Interactive Graphical Proof Editor for Natural Deduction

Daniel Juranec

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 28, 2018

Abstract

Natural deduction proofs, when done using pen and paper, can take a lot of space, be time-consuming to write, and difficult to edit. The project described in this report – a web application – allows for a possibility to edit such proofs interactively, with the help of a graphical interface, and additional functionality such as ”live” proof verification and loading and saving of proofs in progress. Verification is done employing a top-down PEG parser, and using ASTs generated for further checks. Application’s usability and ease of use was scored highly during the evaluation.

Contents

Chapter 1

Introduction

1.1 Natural deduction

In proof theory, natural deduction is a special kind of proof system, in which logical reasoning is expressed through inference rules [7]. These rules are closely related to the ”natural” way of reasoning, hence the name of the system. Modern form of natural deduction was proposed in 1934 by the German mathematician Gerhard Gentzen in his dissertation [6], which also mentioned the term ”natural deduction” for the first time. This system contrasts the Hilbert system - another kind of proof system, which uses logical axioms for the most part as opposed to inference rules.

In logical deduction, there are one or more premises (logical statements) given, and a conclusion has to be reached by following a proof system. When using natural deduction, the steps performed through the proof process use inference rules – on each step a rule is used to ”combine” one or several propositions into a new one. A simple example would be to use a Modus ponens inference rule to infer that q is true from two already known propositions – p and $p \to q$. A table listing the rules of inference can be found in Figure 1.3.

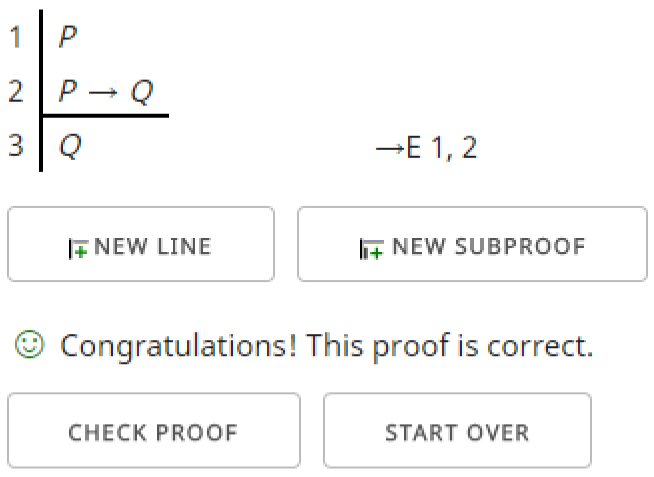
The process of inferring a final conclusion from initial propositions can be represented in a tree form, with premises on the top and final conclusion, being the tree root, at the bottom. Such form of notation is called Gentzen-style presentation. It is simple to follow and should be intuitive to a person seeing such proofs for the first time; an example using the previously described short proof can be seen in Figure 1.1. The other style, called Fitch notation, puts each assumption on a separate row, and row indentations show which assumptions are used to infer the next one (Figure 1.2).

1.2 Motivation and similar projects

Natural deduction proofs can be written on paper, but such process can be quite cumbersome – the proof tree can take a lot of space, take too much time to write, and, probably most importantly, it is difficult to edit. Another problem is that there is no way to straightforwardly check if the proof is correct.

$$\frac{\mathrm{p} \qquad \mathrm{p} \to \mathrm{q}}{\mathrm{q}} \ (\to \mathrm{E})$$

Figure 1.1: Representation of a simple logical deduction proof using Gentzen-style notation.

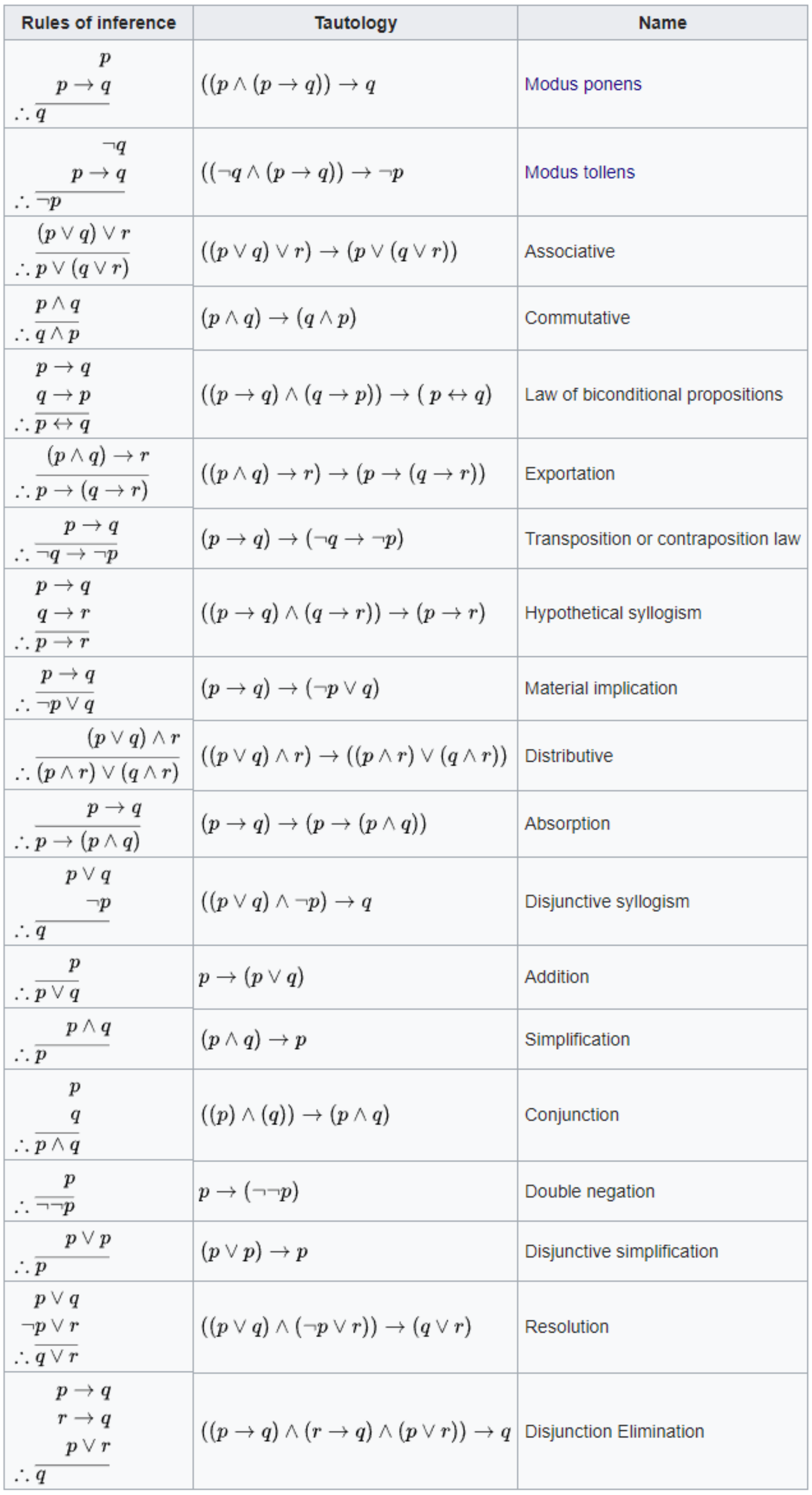


1 | P
2 | $P \to Q$
3 | Q $\to$E 1, 2

Congratulations! This proof is correct.

Figure 1.2: An example of a natural deduction editor [8] using Fitch notation.

It is possible to undertake this task on a computer, since there are programs available for editing natural deduction proofs already. They work well, but do not necessarily fit all of the requirements set in this project. Some of them [8] use Fitch notation, which, while preferred by some people, can be more difficult to understand than Gentzen-style presentation. Others [12] require installation of additional software; sometimes this can be impossible, for example on public computers – in such cases a web application is strongly preferred. Therefore with this project it was decided to create a web application (not requiring any additional plugins or software to be on user's device, just an internet browser), which would use Gentzen style to display proof progress – a somewhat novel take on already existing projects.



Rules of inference	Tautology	Name
p $p \to q$ $\therefore q$	$((p \land (p \to q)) \to q$	Modus ponens
$\neg q$ $p \to q$ $\therefore \neg p$	$((\neg q \land (p \to q)) \to \neg p$	Modus tollens
$(p \lor q) \lor r$ $\therefore p \lor (q \lor r)$	$((p \lor q) \lor r) \to (p \lor (q \lor r))$	Associative
$p \land q$ $\therefore q \land p$	$(p \land q) \to (q \land p)$	Commutative
$p \to q$ $q \to p$ $\therefore p \leftrightarrow q$	$((p \to q) \land (q \to p)) \to (p \leftrightarrow q)$	Law of biconditional propositions
$(p \land q) \to r$ $\therefore p \to (q \to r)$	$((p \land q) \to r) \to (p \to (q \to r))$	Exportation
$p \to q$ $\therefore \neg q \to \neg p$	$(p \to q) \to (\neg q \to \neg p)$	Transposition or contraposition law
$p \to q$ $q \to r$ $\therefore p \to r$	$((p \to q) \land (q \to r)) \to (p \to r)$	Hypothetical syllogism
$p \to q$ $\therefore \neg p \lor q$	$(p \to q) \to (\neg p \lor q)$	Material implication
$(p \lor q) \land r$ $\therefore (p \land r) \lor (q \land r)$	$((p \lor q) \land r) \to ((p \land r) \lor (q \land r))$	Distributive
$p \to q$ $\therefore p \to (p \land q)$	$(p \to q) \to (p \to (p \land q))$	Absorption
$p \lor q$ $\neg p$ $\therefore q$	$((p \lor q) \land \neg p) \to q$	Disjunctive syllogism
p $\therefore p \lor q$	$p \to (p \lor q)$	Addition
$p \land q$ $\therefore p$	$(p \land q) \to p$	Simplification
p q $\therefore p \land q$	$((p) \land (q)) \to (p \land q)$	Conjunction
p $\therefore \neg\neg p$	$p \to (\neg\neg p)$	Double negation
$p \lor p$ $\therefore p$	$(p \lor p) \to p$	Disjunctive simplification
$p \lor q$ $\neg p \lor r$ $\therefore q \lor r$	$((p \lor q) \land (\neg p \lor r)) \to (q \lor r)$	Resolution
$p \to q$ $r \to q$ $p \lor r$ $\therefore q$	$((p \to q) \land (r \to q) \land (p \lor r)) \to q$	Disjunction Elimination

Figure 1.3: List of rules of inference [10].

Chapter 2

Development prerequisites

2.1 Epic user story

A frequent user of the application could be a Mathematics student, currently studying logical reasoning at the university. He is tired of writing natural deduction proof trees by hand, therefore he wants to find a way to write and edit them on a computer. His preferred notation is Gentzen-style, and since he often does his work from the library or the laboratory computers, he does not want to install any additional programs. He also wants to have the possibility of saving in-progress proofs in order to work on them later on. As he is often unsure if his proofs go well, he would like the app to show him whether the assumptions he does are correct with the inference rules he selects.

2.2 User stories

The popular "As a $<$ type of user $>$, I want $<$ some goal $>$ so that $<$ some reason $>$" template for the user stories [5] was employed.

- As a user, I want to edit natural deduction proofs on a computer, so that I do not have to waste paper
- As a new student, I want to use Gentzen-style presentation, so that I can understand the proof flow easier
- As a user, I want to be able to save and load the proofs, so that my progress is not lost if I have to leave
- As a user, I want a convenient way to enter special symbols, so that I do not have to copy-paste them from elsewhere
- As a student, I want to know if my proof progress is correct, so that I can learn about my mistakes

2.3 Functional requirements

Based on the user stories, a list of functional requirements was compiled.

- Web application for editing natural deduction proofs, working with just an internet browser without additional software

- Gentzen-style tree-like presentation for the proofs
- Problem can be entered manually, or by uploading a file containing the problem in a predefined format
- User should be able to input his proof via a WYSIWYG (”what you see is what you get”) editor
- Functionality of adding nodes to the tree, editing and deleting them
- Available list of inference rules to help the user
- Buttons for frequently-used symbols ($\rightarrow$, $\neg$, etc.) for convenience
- Application should constantly verify if the proof is correct and show that on the screen (by highlighting problematic areas)
- User can save his in-progress solution to a file, to continue later by loading the file into the app

2.4 Non-functional requirements

- Easily understandable user interface
- Fast app operation without slowdowns, quick response time

2.5 Technologies used

JavaScript was chosen for the back-end part of the application for its relative simplicity and robustness. For the front-end, HTML and CSS were deemed to be sufficient; option of displaying the tree in a `<canvas>` element (usually used to draw various graphics) was explored, but the support for entering text inside it was insufficient (external libraries [11] would likely have to be used), and adding text entries separately (outside of `<canvas>`) would not be particularly user-friendly; so simple `<ul><li>` lists with custom CSS rules to make them look like trees were chosen instead.

Chapter 3

System design and implementation

3.1 System components

There are three main components in the app (Figure 3.1):

- The front-end – graphical user interface, responsible for interacting with the user
- The back-end – responsible for manipulating the data shown in the proof tree in order to display it properly
- The verifier – responsible for verifying the proof in real time

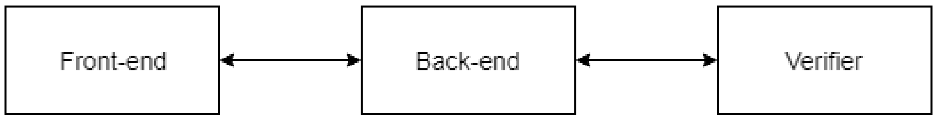


Figure 3.1: High level overview of the application.

3.2 Front-end

The front-end of the app consists of just two files. `index.html` is used for the mark-up, and `styles.css` is used for customizing the look of the HTML elements.

On initial load (Figure 3.2), there are two buttons: one for loading the problem or an in-progress proof from a file, and the other for saving the current progress to a file. For the file open button, an event listener is added on the page load, which triggers when the input is changed; for the file input it means that the listener is triggered when a file is selected.

The save button is implemented as follows: when the button is clicked, a `saveFile` function is called, which firstly converts the tree data structure into a JSON string using `JSON.stringify`. Next, a `Blob` [9] - a file-like object containing some raw data - is created, with its contents being the previously generated JSON string. In order to make the browser save the file, a hyperlink is created temporarily, linking to the blob; this link is then automatically ”clicked”, and then removed. Depending on the browser settings, the user is able to change the download location and the file name via standard operating system dialog box (Figure 3.3).

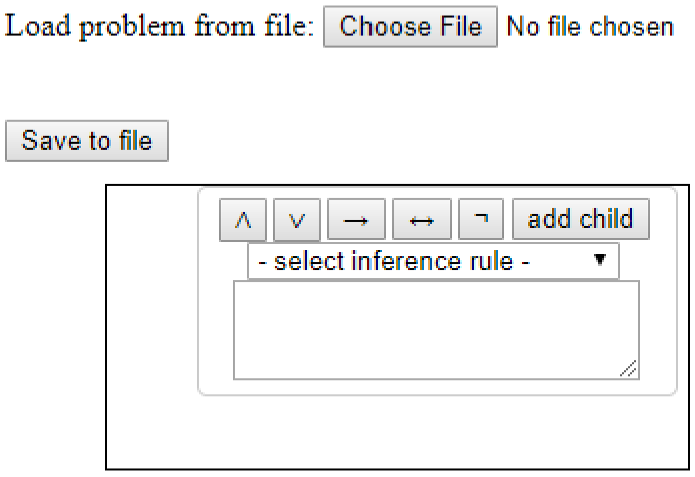
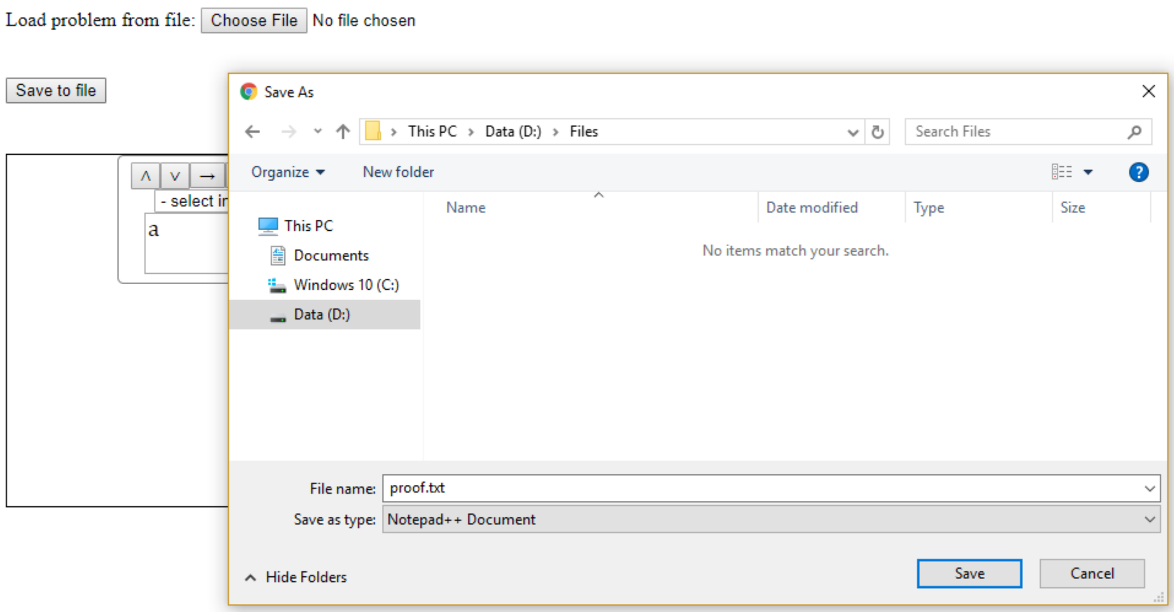


Figure 3.2: The user interface of the app on the first load.



No items match your search.

File name: proof.txt

Save as type: Notepad++ Document

Figure 3.3: Save dialog presented by Google Chrome after clicking ”Save to file”.

Below the two buttons, there is the proof tree area, with a single node added. Each node has its separate buttons for adding frequently used logic symbols, a button for adding children nodes, a button for removing the node (unless it is the root node), and a drop-down list of inference rules. When selecting an inference rule from the list, several children nodes are added automatically if necessary, so that the total number of them corresponds to the number required by the rule selected. The text area is used for entering the proof information. If the verifier determines that the node data is not correct, it marks that node in red (Figure 3.4).

To display the tree with its root at the bottom, the custom CSS rules make the whole tree flip vertically, and then flip each separate node `<div>` vertically again, using `transform:scaleY(-1)`. CSS is also used to make the `<ul>` and `<li>` elements look like a tree, with the help of `before`, `after` and `child` CSS selectors [13].

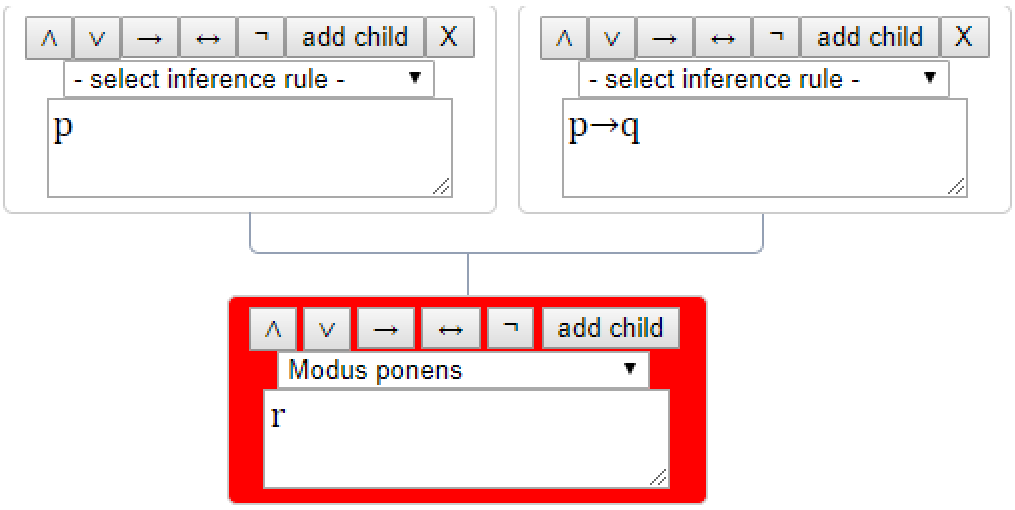


Figure 3.4: The verifier determined that the proof data is wrong, marks the problematic node accordingly.

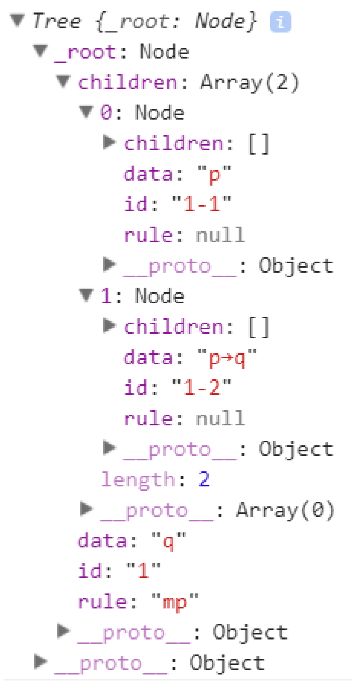
3.3 Back-end

Since the app can be considered a single-purpose app, it was decided to not use any of the popular JavaScript frameworks such as AngularJS [1] – they would be too complex for the purposes of this particular app. Instead, simple JavaScript scripts are used. Button presses from the front-end call the appropriate JavaScript functions, which modify the internal data structures accordingly, and then change the HTML code of the page to show the changes.

The data structure responsible for containing the data from the proof tree is a tree-like structure. Each node element of the tree has an ID, a string corresponding to the text currently entered into the node, a reference to the inference rule used, and a list of children nodes (Figure 3.5). From the logical standpoint, all children nodes of any single node are ”combined” to infer that node using the inference rule defined in it. A number of children nodes is technically unlimited – zero children nodes would mean that this node is the initial assumption of the problem; the number of them is, however, usually limited by the inference rule used.

On the first application load, the data structure (Figure 3.6) and the HTML code for the first empty node are already there. The drop-down list of inference rules is generated dynamically based on the dictionary, in which full rule names (values) are mapped to two-letter abbreviations (keys), e.g. ”Modus ponens” is mapped to ”mp”. Short name versions (as opposed to full names or numerical IDs) allowed for easier development.

Buttons for adding special symbols into the text fields are able to determine the correct place for them based



```
▼ Tree {_root: Node} i
  ▼ _root: Node
    ▼ children: Array(2)
      ▼ 0: Node
        ▶ children: []
          data: "p"
          id: "1-1"
          rule: null
        ▶ __proto__: Object
      ▼ 1: Node
        ▶ children: []
          data: "p→q"
          id: "1-2"
          rule: null
        ▶ __proto__: Object
        length: 2
      ▶ __proto__: Array(0)
      data: "q"
      id: "1"
      rule: "mp"
    ▶ __proto__: Object
  ▶ __proto__: Object
```

Figure 3.5: The data structure for the proof from Figure 1.1.

```
function Tree(data) {
    var node = new Node("1", data);
    this._root = node;
}
```

Figure 3.6: Tree data structure created on first load.

on current cursor position (Figure 3.7); the value after the button press is therefore the text before the cursor, concatenated with the symbol selected, and with the text after the cursor. This functionality was added after user feedback (section 4.3).

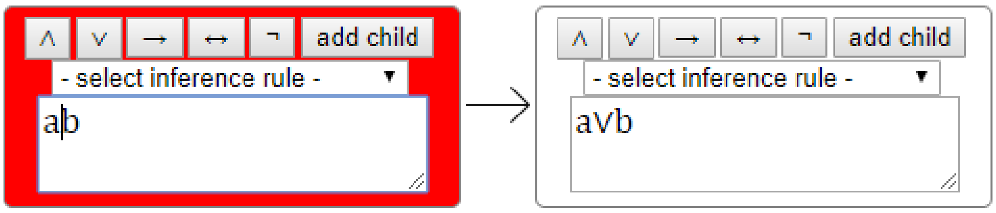
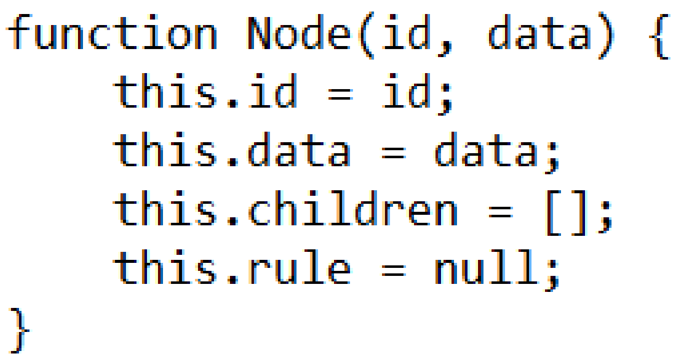


Figure 3.7: Symbol buttons insert the character at the cursor position.

”Add child” button calls the `addChild` function, which creates new `<ul>` and `<li>` elements (based on whether a node has any children nodes already), creates a new Node variable (Figure 3.8), and adds it to the tree data structure as a child of the relevant node.



```
function Node(id, data) {
    this.id = id;
    this.data = data;
    this.children = [];
    this.rule = null;
}
```

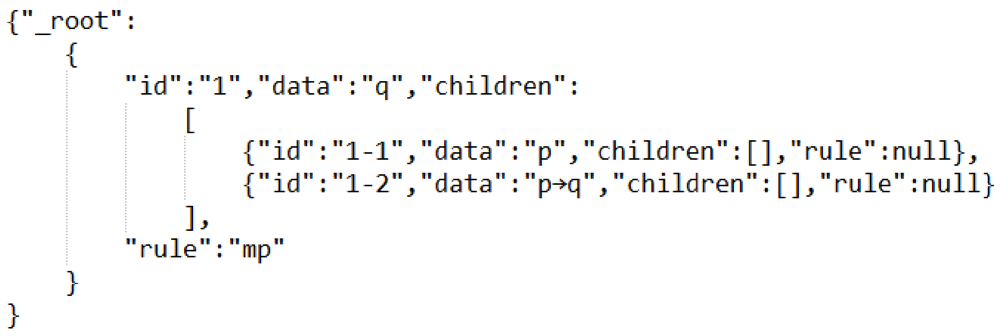
Figure 3.8: Constructor for the Node object type.

Similarly, changing the rule from the drop-down list calls the `changeRule` function. It calls the `addChild` function as many times as necessary, depending on the inference rule selected and the current number of children nodes. For example, if the Modus ponens rule is chosen, and there is already one child node present, only one new node is added. The `rule` field of the node is also modified to store the two-letter abbreviation of the rule.

The file opening functionality is implemented as follows. Once the event listener mentioned in section 3.2 is triggered, the contents of the file are stored into a variable. Since the contents are a JSON string (Figure 3.9), they can be recreated back into an object using `JSON.parse`. However, this object cannot be directly used by the app, as it is of generic Object type, and not Tree type, as required by the app. Therefore, the object has to be iterated through to retrieve the necessary data and put it into the Tree object. A recursive function is used, which firstly retrieves the data from the root node, and then does the same for the children nodes, if there are any. At the same time the HTML elements are also added to the page via the `addChild` function.

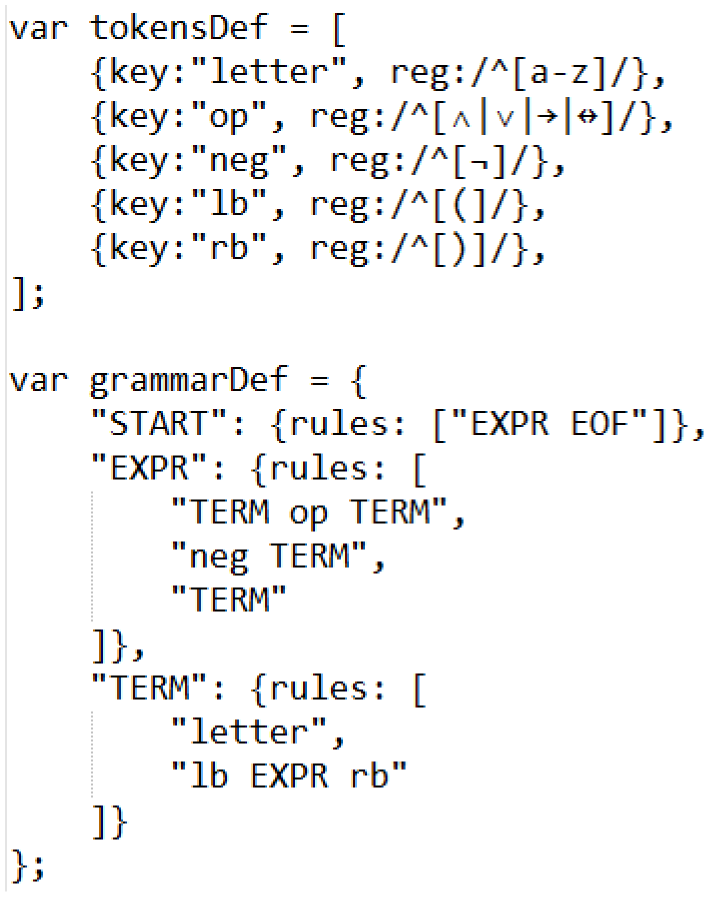
3.4 Verifier

For verification of the proof progress, several components are used. The first one is an top-down parsing expression grammar (PEG) parser EPEG.js [3], which interprets the logical expressions used in the proof as a formal language. It validates the expressions received from the back-end component according to the predefined grammar (Figure 3.10). The variable returned from the parser (Figure 3.11) is an abstract syntax tree (AST), which is then used by the next verification component - the inference rules checker.



```
{"_root":
    {
        "id":"1","data":"q","children":
            [
                {"id":"1-1","data":"p","children":[],"rule":null},
                {"id":"1-2","data":"p→q","children":[],"rule":null}
            ],
        "rule":"mp"
    }
}
```

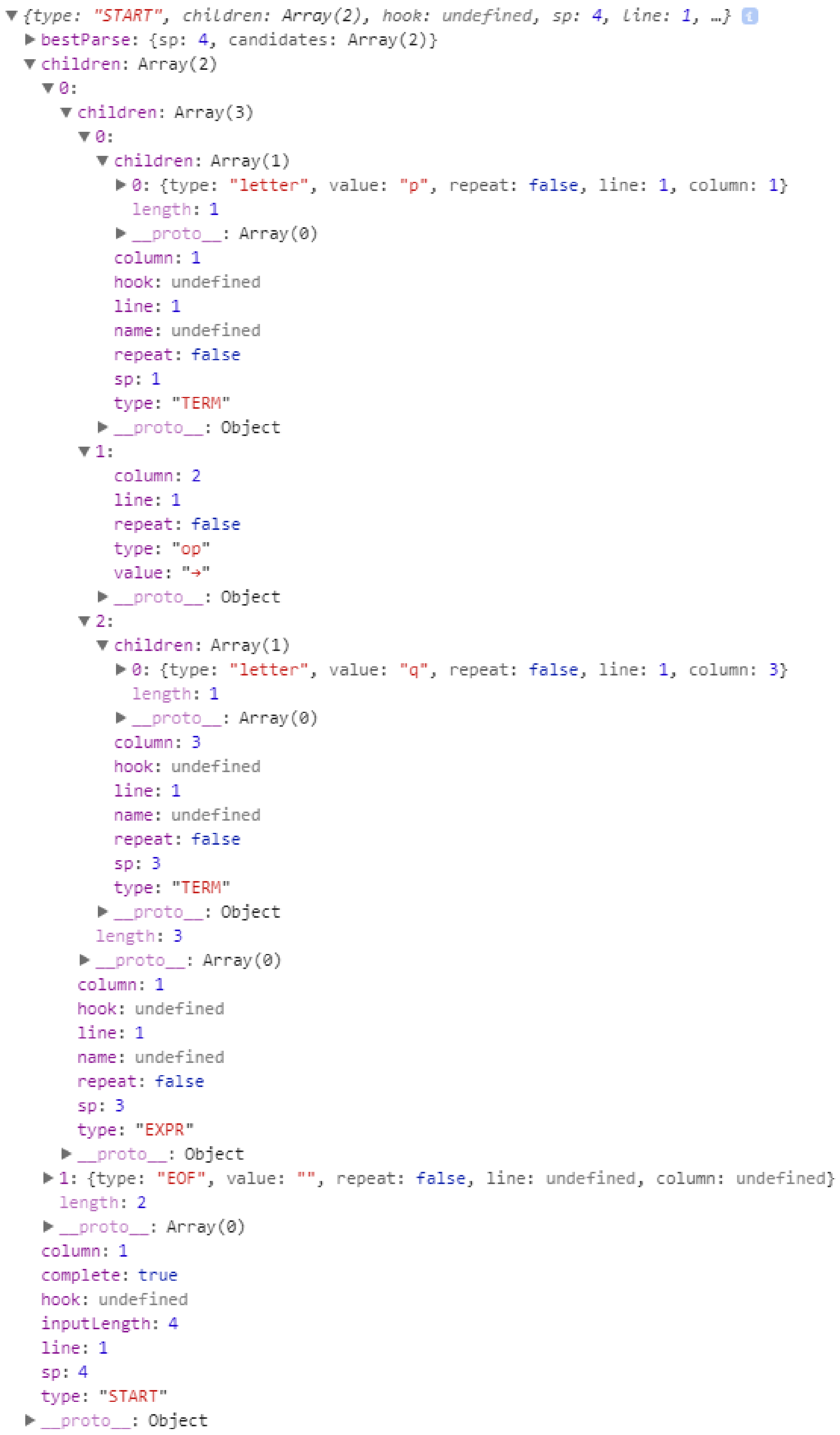
Figure 3.9: Contents of the text file with a saved proof (formatted for readability).



```
var tokensDef = [
    {key:"letter", reg:/^[a-z]/},
    {key:"op", reg:/^[∧|∨|→|↔]/},
    {key:"neg", reg:/^[¬]/},
    {key:"lb", reg:/^[(]/},
    {key:"rb", reg:/^[)]/},
];

var grammarDef = {
    "START": {rules: ["EXPR EOF"]},
    "EXPR": {rules: [
        "TERM op TERM",
        "neg TERM",
        "TERM"
    ]},
    "TERM": {rules: [
        "letter",
        "lb EXPR rb"
    ]}
};
```

Figure 3.10: Grammar and tokens definitions for the EPEG.js parser.

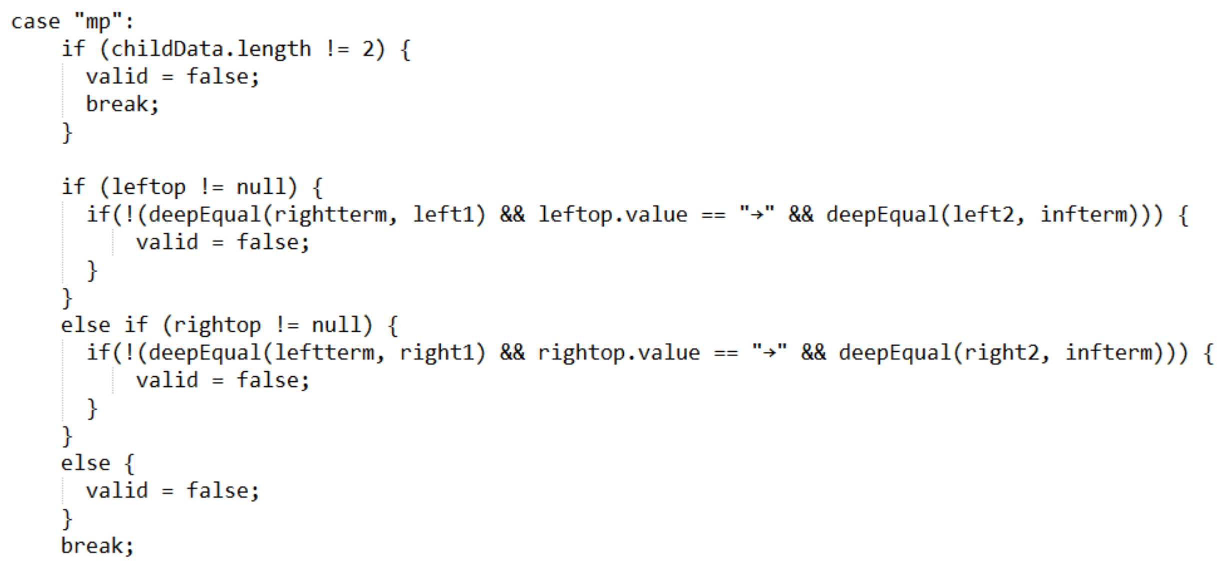


```
▼{type: "START", children: Array(2), hook: undefined, sp: 4, line: 1, …} i
  ▶bestParse: {sp: 4, candidates: Array(2)}
  ▼children: Array(2)
    ▼0:
      ▼children: Array(3)
        ▼0:
          ▼children: Array(1)
            ▶0: {type: "letter", value: "p", repeat: false, line: 1, column: 1}
              length: 1
            ▶__proto__: Array(0)
            column: 1
            hook: undefined
            line: 1
            name: undefined
            repeat: false
            sp: 1
            type: "TERM"
          ▶__proto__: Object
        ▼1:
            column: 2
            line: 1
            repeat: false
            type: "op"
            value: "→"
          ▶__proto__: Object
        ▼2:
          ▼children: Array(1)
            ▶0: {type: "letter", value: "q", repeat: false, line: 1, column: 3}
              length: 1
            ▶__proto__: Array(0)
            column: 3
            hook: undefined
            line: 1
            name: undefined
            repeat: false
            sp: 3
            type: "TERM"
          ▶__proto__: Object
          length: 3
        ▶__proto__: Array(0)
        column: 1
        hook: undefined
        line: 1
        name: undefined
        repeat: false
        sp: 3
        type: "EXPR"
      ▶__proto__: Object
    ▶1: {type: "EOF", value: "", repeat: false, line: undefined, column: undefined}
      length: 2
    ▶__proto__: Array(0)
    column: 1
    complete: true
    hook: undefined
    inputLength: 4
    line: 1
    sp: 4
    type: "START"
  ▶__proto__: Object
```

Figure 3.11: Abstract syntax tree data structure for $p \rightarrow q$ expression.

If the AST returned by the parser is incomplete, the node can already be marked as invalid, and no future checks are required. However, if the AST is complete, then the parser also performs its task on the children nodes of that node, getting their ASTs. The syntax trees are parsed to retrieve the necessary elements, on which the inference rules are then checked.

Each inference rule has its own `case` in the `switch(rule)` operator. Firstly, the number of children nodes is checked, as it is the simplest check to perform; for example, Modus ponens requires two children nodes, so if there are only one, or three and more of them, the node can already be marked as invalid. Next, the element values which should be matching are compared. To compare values, a custom `deepEqual` function is run – AST objects provided by EPEG.js contain several fields which are unnecessary for the comparison to be performed properly. Therefore, those fields are removed; then the elements are compared. A sample inference rule check case for Modus ponens can be seen in Figure 3.12.



```
case "mp":
    if (childData.length != 2) {
        valid = false;
        break;
    }

    if (leftop != null) {
        if(!(deepEqual(rightterm, left1) && leftop.value == "→" && deepEqual(left2, infterm))) {
            valid = false;
        }
    }
    else if (rightop != null) {
        if(!(deepEqual(leftterm, right1) && rightop.value == "→" && deepEqual(right2, infterm))) {
            valid = false;
        }
    }
    else {
        valid = false;
    }
    break;
```

Figure 3.12: Modus ponens inference rule check case.

The verifier function for a node is run in these cases:

- When the text of the node is changed
- When the text of any of the children nodes is changed
- When the inference rule of the node is changed

This ensures the quickest feedback to the user, as there is no need to press a separate button for the check to be performed.

Chapter 4

Evaluation

To evaluate the application, a formative usability evaluation was conducted. Five Computing Science students were invited to take part in an experiment, where a logical problem was given, and the participants had to solve it using the application. The performance was observed, and afterwards a survey was conducted to collect the feedback.

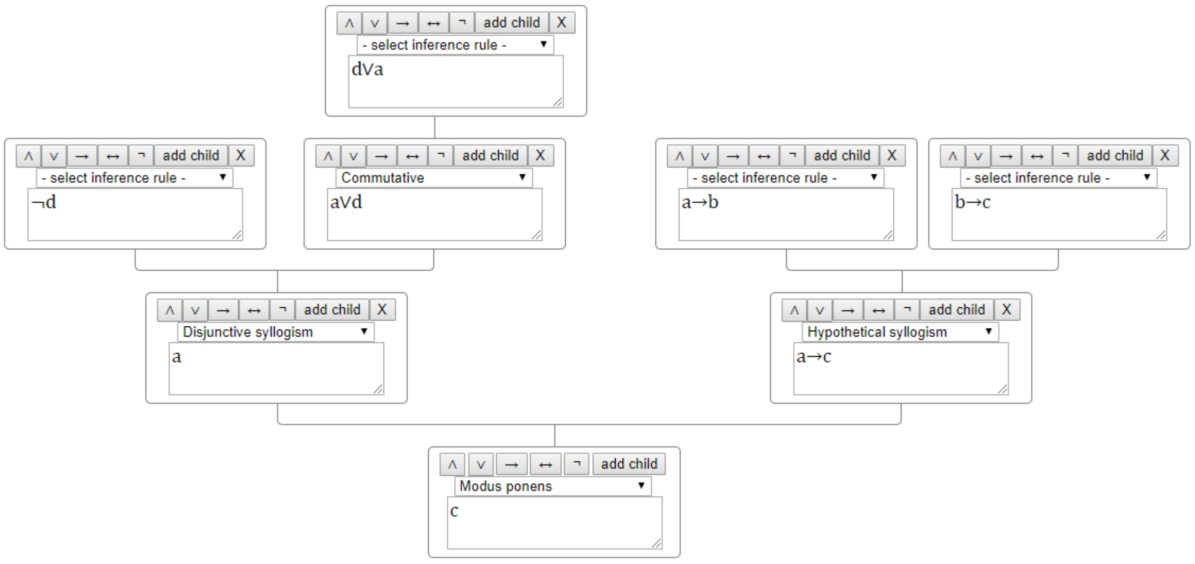


Figure 4.1: Solved evaluation problem.

4.1 Observation

All of the participants were aware of the concepts of logic proofs and inference rules, so no additional explanation was needed. During the proof, some difficulties occurred in regards to the inference rules – since only the names of the rules are displayed in the drop-down list, the participants did not know what each of those meant. The rules relevant to the completion of the exercise therefore had to be explained verbally. No other major problems were encountered during the evaluation; the completion time for the exercise was between 2 and 4 minutes.

4.2 Survey results

To collect the feedback, the System Usability Scale (SUS) was used [4]. It is a 10 items questionnaire with answers varying from ”Strongly disagree” to ”Strongly agree”. The questions are as follows:

- I think that I would like to use this system frequently
- I found the system unnecessarily complex
- I thought the system was easy to use
- I think that I would need the support of a technical person to be able to use this system
- I found the various functions in this system were well integrated
- I thought there was too much inconsistency in this system
- I would imagine that most people would learn to use this system very quickly
- I found the system very cumbersome to use
- I felt very confident using the system
- I needed to learn a lot of things before I could get going with this system

The answers are converted to numeric values, added together and multiplied by 2.5 to provide a value in the range of 0-100. The SUS score of 68 is considered average [2]. The summary of the results is as follows:

Sample size	Mean	Standard deviation	95% confidence interval
5	81.5	7.62	[72.03, 90.97]

The results indicate that the application is fairly easy to use – its SUS score is well above average. Considering that the participants were all Computer Science students with some Maths background, the results might not be very representative; Mathematics students (considered a major part of the application's target user population), while likely having more experience with natural deduction proofs, are possibly not as skillful in using web apps. A more representative study could be conducted, including Mathematics students from various years, and possibly students from other courses who have Maths in their curriculum.

4.3 Actions taken

As evidenced during the evaluation, some explanation on the inference rules and their meaning was lacking. As a result, a link to the table with all of the inference rules and usage examples was added next to the proof area.

Several bugs in regards to proof verification status – the red node background – not being updated frequently enough, and special symbols buttons inserting the characters in the wrong places, were noticed by the evaluation participants; those bugs were fixed afterwards.

Chapter 5

Conclusion

5.1 Summary

The objective of this project was to create an easy to use web application for editing natural deduction logical proofs. The final version of the app meets the functional requirements set in Chapter 2 of this report. Results of the conducted usability evaluation indicate that the non-functional requirements were met as well. The application could potentially be used by students in their learning process.

5.2 Future work

Even though the application is usable in its current state, various other functionality could be added. At this point, saving and loading of problems is done through text files – the task of storing of them lies on the user. A user registration system with a database could be added, so that all in-progress proofs would be stored on a server.

The app uses Gentzen-style notation at the moment. As the verifier functionality is already in place, Fitch notation support could also potentially be added to the app.

5.3 Acknowledgements

- Dr. John O'Donnell for his involvement and guidance throughout the project development.
- Evaluation participants for their time and feedback provided.

Appendices

Appendix A

Ethics checklist

School of Computing Science
University of Glasgow

Ethics checklist for 3rd year, 4th year, MSci, MRes, and taught MSc projects

This form is only applicable for projects that use other people ('participants') for the collection of information, typically in getting comments about a system or a system design, getting information about how a system could be used, or evaluating a working system.

If no other people have been involved in the collection of information, then you do not need to complete this form.

If your evaluation does not comply with any one or more of the points below, please submit an ethics approval form to the Department Ethics Committee.

If your evaluation does comply with all the points below, please sign this form and submit it with your project.

---

1. Participants were not exposed to any risks greater than those encountered in their normal working life.
   Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g. ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback

2. The experimental materials were paper-based, or comprised software running on standard hardware.
   Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, mobile phones, and PDAs is considered non-standard.

3. All participants explicitly stated that they agreed to take part, and that their data could be used in the project.
   If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant.

   Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.

4. No incentives were offered to the participants.
   The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

5. No information about the evaluation or materials was intentionally withheld from the participants.
   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

6. No participant was under the age of 16.
   Parental consent is required for participants under the age of 16.

7. No participant has an impairment that may limit their understanding or communication.
   Additional consent is required for participants with impairments.

8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
   A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. All participants were informed that they could withdraw at any time.
   All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.

10. All participants have been informed of my contact details.
    All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.

11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
    The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation.

12. All the data collected from the participants is stored in an anonymous form.
    All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

Project title Interactive Graphical Proof Editor for Natural Deduction

Student's Name Daniel Jwanec

Student's Registration Number 2145171

Student's Signature [illegible]

Supervisor's Signature ______

Date 27/02/18

Bibliography

[1] AngularJS. https://angularjs.org.

[2] System usability scale (SUS). https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html.

[3] Batiste Bieler. EPEG.js - expressive parsing expression grammar. https://github.com/batiste/EPEG.js.

[4] John Brooke. SUS: A quick and dirty usability scale. http://www.usabilitynet.org/trump/documents/Suschapt.doc.

[5] Mike Cohn. User stories and user story examples. https://www.mountaingoatsoftware.com/agile/user-stories.

[6] Gerhard Karl Erich Gentzen. *Untersuchungen ber das logische Schlieen. I.* Mathematische Zeitschrift, 1934.

[7] Andrzej Indrzejczak. Natural deduction. http://www.iep.utm.edu/nat-ded/.

[8] Kevin Klement. Natural deduction proof editor and checker. http://proofs.openlogicproject.org.

[9] Mozilla and individual contributors. Blob - web APIs. https://developer.mozilla.org/en-US/docs/Web/API/Blob.

[10] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw Hill, fifth edition, 2003.

[11] James Simpson. HTML5 canvas text input. https://github.com/goldfire/CanvasInput.

[12] Bernard Sufrit and Richard Bornat. Jape. http://www.cs.ox.ac.uk/people/bernard.sufrin/personal/jape.org/.

[13] TheCodePlayer. CSS3 family tree. http://thecodeplayer.com/walkthrough/css3-family-tree.