), respektíve keď sú formuly $A \to B$ a $B \to A$ platné. Inými slovami, formula $B$ musí byť dokázateľná z predpokladu $A$, a naopak, formula $A$ musí byť dokázateľná z predpokladu $B$, pretože podľa vety o úplnosti pre logiku prvého rádu je formula dokázateľná práve vtedy, keď je platná a podľa vety o dedukcii je implikácia $A \to B$ dokázateľná práve vtedy, keď je z $A$ dokázateľná $B$.

Overiť, či sú dve formuly v logike prvého rádu ekvivalentné, je nerozhodnuteľný problém [10]. Je to tak z toho dôvodu, že do tohto problému sa dá redukovať problém vyplývania, do ktorého sa dá zasa redukovať problém zastavenia, ktorý je nerozhodnuteľný. Problém ekvivalencie dvoch formúl je ale rekurzívne vyčísliteľný. Každá platná formula sa totiž dá v konečnom čase dokázať. Problém viazne v identifikovaní formúl, ktoré dokázateľné nie sú.

# Kapitola 2

# Požiadavky na aplikáciu

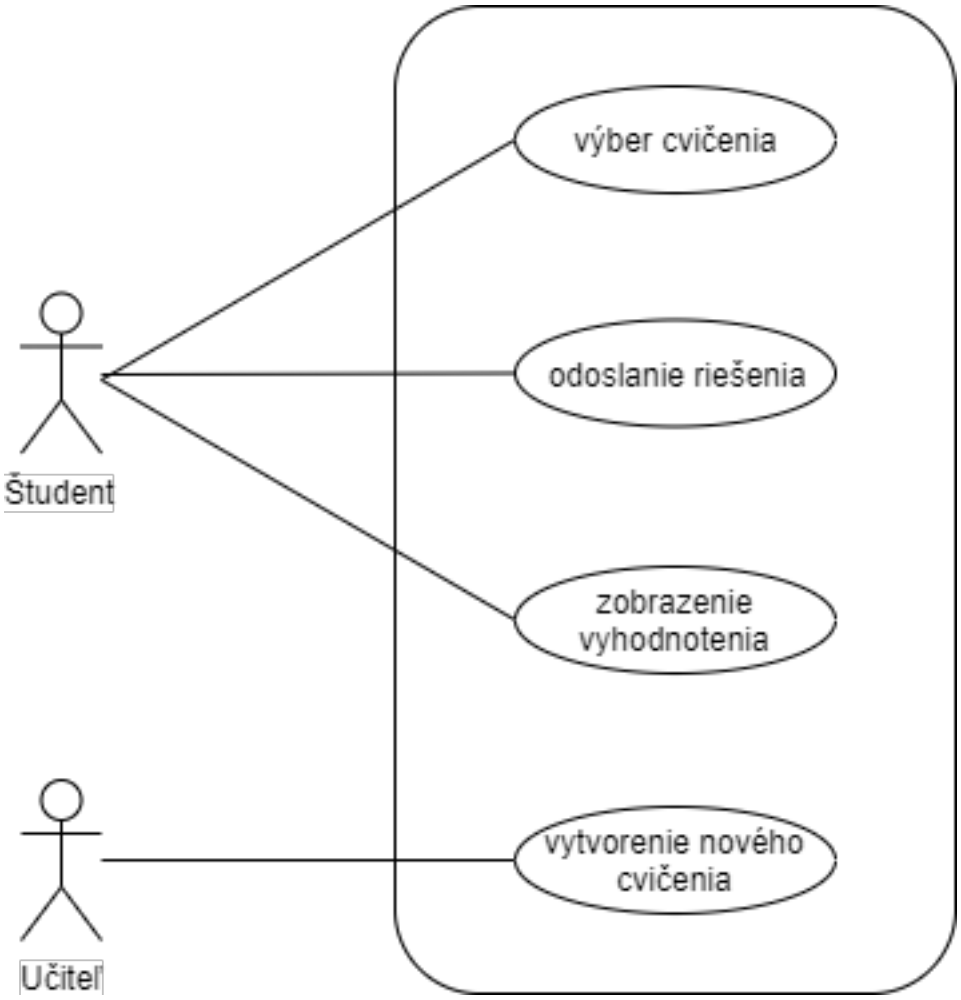
Základnou funkcionalitou, ktorá bude prístupná pre používateľov aj bez nutnosti prihlásenia, je možnosť zvoliť si cvičenie z ponuky cvičení, vyriešiť ho a odoslať tieto riešenia na vyhodnotenie s tým, že aplikácia toto vyhodnotenie následne zobrazí.

Pri prístupe na domovskú stránku sa používateľovi zobrazia náhľady cvičení, ktoré sú uložené v databáze cvičení. Používateľ si zvolí cvičenie, ktoré chce riešiť. Zobrazí sa mu stránka so zadaním cvičenia a textovými poľami, do ktorých bude následne vpisovať svoje riešenia.

Každé cvičenie má v zadaní definované mimologické symboly, ktoré študent vo výsledných formulách môže používať. Pod zadaním sú tvrdenia v prirodzenom jazyku, ktoré je potrebné sformalizovať. Ku každému tvrdeniu je umiestnené textové pole, do ktorého sa vpisuje riešenie, a tlačidlo, pomocou ktorého pošleme riešenie na vyhodnotenie. Používateľ musí mať možnosť posielať jeho riešenia na vyhodnotenie každé jednotlivo, bez nutnosti vyriešenia všetkých úloh v cvičení. Jednotlivé formalizácie teda môže poslať na vyhodnotenie každú zvlášť. Vyhodnotenie sa následne zobrazí pod odpoveďou.

Riešenia (formalizácie tvrdení) bude musieť používateľ zadať v špeciálnom vopred dohodnutom jazyku pre zápis formúl. Obsah textového poľa s riešením sa zároveň bude v reálnom čase parsovať (t.j. syntakticky analyzovať). Ak riešenie obsahuje syntaktické chyby, pod textovým poľom sa o tom vypíše informácia. Výpis chyby sa pri zmenách obsahu textového poľa bude pravidelne aktualizovať. Riešenie bude možné odoslať na vyhodnotenie až vtedy, keď bude obsah textového poľa s riešením bez syntaktických chýb.

Vyhodnotenie riešenia bude obsahovať správu o tom, či bolo riešenie správne alebo nie. V prípade nesprávneho riešenia sa navyše zobrazí kontrapríklad, ktorý ukáže rozdiel medzi tvrdením zadaným v prirodzenom jazyku a formalizáciou, ktorú zadal používateľ. Tento kontrapríklad bude príkladom konkrétnej štruktúry, v ktorej je prvé z tvrdení




splnené, ale druhé nie je, prípadne naopak.

Ďalej bude aplikácia obsahovať funkcionalitu, ktorá bude dostupná iba prihláseným používateľom. Prihlásený používateľ bude môcť vytvoriť nové cvičenie - zadefinovať mu mimologické symboly (množinu symbolov pre konštanty, množinu pre predikátové symboly a množinu pre funkčné symboly) a popridávať tvrdenia v prirodzenom jazyku, ktoré treba sformalizovať. Pre každé tvrdenie v prirodzenom jazyku treba zadať jeho správnu formalizáciu.

Obsahy jednotlivých textových polí pre definíciu mimologických symbolov sa taktiež budú v reálnom čase parsovať a bude sa vypisovať informácia o prípadnej syntaktickej chybe. Takisto sa budú kontrolovať kolízie medzi mimologickými symbolmi (napríklad ak bude niektorý symbol pre konštantu zhodný s niektorým predikátovým symbolom, vypíše sa o tom správa). Ďalej sa budú parsovať obsahy textových polí, do ktorých sa vpisujú správne formalizácie.

Aplikácia bude mať aj podstránku s prihlasovacím formulárom.

# Kapitola 3

# Návrh aplikácie

## 3.1 Architektúra aplikácie

Aplikácia bude typu klient-server. Klient bude posielať požiadavky v nasledovných prípadoch:

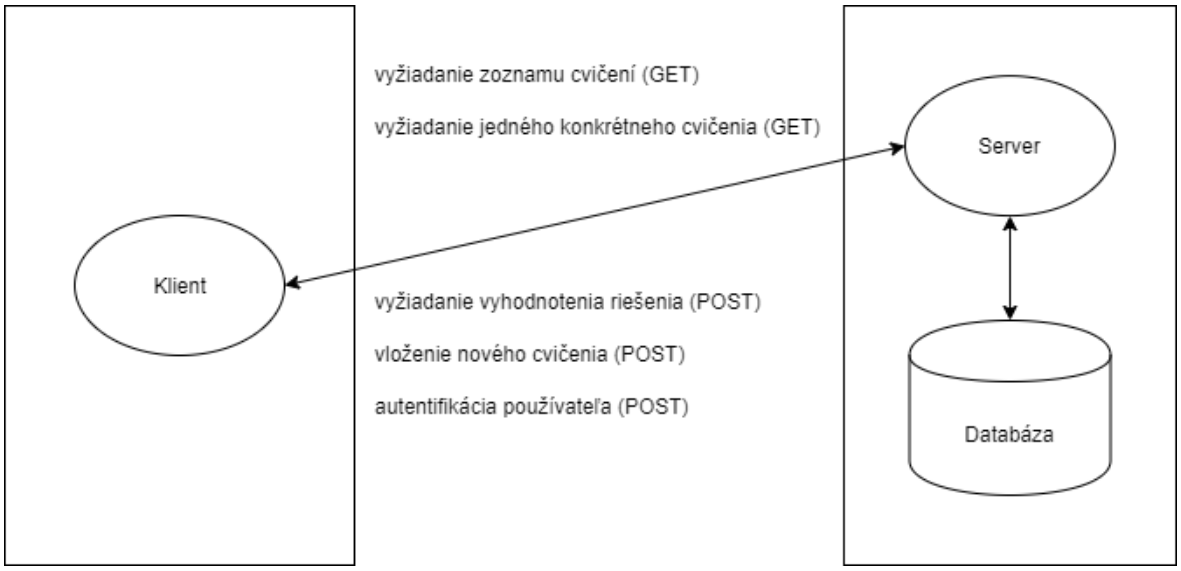
- Používateľovi sa na domovskej stránke majú zobraziť náhľady cvičení. Zoznam týchto cvičení si klient vyžiada zo servera. Ten načíta tieto údaje z databázy cvičení a odošle ich klientovi.

- Používateľ si zvolí cvičenie a má sa mu zobraziť stránka pre toto konkrétne cvičenie. Zadanie tohto cvičenia si klient vyžiada zo servera, ktorý tieto údaje načíta z databázy cvičení a odošle ich klientovi.

- Používateľ odošle riešenie na vyhodnotenie. Klient pošle serveru požiadavku na vyhodnotenie tohto riešenia. Server z databázy cvičení načíta správne riešenie, na základe ktorého vyhodnotí študentovo riešenie, a následne sa vyhodnotenie odošle klientovi.

- Prihlásený používateľ vytvoril nové cvičenie. Klient pošle serveru požiadavku na uloženie cvičenia. Server po úspešnej autorizácii používateľa uloží cvičenie do databázy cvičení a odpovie klientovi, či sa to podarilo.

- Používateľ sa prihlasuje do aplikácie. Klient pošle serveru požiadavku na overenie prihlasovacích údajov. Server overí prihlasovacie údaje na základe údajov z databázy používateľov a v prípade úspešného prihlásenia odošle klientovi token.




## 3.2 Štruktúra cvičenia

Cvičenia budú pozostávať z nasledovných údajov:

- meno cvičenia
- množina symbolov pre konštanty
- množina predikátových symbolov
- množina funkčných symbolov
- vysvetlenie významu jednotlivých predikátových a funkčných symbolov
- tvrdenia v prirodzenom jazyku, ktoré budú študenti formalizovať
- ku každému tvrdeniu jeho správna formalizácia

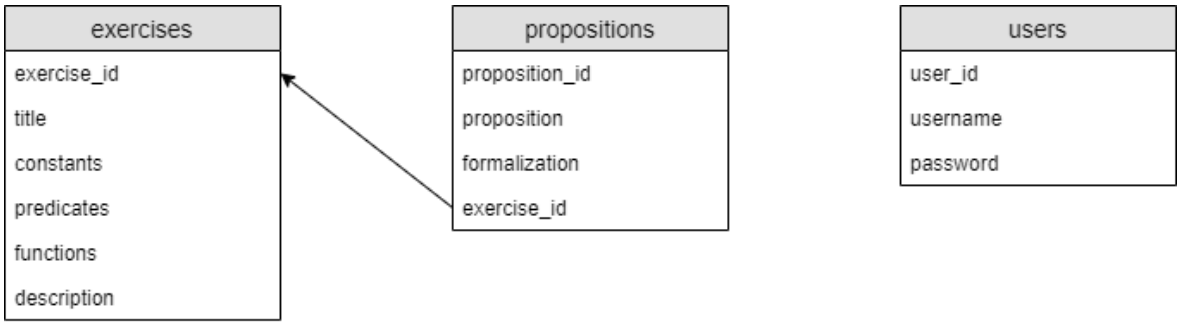
## 3.3 Štruktúra databázy

Cvičenia potrebujeme uložiť do databázy. Ako bolo stanovené v predchádzajúcej podkapitole, jedno cvičenie obsahuje viacero formúl. Preto potrebujeme mať na ukladanie cvičení k dispozícii dve tabuľky - jedna bude obsahovať samotné cvičenia s definíciami mimologických symbolov, druhá bude obsahovať formuly, pričom v tabuľke bude pre každú formulu špecifikované, ku ktorému cvičeniu patrí.

Okrem databázy cvičení budeme potrebovať aj databázu používateľov. Heslá používateľov budú v tejto databáze zahešované.

Databáza bude teda pozostávať z troch tabuliek:

- `exercises` (tabuľka cvičení) - obsahuje unikátne ID cvičenia, meno cvičenia, definíciu mimologických symbolov (konštanty, predikáty a funkcie) a vysvetlenie významu predikátových a funkčných symbolov
- `propositions` (tabuľka formúl) - obsahuje unikátne ID formuly, ďalej znenie formuly v prirodzenom jazyku, potom správnu formalizáciu tejto formuly a ID cvičenia, do ktorého táto formula patrí (toto ID referencuje ID cvičenia v tabuľke `exercises`)
- `users` (tabuľka používateľov) - obsahuje unikátne ID používateľa, používateľské meno a zahešované heslo



# Kapitola 4

# Použité technológie

Táto kapitola má za úlohu pre čitateľa zoskupiť a sprehľadniť väčšinu technológií a knižníc, ktoré boli pri implementácii aplikácie použité.

Rozhranie na strane klienta bolo vytvorené pomocou knižnice React, pričom na manažovanie stavu aplikácie sme využili Redux. Na strane servera bol použitý framework Express. Autorizácia používateľov je vyriešená pomocou JSON Web Tokens.

Aplikácia používa okrem technológií a knižníc opísaných v tejto kapitole aj dokazovač Vampire, pomocou ktorého vyhodnocuje riešenia formalizačných cvičení. Softvéru Vampire a jeho použitiu v tejto bakalárskej práci je dedikovaná samostatná kapitola 5.

## 4.1 React

React je javascriptová knižnica [3], ktorá umožňuje deklaratívnym spôsobom vytvárať interaktívne používateľské rozhrania. Výhoda pri práci s touto knižnicou spočíva predovšetkým v tom, že logika renderovania a aktualizovania jednotlivých elementov používateľského rozhrania (v súvislosti s knižnicou React tieto elementy nazývame *komponenty*) je oddelená od deklarovania toho, čo sa má používateľovi zobraziť a za akých podmienok.

Používateľské rozhranie vytvorené pomocou knižnice React sa skladá z *komponentov*. *Komponent* je enkapsulovaná trieda, ktorá spravuje svoj vlastný stav a má definovanú metódu `render`, ktorá vracia JSX element. JSX element je výraz, resp. špeciálny typ syntaxe, ktorý vytvorí React element. Príklad:

```
1 const el = <h3>Current time: { getCurrentTime() }</h3>;
```

Do množinových zátvoriek v JSX elemente môžeme namiesto `getCurrentTime()` na-

písať ľubovoľný platný výraz v JavaScripte, napríklad matematický výraz, výraz s ternárnym operátorom pre podmienky alebo aj iný JSX element.

Jednotlivé React komponenty v aplikácii dokopy tvoria hierarchickú štruktúru alebo strom, kde komponent na vyššej úrovni v strome je poskladaný z komponentov na nižšej úrovni. Komponenty môžu byť *bezstavové* alebo *stavové*.

*Bezstavové komponenty* (ukážka 4.1) dostanú potrebné dáta od rodičovského komponentu ako argument v konštruktore `props`, ktorý je obyčajný JSON objekt. Bezstavové komponenty si neudržiavajú žiaden vnútorný stav vo vlastnosti `this.state`.

```
1  class StatelessComponent extends React.Component {
2      constructor(props) {
3          super(props);
4          this.txt = props.txt;
5      }
6
7      render() {
8          return <h1>{ this.txt }</h1>;
9      }
10 }
11
12 ReactDOM.render(
13     <StatelessComponent txt='Hello world!' />,
14     document.getElementById('root')
15 );
```

Listing 4.1: Príklad jednoduchého bezstavového React komponentu a jeho použitie

*Stavové komponenty* taktiež môžu dostať v argumente `props` nejaké dáta, avšak navyše si vo vlastnosti `this.state` udržiavajú svoj vlastný interný stav, ktorý sa nejakým spôsobom inicializuje (napríklad v konštruktore). Následne môže stavový komponent k obsahu svojej vlastnosti `this.state` ľubovoľne pristupovať a takisto ho ľubovoľne meniť. Meniť stav sa môže napríklad na základe vstupu od používateľa prostredníctvom tlačidla alebo nejakého iného elementu umožňujúceho interakciu používateľa s aplikáciou, prípadne si komponent môže pri svojom prvotnom vygenerovaní vyžiadať nejaké dáta zo servera a uložiť ich do `this.state`.

Nepriamy prístup k vlastnosti `this.state` nejakého komponentu môžu mať komponenty, ktoré sú jeho potomkami (t.j. také komponenty, ktoré sa v strome všetkých React komponentov aplikácie nachádzajú v podstrome, ktorého koreň je nami uvažovaný sta-

vový komponent). Dáta z `this.state` im totiž tento komponent môže podsunúť ako `props`, a takisto im môže aj podsunúť funkcie, ktoré vlastnosť `this.state` upravujú. Iné komponenty prístup k `this.state` nemajú. Ak chceme mať dáta zo stavu komponentu prístupné aj mimo jeho potomkov (a používame iba React bez Reduxu), musíme správu týchto dát premiestniť na vyššie miesto v hierarchii komponentov, aby sa v podstrome nachádzali všetky komponenty, v ktorých chceme k týmto dátam pristupovať.

Každý komponent má definovanú metódu `render`, ktorá sa zavolá vždy pri prvotnom vygenerovaní komponentu v rozhraní a následne pri každom rerenderovaní komponentu. Komponent sa rerenderuje v prípade, keď sa nejakým spôsobom zmení stav komponentu uložený v `this.state` (v prípade stavového komponentu) alebo sa aktualizujú argumenty konštruktora `props`, ktoré mu boli odovzdané z rodičovského komponentu.

Komponenty v našej aplikácii nie sú písané spôsobom, ktorý vidíme na ukážke 4.1. Od verzie Reactu 16.8 je možné manipulovať so stavom elementu prostredníctvom tzv. React Hooks, ktoré výrazne zjednodušujú programovanie stavových komponentov. Namiesto písania komponentov vo forme javascriptových tried s konštruktorom tentoraz stačí celý komponent napísať iba ako javascriptovú funkciu, ktorá vracia to, čo by v klasickom komponente vrátila metóda `render` (ukážka 4.2). Funkcia v prípade potreby môže takisto dostať argument `props`.

```
1  function StatefulComponent() {
2      // React Hook
3      const [txt, setTxt] = useState('');
4
5      return (
6          <div>
7              <input
8                  value={txt}
9                  onChange={(e) => setTxt(e.target.value)}
10             />
11             <p><b>Current value: </b>{ txt }</p>
12         </div>
13     );
14 }
```

Listing 4.2: Stavový komponent, ktorý vygeneruje jednoduché textové pole a pod ním text, ktorý sa mení na základe obsahu textového poľa

React umožňuje renderovať obsah iba jednej stránky, takže v prípade webových ap-

likácií s viacerými podstránkami je potrebné používať dodatočné knižnice určené na routovanie, teda výber podstránok na základe cesty v URL, pod ktorým webový prehliadač pristupuje k aplikácii. V našej aplikácii bola pre tento účel použitá knižnica React Router, ktorá poskytuje špecializované React komponenty, ktoré umožňujú vytvárať React aplikácie pozostávajúce z viacerých stránok. Takýmito komponentmi sú napríklad `BrowserRouter`, `Switch`, `Route` alebo `Redirect`. Ich použitie v rámci zdrojového kódu React aplikácie je v princípe podobné ako pri iných React komponentoch.

Ako môžeme vidieť v príklade 4.3, každému komponentu `Route` v jeho `props` špecifikujeme konkrétnu cestu v URL a React komponent, ktorý sa má pri vstupe na danú URL vyrenderovať.

```
function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Route exact path='/' component={HomePage} />
        <Route path='/login' component={LoginPage} />
      </Switch>
    </BrowserRouter>
  );
}
```

Listing 4.3: Routovanie pomocou knižnice React Router

## 4.2 Redux

Ak je potrebné, aby bol stav nejakého React komponentu prístupný vo všetkých ostatných komponentoch aplikácie, musíme manažovanie týchto dát presunúť do koreňového komponentu v strome komponentov. Toto môže byť trochu nepohodlné v aplikáciách, kde je potrebné zdieľať nejaké dáta medzi veľkým množstvom komponentov, pretože komponenty, ktoré sú vyššie v hierarchii, sa môžu pomerne rýchlo stať neprehľadnými a komplikovanými.

Redux je knižnica [1], ktorá centralizuje manažovanie a aktualizovanie stavu aplikácie. Umožňuje, aby dáta boli prístupné na všetkých takých miestach aplikácie, na ktorých je to potrebné, čím sa dá vyhnúť kumulovaniu týchto dát v komponentoch, ktoré sú vyššie v hierarchii komponentov, a tým zneprehľadňovaniu zdrojového kódu. Použiť Redux je výhodné aj vtedy, ak je logika pre aktualizovanie stavu pomerne zložitá.


Obr. 4.1: Tok dát v aplikácii

V aplikácii používajúcej Redux tečú dáta nasledovným spôsobom (obr. 4.1):

(i) Aplikácia je v určitom čase v nejakom stave (pamätá si nejaké konkrétne dáta). Tento stav je uložený v objekte, ktorý sa nazýva `store`.

(ii) Komponenty pristúpia k aktuálnemu stavu aplikácie uloženému v objekte `store` a na základe neho sa rozhodnú, aký konkrétny obsah sa má zobraziť používateľovi.

(iii) Používateľ interaguje s rozhraním (napr. stlačí tlačidlo alebo napíše text) a vyvolá sa akcia. Následne sa na základe aktuálneho stavu a vyvolanej akcie prepíše stav aplikácie v objekte `store` novými dátami. Potom sa znovu prejde do bodu (i).

V našej aplikácii boli okrem samotného Reduxu použité aj knižnice React-Redux a Redux Toolkit, ktoré zjednodušujú komunikáciu medzi React komponentmi a stavom aplikácie a umožňujú predchádzať bežným chybám pri práci s Reduxom.

## 4.3 Express

Express (alebo Express.js) [8] je framework pre Node.js, pomocou ktorého bola vytvorená strana servera v našej aplikácii.

```
1  const express = require('express');
2  const server = express();
3
4  const PORT = process.env.PORT;
5
6  // middleware
7  server.use(express.json());
8
9  server.get('/', (req, res) => {
10     res.status(201).json({ greeting: 'Hello world!' });
```

```
11 });
12
13 server.listen(PORT, () => {
14     console.log(`Server started listening on port ${PORT}`)
          ;
15 });
```

Listing 4.4: Jednoduchý príklad použitia Express.js

Jednoduchý príklad použitia Express.js môžeme vidieť na ukážke 4.4. Na začiatku sa vytvorí inštancia `express`. Následne jej pomocou metódy `use` môžeme pridať rôzny middleware (napr. parsery alebo funkcie pre logovanie alebo autorizáciu). Metódami `get`, `post`, `put` a `delete` zadáme, akým spôsobom má na špecifikovanej ceste na serveri aplikácia odpovedať na príslušné HTTP požiadavky. Pomocou metódy `listen` spustíme server na požadovanom porte. Jednoduchá aplikácia na ukážke 4.4 používa JSON parser ako middleware a pri HTTP požiadavke GET na cestu `'/'` aplikácia odpovie so status kódom 201, posielajúc naspäť JSON objekt s dátami. Port, na ktorom aplikácia beží, je získaný z `.env` súboru.

## 4.4 JSON Web Tokens

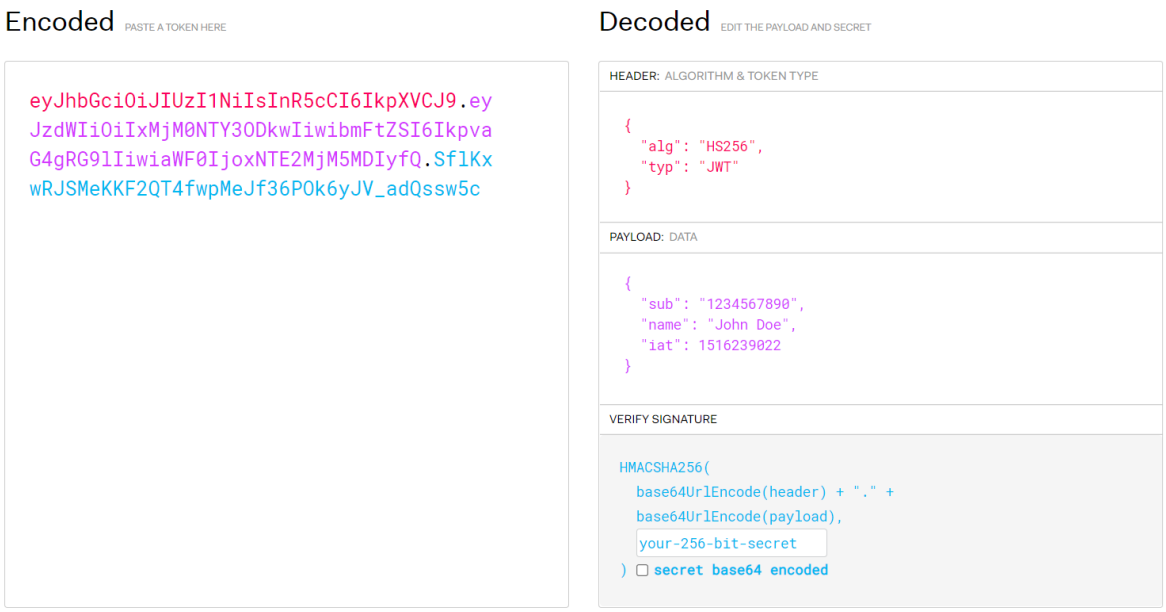
JSON Web Tokens (JWT [4]) umožňujú autorizovať používateľa po zaslaní HTTP požiadavky na niektorú chránenú URL na serveri. Autorizácia používateľov na serveri pomocou tokenov (narozdiel od autorizácie pomocou sessions) nevyžaduje ukladanie dodatočných dát na strane servera a spolieha sa iba na tokeny.

Keď používateľ vyplní prihlasovacie údaje a server ich overí, vytvorí sa pre tohto používateľa po úspešnom prihlásení token, ktorý sa odošle na stranu klienta. Server si tento token neukladá, uloží si ho iba klient. Následne klient pri každom prístupe na chránenú URL servera pripojí k požiadavke tento token. Token zapíše do požiadavky do hlavičky `Authorization` vo formáte:

```
1 Authorization: Bearer <token>
```

Po prijatí požiadavky server overí platnosť tokenu a na základe výsledku overovania povolí alebo zamietne prístup.

Samotný token má štruktúru `X.Y.Z`, kde časti `X`, `Y`, `Z` sú reťazce v kódovaní Base64-URL a v tokene sú oddelené bodkou. Ich význam je nasledovný:




Obr. 4.2: Štruktúra tokenu JWT

(i) `X` - hlavička vo formáte JSON, v ktorej sú obvykle zakódované dva údaje:

1. reťazec s menom kryptografického hashovacieho algoritmu, ktorý bol použitý na vytvorenie podpisu v tokene
2. reťazec popisujúci typ tokenu (v tomto prípade JWT)

(ii) `Y` - sprievodné dáta vo formáte JSON, ako napríklad informácie o používateľovi (napr. jeho identifikačné číslo a meno), čas vytvorenia/expirácie tokenu, prípadne ďalšie údaje (v prípade dôverných údajov je nutné, aby boli vopred zašifrované)

(iii) `Z` - podpis

Podpis je vygenerovaný pomocou algoritmu špecifikovaného v hlavičke. Vytvára sa na základe zakódovanej hlavičky, zakódovaných sprievodných dát a tajného kľúča uloženého na serveri. Ak sú sprievodné dáta alebo hlavička akýmkoľvek spôsobom modifikované, podpis už ďalej nebude so zvyškom tokenu korešpondovať a pri procese overovania tokenu na serveri bude zistené, že s údajmi bolo nejakým spôsobom manipulované a server im nemôže dôverovať. Ak sa podpis podarí overiť, server môže dáta z tokenu použiť na autorizáciu operácií.

Príklad JWT tokenu vidíme na obrázku 4.2. Uvedený debugger pre JWT tokeny je dostupný na stránke `https://jwt.io/` ([4]).

# Kapitola 5

# Práca s dokazovačom Vampire

Kľúčovou časťou našej aplikácie je samotné vyhodnocovanie formalizačných cvičení. Ako už bolo spomenuté v kapitole 3, v databáze s cvičeniami je ku každému tvrdeniu uložená aj jeho správna formalizácia. Táto formalizácia ale takmer nikdy nie je jediným akceptovateľným riešením. Ak totiž správna formalizácia obsahuje napríklad symbol pre premennú, tak už len z toho dôvodu, že symbolov pre premenné je nekonečne veľa, existuje nekonečne veľa správnych riešení, ktoré sa líšia iba v menách premenných. Okrem toho ale môže existovať veľa formalizácií, ktoré sú správnymi riešeniami, ale líšia sa poradím predikátov, prípadne je odlišná celá štruktúra formuly, ale so správnou formalizáciou sú ekvivalentné. Je potrebné, aby boli aj takéto riešenia akceptované. Preto potrebujeme overiť, či je študentovo riešenie ekvivalentné správnemu riešeniu uloženému v databáze.

Pre jednoduchosť vyjadrovania si označme riešenie študenta ako formulu $R$ a správne riešenie uložené v databáze ako formulu $S$. V kapitole 1 sme si definovali pojem ekvivalencie formúl. Naším cieľom je zistiť, či je formula $R$ pravdivá práve vtedy, keď je pravdivá formula $S$, teda či sú formuly ekvivalentné. Pre ekvivalentné formuly platí $R \iff S$, čo teda znamená, že $R \Rightarrow S$ a zároveň $S \Rightarrow R$.

Ak formuly nie sú ekvivalentné, vieme nájsť konkrétnu štruktúru, pre ktorú jedna z formúl $R$ a $S$ platí, ale druhá nie. V princípe môžu nastať dve situácie:

(i) Neplatí ani jedna z implikácií $R \Rightarrow S$ a $S \Rightarrow R$. V takomto prípade vieme nájsť minimálne dva kontrapríklady. Jeden z nich je štruktúra, v ktorej platí formula $R$ a neplatí $S$. Druhý kontrapríklad je zase štruktúra, v ktorej naopak platí formula $S$, ale neplatí formula $R$.

(ii) Neplatí práve jedna z implikácií $R \Rightarrow S$ a $S \Rightarrow R$. V tomto prípade vieme nájsť kontrapríklady iba pre jednu implikáciu, čiže minimálne jeden kontrapríklad.

Overiť, či sú dve formuly ekvivalentné, je nerozhodnuteľný problém, ako sme spomenuli v kapitole 1. Napriek tomu ale existuje viacero automatických dokazovačov, ktoré sa snažia tento problém riešiť. Teoreticky, ak sa vyplývanie dá dokázať, dôkaz sa dá nájsť v konečnom čase, ale ak sa dokázať nedá, nemusí sa to dať zistiť a neúspešné hľadanie dôkazu môže bežať do nekonečna.

Na overovanie bol teda použitý softvér Vampire [13], ktorý vie zo zadaných axióm a predpokladov v logike prvého rádu automaticky dokazovať špecifikované teorémy, prípadne zistiť, že to nie je možné. Navyše vie nájsť konečné modely - kontrapríklady, ktoré nám názorne ukážu, prečo sa teoréma z uvedených predpokladov nedá dokázať. Napriek nerozhodnuteľnosti týchto problémov je Vampire jeden z najúspešnejších dokazovačov, čo je podložené jeho opakovanými úspechmi v súťaži CADE ATP System Competition ([9]). Vampire počíta svoju odpoveď v obmedzenom časovom limite, ktorý je predvolený na 60 sekúnd, avšak v prípade potreby ho vieme upraviť. V našej aplikácii je tento časový limit znížený na 10 sekúnd, pretože dokazovač Vampire je veľmi rýchly a na vzorových príkladoch zakaždým zbehol za pár stotín sekundy.

## 5.1 Spustenie z príkazového riadka

Softvér Vampire sa v predvolenom režime spúšťa z príkazového riadka nasledovným príkazom:

```
vampire problem.p
```

Súbor `problem.p` obsahuje logický problém v TPTP formáte, ktorý si bližšie opíšeme v ďalšej podkapitole. Výpočet beží v časovom limite 60 sekúnd. Výsledok sa vypíše na štandardný výstup.

Ak chceme zmeniť predvolený časový limit (napr. na 300 sekúnd), spustíme Vampire spolu s prepínačom `-t` a časom v sekundách:

```
vampire -t 300 problem.p
```

V prípade, že sme si istí, že sa teoréma z daných predpokladov dokázať nedá, a teda existuje kontrapríklad, doplnením `-sa fmb` docielime výpis kontrapríkladu:

```
vampire -sa fmb problem.p
```

Vampire má aj mnoho iných rôznych nastavení, nám stačí obmedziť sa iba na tieto.

Pri overovaní správnosti študentovho riešenia budeme spúšťať Vampire viackrát. Ak chceme zistiť, či sú formuly $R$ a $S$ ekvivalentné, musíme overiť, či je možné dokázať

formulu $S$ z predpokladu $R$, a naopak, či je možné dokázať formulu $R$ z predpokladu $S$. Vampire teda spustíme dvakrát, najskôr pre implikáciu $R \Rightarrow S$ a následne pre $S \Rightarrow R$. V prípade, že sú obe implikácie platné, sú formuly ekvivalentné a študentovo riešenie je správne. Ak pre niektorú implikáciu dostaneme zápornú odpoveď, spustíme Vampire pre túto neplatnú stranu znova, ale tentoraz s prepínačom `-sa fmb`, aby našiel kontrapríklad. Vampire vypíše tento kontrapríklad vo formáte TPTP. V prípade, že neplatí ani jedna implikácia, spustíme Vampire postupne pre obe strany, aby sme našli dva kontrapríklady.

Keďže je časový limit v našej aplikácii obmedzený na 10 sekúnd, teoreticky sa môže stať, že Vampire sa za tento čas k žiadnemu výsledku nedostane. Toto je ale dosť nepravdepodobné, pretože formuly, ktoré sa vyskytujú vo formalizačných cvičeniach, sú pomerne jednoduché.

## 5.2 TPTP formát

Softvér Vampire akceptuje vstupné dáta v TPTP formáte [11] a takisto v tomto formáte aj vypisuje výsledky. Súbor v TPTP formáte má príponu `.p`. Spôsob zapisovania formúl v TPTP syntaxi si ukážeme a vysvetlíme na jednoduchom príklade.

Zadefinujme si množinu symbolov pre konštanty $\mathcal{C}$, množinu predikátových symbolov $\mathcal{P}$ a množinu funkčných symbolov $\mathcal{F}$ nasledovne:

$$\mathcal{C} = \{\ \texttt{jozko},\ \texttt{evka},\ \texttt{A},\ \texttt{B},\ \texttt{C},\ \texttt{D},\ \texttt{E},\ \texttt{Fx}\ \}$$
$$\mathcal{P} = \{\ \texttt{student},\ \texttt{znamka},\ \texttt{hodnotenie}\ \}$$
$$\mathcal{F} = \emptyset$$

Predikátové symboly `student` a `znamka` majú jeden argument, symbol `hodnotenie` má dva argumenty. Formula $\texttt{student}(x)$ znamená, že $x$ je študent, formula $\texttt{znamka}(y)$ znamená, že $y$ je známka a $\texttt{hodnotenie}(x, y)$ znamená, že $x$ je hodnotený známkou $y$. Chceme formalizovať jednoduché tvrdenie:

„Každý študent je hodnotený nejakou známkou. “

Správna formalizácia tohto tvrdenia môže mať jednu z nasledovných dvoch podôb:

$$(\forall x)(\texttt{student}(x)\ \Rightarrow\ (\exists y)(\texttt{znamka}(y)\ \wedge\ \texttt{hodnotenie}(x, y)))$$

$$\neg(\exists x)(\texttt{student}(x)\ \wedge\ (\forall y)(\neg\texttt{znamka}(y)\ \vee\ \neg\texttt{hodnotenie}(x, y)))$$

V TPTP syntaxi sa obe formuly zapisujú takto:

```
! [X] : (student(X) => ? [Y] : (znamka(Y) & hodnotenie(x,y)))
```

```
~( ? [X] : (student(X) & ! [Y] : (~ znamka(Y) | ~ hodnotenie(x,y))))
```

Je dôležité poznamenať, že všetky premenné sa musia začínať veľkým písmenom. Uvedený zápis avšak stále nie je kompletný. Vo vstupe, ktorý má byť v TPTP formáte, treba špecifikovať, ktoré formuly sú axiómy, a ktoré formuly sú vety, ktoré treba dokázať. Každá formula teda bude ešte doplnená ďalšími informáciami, a výsledný zápis jednej formuly je takýto:

```
fof(<name>, <role>, <formula>).
```

Položka `name` je meno formuly. Položka `role` nám určuje, akú rolu zohráva táto formula v probléme (my budeme používať iba roly `axiom` a `conjecture`, aby Vampire pomocou formuly s rolou `axiom` dokázal formulu s rolou `conjecture`). Položka `formula` je formula v TPTP syntaxi (ako je uvedené vyššie).

## 5.3 Výstup

Aby sme overili, či sú formuly ekvivalentné, potrebujeme overiť platnosť oboch implikácií. Vstup pre overenie jednej strany vyzerá takto:

```
1  fof(a,axiom,
2      ! [X] : (
3          student(X) => ? [Y] : (
4              znamka(Y) & hodnotenie(X,Y)
5          )
6      )
7  ).
8
9  fof(b,conjecture,
10     ~ (
11         ? [X] : (
12             student(X) & ! [Y] : (
13                 ~ znamka(Y) | ~ hodnotenie(X,Y)
14             )
15         )
16     )
```

```
% SZS output end Proof for example1
% ------------------------------
% Version: Vampire 4.5.1 (commit 9d08e223 on 2021-01-26 16:29:53 +0000)
% Termination reason: Refutation

% Memory used [KB]: 4861
% Time elapsed: 0.031 s
% ------------------------------
% ------------------------------
```

Obr. 5.1: Časť výstupu softvéru Vampire

```
17 ) .
```

Po spustení programu Vampire na tomto konkrétnom vstupe sa nám vypíše samotný dôkaz formuly `b` pomocou formuly `a`, avšak tento dôkaz pre nás teraz nie je dôležitý. Dôležitejšie pre nás sú komentáre na konci výstupu (obr. 5.1), ktoré hovoria o výsledku výpočtu, konkrétne riadok

```
% Termination reason: Refutation
```

kde `Refutation` pre nás znamená, že Vampire vedel z prvej formuly odvodiť druhú. Ak by sme formuly vymenili a dokazovala by sa opačná implikácia, verdikt by bol rovnaký, pretože formuly sú ekvivalentné.

Ak by Vampire dospel k výsledku, že sa to dokázať nedá, a teda existuje kontrapríklad, vypísal by namiesto toho `Satisfiable`. V prípade, že by Vampire v danom časovom limite nedospel k žiadnemu výsledku, ukončil by výpočet s dôvodom `Time limit`.

Majme teraz príklad formúl, pre ktoré dostaneme výsledok `Satisfiable`:

$$(\forall x)(\texttt{student}(x) \Rightarrow (\exists y)(\texttt{znamka}(y) \wedge \texttt{hodnotenie}(x, y)))$$

$$(\forall x)(\texttt{student}(x) \wedge (\exists y)(\texttt{znamka}(y) \wedge \texttt{hodnotenie}(x, y)))$$

Môžeme vidieť, že druhá formula je istým spôsobom striktnejšia, pretože je pravdivá iba v takých štruktúrach, v ktorých sú všetky entity v doméne študenti. Prvá formula povoľuje v doméne výskyt aj takých entít, ktoré študentmi nie sú.

Po prevode do TPTP formátu a spustení programu Vampire s prepínačom `-sa fmb` by sa na výstupe mal objaviť kontrapríklad - štruktúra, v ktorej platí prvá formula, ale neplatí druhá. V tomto konkrétnom prípade je ale kontrapríklad štruktúry $(D, i)$ celkom triviálny: doména $D$ je neprázdna (to vieme z definície), ale množiny $i(\texttt{student})$, $i(\texttt{znamka})$, $i(\texttt{hodnotenie})$ budú všetky prázdne. Vampire teda vypíše tento triviálny

kontrapríklad, v ktorom sú ale všetky relácie prislúchajúce jednotlivým predikátom prázdne, takže nevypíše nič, pretože nie je čo vypísať.

Aby sme teda dostali výpovednejší kontrapríklad, upravíme vstup nasledovne:

```
1  fof(a,axiom,
2      ! [X] : (
3          student(X) => ? [Y] : (
4              znamka(Y) & hodnotenie(X,Y)
5          )
6      )
7  ).
8
9  fof(c,axiom,
10     student(jozko)
11 ).
12
13 fof(b,conjecture,
14     ! [X] : (
15         student(X) & ? [Y] : (
16             znamka(Y) & hodnotenie(X,Y)
17         )
18     )
19 ).
```

Týmto vynútime, aby množina študentov nebola prázdna, a Vampire nám vypíše iný kontrapríklad. Na obrázku 5.2 môžeme vidieť, že kontrapríklad je takisto vo formáte TPTP. Opisuje štruktúru, v ktorej máme jediného študenta (s menom `jozko`) a jedinú známku (s vygenerovaným označením `fmb_$i_2`). Ďalej v štruktúre platí `hodnotenie(jozko, fmb_$i_2)`. Je zjavné, že prvá formula je v tejto štruktúre pravdivá, pretože každý študent je hodnotený známkou. Avšak druhá formula neplatí, pretože nie všetky entity v doméne sú študentmi - entita `fmb_$i_2` totiž nie je študent.

```
TRYING [1]
TRYING [2]
Finite Model Found!
% SZS status CounterSatisfiable for example3
% SZS output start FiniteModel for example3
tff(declare_$i,type,$i:$tType).
tff(declare_$i1,type,jozko:$i).
tff(declare_$i2,type,fmb_$i_2:$i).
tff(finite_domain,axiom,
      ! [X:$i] : (
         X = jozko | X = fmb_$i_2
      ) ).

tff(distinct_domain,axiom,
         jozko != fmb_$i_2
).

tff(declare_student,type,student: $i > $o ).
tff(predicate_student,axiom,
           student(jozko)
         & ~student(fmb_$i_2)

).

tff(declare_znamka,type,znamka: $i > $o ).
tff(predicate_znamka,axiom,
           ~znamka(jozko)
         & znamka(fmb_$i_2)

).

tff(declare_hodnotenie,type,hodnotenie: $i * $i > $o ).
tff(predicate_hodnotenie,axiom,
           ~hodnotenie(jozko,jozko)
         & hodnotenie(jozko,fmb_$i_2)
         & ~hodnotenie(fmb_$i_2,jozko)
         & ~hodnotenie(fmb_$i_2,fmb_$i_2)

).

% SZS output end FiniteModel for example3
% ------------------------------
% Version: Vampire 4.5.1 (commit 9d08e223 on 2021-01-26 16:29:53 +0000)
% Termination reason: Satisfiable

% Memory used [KB]: 4861
% Time elapsed: 0.050 s
% ------------------------------
% ------------------------------
```

Obr. 5.2: Výpis kontrapríkladu pomocou softvéru Vampire

# Kapitola 6

# Implementácia

## 6.1 Strana klienta

Zdrojový kód React aplikácie je organizovaný do dvoch hlavných balíčkov `public` a `src`. Obsah balíčka `public` bol celý vygenerovaný pri inicializácii React aplikácie, a až na drobné detaily (ako je napríklad titulok stránky v súbore `index.html`) sme ho vôbec nemenili. V súbore `index.html` je v tele stránky vložený jediný `div` element s `id="root"`, do ktorého sa renderuje hlavný React komponent `App`.

Zdrojový kód je teda umiestnený v balíčku `src`. Na najvyššej úrovni sú pre nás dôležité tri súbory - `index.js`, `App.js` a `config.js`. Okrem týchto súborov sú tu definované napríklad aj základné štýly, ale tým sa zaoberať nemusíme, pretože v rámci komponentov boli takmer všetky štýly popridávané priamo v definíciách komponentov pomocou knižnice React Bootstrap.

V súbore `index.js` máme príkaz, ktorým sa vyrenderuje ReactDOM s hlavným komponentom `App`:

```
1 ReactDOM.render(
2   <React.StrictMode>
3     <Provider store={store}>
4       <App />
5     </Provider>
6   </React.StrictMode>,
7   document.getElementById('root')
8 );
```

Komponent `Provider` nám zabezpečí prepojenie React aplikácie s objektom `store`

z knižnice Redux, v ktorom bude uložený celý stav aplikácie so všetkými náležitými dátami.

V súbore `App.js` je zadeklarovaný hlavný komponent `App`. Vyrenderuje navigáciu a takisto rieši problém routovania v našej aplikácii pomocou knižnice React Router:

```
<Switch>
  <Route exact path="/" component={ExerciseList} />
  <Route exact path="/login" component={LoginForm} />
  <Route path="/solve/:id" component={SolveExercise} />

  <ProtectedRoute
    exact path="/add"
    component={AddExercise}
  />

  <Route path="*" component={() => {
    <Alert variant="danger">404 Not Found</Alert>
  }} />
</Switch>
```

Komponent `ProtectedRoute` je nami definovaný špeciálny komponent pre URL, ktoré sa majú používateľovi zobraziť iba po prihlásení. V prípade, že používateľ prihlásený nie je a v prehliadači pristupuje na URL pre pridanie nového cvičenia, komponent `ProtectedRoute` ho automaticky presmeruje na stránku prihlásenia. Po úspešnom prihlásení bude používateľ presmerovaný naspäť na stránku pre pridávanie cvičenia.

Úlohou `config.js` je vyexportovať premenné prostredia, ktorých hodnoty sú používané v aplikácii. V našom prípade máme na strane klienta iba jednu takúto premennú - URL servera, ktorému klient posiela požiadavky.

Väčšina zdrojového kódu aplikácie je umiestnená v balíčkoch `components` a `redux`, ktoré od seba oddeľujú deklarácie komponentov (v balíčku `components`) a manažovanie stavu aplikácie (v balíčku `redux`).

### 6.1.1 Komponenty

Balíček `components` sa ďalej rozvetvuje na tri balíčky, v ktorej sú všetky naše React komponenty definované:

(i) `addExercise` - deklarácie komponentov tvoriacich podstránku pre pridanie no-

vého cvičenia do databázy

(ii) `solveExercise` - deklarácie komponentov tvoriacich podstránku so zoznamom všetkých cvičení a takisto podstránku pre riešenie konkrétneho cvičenia

(iii) `login` - deklarácia komponentu pre prihlásenie používateľa a deklarácia komponentu `ProtectedRoute`

Keďže používame na manažovanie stavu aplikácie knižnicu Redux, všetky komponenty sú bezstavové a všetky dáta sú im v prípade potreby odovzdané cez argument `props`. Niektoré komponenty prístup k stavu aplikácie nepotrebujú, avšak ostatným komponentom bolo treba dodefinovať funkciu `mapStateToProps` a objekt `mapDispatchToProps`, ktoré im do argumentu `props` vložia potrebné dáta z Redux objektu `store` a takisto akcie, ktoré komponentom umožnia meniť stav aplikácie. Uvedieme príklad (zo súboru `addExercise/Proposition.js`):

```
1  const mapStateToProps = (state, ownProps) => {
2    return {
3      informalValue: selectInformalValue(state, ownProps.i),
4      formalization: selectFormalization(state, ownProps.i)
5    };
6  };
7
8  const mapDispatchToProps = {
9    remove: removeProposition,
10   update: updateInformalValue
11 };
12
13 export default connect(
14   mapStateToProps, mapDispatchToProps
15 )(Proposition);
```

Funkcia `mapStateToProps` odovzdá komponentu potrebné dáta z aktuálneho stavu. Výsledkom tejto funkcie je objekt s dátami, ktoré sa následne zahrnú do argumentu `props` daného komponentu, a teda komponent bude mať k týmto dátam prístup. Funkcie `selectInformalValue` a `selectFormalization` sú tzv. *selektory*, pomocou ktorých sa z objektu `state` tieto dáta vyextrahujú. Použitie selektorov by sme ale mohli považovať za nepovinné, pretože k potrebným dátam je možné pristúpiť aj priamo pomocou objektu `state`.

Objekt `mapDispatchToProps` zasa medzi `props` pridá akcie, pomocou ktorých sa bude dať zmeniť stav aplikácie.

Pomocou príkazu `connect` nakoniec prepojíme React komponent `Proposition` s funkciou `mapStateToProps` a objektom `mapDispatchToProps`.

### 6.1.2 Manažovanie stavu aplikácie

V priečinku `redux` sa nachádza skupina súborov, ktorých mená majú sufix `Slice.js`. Celá štruktúra stavu aplikácie je pomerne komplikovaná, preto je stav rozdelený do menších celkov, z ktorých každý je opísaný v jednom súbore:

(i) `exercisesSlice.js` - časť stavu, ktorá prináleží podstránke so zoznamom všetkých cvičení. Obsahuje náhľady všetkých cvičení v databáze.

(ii) `solveExerciseSlice.js` - časť stavu, ktorá prináleží podstránke pre vyriešenie konkrétneho cvičenia. Obsahuje samotné cvičenie, riešenia zadané študentom a takisto vyhodnotenia riešení.

(iii) `addExerciseSlice.js` - časť stavu, ktorá prináleží podstránke pre vloženie nového cvičenia do databázy. Obsahuje cvičenie, ktoré je práve vytvárané.

(iv) `userSlice.js` - časť stavu, v ktorej sú uložené dáta o stave prihlásenia

V súbore `store.js` sú všetky tieto časti stavu zlúčené do jedného objektu `store`.

Uvedené štyri súbory s časťami stavu aplikácie obsahujú deklarácie nasledovných položiek:

(i) začiatočný stav

(ii) *reducery* - funkcie, ktoré menia stav po vyvolaní akcií na základe vstupu od používateľa

(iii) *extra reducery* - funkcie, ktoré menia stav po vyvolaní asynchrónnych akcií, napríklad pri vyžiadaní nejakých dát zo servera

(iv) *selektory* - funkcie, ktoré zo stavu vyextrahujú nejaké údaje

Ako bolo spomenuté v predchádzajúcej časti, používanie selektorov nie je nutné. Avšak v našej aplikácii je logika výberu potrebných dát trochu komplikovanejšia. Používateľ totiž do niektorých textových polí zadáva vstupy, ktoré sú v špeciálnej syntaxi, a teda je ich potrebné skontrolovať prislúchajúcim parserom. Zároveň aplikácia používateľovi v reálnom čase vypisuje chybové správy obsahujúce informácie o syntaktickej

chybe vo vstupe. Ukladať si tieto informácie do stavu je trochu neprehľadné. Vstup sa teda kontroluje až pri extrahovaní dát zo stavu, teda v selektoroch. Na parsovanie týchto vstupov bola použitá knižnica parserov `@fmfi-uk-1-ain-412/js-fol-parser` (dostupná na GitHube).

## 6.2 Strana servera

V súbore `server.js` sa inicializuje inštancia `express`, pridá sa jej middleware, špecifikujú sa ďalšie cesty na serveri a spustí sa samotný server.

Úlohou modulu `config.js` je vyexportovať všetky premenné prostredia, ktoré sú potrebné v našej aplikácii. Medzi tieto premenné patria napríklad údaje pre prístup k PostgreSQL databáze, číslo portu a rôzne tajné kľúče, ktoré potrebujeme pri vytváraní tokenov alebo hešovaní niektorých dát.

V priečinku `routes` sú moduly, v ktorých sú definované spôsoby spracovania požiadaviek na rôznych cestách na serveri. Cesta pre pridanie cvičenia do databázy je chránená a prístupná iba prihláseným používateľom. Z tohto dôvodu je definovaná v samostatnom súbore `add.js`, aby sme jej mohli pridať middleware pre autorizáciu pomocou tokenov.

V priečinku `db` sú moduly, v ktorých sú zadefinované funkcie pre prácu s databázou. Takisto sa tu nachádza modul `db.js` pre spojenie s našou PostgreSQL databázou a súbor `db_init.sql`, ktorý môžeme použiť pri prvotnom inicializovaní databázy.

Funkcie pre vyhodnocovanie cvičení nájdeme v priečinku `helpers`. Tento priečinok obsahuje súbor `formula_classes.js`, v ktorom sú definované triedy reprezentujúce stavebné prvky formúl, čiže premenné, koštanty, predikátové atómy, aplikácie funkcií, predikátový symbol rovnosti a všetky logické symboly, ktoré patria do jazyka logiky prvého rádu. Použili sme návrhový vzor Composite, pretože všetky tieto triedy musia mať rovnaké rozhranie. Zložité formuly budeme pomocou týchto tried vytvárať komponovaním do stromovej štruktúry. Všetky tieto triedy majú definovanú metódu `toVampire`, ktorá vráti reťazcovú reprezentáciu daného objektu v syntaxi TPTP.

### 6.2.1 Vyhodnocovanie cvičení

Ako sme už spomínali v kapitole 2, študenti zapisujú formuly v dohodnutej syntaxi, ktorú rozoznáva parser `@fmfi-uk-1-ain-412/js-fol-parser`. Softvér Vampire, ktorým budeme zisťovať, či je študentovo riešenie ekvivalentné so správnym riešením v databáze, akceptuje vstupné dáta iba vo formáte TPTP. Za účelom vyhodnotenia štu-

dentovho riešenia je teda nutné prekonvertovať formuly z nášho dohodnutého jazyka do formátu TPTP. Budeme postupovať v nasledovných krokoch:

1. Vyberieme správne riešenie daného cvičenia z databázy.
2. Obe formuly odovzdáme parseru, ktorý rozoznáva našu syntax. Pri volaní parsera zároveň treba odovzdať argument `factories` so špecifikovaním konkrétnych tried, ktoré prislúchajú jednotlivým stavebným prvkom formuly. My použijeme triedy z modulu `formula_classes.js`. Týmto získame formulu reprezentovanú inštanciami týchto tried.
3. Pomocou metódy `toVampire` obe formuly skonvertujeme do syntaxe TPTP.
4. Vyhodnotíme správnosť riešenia pomocou softvéru Vampire. Táto časť sa nachádza v module `vampire.js` v priečinku `helpers`.

Treba ešte poznamenať, že TPTP syntax má od našej syntaxe formúl odlišné konvencie v pomenovaniach mimologických symbolov (napríklad symboly pre premenné v TPTP musia začínať veľkým písmenom). Tento problém je vyriešený tak, že všetky mimologické symboly jednoducho premenujeme a všetky mená si uložíme do slovníkov. Jeden slovník bude obsahovať dáta v tvare `stare_meno: nove_meno`, druhý zasa `nove_meno: stare_meno`. Vďaka tomu bude možné mená preložiť naspäť.

V prípade nesprávneho riešenia softvér Vampire vypíše kontrapríklad. TPTP syntax ale študentovi pravdepodobne nie je známa, preto je potrebné tento výstup spracovať a vypísať ho spôsobom, ktorému študent rozumie. Pomocou generátora parserov pre JavaScript PEG.js [6] sme teda vytvorili parser pre TPTP syntax. Gramatika pre TPTP syntax [12] je napísaná v súbore `grammar.pegjs`.

# Záver

Cieľom tejto bakalárskej práce bolo vytvoriť klient-server webovú aplikáciu, ktorá študentom umožňuje riešiť formalizačné cvičenia a následne si ich nechať touto aplikáciou vyhodnotiť. Po vyhodnotení sa následne študentovi vypíše informácia o tom, či je jeho formalizácia správna alebo nesprávna. V prípade nesprávneho riešenia je táto informácia doplnená kontrapríkladom, ktorý pomôže študentovi lepšie pochopiť, prečo je jeho riešenie nesprávne.

Samotné vyhodnocovanie správnosti riešenia je realizované na strane servera. Je to tak z dvoch dôvodov. Prvým je, že samotné vyhodnocovanie vykonáva dokazovač Vampire, ktorý na strane klienta samozrejme spúšťať nemôžeme. Druhým dôvodom je utajenie správneho riešenia. Správne riešenie cvičenia je totiž uložené v databáze a keby sme ho zaslali na stranu klienta, hocikto by k nemu mal prístup.

Aplikácia je naprogramovaná v programovacom jazyku JavaScript. Rozhranie na strane klienta bolo vytvorené pomocou knižnice React, pričom na manažovanie stavu aplikácie bola použitá knižnica Redux. Aplikácia bežiaca na strane servera bola vytvorená pomocou frameworku Express.js. Ako databázový systém pre našu aplikáciu bol zvolený PostgreSQL.

Funkcionalita aplikácie na strane klienta bola doplnená aj o rozhranie, ktoré umožňuje prihláseným používateľom vytvárať nové cvičenia. Tieto cvičenia sa následne po odoslaní uložia do databázy s cvičeniami.

V našej aplikácii je ale stále priestor na zlepšovanie. Takýmto ďalším vylepšením by mohlo byť napríklad vylepšenie zobrazovania kontrapríkladov, aby boli zrozumiteľnejšie. Prípadne je možné experimentovať so vstupmi pre program Vampire tak, aby nám zakaždým našiel čo najvýpovednejší kontrapríklad. Takisto by sa mohlo overiť použitie aplikácie na zhromaždených riešeniach domácich úloh z predmetu Matematika (4) – Logika pre informatikov v programe aplikovanej informatiky na FMFI, prípadne by sme aplikáciu mohli prakticky nasadiť pri výučbe.

# Literatúra


[1] Dan Abramov and the Redux documentation authors. Redux. [Citované 2021-05-08] `https://redux.js.org/`.

[2] Dave Barker-Plummer, Robert Dale, Richard Cox, and John Etchemendy. Automated assessment in the internet classroom. In *Proc. AAAI Fall Symp. Education Informatics, Arlington, VA*, 2008.

[3] Facebook Inc. React. [Citované 2021-05-08] `https://reactjs.org/`.

[4] JSON Web Tokens. [Citované 2021-05-08] `https://jwt.io/`.

[5] Ján Kľuka and Jozef Šiška. Prednášky z Matematiky (4) – Logiky pre informatikov. Letný semester 2019/2020. [Citované 2021-01-21] `http://dai.fmph.uniba.sk/courses/lpi/lpi/prednasky/poznamky-z-prednasok.pdf`.

[6] PEG.js. [Citované 2021-05-14] `https://pegjs.org/`.

[7] Isidoros Perikos, Foteini Grivokostopoulou, and Ioannis Hatzilygeroudis. Automatic marking of NL to FOL conversions. In *Proc. of 15th IASTED International Conference on Computers and Advanced Technology in Education (CATE), Napoli, Italy*, pages 227–233, 2012.

[8] StrongLoop, IBM, and other expressjs.com contributors. Express. [Citované 2021-05-08] `https://expressjs.com/`.

[9] Geoff Sutcliffe. Proceedings of the 10th IJCAR ATP System Competition (CASC-J10). [Citované 2021-05-13] `http://www.tptp.org/CASC/J10/Proceedings.pdf`.

[10] Vítězslav Švejdar. Logika: neúplnost, složitost a nutnost. *Praha: Academia*, 2002.

[11] TPTP Format for Problems. [Citované 2021-05-12] `http://www.tptp.org/TPTP/QuickGuide/Problems.html`.

[12] TPTP syntax. [Citované 2021-05-14] `http://www.tptp.org/TPTP/SyntaxBNF.html`.

[13] Vampire. [Citované 2021-05-08] https://vprover.github.io/.

# Príloha A: Elektronická príloha

Prílohou tejto bakalárskej práce je zdrojový kód aplikácie. Obsahuje dva repozitáre:

- `formalization-checker-front-end` - zdrojový kód aplikácie na strane klienta
- `formalization-checker-back-end` - zdrojový kód aplikácie na strane servera

Najaktuálnejšia verzia zdrojového kódu s prípadnými vylepšeniami sa nachádza na GitHube:

- `https://github.com/gombarova/formalization-checker-front-end`
- `https://github.com/gombarova/formalization-checker-back-end`