Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Educational tools for first order logic

Bachelor thesis

2018 Alexandra Nyitraiová

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Educational tools for first order logic

Bachelor thesis

Study programme:	Applied Computer Science
Study field:	Applied Informatics
Department:	Department of Applied Informatics
Supervisor:	RNDr. Jozef Šiška, PhD.
Consultant:	Mgr. Ján Kľuka, PhD.

Bratislava 2018 Alexandra Nyitraiová

41026145

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Alexandra Nyitraiová
Study programme:	Applied Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Applied Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Educational tools for first order logic

Annotation: Training and exercises are an important part of education process, especially when learning formalisms and constructs from logic such as formal proofs or semantics. For such exercises to be effective, the students need active feedback on what they are doing right or wrong. Interactive applications can make such feedback available even in the absence of teachers and thus allow students to work on the exercises according to their own time schedule.

Aim: The goal of this thesis is to create or extend tools that allow students to practice some of the tasks from logic courses: proofs, semantic tableaux or resolution. These shall be in the form of interactive, client-side web based applications.

Keywords: logic tools, first-order logic, client-side web application

Supervisor:	RNDr. Jozef Šiška, PhD.
Consultant:	Mgr. Ján Kľuka, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	15.10.2017
Approved:	16.10.2017

doc. RNDr. Damas Gruska, PhD.
Guarantor of Study Programme

Student

Supervisor

41026145

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov: Educational tools for first order logic
Výučbové nástroje pre prvorádovú logiku

Anotácia: Trénovanie a precvičovanie úloh tvoria dôležitú súčasť procesu výuky, hlavne v prípade formalizmov a konštrukcií z matematickej logiky. Aby takéto precvičovanie bolo efektívne, študenti musia mať dostatočnú spätnú väzbu o tom, čo spravili dobre a čo zle. Interaktívne aplikácie môžu sprístupniť takúto spätnú väzbu aj bez učiteľov, čo umožňuje študentom pracovať na úlohách podľa ich časových možností.

Cieľ: Cieľom práce je vytvoriť alebo rošíriť nástroje, ktoré umožňujú študentom trénovať si niektoré úlohy z predmetov venujúcim sa logike: dôkazy, sémantické tablá alebo rezolvencia. Nástroje budú vo forme interaktívnych webových aplikácií na strane klienta.

Kľúčové slová: nástroje pre logiku, logika prvého rádu, webová aplikácia na strane klienta

Vedúci:	RNDr. Jozef Šiška, PhD.
Konzultant:	Mgr. Ján Kľuka, PhD.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	prof. Ing. Igor Farkaš, Dr.
Dátum zadania:	15.10.2017

Dátum schválenia: 16.10.2017

doc. RNDr. Damas Gruska, PhD.
garant študijného programu

....
študent

....
vedúci práce

Acknowledgement: I would like to thank to my supervisor Jozef Šiška for his help and time he spend explaining me the curriculum. I found it very useful. I would like to thank to my other supervisor Ján Kľuka for spending his time with me and my work whenever I needed it. Thank you both very much for your patience, kindness and sharing your knowledge. I would like to thank to the students who tested our new tool. Thanks goes also to my boyfriend Zoltán Onódy for his support and patience.

Abstrakt

V tejto práci bolo našou snahou pomôcť študentom predmetu *Matematika (4) - Logika pre informatikov* naučiť sa ako funguje dôkaz analytickým tablom. Na základe práce našich školiteľov sme vytvorili nástroj, ktorý sa používa na vytvorenie dôkazu podľa metódy analytického tabla. Náš editor nevytvára dôkaz automaticky, ale validuje užívateľov dôkaz po každej akcii. Editor zdôrazňuje chyby a každú chybu vypíše pri príslušnej nevalidnej formule. Týmto nadobúda edukačný charakter. Náš editor podporuje dokazovanie v prvorádovej logike ale aj vo výrokovej logike. Dôkaz je vizualizovaný ako strom a každý vrchol je reprezentovaný nejakou formulou. Formula môže byť buď predpoklad o ktorom vieme, že platí alebo môže byť odvodená jedným zo štyroch pravidiel z nejakej formuly vyššie v strome. Formuly odvodzujeme aplikovaním nasledujúcich štyroch pravidiel: *alfa*, *beta*, *gama* a *delta*.

Náš editor sme naprogramovali ako webovú aplikáciu vo funkcionálnom programovacom jazyku Elm. Niektorí zo študentov predmetu *Mathematika (4)* nám ohodnotili náš editor. Podľa nich je náš nástroj vhodný pre naučenie sa dokazovania podľa metódy analytického tabla. Mnohí z nich by použili náš editor na vypracovanie domáceho zadania. Dali nám užitočnú spätnú väzbu, ktorá nám pomohla zlepšiť užívateľské rozhranie nášho editora. Predpokladáme, že v budúcnosti sa bude do nášho editora prispievať. Hoci sme dosiahli náš cieľ, na editore je stále čo zlepšovať. Napríklad implementovať prvorádovú logiku s rovnosťou.

Kľúčové slová: formula, tablový editor, prvorádová logika, výroková logika, webová aplikácia, alfa pravidlo, beta pravidlo, gama pravidlo, delta pravidlo, dôkaz analytickým tablom, funkcionálne programovanie, Elm

Abstract

In our work we wanted to help students of *Mathematics (4)* class to learn how analytical tableau proof works. We built a tool based on the editor of Jozef Šiška and Ján Kľuka which is used to create proofs according to the analytical tableau method. The editor does not create proofs automatically, but it validates the user's proof after every user action. It highlights mistakes and displays them near every formula. Thus it gains an educational character. Our tool supports proving in first-order logic as well as in propositional logic. The proof is visualized as a tree. A node is represented by a formula. A formula can be either a premise or derived from a predecessor located somewhere above it. The formula can be derived from a predecessor by applying one of the four rules: *alpha*, *beta*, *gamma* and *delta*.

We built the tool as a web application in functional programming language Elm. Some of the students of the *Mathematics (4)* class evaluated our tool. They found our tool useful and would use it to learn how the analytical tableau proof works or to complete a homework assignment. They gave us useful feedback which helped us to improve the user interface. We expect future contributions in our tool. Although we reached our goal, there are still some improvements which would make our tool more useful, such as implementing first-order logic with equality.

Keywords: formula, tableau editor, first-order logic, propositional logic, web application, alpha rule, beta rule, gamma rule, delta rule, analytical tableau proof, functional programming, Elm

Contents

Introduction

The goal of this thesis is to create a web application, which enables students of mathematical logic to create an analytical tableau and thus ease solving their homework.

Proof editors are suitable for teaching mathematical proofs. An appropriate implementation of a proof editor for a given method makes it easier for a student to study that method. An editor can make it easier for a teacher as well. For example, when correcting homework or while giving a lecture.

Mathematical logic, the science whose primary concern is deduction, is included in the curriculum of our computer science study. For the future work of students of this discipline, it is essential to know how to use deductive methods and thus derive and prove possible conclusions from a given theory.

Tableau calculus is one of the methods of writing a formal proof in propositional and first-order logic. The analytical tableau has a tree structure whose nodes represent signed formulas. By constructing such a tree, we can prove or disprove a formula or the feasibility of a theory.
There are other formal methods of proving in mathematical logic. It is worth mentioning, in particular, the Hilbert calculus[6] and the Sequent calculus. Sequent calculus has similar advantages as the Tableau calculus. It also constructs a tree according to some rules.

The method of analytical tableau differs from the Hilbert calculus in the way of formating the proof. The method of analytical tableau forms a proof in the form of a tree. The Hilbert calculus forms the proof as a sequence of formulas.
The relationship between Hilbert calculus and the natural deduction system is like the relationship between the low-level programming language and the high-level programming language. This means that it is often an inappropriate method of proving because of its cumbersome nature.
When using Hilbert calculus, we need to prove at least some metatheorems before we can use such a system without too much overhead.
The advantage of the analytical tableau method is that it is analytical. That is, we know exactly which rule we have to use and how. This method is therefore intuitive and straightforward and is rarely mistaken. Unfortunately, this does not apply when creating a tableau in first-order logic. Visualizing the tree structure at the same time

simplifies finding conclusions after designing the tree.

Inattention causes the most common mistakes when constructing a tableau. A student misapplies a rule or copies a variable or a term incorrectly.
Other mistakes occur in situations where the tableau tree structure is really large, and thus the tableau is unclear. The investigator may not notice the formulas to which the rules were not applied or are misapplied.

A specific requirement for our application is that it allows the student to create an incorrect tableau while also highlighting mistakes. Our application thus acquires an educational character and gives students the opportunity to learn from their own mistakes.
The application also evaluates whether something may be resulting, is resulting or is not resulting from a tableau.

Chapter 1 describes the theory needed to create a tableau. We define basic concepts such as formula, signed formula, truth-functionally satisfiable formulas, the theory of satisfiability, analytical tableau, open and closed branches of a tableau and alpha, beta, gamma, and delta rules. We mention similar tools and how they differ from ours. We explain the technologies we build our application in.
In Chapter 2, we set the exact requirements for our final application, and present the design of the application in Chapter 3. Chapter 4 explains how was the editor implemented and how it can be used. In Chapter 5 we give an evaluation to our work and mention what has been done according to the design and what differs from the design and why.

Chapter 1

Background

Formally was the Tableau calculus defined, described, and explained for the first time by Raymond M. Smullyan in his book First-Order Logic [14]. He defined the basic terminology and definitions we will be using in our work. We give a formal description of analytical tableau based on R.M. Smullyan's work [14].
In this chapter we explain mathematical background of analytical tableau in propositional and first-order logic for better understanding.

1.1 Propositional logic

Definition 1 (Symbols of propositional logic[14]). *Symbols of propositional logic are propositional variables from a denumerable set $V = \{p_1, p_2, \dots, p_n\}$, which does not contain symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$, (and). Even its elements do not include these symbols. Other symbols of propositional logic are the two symbols (and) (left parenthesis and right parenthesis). They are used for purposes of punctuation. [14] Symbol $\neg$ is unary connective (has one argument). Symbols $\wedge, \vee$ $\rightarrow$ are binary connectives (have two arguments).*

Definition 2 (Formula [14]). *Every propositional variable is a formula. If A is formula, so is $\neg$ A. If A, B are formulas, so are $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ - conjunction, disjunction, implication of formulas A and B.*

Definition 3 (A satisfiable formula in an evaluation [14]). *Let V be the denumerable set of propositional variables. Let v be an evaluation of the set V. Then for every propositional variable p from V and every formula A, B above we say that:*

- *v satisfies atomic formula p if and only if $(p) = t$,*
- *v satisfies formula $\neg A$ if and only if v does not satisfy A,*
- *v satisfies formula $(A \wedge B)$ if and only if v satisfies A and v satisfies B,*

- *v satisfies formula $(A \vee B)$ if and only if v satisfies A or v satisfies B,*
- *v satisfies formula $(A \rightarrow B)$ if and only if v does not satisfy A or v satisfies B.*

Definition 4 ((Non)Satisfiability, tautology, falsifiability of formulas in propositional logic[14])**.** *Formula X is satisfiable if and only if it is satisfied with at least one evaluation of the propositional variables. Formula X is called non-satisfiable if it is not satisfiable. Formula X is called falsifiable if and only if it is not satisfied in at least one evaluation of the propositional variables. Formula X is called tautology if and only if it is satisfied at each evaluation of the propositional variables.*

Definition 5 (Theory in propositional logic[14])**.** *We call each set of formulas a theory in propositional logic. Whether a theory T is satisfied or not, depends only on the evaluation of the propositional variables that are in the formulas of the theory T.*

Definition 6 (Definition of theory satisfiability in propositional logic [14])**.** *Let T be the theory in propositional logic. Evaluation satisfies theory T ($v \models T$) if and only if v satisfies every formula X from the set T. We call the satisfying evaluation a model of theory T.*

Definition 7 (Signed formulas [14])**.** *Let X be the formula of the propositional logic. Sequence consisting of symbols $\mathbf{T}X$ and $\mathbf{F}X$ is called signed formula. [10] Let v be the evaluation of propositional variables and X is the formula. Then*

- *v satisfies $\mathbf{T}X$ if and only if v satisfies X.*
- *v satisfies $\mathbf{F}X$ if and only if v does not satisfy X.*

1.1.1 Analytic tableau in propositional logic and related definitions

Definition 8 (Alpha and Beta rules[14])**.** *According to the definition of satisfying a formula we have formulated α and β rules as follows.*

$$\begin{array}{c}\alpha\\\hline\alpha_1\\\alpha_2\end{array}\qquad\begin{array}{c|c}\multicolumn{2}{c}{\beta}\\\hline\beta_1&\beta_2\end{array}$$

$$\begin{array}{c}\mathbf{T}(X\wedge Y)\\\hline\mathbf{T}X\\\mathbf{T}Y\end{array}\qquad\begin{array}{c|c}\multicolumn{2}{c}{\mathbf{F}(X\wedge Y)}\\\hline\mathbf{F}X&\mathbf{F}Y\end{array}\qquad\begin{array}{c}\mathbf{T}\neg X\\\hline\mathbf{F}X\end{array}$$

$$\begin{array}{c}\mathbf{F}(X\vee Y)\\\hline\mathbf{F}X\\\mathbf{F}Y\end{array}\qquad\begin{array}{c|c}\multicolumn{2}{c}{\mathbf{T}(X\vee Y)}\\\hline\mathbf{T}X&\mathbf{T}Y\end{array}\qquad\begin{array}{c}\mathbf{F}\neg X\\\hline\mathbf{T}X\end{array}$$

$$\begin{array}{c}\mathbf{F}(X\to Y)\\\hline\mathbf{T}X\\\mathbf{F}Y\end{array}\qquad\begin{array}{c|c}\multicolumn{2}{c}{\mathbf{T}(X\to Y)}\\\hline\mathbf{F}X&\mathbf{T}Y\end{array}$$

In propositional logic, we have two types of rules. Alpha rule and Beta rule. If there is an conjunction signed with T this indicates, we have to use the alpha rule as described above to simplify the original formula and detach the sub-formulas. If there is a disjunction signed with T this means, this disjunction can be satisfied if at least one of the sub-formulas is true. Therefore we apply the Beta rule as demonstrated above to detach the sub-formulas.

Definition 9 (Analytic tableau [14]). *An analytic tableau for formula X is an ordered binary tree, whose nodes are represented by formulas. Such thee is constructed as follows. We start by placing X at the root. Now suppose T is a tableau for X which has already been constructed; let Y be a leaf. Then we may extend T by either of the following two operations.*

(A) If some α occurs on the path P_y then we may adjoin either α_1 or α_2 as the sole successor of Y.

(B) If some β occurs on the path P_y then we may simultaneously adjoin β_1 as the left successor of Y and β_2 as the right successor of Y.[14]

Note: Successor, of the node X, is such node Y, which is located in the tree somewhere below X. Direct successor of the node X is such node Y, which is located directly under X. Depth of Y is exactly one greater than X's.

Definition 10 (Open and closed branches in a tableau [14]). *A branch of the tableau T is a path from the root of T to some of its leafs. Signed formula X^+ is on the branch π in T if and only if it is in some of the nodes on π.*

Branch π of tableau T is closed if and only if it contains signed formulas FY and TY for some formula Y. Otherwise is π open. Tableau T is closed if and only if its *every branch is closed. On the contrary, tableau T is open if and only if at least one of its branches is open.*

Example 1 (Example of tableau in propositional logic)**.** *The following example demonstrates the proof using method of analytical tableau in propositional logic. We prove by contradiction that the formula $(p \to (q \to (p \wedge q)))$ is a tautology.*
Let's suppose the given formula is not a tautology therefore we sign it with F sign. Now we need to apply alpha or beta rules until no rule can be applied. Formulas in a branch are satisfied simultaneously.

$$\begin{array}{llll}
(1) & & \mathbf{F}(p \to (q \to (p \wedge q))) & \\
(2) & & \mathbf{T}p & \textit{from (1)} \\
(3) & & \mathbf{F}(q \to (p \wedge q)) & \textit{from (1)} \\
(4) & & \mathbf{T}q & \textit{from (3)} \\
(5) & & \mathbf{F}(p \wedge q) & \textit{from (3)}
\end{array}$$

$$\begin{array}{lll|lll}
(6) & \mathbf{F}p & \textit{z (5)} & (7) & \mathbf{F}q & \textit{z (5)} \\
 & * & \textit{between (2) and (6)} & & * & \textit{between (4) and (7)}
\end{array}$$

By applying alpha and beta rules we get a conflict in every branch. If X is a formula then by conflict we mean there are TX and FX in the same branch. They can not be satisfied simultaneously. If there is a conflict in every branch we got a conflict with the assumption and thus the given formula is a tautology.

1.2 First-order logic

Definition 11 (Symbols of first-order logic [14])**.** *For first-order logic we shall use the following symbols:*

- *symbols of (individual) variables from some infinite countable set $\mathcal{V}_L$ (we denote them as x, y)*
- *non-logical symbols:*
  - *symbols of constants from a denumerable set $\mathcal{C}_L$ (a,b,...)*
  - *functional symbols from a denumerable set $\mathcal{F}_L$ (f,g,...)*
  - *predicate symbols from a denumerable set $\mathcal{P}_L$ (p,r,...)*
- *logical symbols:*
  - *logical connectives: $\neg$, $\wedge$, $\vee$, $\to$*
  - *quantifiers: $\forall$ a $\exists$ (universal a existential quantifiers)*

- *punctuation symbols:* (,) *and* , *(left and right parenthesis and comma)*

Sets $\mathcal{C}_L$, $\mathcal{F}_L$, $\mathcal{P}_L$ are disjoint. Non-logical symbols do not contain logical and punctuation symbols. Every symbol such that $S \in \mathcal{P}_L \cup \mathcal{F}_L$ is assigned an arity (argument count) $ar(S) \in N^+$

Example 2. *Symbols of constants represent concrete objects or values, just like own names in the natural language or constants in a programming language: $Saska$, $Zoli$, $Train24$, 9, 0. Predicate symbols represent features and relationships: $loves^2$, $<^2$, $buys^3$. Functional symbols represent relationships between uniquely designated objects: $price^1$, $valuation^2$, $+^2$.*

Definition 12 (Term and set of terms [14])**.** *We define the set T_L of terms in the language of the logic L as the smallest set of sequences of language L's symbols.*
Each symbol of variable $x \in \mathcal{V}_L$ is a term.
Each symbol of constant $c \in \mathcal{C}_L$ is a term.
If f is a functional symbol with arity n and $t_1, ..., t_n$ are terms, then also $f(t_1, ..., t_n)$ is a term.

Example 3. *Terms represent specific objects which can be directly named by symbols of constants: $Meghan$, $Train25$, 3, 2. Terms can be also indirectly named by unambiguous relationships: $mother(Sam)$, $price(Train25)$, $seller(Train25)$, $+(2,4)$. Terms may be arbitrarily nested: $mother(mother(Kate))$, $price(seller(train1))$, $+(0,1)$.*

Definition 13 (Atomic formula in first order logic[14])**.** *Atomic formulas are also called atoms. The set of all the atomic formulas of the language L is called A_L. If P is the predicate symbol with the arity n and $t_1, ..., t_n$ are terms, so the sequence of the symbols $P(t_1, ..., t_n)$ is called a predicate atom of the language L. Predicate atoms of the language are called atomic formulas (atoms) of language L.*

Example 4. *These are examples of atomic formulas in first-order logic:*
$woman(mother(x))$, $older(Erika, x)$, $child(Michael, mother(Emma))$

Definition 14 (Formula and set of formulas in first order logic[14])**.** *A set of formulas E_L of the first-order logic language L is the smallest set of sequences of language L, for which:*

- *All atomic formulas from A_L are formulas.*
- *If A is a formula, so is $\neg A$.*
- *If A and B are formulas, so are $(A \wedge B)$, $(A \vee B)$ and $(A \rightarrow B)$.*
- *If x is an individual variable and A is a formula, so are $\exists x A$ a $\forall x A$. (existential and universal quantification of formula A with respect to x)*

Nothing else is a formula in first-order logic.

Example 5. *This is an example of formula in first order logic:*
$(mother(x, y) \wedge child(y)) \rightarrow woman(Adriana)$

Definition 15 (Free and bound variables[14])**.** *We define an occurrence of a variable x in a formula A to be* bound *if it is either within the scope of some occurrence of $\forall x$ or $\exists x$, or else is itself immediately preceded by $\forall$ or $\exists$. An occurrence of x in A is called* free *if it is not bound. Finally x is said to have a* free occurrence *in A if at least one occurrence of x in A is free.*

Example 6. *(a) $\forall x(P(x) \wedge Q(x))$ - x is bound in this formula*

(b) $\forall x P(x) \wedge Q(x)$ - there is one free occurrence of x in the formula - $Q(x)$

(c) $P(x) \rightarrow Q(x)$ - both x are free variables in this formula

(d) $\exists y(P(x) \vee Q(x))$ - both x are free variables in this formula

Definition 16 (Open, closed formula and evaluation of variables[14])**.** *Let A be the formula of language L. Formula A is closed if and only if it does not contain any free occurrences of variables (e.g.* $\text{free}(x) = \emptyset$*). Formula A is open if and only if it does not contain any quantifiers.*

Example 7.

$\neg richer(x, y) \wedge hates(z, y)$	***open***	*not closed*
$\exists y(\neg richer(x, y) \wedge \forall z\, hates(z, y))$	*not open*	*not closed*
$\exists y \exists z(\forall x \neg richer(x, y) \wedge hates(z, y))$	*not open*	***closed***

Definition 17 (Structure[14])**.** *Let L be the language of first-order logic. The couple $\mathcal{M} = (M, i)$ is a structure for language L where*

- *M is a non-empty set, called* domain of the structure *$\mathcal{M}$;*
- *i is a representation function called interpretive function of the structure $\mathcal{M}$, which*
  - *assigns element $i(c) \in M$ to every symbol of the constant c of L language,*
  - *assigns function $i(f)\colon M^n \rightarrow M$ to every functional symbol f with arity n of L language,*
  - *assigns a set $i(P) \subseteq M^n$ to every predicate symbol P with arity n of L language.*

Example 8. *Let's find a structure for language L_{family} for simplified relationships with symbols of constants $\mathcal{C}_L = \{Daphne, Andy, Meghan, Sam\}$, predicate symbols and $\mathcal{P}_L = \{parent^2, woman^1\}$, functional symbols $\mathcal{F}_L = \{mother^1, sibling^1\}$ and symbols*

of individual variables $\mathcal{V}_L = \{x, y, z\}$.
$\mathcal{M}$ = (M, i)
M = d, s, m, a, l
$i(Daphne) = d$, $i(Sam) = s$, $i(Meghan) = m$, $i(Andy) = a$
$i(mother) = \{d \to m, a \to m\}$
$i(sibling) = \{d \to a, a \to d\}$
$i(woman) = \{d, m\}$
$i(parent) = \{(m, s), (s, m)\}$

Definition 18 (Evaluation of variables[14]). *An evaluation of (individual) variables is any function* $e\colon \mathcal{V}_{\mathcal{L}} \to M$ *(assigns variables to a domain elements).*

By writing $e(x/v)$ *we denote the valuation of the individual variables that assigns the variable* x *the value* v *from domain* $\mathcal{M}$ *and all other variables the same value as* e.

Example 9. *We will use the structure from example 8. We define a term* $t_1 = sibling(x)$. *We define evaluation* $e = \{x \to d\}$.
We assigned the variable x *from the domain* $\mathcal{M}$ *to* d. *Therefore the value of our term* t_1 *under evaluation* e *is* $t_1 = sibling(x) = sibling(d) = a$.

Definition 19 (Substitution and its application[14]). *Let* L *be the language of first-order logic. Substitution (in language* L*) for any set of individual variables is every mapping* $\sigma\colon V \to T$ *of variables from* V *to terms of language* L.

Example 10. *Let* $\mathcal{C}_{\mathcal{L}} = \{a, b\}$, $\mathcal{F}_{\mathcal{L}} = \{g^2, f^3\}$. *Then for example* $\sigma_1 = \{x \mapsto a, y \mapsto f(a, x, y)\}$ *is substitution.*

We want to use the substitution to replace variables in terms and formulas. However, we must be careful about some special cases.
Let A *be a sequence of symbols. Term* t *is substitutable for variable* x *in* A *if and only if there is no free occurrence of the variable* x *in* A. *Any variable* y *present in* t, *must not be within the scope of the formula's quantifier* $\exists y$ *or* $\forall y$.
Substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ *is applicable on* A *if and only if for every* i, $1 \leq i \leq n$, *term* t_i *is substitutable for* x_i *in* X.

Example 11. *No term, which contains* y, *(ex.* y, $bff(y)$*), is substitutable for variable* x *in formula* $\forall y$ *hates(x, y). The variable* y *is bound in the formula.*

Definition 20 (Satisfying a set of formulas in a structure[14]). *Let* S *be the set of formulas of language* L, *let* $\mathcal{M}$ *be a structure for* L, *let* e *be an evaluation of propositional variables.*

- *Structure* M *satisfies the set* S *in evaluation* e *(in short* $M \models S[e]$*) if and only if* $M \models Y[e]$ *is true for every formula* Y *from* S.

- *Structure M satisfies the set S (in short $M \models S$) if and only if $M \models Y$ is true for every formula Y from S.*

Definition 21 (Satisfiability of first-order formula and set of formulas[14])**.** *Let X be a formula of language L and S be a set of formulas of language L.*

- *Formula X is satisfiable if and only if at least one structure M for L satisfies X under at least one evaluation e.*
- *The set of formulas S is satisfiable if and only if at least one structure M for L satisfies S in at least one evaluation e.*
- *Formula X (set of formulas S) is non-satisfiable if and only if it is not satisfiable.*

A formula is called valid if and only if every structure M for language L satisfies X under each evaluation e. Validity of formula can be understood as a special case of satisfiability, when every structure M for L satisfies a set of formulas S under their every evaluation e. Valid formulas are a first-ordinal analogy of tautology.

Definition 22 (First-order entailment of formula from a set of formulas[14])**.** *Let X be a formula in language L. Let S be the set of formulas in language L. Formula X (first-orderly) follows from S (in short $S \models X$) if and only if for every structure M for L and every evaluation e is true, that if M satisfies S in e, then M satisfies X in e.*

Definition 23 (Gama and Delta rules[14])**.** *When creating a tableau in first order logic, we use Alpha, Beta, Gamma and Delta rules. The former two were described in definition 8. We will describe the latter two. The gamma and the delta rules are defined as follows:*

$$\begin{array}{cccc}
\gamma & \dfrac{\mathbf{T}\forall x A}{\mathbf{T}A\{x \mapsto t\}} & \dfrac{\mathbf{F}\exists x A}{\mathbf{F}A\{x \mapsto t\}} & \textit{unanimously: } \dfrac{\gamma(x)}{\gamma_1(t)} \\[2ex]
\delta & \dfrac{\mathbf{F}\forall x A}{\mathbf{F}A\{x \mapsto y\}} & \dfrac{\mathbf{T}\exists x A}{\mathbf{T}A\{x \mapsto y\}} & \textit{unanimously: } \dfrac{\delta(x)}{\delta_1(y)}
\end{array}$$

We can use gamma or delta rule only if there is a quantifier which applies to the whole formula.

Example 12. *These are examples of formulas which a gamma or delta rule can be applied to: $\forall x(P(x) \vee Q(x))$, $\forall x(P(x) \vee Q(y))$*

Example 13. *In the following examples, we can not apply gamma or delta rule: $\forall x P(x) \vee Q(x)$, $(P(x) \vee Q(x))$, $\forall y P(x) \vee Q(y)$ or in $\forall x(P(x) \vee \exists y Q(y))$ when substituting for y.*

The following rule is specific for substitution when we use gamma. We can substitute the original variable with a term or a constant. The substitution has to be applicable. The substitution is not applicable if a substituting constant or term are incorrect. The new substituting constant is incorrect if it looks like a variable which is bound in referenced formula. The new substituting term is incorrect if it contains a variable which is bound in referenced formula.

The following rules are specific for substitution when we use delta. We can substitute the original variable only with a constant. A substituting term can not be used in substitution. The substitution has to be applicable. The substitution is not applicable if a substituting constant is incorrect. The new constant is incorrect if it looks like a variable which is bound in referenced formula. Substituting constant can not be located somewhere above in the tableau as the free variable.

In general, when applying gamma, we can use any *constant. When applying delta, we have to use* new *constant.*

We have to be careful when to choose which one. We usually choose delta over gamma always if possible.

Example 14 (Example of tableau in first-order logic)**.** *The following example demonstrates the proof using method of analytical tableau in fist-order logic. We prove by contradiction that the following formula is valid. Formula in first order logic is valid if it is satisfied in every structure and under every evaluation.*
$(\forall x P(x) \rightarrow Q(x)) \rightarrow (\forall x P(x) \rightarrow \forall x Q(x))$

Let's suppose the given formula is not valid. Therefore we sign it with F sign. Now we need to apply alpha, beta, gamma or delta rules until no rule can be applied. Formulas in a branch are satisfied simultaneously.

$$\begin{array}{llll}
(1) & \mathbf{F}(\forall x P(x) \rightarrow Q(x)) \rightarrow (\forall x P(x) \rightarrow \forall x Q(x)) & & \\
(2) & \mathbf{T}\forall x P(x) \rightarrow Q(x) & & \alpha\textit{(1)} \\
(3) & \mathbf{F}(\forall x P(x) \rightarrow \forall x Q(x)) & & \alpha\textit{(1)} \\
(4) & \mathbf{T}\forall x P(x) & & \alpha\textit{(3)} \\
(5) & \mathbf{F}\forall x Q(x) & & \alpha\textit{(3)} \\
(6) & \mathbf{F}Q(a) & & \delta\textit{(5)}\ \{x \rightarrow a\} \\
(7) & \mathbf{T}P(a) & & \gamma\textit{(4)} \\
(8) & \mathbf{T}P(a) \rightarrow Q(a) & & \gamma\textit{(2)}\ \{x \rightarrow a\}
\end{array}$$

$$\begin{array}{lll|lll}
(9) & \mathbf{F}P(a) & \beta\textit{(8)} & (10) & \mathbf{T}Q(a) & \beta\textit{(8)} \\
 & * & \textit{between (7) and (9)} & & * & \textit{between (10) and (6)}
\end{array}$$

As we see, formulas $\mathbf{F}P(a)$ *and* $\mathbf{T}P(a)$ *are in one branch but can not be satisfied simultaneously. Therefore there is a conflict in the branch. Since we got conflicts in every branch of this tableau, we have a conflict with the assumption. It means, that the given formula is valid.*

1.3 Similar work

We have done some research to find out and explore other similar works. We are going to describe their features in comparison with what features we want from our tool to have.

1.3.1 Tableau Editor

Ján Kľuka and Jozef Šiška have created a web application called Tableau Editor. [11] This editor has embedded basic binary connectives, one unary connective, parsing formulas, signed formulas with **T** and **F** characters, alpha and beta rules. It allows users to create an incorrect tableau and indicates mistakes. Tableau is also rendered visually as a tree. The editor works only for propositional logic, not for first order logic.

1.3.2 Ruzsa

Author of Ruzsa, the similar tableau editor implemented as web application, is Tamás Bitai [1]. After thoroughly reviewing his application, we found that it also works for first-order logic. It implements basic binary and unary connectives, quantifiers, formulas, alpha, beta, gamma and delta rules, and many other nonstandard features lacking documentation. The application works for both propositional and first-order logic, but it is unfortunately counterintuitive. It does not allow the creation of an incorrect tableau. The application is used to proove the Tarski's world [17] with the method of analytical tableau. It is programmed in JavaScript and Angular.

1.3.3 Logitext

We found Logitext an online editor for teaching how the Sequent calculus work [19]. Its author is Edward Yang. It includes a very accurate tutorial which is very helpful. The author explains how the Sequent calculus works and how to prove with his tool. We tried the editor and discovered, that it is possible to brute force a proof by clicking on the particular right arrow. The tool works for first order logic as well as for propositional logic. It is implemented in Haskell - a functional programming language and in other two languages.

1.3.4 Clausal Language

CL (Clausal Language) is a declarative programming language with the look and feel of a modern functional programming language. [8] CL comes with its own proof system (intelligent proof assistant) in which the user can state and prove properties of his

functions and predicates. [8]
Clausal language comes with its own proof system, which can also be used for methods other than the analytical tableau method. This means that it does not draw a tableau as a tree structure, unlike the above-mentioned similar works. The idea of a tree tableau is lost in this system and is therefore not suitable for teaching the tableau method. Its author is Ján Komara.

1.3.5 Other tools for teaching first-order logic

There are also other non-tableau based logic tools as Fitch [16], Boole [15], Tarski [17], and others [18]. Their authors are people from the Stanford University. Monika Švaralová worked on a program for teaching mathematics in her bachelor thesis. [20] This program can teach students the principals of only propositional logic.

1.4 Technologies

We are going to build our application upon the work of Ján Kľuka and Jozef Šiška and improve their tool. As the original tableau editor was written in Elm, we decided to use Elm in our work.

1.4.1 Elm

Elm has its own DOM implementation and is compiled to JavaScript. We can create a web application in it. Elm uses type inference and therefore is a strongly typed language. This helps to detect runtime errors during compilation. If an error occurs a friendly error message is shown in the console. Therefore, an application written in Elm can not throw a runtime error[4]. In case of interacting with JavaScript there can occur a runtime error, but stays on the JavaScript's side of the application.

The outputs of an Elm program are not streamed - printed in run time. They are printed once when the whole process is finished. As long as we are waiting for the code to finish evaluation, we will not see anything.

It is possible to create an infinite loop in Elm. Let's suppose we have created one and there is a Debug.log command inside it to print something out in every iteration. If we hit ctrl+c to end evaluation prematurely, Elm ignores the output that is returned to it. So nothing will be printed.

We will use elm-live package [2] to create a development server. Thanks to its hot-reload feature it is very developer-friendly to use it while development process.

Consequences of an Elm application

Elm is a functional language. Any function that is given the input x always returns the same output y. The disadvantage is that it is almost impossible to generate a random number. Elm solves this with commands.

When creating an Elm application, we need to define four functions. One of them returns the initial state, the second updates the state based on the action message, the third renders the data from the model and the fourth handles requests which may interact with JavaScript.

If we want to change anything in the view, we need to make some action - for instance, click a button. The dispatcher is triggered and returns a message. The state is updated according to this message and subsequently rerendered. The data flows only in one direction. This is called the Flux principle [13]. The advantage of the Flux principle is a separation of concepts view and model. This brings modularity to our code. We can redesign our view without touching the model. The other advantage is that we spend less time with debugging our application. The state is not mutated directly but only if an appropriate message comes. It is hard to debug when any object can change itself in any time. Using Flux principle we have better control over the data flow in an application. Another advantage is that we can easily track and save every state which is our application currently in. Thanks to this we can easily implement undo and redo functionality. The easiest and usual way to implement undo-redo functionality in functional languages is using the structure shown in Listing 1.1.

```elm
type alias History state =
    { past: List state
    , present: state
    , future: List state
    }
```

Listing 1.1: Undo/redo implementation in Elm

We can notice that concatenating $reverse(past) + +[present] + +future$ creates one array with all states in order as they were created.

We keep all of the past states, we remember current state and remember all of the future states which we went back from. This structure is easy to work with. When we want to move one state forward, we prepend the *present* state to the list of *past* states and put the first state from *future* states into the variable for the present state.

Zipper

Zipper is a data structure used in functional programming languages. It is usually used for representing and moving up and down a tree, list or a tuple. It has few advantages. We will demonstrate them in analogy to the list.

If we want to modify an element in an array in functional programming, we need to copy first (n-1) elements. Then we modify the element and create the new array from the copied elements and the modified one. The time complexity of this operation is O(n). Instead of array, we can represent list in the zipper. For example, the list of `[first, second, third, fourth, fifth]` can be represented as

- `((second first) third (fourth fifth))`.

For every node we will keep three types of information. The previous element, the content of actual element and the next element. In this structure we keep the focus on the actual element, so it can be modified in O(1) time complexity. We can also easily shift the focus to the left `((first) second (third fourth fifth))` and to the right `((third second first) fourth (fifth))`. In a tree, we can shift focus to the parent or to the child - up or down. The nodes at the focus are easily modifiable.

Breadcrumbs are used to retain information about parts of the tree that move out of focus. As the tree is navigated, the needed context is pushed onto the list of breadcrumbs, and they are maintained in the reverse order in which they are visited. [7]

1.4.2 Yarn

Yarn is a package manager for packages designed for JavaScript. Since npm - the other package manager for Javascript packages - had some issues when installing Elm, we decided to use Yarn as package manager. Package manager allows us to install packages in the scope of a particular project. Thus we can use different version of a particular package for other project. We install packages elm[4], elm-live[2] and elm-make[5].

Chapter 2

Problem specification

Our work aims to create a web application, which will serve as an assistant during the creation of an analytic tableaux proof. In section 1.3.1 we mentioned similar application. We decided to continue in that work and add features to it to meet our requirements. The requirements for our final application are as follows:

Visualisation, validation and mistakes highlighting.

- In our tableau assistant, it must be possible to create a proof in propositional logic and first-order logic.
- The editor will be able to recognize and parse first-order formulas.
- The proof must be visualized as a tree.
- It should be possible to distinguish between premise and non-premise.
- Formulas will be validated according to the alpha, beta, gamma and delta rules.
- The tableau assistant must validate our proof and highlight our mistakes.
- It should be possible to prettify the formulas. Characters as $- >, /\backslash, \backslash/, \backslash forall, \backslash exists$ must be replaced by $\rightarrow, \wedge, \vee, \forall, \exists$. It should also remove unnecessary parenthesis.

Manipulating the proof.

- The root of the proof can be only an alpha node.
- It should be possible to extend the tableau by adding a new node wherever needed. Not only under the leaves.
- When adding a beta below a node n, the direct successor of n becomes the direct successor of one of the beta formulas.

- It must be possible to delete an arbitrary node in a tree and a sub-tree of a node.
- Deleting one of the beta trees should be possible only if the beta node does not contain a formula and any sub-tree. The other beta becomes an alpha continuation of the tableau.
- It must be possible to undo or redo our steps.
- Sub-trees of type beta can be easily swapped. Left to right and right to the left.
- It should be possible to close a branch and reopen it.
- Changing the type of formula should be possible. When changing between gamma and delta, the substitution must remain the same.

Persistance.

- It must be possible to export our tableau and import it later to continue in proving.
- It must be possible to print out our tableau or save it in pdf format.

Chapter 3

Design

To satisfy requirements from Chapter 2 we need to reimplement the proof structure, implement validation of Gamma and Delta rules, implement undo-redo functionality and other parts of editor's logic. We are going to improve the user interface so that the user can interact more intuitively with the tableau editor. The main two groups of concern in this chapter are application's new logic features and improvement of the user interface.

3.1 Application logic

It may not be evident to the user that some parts of the original application's logic were reimplemented. They will work the same way. However, refactoring is very beneficial for every application. Not refactoring the original code precisely before adding new lines of code would lead to the messy and unmaintainable code.

3.1.1 Proof structure

In the old tree representation we parsed the formula every time we wanted to validate it. It means every time the state changed, the update and view functions were called, and all of the formulas were parsed and validated again. In the new tree representation, we store the parsed formula in the state. It is not necessary to parse it every time after a state change, and therefore it saves resources.

The content of the node and sequel of the tableau below the node were not separated in the old tree structure. Work with this structure was more arduous as we heavily used the Elm's *Maybe* type. For example, every *Leaf* kept the information whether its branch is closed or not. This information was represented by *Leaf Maybe (reference1, reference2)*. If the branch was not closed, there was no tuple to keep. To check if the referred nodes are in contradiction, we had to check if there is some tuple at all.

The design of our new proof structure is as can be seen in Listing 3.1.

```
type alias Tableau =
    { node : Node, ext : Extension }

type alias Node =
    { id : Int
    , value : String
    , reference : Ref
    , formula : Result Parser.Error (Formula.Signed Formula.Formula
   )
    , gui : GUI
}

type Extension
    = Open
    | Closed Ref Ref
    | Alpha Tableau
    | Beta Tableau Tableau
    | Gamma Tableau Substitution
    | Delta Tableau Substitution
```

Listing 3.1: Tableau structure

We created *Tableau* record which has 2 variables. The *node* variable for storing the node's content and the *ext* variable for keeping the sequel of the tableau. We store five types of information in every *node*. The node's unique identifier, the string representation of a formula, the reference to the node which was the actual node derived from, the parsed formula and the variable *gui* which we use when rendering the proof to HTML. The *ext* variable keeps the objects of type *Extension*. The extension can be *Alpha* which contains a tableau as a child, *Beta* which contains left and right tableau children, *Gamma* and *Delta* which contain substitutions and a tableaux as children, *Open* which serves as the tail of a branch and *Closed* node which keeps two references of contradicting nodes. This way we created a structure which is easy to work with and separated concepts - node's content and the sequel of a tableau.

3.1.2 New zipper structure

We modified zipper's structure to represent also gamma and delta nodes with their substitutions. We can see the zipper's design in the Listing 3.2. Zipper for a node consists of the *Tableau* for the actual node and the *Breadcrumbs*. The type *Breadcrumbs* is list of all predecessor nodes of an actual node. Every *Crumb* type keeps a node's direct ancestor. We defined five types of *Crumb*.

The type *BetaLeftCrumb* stores the *Tableau* of its sibling and parent's node. The same for *BetaRightCrumb*. The type *GammaCrumb* and *DeltaCrumb* store the node of their parent and their *Substitution*. The type *AlphaCrumb* stores only the direct ancestor's node.

```
type alias Zipper =
    ( Tableau, BreadCrumbs )

type alias BreadCrumbs =
    List Crumb

type Crumb
    = AlphaCrumb Node
    | BetaLeftCrumb Node Tableau
    | BetaRightCrumb Node Tableau
    | GammaCrumb Node Tableau.Substitution
    | DeltaCrumb Node Tableau.Substitution
```

Listing 3.2: Zipper structure

3.1.3 Parsing and validating first-order formulas

Ján Kľuka already programmed the parsing of first-order formulas in the original tableau editor [11]. He used Elm's Parser module. This module was written by the creator of Elm language and is used for parsing strings in Elm. We will use Ján's parser in our work and parse a formula every time the user changes the string in the input for a formula.

3.1.4 Validation

As validation of Alpha and Beta rules work the same way in the first-order logic as in propositional, there is no need to implement them again. The only feature we will implement is that beta sub-formula cannot be a premise.

3.1.5 Gamma rule validation in first-order formulas

A gamma sub-formula is valid if and only if it meets the specific assumptions which should be checked in the following order:

1. It has to have a valid reference. The reference is valid if it refers to some formula above.

2. Gamma sub-formula cannot be a premise. Its reference cannot point to itself.

3. Applying gamma rule to the referenced signed formula must result in the given formula. The referenced formula has to have a form of $T\forall xP(x)$ or $F\exists xP(x)$
4. The substitution has to be applicable. The new term is incorrect if it contains a variable which looks like a bound in the referenced formula. We refer to the Definition 19, which explains the theory around the applicability of a substitution.
5. Only the quantified variable right behind the T or F sign can be substituted. If there is no quantifier, the gamma rule cannot be applied.
6. The substitution has to be written correctly. The term replaces the given variable without a typo. No other variable is replaced.

3.1.6 Delta rule validation in first-order formulas

A delta sub-formula is valid if and only if it meets the specific assumptions which should be checked in the following order:

1. It has to have a valid reference. The reference is valid if it refers to some formula above.
2. Delta sub-formula cannot be a premise. Its reference cannot point to itself.
3. Applying delta rule to the referenced signed formula must result in the given formula. The referenced formula has to have a form of $F\forall xP(x)$ or $T\exists xP(x)$.
4. The chosen substituting term has to be a constant and not a function.
5. The substitution has to be applicable. The new term is incorrect if contains a variable which looks like a bound in the referenced formula. We refer to the Definition 19, which explains the theory around the applicability of a substitution.
6. Only the quantified variable right behind the T or F sign can be substituted. If there is no quantifier, the delta rule cannot be applied.
7. The substitution has to be written correctly. The term replaces the given variable without a typo. No other variable is replaced.
8. Substituting constant can not be located somewhere above in the tableau as the free variable.

3.1.7 Substitution

Substitution is performed during validation according to the Gamma and Delta rules. In a formula that starts with a quantifier and a quantified variable, we can substitute this variable for a suitably chosen term.

Ján Kľuka already programmed the substitution. This function takes the formula and a *Substitution* record and returns either correctly substituted formula or an error message if the substitution is not applicable. We will use this function to validate the user's substitution.

3.1.8 Undo, Redo

As a tableau grows big and become complex, the modification may be time-consuming. Because of the complexity of the tableau, every modification may be a destructive operation which may cause an undesired effect. Therefore it is essential to implement the undo-redo functionality.

Thanks to the Elm architecture, application's data flow in one direction only. It means, every time a button is clicked or a character is typed, the state changes. We will store all past states of the application. In case of a mistake, we can go to the very first state of our tableau or go few steps back.

We will implement two buttons. One for undo step and one for redo step.

3.2 User interface improvements

The user interface of the original tableau editor[11] was usable but not user-friendly. If we wanted to delete a node, we had to delete the node with its whole sub-tree. If we wanted to add another premise to our semi-finished tableau, we could not add it right below the last premise, but only below an opened branch.

The action buttons - for import, export, prettify and print - were below the tableau. Every time the user added a node, the section with these buttons jumped down. This behavior is not very user-friendly. We place these buttons to above the tableau.

Students usually create tableau with many branches so it can be enormous. An advantage of the original user interface was that it saved space. Therefore the buttons, inputs and all of the elements will be small and designed simple enough to retain this advantage.

The error messages were not clear enough as they were shown all together below the tableau. We show an error message near every node if there is one.

For better user interaction we add the E button (as edit) next to every node. This button will hide and show controls which modify the node or node's extension. This way we save the space near every node and manipulating the node is possible as well.

3.2.1 Render the tableau using html *div* elements

The original tableau structure was being rendered using tables. Rendering the tree to HTML table was very complicated way of displaying the tableau. The rows of sub-trees had to be merged correctly so that the very top layer of the tableau had to have as many columns as the number of leaves in the tree. Rendering the tree with the *div* elements is more reasonable than rendering the tree using a table.

3.2.2 Add a node anywhere in the tree

It would be nice to be able to add a premise below the last premise even if there is some sub-tree. Adding a node will be possible almost anywhere in the tree. However, we cannot add a node above the root. We cannot replace one of the beta nodes with alpha node since it would break the structure of the proof. Adding a node will be possible by clicking on the button. This button will be shown under every node when the *E* button is clicked. The node, below which we add the new node, becomes parent of the new node. The extension of the parent becomes the extension of the newly added node.

3.2.3 Deleting arbitrary nodes

Deletion of a single node has several criteria:

- The child of the deleted node becomes the child of the deleted node's parent.
- A student can delete the root if only if the node below the root is an alpha node.
- A student can delete one of the beta sub-trees if and only if it does not have any sub-tree and any value in the input for the formula.
- If a student deletes a beta sub-tree, the remaining beta sub-tree converts to an alpha sub-tree. This alpha sub-tree will be the sequel of the tableau instead of the original two beta formulas.

3.2.4 Delete the sub-tree of a node

Students will be able to delete the whole sub-tree of a particular node in the tree. This feature will be useful when getting rid of a large and redundant sub-trees. To delete the sub-tree, we need to show the node's controls by clicking on *E*. Then we click on the *Delete* button and choose one of the options - sub-tree or node.

3.2.5 Swapping beta sub-trees

The student would like to arrange his tableau visually and move open branches to the left or right. We want to allow him do that.

The student will be able to swap beta sub-trees in our tableau editor. Under every node, which has direct beta sub-trees, will be a button for swapping beta sub-trees.

3.2.6 Change of formula type

Imagine the following situation. The user added a gamma sub-formula below a premise. The user already wrote a formula and substitution to the inputs. However, then the user realized, it should be delta sub-formula. Instead of adding a delta node, copying the inputs manually and deleting the gamma node it may be helpful to change the formula's type.

Changing a node's type will be subject to following conditions:

- The root node can be only an alpha node. It means this node cannot be changed.
- When changing from gamma node to delta node or vice-versa, substitution remains the same.
- A node of type *Beta* cannot be changed to anything else if its sibling has a sub-tree or a value in its formula input.

Chapter 4

Implementation

In this chapter, we describe the main aspects of our implementation. The resulting functionality was already described in Chapter 3.

4.1 New proof structure

We implemented the proof structure as designed in Chapter 3. This new structure influenced some parts of the original application. Thus we had to reimplement them. It is essential to mention that there is a difference between node's parent in the tree and the referenced node. The parent p of the node n is such node, which is located right above the node n in the tree and contains the node n as an extension. The referenced node r is such node which is the node n's formula derived from. A formula does not have to be derived from the node located right above. Therefore we need to keep the information about the referenced node in every node.

A node can be either derived from a predecessor or a premise. We derive the node from a predecessor by applying one of the four mentioned rules: *alpha*, *beta*, *gamma* and *delta*. To set a formula as a premise, it has to refer to itself.

4.1.1 Zipper reimplementation

For better understanding, we are going to demonstrate the changes on the function *down* of the zipper. The argument of this function is a zipper, and it returns the zipper representing the node below. In short, this function moves the focus in the zipper to the node below. It is used in functions like *renumber* when renumbering the identifier of the node.

In the original implementation, the sequel of the tableau was not separated from the content of the node. We used $case\ t\ of$ to ask for the continuation of the tableau. We could access the actual node in every branch. If we wanted to preprocess the node in the *let*...*in* section before matching some of the *case* branches, we could not do that. Formerly *down* worked only on the alpha sub-tree as we can see in the Listing 4.1.

```
down : Zipper -> Zipper
down ( t, bs ) =
    case t of
        Alpha n st ->
            ( st, (AlphaCrumb n) :: bs )
        _ ->
            ( t, bs )
```

Listing 4.1: Original down function

As we separated the node's content from the continuation of the tableau, we gained access to the *t.node* in the *let*...*in* section. This is an advantage if we want to preprocess the node and do not want to duplicate the code in every *case* branch.

We had to add another two cases - Gamma and Delta. These two hold sub-tableau as a sequel and their substitution. As applying Gamma and Delta rules does not create two sub-trees as applying the Beta rule, we can apply *down* function to them. We can see the new *down* function in Listing 4.2.

```
down : Zipper -> Zipper
down ( t, bs ) =
    let
        preprocessed = identity t.node
    in
    case t.ext of
        Alpha subtableau ->
            ( subt, AlphaCrumb preprocessed :: bs )

        Gamma subtableau substitution ->
            ( subtableau, GammaCrumb preprocessed substitution ::
   bs )

        Delta subtableau substitution ->
            ( subtableau, DeltaCrumb preprocessed substitution ::
   bs )

        _ ->
            ( t, bs )
```

Listing 4.2: Down function moves the focus down the tree

4.2 Unit tests

We wrote tests to check that our new functionality is correct.

We tested the cases when the substitution can and cannot be applicable if the formula begins with a quantifier and quantified variable, if there is exactly one variable we want to substitute and if it is the one right after the quantifier.

We also wrote tests for the refactored functions which work with zipper. Other tests we wrote tested whether the tableau as a proof structure is valid - for instance, tests for renumbering the references correctly.

4.3 Substitution

To perform substitution, we have to do it according to the rules mentioned in Chapter 3. Substitution was initially implemented as a part of the tableau editor's code. Zoltán Onódy needed to parse propositional and first-order formulas in his work[12] as well, so he extracted the Formula.elm parser to a separate module [9]. We implemented function *removeQuantifierAndSubstitute* in this separated module and use this function when validating user's substitution. It connects our validation functions with *substitute* in formula module [9].

We had to create a new record to save the substitution in the state for every gamma and delta node. We can see it in Listing 4.3. The user enters which constant or term he wants to substitute and for which variable he wants to substitute it.

```
type alias Substitution =
    { term : String, var : String }
```

Listing 4.3: Substitution record

4.4 User interface

To improve the user interface, we wrote custom CSS styles which was quite time-consuming. These styles enable uz to hide or show options for every node by clicking on the button labeled with the letter *E*. The information whether they are hidden is saved in the application state. The controls are used for adding a specific node type, deleting a node or a sub-tree, changing the node type, closing the branch and reopening the branch.

4.5 Validation of Gamma and Delta rules

The validation functions are written in a way that they chain functions validating the given formula according to the particular rule. They very heavily use the functionality of *Result* and *Maybe* types. We had to adjust our functions used in validation so they can be chained with those types.

4.6 Undo, Redo

To work with history, we used the undo-redo [3] module which was created by Elm community. Implementing history usage to our application changed slightly the way, how we work with the model's variable in the update and view functions.

4.7 Adding and deleting a specific node

We remind that there is a difference between node's parent and a referenced node. This difference was described in Section 4.1

When deleting a particular node, we need to do so from the view of the node's parent. We need to attach the node's parent to the node's child. The referenced node in every node is represented the same way as in the original editor. We can see this representation in Listing 4.4.

```
type alias Ref =
   { str : String, up : Maybe Int }
```

Listing 4.4: Reference record refers to the node which was a formula derived from

Every node has a unique numerical identifier. This identifier is stored in the node's *id* variable. The *str* field in *Ref* record stores the string representation of referenced node's identifier. The *up* field represents how many nodes above is the referenced node located. We need the *up* field in case of renumbering the identifiers. If a node is added or deleted, we renumber the identifiers to have a continuous sequence of them from 1 to n. In this case, it is useful to know how many nodes above is the referenced node located as we cannot trust the changing identifiers.

When deleting the node, we attach the node's parent to the node's child. When deleting one of the beta sub-trees we send the reference of the sub-tree's parent. This information does not help us to determine whether we want to delete the right or the left sub-tree. In case of deleting one of the beta sub-trees, we need to work with breadcrumbs to determine which sub-tree we want to delete.

We have to renumber the references when deleting or adding a node in the tableau. In the original representation, we could delete only the whole sub-tree and add a new

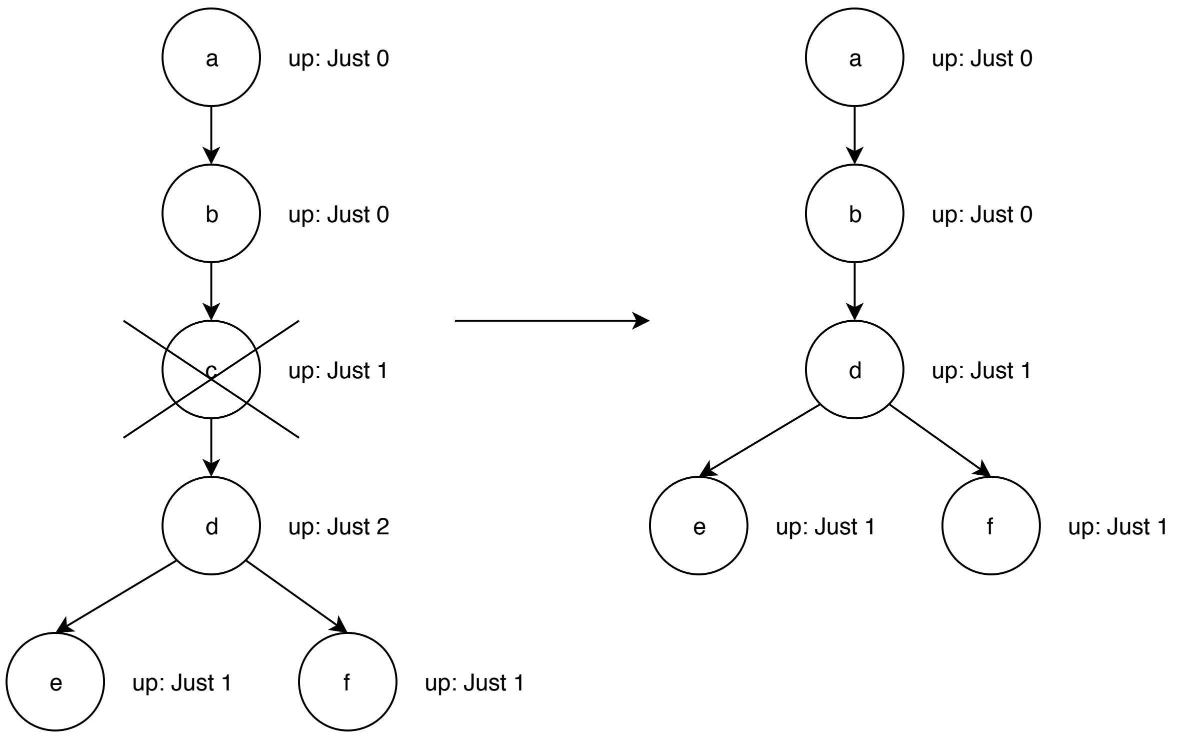

Figure 4.1: Tableau before and after deleting the node

node only below the Leaf. Every identifier renumbered just fine because the only thing that changed in the reference was the *str* variable. We could rely on the fact, that the *up* variable did not change at all. In the new implementation, there might be a problem with renumbering because the *up* variable may change. Consider the situation represented in Figure 4.1

The x in *Just* x represents how many steps above is the referenced formula located. Nodes a and b are premises. Node c and d are deduced from the b node and nodes e and f are deduced from the d node. Now, we would like to delete the c node. Since the reference of node d is pointing somewhere above the deleted node, we have to renumber the mentioned x in the d node's reference. Since the reference of node e and f is pointing somewhere below the deleted node, we cannot renumber their *up* variables. References which point to c remain the same. Renumbering them would not make them valid.

In other words, if the mentioned x would be bigger or equal to one plus the length of the path from the parent of the deleted node we would change it to $x - 1$. Listing 4.5 shows the code which renumbers the *up* variable when deleting a node.

```
if x >= lengthOfPathFromDeletedNodesFather+1 then
   Ref ref.str (Just (x - 1))
```

Listing 4.5: Renumbering up when deleting

Consider the situation when adding the node somewhere in the middle of the tree

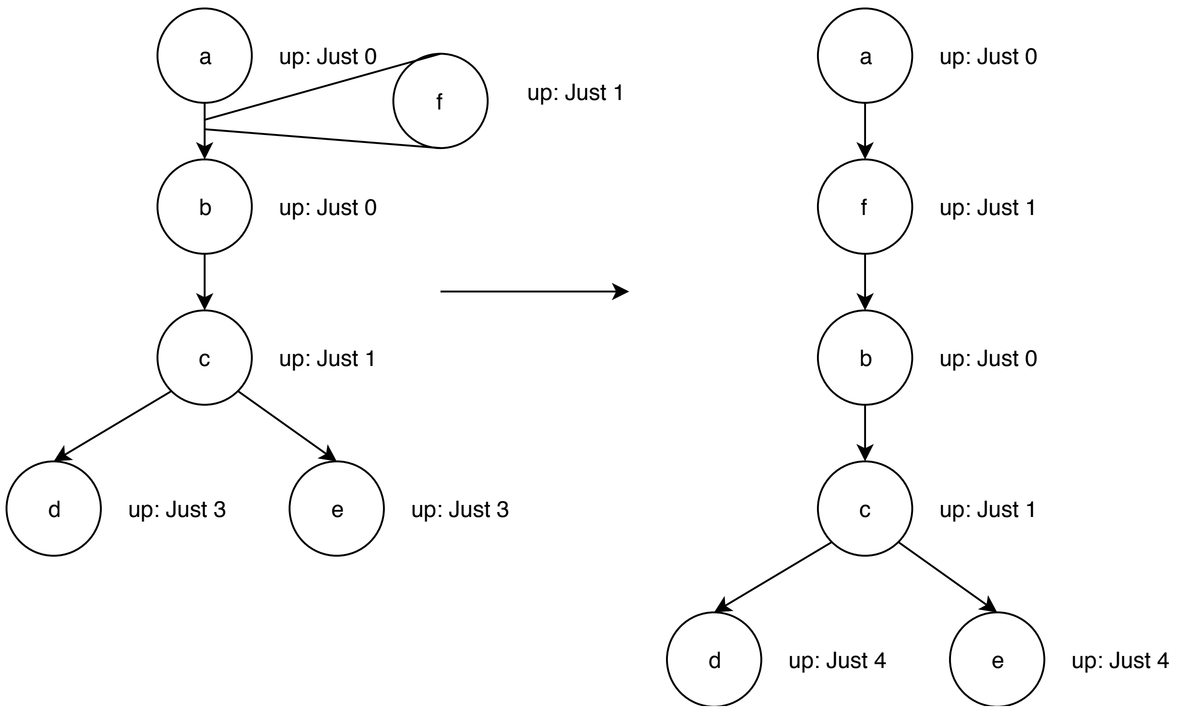

Figure 4.2: Tableau before and after adding the node

as shown in Figure 4.2:

Nodes a and b are premises. Node c is deduced from the node b and nodes d and e are deduced from the node a. We added the node f between the a node and the b node. Since the referenced node of node c is located below the added f node, we do not need to renumber the mentioned x in reference record of node c. Since the referenced node of d and e nodes is located above the added node, we need to renumber their x.

In other words, if the mentioned x would be bigger or equal to the path from added node's father minus one $path - 1$ we change it to $x + 1$. Else, there is no need to renumber. Listing 4.6 shows the code which renumbers the up variable when adding a node.

```
if x >= lengthOfPathFromAddedNodesFather-1 then
    Ref ref.str (Just (x + 1))
```

Listing 4.6: Renumbering up when adding

When we encounter $Just$ 0, there is no need to renumber. A formula which contains $up = Just$ 0 variable in the reference remains the same when adding or deleting a node. Such formula is a premise because it is referencing itself.

Chapter 5

Evaluation

To evaluate our work, we needed to test it. Because our tableau editor is designed for students of the subject *Mathematics (4) - Logic for programmers*, we asked some of the students to help us. They were given specific tasks, and they had to complete. They could ask any time they did not know what to do. After they finished the assignment, we discussed the troubles they had and the features of the editor. In the end, they were asked questions from the questionnaire 5.3.

5.1 The goal of testing

Our testing aimed to find out whether our tableau editor is intuitive, useful and practical for students. For what purpose would they use it and if they would recommend it to a classmate. We wanted to know what features they are missing in our editor and what features they consider useless or even disturbing.

5.2 Task assignments

We presented students with an assignment, where they had to prove in first-order logic, that a particular formula can result from the given theory. They were given two premises and one statement. They had to prove, that the statement can result from the given two premises. The specific assignment was:
$\{\forall x(dieta(x) \to darcek(x)), \exists x dieta(x)\} \models \exists x darcek(x)$

Another assignment included:

1. create beta sub-formulas,
2. change them,
3. delete one whole beta sub-tree.

5.3 Questionnaire

After discussing the useful and disturbing features of our editor, students were asked following questions.

1. At how many points from 0 to 10 was the user interface of our editor intuitive?
2. What purpose would you use the editor for?
3. Would you recommend the editor to your classmates?

5.4 Results of testing

We tested our editor on 12 students. We are going to summarize their feedback, and then we present the results of the questionnaire in table 5.1.

Only one of the tested students tried the original tableau editor. Others could not give us feedback in comparison with the original tableau editor.

For seven of the students, it was not intuitive enough to add a node. Four of the students found disturbing that they had to choose the node type before writing the formula when adding a node. They would like to add a default node first then write a formula and then decide what kind of node will it be. Adding a (+) button near the controls, that adds a default node would solve the problem for them. Even if the default node was an alpha.

The two inputs for substituted variable and substituting term tricked all of them when entering the substitution. On *Mathematics (4)* class, they wrote the substitution contrariwise. Five of them would improve the notation of substitution. They prefer $\{original \rightarrow new\}$ over $|new\ for\ original$. Five of them lacked visual distinction between gamma and delta as they look the same.

Five of them would improve the Help section. The original, which we also used, is not easy to read. This might be the reason, why some of the students did not read it at all and therefore had problems to construct a tableau. They would appreciate some examples or a tutorial in the Help section.

Eight of them could not find the buttons for adding, deleting and changing the node's type. There is a button labeled with the letter E (as edit) next to every node which is quite counterintuitive. They would label the button with a cogwheel to be clear, that its purpose is to show or hide the controls which manipulate the node.

Eight of them did not know how to delete the last node in a closed branch. The students were clicking on the E button next to the last node to find the delete button. Nothing happened. They did not realize that they have to open the branch first and then the rest of the buttons will appear. They did not realize that the o (open) button in the closed node is used for opening the branch. They would label the button with *open* word.

Two of them would label the button for closing the branch with *close* keyword instead of $*$ character.

All of the students were lacking some tooltip when nothing happens. 4 of them said, they would leave the application if there is no reaction to their action. In our case, we wanted to delete one whole beta tree. However, they could not do it because the beta node had some formula inside or sub-tree below. They correctly found the *delete node* button but were surprised when nothing happened when clicked. A tooltip would solve the problem.
We also considered not to wait until the user deletes the sub-tree and the formula in the particular beta sub-formula, but this behavior would be inconsistent. We would delete the whole beta sub-tree by clicking on *delete node* button. When deleting an arbitrary node of type *alpha*, *gamma* or *delta*, we delete only a single node. This is done by clicking on a button labeled with *Delete* and then clicking on an option *node*. Since the beta formulas are strongly bound to each other, there is no way one could exist without the other. Another thing when wanting to delete one beta sub-formula is that it can have a beta as a child. In this case, we can not delete the particular beta sub-formula. Therefore we did not implement the feature to delete only one beta sub-formula without deleting its sub-tree.

5.4.1 Summary

Students found the functionality for changing the node type beneficial. If they had this tool when they did not know the tableau proof yet, they would be delighted to use it. All of them would solve their homework with our tool. They also found the feature of deleting and adding the node anywhere in the proof's structure useful. We noticed they used undo-redo functionality very intuitively.

Although they found some buttons counterintuitive, they got used to them very quickly. The case when there is no reaction to action would discourage 3 of the students to use our tool.

Here is the list of the features they would implement

- Hide other section with node-modifying buttons when we have just opened one. We still cannot work with 2 sections at once.
- They would appreciate to use the latex syntax when writing special characters as $\backslash wedge$ instead of /\, $\backslash vee$ instead of \/, $\backslash rightarrow$ onstead of $->$.
- One of them would appreciate to use keyboard shortcuts to manipulate the tableau as he had a laptop and does not use a mouse.

- Two of them would implement rules as modus ponens, modul tolens and others.
- Two of them would implement a button for deleting the text in the input for the formula.
- Three of them would like to have "import example from Help section" button.

What would you use the tool for?	Number of votes
To check the correctness of my proof.	0
To prove my homework assignment.	0
Both, to prove and to check the correctness.	12
To learn how tableau proof works.	8
To learn for the exam.	2

At how many points from 1 to 10 was our tool intuitive for you?	Number of votes
6	1
6.5	3
7	4
7.5	2
8	2

Would you recommend the tool to a classmate?	Number of votes
Yes.	8
No.	0
I do not know.	4

Table 5.1: Summarized feedback from the questionnaire

Some of the features were implemented right after the testing. For example, we labeled the button for expanding node's controls with a cogwheel. We do not allow to delete a node when the proper conditions are not met. We switched the substituted variable and constant and improved the notation of the substitution. We implemented adding a default node, which internally adds an alpha node. We labeled the buttons for closing and opening a branch with *Close* and *Open* keywords. We improved the help section to be more clear and helpful. It is now easy to navigate in it. Some features do not have as high priority or would be time-consuming to implement them. For example, implementing full latex syntax or manipulating the tableau with keyboard shortcuts.

We would like to thank all of the students for helping us to test our tool.

Conclusion

In out thesis we extended and improved the functionality of the mentioned tableau editor [11]. Our tableau editor works also with first-order logic which was our goal. We improved the user interface which makes it easier for students to work with the editor. A live version of our tool is available at `https://fmfi-uk-1-ain-412.github.io/tableauEditor/`.

Our educational tool allows the student to create an analytical tableau proof. This proof is visualized as a tree. Every node in the tree contains the formula, identifier and a reference to a node which the formula was derived from. We apply four rules to formulas in nodes to create a proof tree. The rules are categorized into four types, *alpha*, *beta*, *gamma* and *delta*.

A node n can be either a premise or is derived by one of the four rules from a node d which is located somewhere above the node n. The node does not have to be derived from its parent but can be derived from any predecessor somewhere in the tree. Applying a beta rule splits the proof into two branches. Gamma and delta nodes contain a substitution.

The proof is validated every time an action is made. The editor highlights the mistakes and so our tool acquires an educational character. A mistake is shown near every invalid node.

We tested our tool on twelve students of the *Mathematics(4) - Logic for informatics* class and implemented some of their suggestions. The evaluation was very helpful, because students gave us a proper feedback. Their feedback also shows, that they would use our tool to learn how analytical tableau proofs work. The majority of the tested students would use our tool for completing a homework assignment.

Although all of the new features were implemented as designed, there are still things which would improve the user's experience. Our tool for proving by analytic tableaux is more useful than the previous one. As some students have already suggested, it would be nice to have a modus ponens or modus tolens functionality. It would be great to manipulate the tableau with keybord shortcuts. As future work, we would also like to implement the rules for equality to be able to prove in first-order logic with equality.

Appendix A: Source code

The source code of the tableau editor can be found on the attached CD. It is also available online at `https://github.com/FMFI-UK-1-AIN-412/tableauEditor/tree/master`. The compiled and functional application is available at `https://fmfi-uk-1-ain-412.github.io/tableauEditor/`.

The source code on the attached CD has the following structure:

/tableauEditor There is a licence `LICENSE` in the folder named `tableauEditor`. There are some information about how to set up the development environment in the file `README.md` as well.

/tableauEditor/build/ contains the compiled code. To open the tableau editor in the browser open `/tableauEditor/build/index.html`,

/tableauEditor/tests/ contains tests for our tableau editor,

/tableauEditor/src/ contains the source code of the tableau editor.

Bibliography

[1] Tamás Bitai. Ruzsa - Tableau Editor for Tarski's World. https://ruzsa.tbitai.me, 2017. [Online; accessed 15-May-2018].

[2] Elm community. Develpment environment. https://github.com/architectcodes/elm-live, 2017. [Online; accessed 15-May-2018].

[3] Elm community. Module which works with state history. http://package.elm-lang.org/packages/elm-community/undo-redo/latest/, 2018. [Online; accessed 15-May-2018].

[4] Evan Czaplicki. Elm lang. http://elm-lang.org/docs, 2012. [Online; accessed 15-May-2018].

[5] Evan Czaplicki. Build tool for elm application. https://github.com/elm-lang/elm-make, 2016. [Online; accessed 15-May-2018].

[6] Haim Gaifman. A hilbert type deductive system for sentential logic, completeness and compactness. http://www.columbia.edu/~hg17/ViewMathLogic/view1-deductive-system.pdf, 2002.

[7] Tom Kidd. Multiwayelmzipper. http://package.elm-lang.org/packages/tomjkidd/elm-multiway-tree-zipper/latest/MultiwayTreeZipper, 2016. [Online; accessed 15-May-2018].

[8] Ján Komara. Clausal Language- Programming Language and Proof Assistant. http://ii.fmph.uniba.sk/cl/view/?lang=sk, 2016. [Online; accessed 15-May-2018].

[9] Ján Kľuka. Module for parsing formulas in elm. https://github.com/FMFI-UK-1-AIN-412/elm-formula, 2018. [Online; accessed 15-May-2018].

[10] Ján Kľuka and Jozef Šiška. Prednášky z matematiky (4) - logiky pre informatikov. https://github.com/FMFI-UK-1-AIN-412/lpi/blob/master/docs/lecs/poznamky-z-prednasok.pdf, 2017. [Online; accessed 15-May-2018].

[11] Ján Kľuka and Jozef Šiška. Tableau Editor. `https://github.com/FMFI-UK-1-AIN-412/tableauEditor/tree/oldmaster`, 2017. [Online; accessed 15-May-2018].

[12] Zoltán Onódy. *A proof assistant for first-order logic.* Comenius University, Bratislava, 2018. Submitted.

[13] Josh Perez. Principles of flux. `https://medium.com/@goatslacker/principles-of-flux-ea872bc20772`, 2015.

[14] Raymond M. Smullyan. *First-order logic [by] Raymond M. Smullyan.* Springer-Verlag Berlin, New York [etc.], 1968.

[15] University Stanford. Boole- application for constructing truth tables. `https://ggweb.gradegrinder.net/support/manual/boole`, 2005. [Online; accessed 15-May-2018].

[16] University Stanford. Fitch- application for constructing formal proofs in first-order logic. `https://ggweb.gradegrinder.net/support/manual/fitch`, 2005. [Online; accessed 15-May-2018].

[17] University Stanford. Tarski's World. `https://ggweb.gradegrinder.net/support/manual/tarski`, 2005. [Online; accessed 15-May-2018].

[18] University Stanford. Tools and proof editors for logic. `http://intrologic.stanford.edu/applications/applications.html`, 2005. [Online; accessed 15-May-2018].

[19] Edward Z. Yang. Logitext. `http://logitext.mit.edu/main`, 2012. [Online; accessed 15-May-2018].

[20] Monika Švaralová. Výukový program demonštrujúci matematický princíp, bakalárska práca. *FMFI UK*, 2015.