

University of Edinburgh

# ClickDeduce

Interactive Inference Rule Exploration

Richard Holmes  
4-4-2024

## Abstract

This project was focused on the design and implementation of “ClickDeduce” [1] – an online tool for creating evaluation and type-checking trees for expressions. It is tailored to closely follow the content of the Elements of Programming Languages course, aiming to be a helpful utility for students completing this course. While user testing has not yet been performed, the application has been evaluated with a combination of accessibility guidelines, performance testing, and analysing expected user journeys. Following a successful implementation it is now freely available online and a wider scope and a range of improvements are planned for the second year of this project.

# Table of Contents

<b>ABSTRACT .....</b>	<b>1</b>
<b>TABLE OF CONTENTS .....</b>	<b>2</b>
<b>1 INTRODUCTION .....</b>	<b>4</b>
<b>2 BACKGROUND .....</b>	<b>5</b>
2.1 ABSTRACT SYNTAX TREES .....	5
2.2 INFERENCE RULES.....	5
2.3 ELEMENTS OF PROGRAMMING LANGUAGES COURSE .....	8
<b>3 RELATED WORKS .....</b>	<b>9</b>
3.1 MACLOGIC.....	9
3.2 BOB ATKEY'S INTERACTIVE NATURAL DEDUCTION .....	9
3.3 HOLBERT .....	11
3.4 PREVIOUS PROJECT BY VALENTAS DICEVICIUS.....	11
<b>4 DESIGN .....</b>	<b>13</b>
4.1 TARGET AUDIENCE .....	13
4.2 GOALS .....	13
4.3 PRIMARY DEVELOPMENT LANGUAGE .....	14
4.4 PLATFORMS.....	14
4.5 ARCHITECTURE.....	14
4.6 PURPOSE .....	16
4.7 INTERFACE FUNCTIONS.....	16
4.8 LANGUAGES .....	19
4.9 TYPE INFERENCE .....	20
<b>5 IMPLEMENTATION .....</b>	<b>21</b>
5.1 OVERVIEW .....	21
5.2 LANGUAGE FEATURES.....	21
5.3 ACTIONS API .....	23
5.4 VIEW MODES .....	23
5.5 TREE STRUCTURE .....	24
5.6 PHANTOM TREES.....	25
5.7 COMMAND HISTORY .....	26
5.8 TRANSITION FROM SERVER-BASED TO SERVERLESS.....	26
5.9 STACK OVERFLOW.....	27
5.10 SAVING AND LOADING .....	27
5.11 ERROR HANDLING.....	28
5.12 CONVERTING TO HTML .....	28
5.13 CONVERTING TO LATEX .....	29
5.14 INTERFACE .....	30
5.15 GUIDE.....	31
5.16 EXPRESSION SELECTION .....	32
5.17 NESTED EXPRESSION ELEMENTS .....	33
5.18 BINDING AND SCOPE .....	33
5.19 CONTINUOUS DEPLOYMENT AND INTEGRATION .....	34

<b>6</b>	<b>EVALUATION .....</b>	<b>35</b>
6.1	AUTOMATED TESTS .....	35
6.2	ACCESSIBILITY.....	35
6.3	PERFORMANCE.....	35
6.4	EXAMPLE USER JOURNEYS.....	37
<b>7</b>	<b>CONCLUSION .....</b>	<b>41</b>
7.1	NEXT STEPS.....	41
<b>8</b>	<b>REFERENCES.....</b>	<b>43</b>
	<b>APPENDIX 1: LATEX OUTPUT EXAMPLES .....</b>	<b>45</b>
	<b>APPENDIX 2: LANGUAGE FEATURE EXAMPLES .....</b>	<b>46</b>
	EXPRESSIONS.....	46
	TYPES.....	47
	VALUES.....	47
	<b>APPENDIX 3: EPL LANGUAGE FEATURES.....</b>	<b>48</b>

# 1 Introduction

Computer Science students may be familiar with writing code, but learning about more abstract concepts surrounding programming languages requires different skills. One common approach for describing the properties of a feature of a programming language is using inference rules. These follow syntax more commonly used in logic, which students may be less familiar with.

At the University of Edinburgh, the “Elements of Programming Languages” course is offered to 3<sup>rd</sup> and 4<sup>th</sup> year students. It presents a wide range of expressions that are fundamental to many programming languages. These are introduced as inference rules and are used in a fairly abstract manner.

Writing out the derivation for an expression requires a good understanding of both the rules used and how to construct a derivation tree correctly. On paper, it is easy to run out of space or make a small mistake early in the derivation which cannot be swiftly corrected. Students could benefit from a more accessible and interactive way to learn about both derivation trees and inference rules for the programming language features they learn about.

While tools exist for such a purpose, most of them are designed for much more advanced users and focus on logic more broadly, rather than programming languages. Setting up the required rules for the languages in the course is not realistic for a student new to the concept.

ClickDeduce is a simple application which closely follows the structure of the “Elements of Programming Languages” course and is intended to be used by students on the course. It aims to make learning about abstract programming language features and constructing corresponding derivation trees much more accessible.

The application also aims to be easily extensible, so that others wanting to use the tool could make adjustments to the existing language features or implement entirely new ones.

This report delves into the problem space, design, implementation, and evaluation of ClickDeduce. We begin by understanding the background context, including abstract syntax trees, inference rules, and the Elements of Programming Languages course. We then proceed to analyse other existing tools that tackle similar problems.

Multiple design problems are then inspected, such as the target audience and which platforms to develop for. An exploration of the possible system architectures, such as a website with a dedicated backend server, or a website with all functionality browser-side. After this, interesting and challenging aspects of the implementation are discussed.

Finally, the tool is evaluated using accessibility guidelines, performance testing, and demonstrating example user journeys.

## 2 Background

### 2.1 Abstract Syntax Trees

Abstract syntax trees are used in computer science as a visual representation of code. Each node in the tree represents a code construct – usually an expression or statement. It is useful for making the structure of code apparent since they are normally part of compact lines of text.

Each node in the tree represents a construct in the source code, such as expressions or statements. Different kinds of nodes represent different constructs.

For example, the abstract syntax tree for the expression  $(1 + 2) \times 3$  is shown in Figure 1. Each number and operator has its own node. It has a top-down structure where the root node is at the top of the tree.

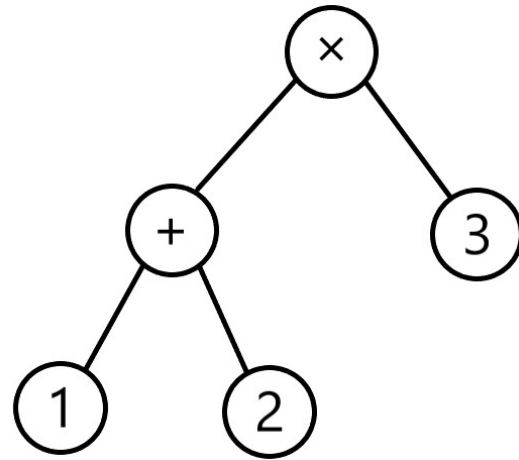


Figure 1: Abstract syntax tree for  $(1 + 2) \times 3$

Both addition and multiplication are binary operators, meaning they each have two subexpressions as arguments. The numbers are variables – leaf nodes with no arguments.

#### 2.1.1 Abstract Binding Trees

A common feature in programming languages is the ability to bind a value to a variable to be used somewhere else. For example, in the expression  $\text{let } x = 4 \text{ in } x + 2$ , the value of 4 is bound to  $x$ , which is then reused in  $x + 2$ .

This binding only occurs within a particular scope, for instance, the expression  $(\text{let } x = 4 \text{ in } x + 2) + x$  is invalid (assuming it is not part of a larger expression) because the variable  $x$  is referenced in a scope where it is not bound. This is shown in Figure 2.

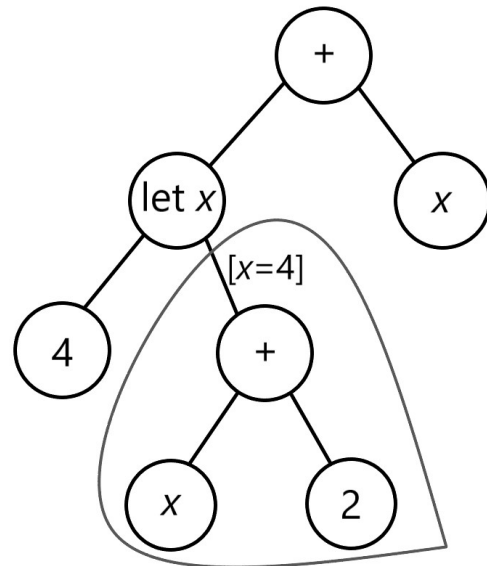


Figure 2: Binding syntax tree for:  
 $(\text{let } x = 4 \text{ in } x + 2) + x$

### 2.2 Inference Rules

In the realm of logic and reasoning, inference rules define valid steps that can be taken between premises and a conclusion. For example, if I know that it is currently snowing and it is always cold when it is snowing, then I could conclude that it is currently cold. This can be expressed in an inference rule as follows:

$$\frac{\text{Snowing} \quad \text{Snowing implies Cold}}{\text{Cold}}$$

This specific usage of inference rules is called a judgement. Any number of premises can appear above the line, with a single conclusion below the line. The premises of the judgement must be valid for the use of the rule to be correct.

### 2.2.1 Axioms

An axiom is simply a rule which has zero premises. This means that the conclusion is always true. Any route taken through the valid resulting tree will end in an axiom; they act as terminals.

### 2.2.2 Terms for Inference Rules in Programming Languages

- **Expression:** the basic unit of a written program. An expression is associated with a particular operation (e.g. addition or multiplication) and can contain other expressions as arguments. Throughout this report, they will be shown in either named form or symbolic form. A named form expression could appear like “Add(Times(1,2), 3)” which in symbolic form would be “ $(1 \times 2) + 3$ ”.
- **Value:** the result of evaluating an expression. There is a variety of possible values depending on the rules of the language. Examples include integers and Booleans. Each value also has a corresponding type.
- **Type:** the possible kinds of values in a language. In a simple arithmetic language, it may be the case that only integers are values. More complex languages could have types such as strings or even types which are parameterised by other types (e.g. a list of integers).
- **Literal:** essential parameters to some expressions, cannot be evaluated or type-checked on their own. For example, for the expression “Num(78)”, 78 is a literal, or in “Lambda( $x$ , Integer, Times(2,3))”, the variable name “ $x$ ” is a literal. Note that it is important to distinguish between a schematic symbol that appears in a type inference or evaluation rule and the concrete variable names used in expressions (such as the literal “ $x$ ” in this example).

In summary, a well-formed expression can be evaluated to a value with a particular type.

Imperative programming languages also introduce statements, where each statement is an individual action that is carried out rather than something that produces a value, but these are not included in this project.

### 2.2.3 Evaluation

How an expression should be evaluated in a programming language can be expressed similarly to how type-checking rules can be expressed using inference rules.

Throughout this project, we will use the style of big-step evaluation introduced in the Elements of Programming Languages Course.

The essence of big-step evaluation is captured in its name: when evaluating an expression, we take a “big step” to directly obtain the final value without recording or detailing intermediate computational steps or states. The notation for big-step evaluation is the double-down arrow ( $\Downarrow$ ), which indicates that an expression evaluates to a certain value.

For example, here is an inference rule for an addition operation using big-step evaluation:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2}$$

The “ $+_{\mathbb{N}}$ ” operator is employed to denote the numeric addition operation, distinguishing it from the symbolic “ $+$ ” used for the expression itself.

By using these judgements, an inference rule derivation tree can be constructed. It is similar to an abstract syntax tree, but each node represents the usage of an evaluation rule and it grows bottom-upwards.

Here is an example of the big-step evaluation tree for  $(1 + 2) + 4$ :

$$\frac{\frac{\frac{1 \Downarrow 1}{1 + 2 \Downarrow 3} \quad \frac{2 \Downarrow 2}{4 \Downarrow 4}}{(1 + 2) + 4 \Downarrow 7}}$$

## 2.2.4 Type-Checking

Type-checking is a fundamental aspect of many modern programming languages, designed to ensure that programs adhere to specified behaviours based on the types assigned to values. Within these systems, every value is associated with a specific type that dictates its behaviour and interactions within the program.

Each operation or function in a program expects input values of certain types and often produces an output of a specific type. When these operations receive values of the expected types, they behave as intended. However, when provided with values of mismatched or unexpected types, they can cause unexpected or erroneous behaviours.

There are two primary approaches to type-checking:

1. **Static Type-Checking:** Here, the type-checker scrutinizes the code for type errors before the program is executed, often during the compilation phase. This method offers the advantage of catching type-related errors early on, even before the program runs. It provides clear feedback to developers about where the issue lies, enabling timely and precise corrections. However, in some cases, it may result in more verbosity or boilerplate code. Languages like C, Java, and Haskell are examples of statically typed languages.
2. **Dynamic Type-Checking:** In dynamically checked systems, the validation of types happens during the program's execution, at runtime. This means that type-related errors only surface when the problematic piece of code is executed. While this offers flexibility in writing code and can be more lenient in some scenarios, it can also lead to runtime errors which might be harder to debug since the program may fail unexpectedly during its execution. Python, Ruby, and JavaScript are examples of dynamically typed languages.

Here is a type-checking rule for multiplying integers. A turnstile ( $\vdash$ ) precedes the expression which is then followed by a colon “ $:$ ” and then the type of the expression.

$$\frac{\vdash e_1 : \text{Integer} \quad \vdash e_2 : \text{Integer}}{\vdash e_1 \times e_2 : \text{Integer}}$$

Consider the example of the rule for multiplying numbers in a simple example language which is statically type-checked. It is shown above as requiring both the left and right-hand sides of the expression to type-check to integers so that it can produce another integer. The premises of the rule are that both the sub-expressions will be integers. The conclusion of this rule is that the



expression will result in an integer. From the perspective of the type-checker, the exact values of the inputs and the result are unimportant; it only cares about which type they are.

Attempting to multiply an integer by a string is not supported and would result in an error during type-checking.

$$\frac{\frac{}{\vdash 5: \text{Integer}} \quad \frac{}{\vdash \text{"Hello"}: \text{String}}}{\vdash 5 \times \text{"Hello"}: \text{ERROR}}$$

In essence, type-checking acts as a safeguard, ensuring that operations within a program receive appropriate data types, thereby preventing potential type-induced anomalies. This enhances the robustness and predictability of software, contributing to the creation of reliable and error-free applications.

## 2.3 Elements of Programming Languages Course

As mentioned in the introduction, the Elements of Programming Languages course [2] is taught at the University of Edinburgh by the School of Informatics. It teaches students about common programming language features, using inference rules to define simple versions of them.

### 2.3.1 Languages

Throughout the Elements of Programming Languages course, a series of abstract languages are introduced to students. These begin with the simple LArith language, which implements integers, addition, and multiplication. This is followed by the LIf language, extending LArith with Boolean values and if-then-else expressions.

This pattern of extending the previous language with new features continues, adding variables, lambda functions, recursive functions, pair and union values, polymorphic types, and loops. Other language features such as objects and classes are mentioned during the course, but not formalised in a language that the students are expected to use.

## 3 Related Works

### 3.1 MacLogic

This is a deprecated piece of software written exclusively for the Classic Macintosh – the latest version of the operating system was released in 1999. Despite this, it is still regarded by some [3] as one of the best educational programs for natural deduction proofs.

It is designed for logic rather than programming languages. Due to this difference and the lack of a tree interface, it does not seem particularly relevant to this project. There may be some helpful features in its interface to be analysed, but without access to a functional version of it, this was not possible.

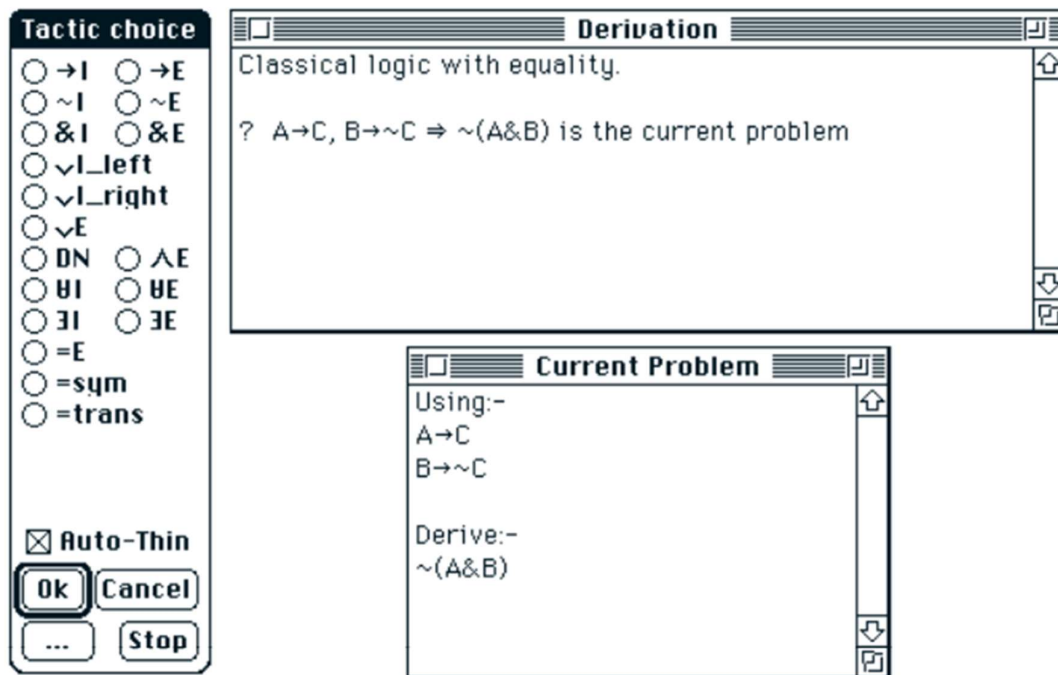


Figure 3: Example of MacLogic interface [4]

### 3.2 Bob Atkey's Interactive Natural Deduction

This is a simple web app where the user is invited to complete a natural deduction proof, including implications and conjunctions. It is freely accessible online [5].

This website is not for the kinds of expressions that ClickDeduce is intended to be used for, but its use of a tree-based interface for inference rules is relevant.

The project has remained in an experimental stage, with its last update occurring over six

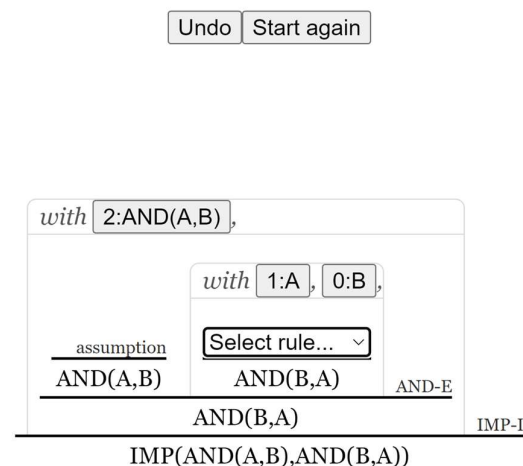


Figure 4: Interface of Interactive Natural Deduction

years ago. The project's scope is narrow, focusing on demonstrating the concept through a singular, illustrative example.

The actions a user can take with its interface are as follows:

- Where the focused branch is not completed, the user can select a rule using a drop-down menu which lists all the available rules.
- Depending on the rules selected, an available assumption may appear at the top. Clicking on that assumption will use it in the focused branch.
- If there is another incomplete branch that is not focused, then a button appears which can be clicked to focus on that branch.
- The user can return to an earlier point in the tree by clicking on the conclusion of a rule. This deletes the section of the tree above that rule.
- There is an “Undo” button which reverts the most recent action.
- There is a “Start again” button that resets the tree to its initial state. This is equivalent to the user clicking on the bottom line but resets the option to undo.

The best part of this interface is how it allows the user to undo an action or reset a node by clicking on it. The node being hovered over is highlighted to make it clear that it is actionable.

I believe the interface can be improved to make it easier to understand the available actions:

- Without a pre-existing understanding of the rules, their names are unclear and there is no description provided for their intended usage.
- Automatically introduced variable names are potentially intimidating, with the first names being “?X10” and “?X9”.
- The way to use available assumptions should be clearer.
- Executing the same action multiple times in a row, such as repeatedly clicking on the same conclusion, results in the “Undo” action building up a meaningless queue of actions.
- Expressions are displayed using the names of operators rather than their symbols (for example, “IMP(AND(A, B), AND(B, A))” could instead be written as “ $(A \wedge B) \rightarrow (B \wedge A)$ ”).

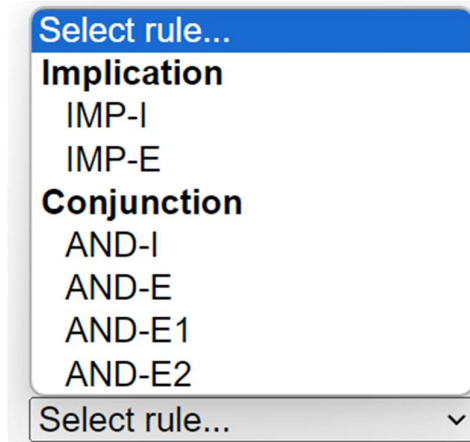


Figure 5: Dropdown menu for selecting a rule kind

### 3.3 Holbert

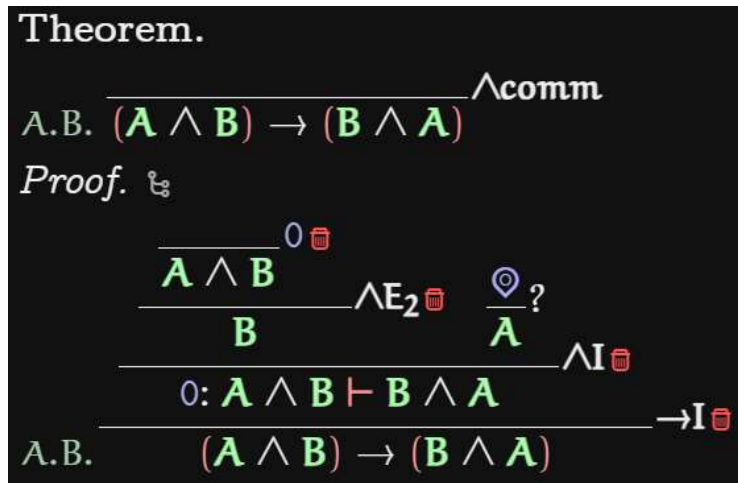


Figure 6: Example of theorem and proof interface in Holbert.

Holbert [6] is an in-development online textbook platform specialising in logic proofs for teaching. It is developed by Liam O'Connor and Rayhana Amjad from the University of Edinburgh. An interactive demonstration of the tool is available online.

This tool, similarly to other tools mentioned, is designed to visualise and create logic proofs using natural deduction rules. This tool could be used to create inference trees for programming language expressions, but this would require lots of predefined rules and the interface is generally not tailored towards this specific use case.

There are two main forms of entering information for the proofs. Rules and theorems are entered as text, using post-fix operator notation, for example,  $\_ \rightarrow \_ (\_ / \_ A B) (\_ / \_ B A)$  corresponds to  $(A \wedge B) \rightarrow (B \wedge A)$ . This may work well for experienced users, but it requires the user to know this syntax, which may not be intuitive for novice users.

The second form of input is progressing from a theorem towards a goal. A panel at the side of the screen displays the currently valid assumptions and available rules that can be used. The user can then click on them to use them to continue the tree. This is more suitable for novice users since it displays information in the same format as written rules and does not require additional tool-specific knowledge.

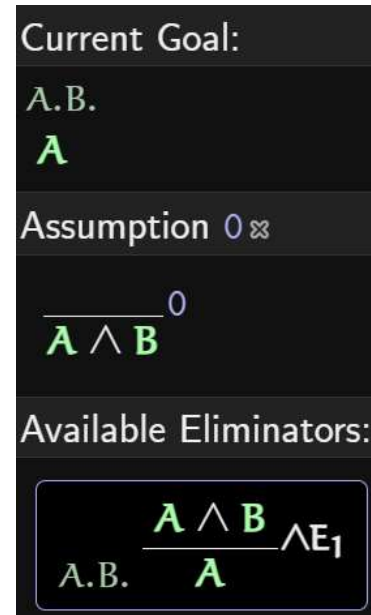


Figure 7: Assumption and eliminators sidebar element in Holbert.

### 3.4 Previous Project by Valentas Dicevicius

Valentas Dicevicius, a University of Edinburgh alumni, completed a similar project in 2017 [7]. While the report was available on request, neither the application that was developed nor the source code is currently accessible online.

From what is discussed in the report, it appears that this tool was a website that also followed the languages defined in the Elements of Programming Languages course. It began by prompting the user to enter a full expression as text, which was then parsed by the program. That entered expression would be the root node, and then the user could click a button to

advance the construction of the tree, adding a child node to one of the nodes that does not yet have all of its required children. Eventually, the entire evaluation tree would be displayed.

Additionally, it had a “challenge mode”, where the user could specify the chance that instead of the expression for a node being provided automatically, the user instead has to enter it correctly. This catered towards its purpose of reinforcing students’ ability to complete these derivation trees.

## 4 Design

### 4.1 Target Audience

This project primarily aims to be useful to the future students of the Elements of Programming Course. It also aims to be useful in a broader context for teaching about expression trees and for more experienced users to be able to construct expression trees and use them elsewhere.

#### 4.1.1 Students

Students are the primary target audience for this tool. They are aiming to use this tool to reinforce their understanding of how the expressions presented in lectures and tutorials work. They may also be used as part of a tutorial or for practising derivations for an exam.

Students will be much more likely to use a free tool that is easy to begin using, rather than one that costs money or time to try. To cater towards students who are only just becoming familiar with this topic, the interface needs to be simple and intuitive.

Additionally, it would make the tool much more useful if it matched what the students were being taught within their course.

#### 4.1.2 Teaching Staff

As part of demonstrations or for preparing resources for a course, a lecturer or tutor may want to use this tool.

It will be much easier for them to use the tool if it closely corresponds to the content of the lectures. They may also need the option to export to other formats, for example, LaTeX, so that trees can be displayed in PDFs such as tutorial sheets or slideshows.

#### 4.1.3 Academics

While not the primary focus of this project, there may be some academics who want to use a tool like this for creating visualisations of complex and possibly novel programming language features.

Their requirements are similar to those of teaching staff and there is likely to be overlap between these user groups. They will need to be able to export trees for use in their papers.

<b>Feature</b>	<b>Students</b>	<b>Teachers</b>	<b>Academics</b>
<i>Simple interface</i>	✓		
<i>Free and easy setup</i>	✓		
<i>Accuracy and correctness</i>	✓	✓	✓
<i>Matching lecture content</i>	✓	✓	
<i>Export to alternative formats</i>		✓	✓
<i>Adding new language features</i>			✓

Figure 8: Summary of important features per target user group

### 4.2 Goals

Considering the needs of the different target audiences, the following goals are defined:

- Convenience: Students and other users should be able to quickly and easily begin using the tool. A long or complex setup process needs to be avoided.
- Simplicity: Students who have only recently been introduced to the concept should be able to use the tool, despite their relative unfamiliarity with the concept.
- Availability: The tool should be open and free for anyone to use.
- Correctness: The users must be able to trust that any outputs are reliable and accurate. With more complex language features, evaluations can become very complex, and the last thing a student needs is incorrect outputs.
- Extensibility: More advanced users should be able to add new languages and expressions to the tool. The design should strive to be as modifiable as possible.

### 4.3 Primary Development Language

The Elements of Programming Languages course uses Scala throughout its tutorials and coursework assignments. To best fit with the course, Scala was selected for this application. This means that more advanced students who want to experiment with implementing their own features would be able to use their experience from earlier work in the course.

Scala is well suited for this problem. Evaluating and type-checking expression trees does not have a global scope for variables and does not involve any side effects, so the functional programming features in Scala are well suited. The object-oriented features make it easy to define expressions, types, and values as new classes which the rest of the code can use according to the appropriate interfaces.

### 4.4 Platforms

It is essential to decide which devices should be supported early on in development. Each platform supported significantly increases the development time and testing requirements.

While there are platform-agnostic development tools, how an interface will be used varies significantly based on the usage of a mouse and keyboard versus a touchscreen.

A poll from 2021 [8] found that 94% of students use either a laptop or desktop as their primary device. Therefore, it makes sense to target this as the main platform.

Considering this, it makes most sense to develop the tool for laptops/desktops. This would either take the form of an application that runs on Windows/Mac/Linux or a website.

A website is the easiest way to support many platforms since it can be viewed on any device with an internet browser. The downside is that different browsers may render the page slightly differently and that different devices have very different aspect ratios.

Derivation trees can become very large, which makes them poorly suited to small screens such as mobile devices. Furthermore, in almost all cases they are much wider than they are tall, so a landscape screen orientation is more suitable than a portrait view.

### 4.5 Architecture

Multiple different approaches to the tool were considered. They ranged from a local application with custom UI implementation to a server which hosts a website with the interface and responds to requests, to a simpler webpage which does not communicate with a server after loading.

### 4.5.1 Local Application

This would consist of a Scala program that the end user would run on their computer. Since Scala can use Java libraries, the popular Java Swing GUI library could be used.

The advantages of this approach include having better control over the tool and not needing to worry about server infrastructure, security, or costs.

The simpler mechanism would be to distribute an executable application that the user runs. This could encounter issues with different operating systems and become outdated (like how MacLogic is only available on decades-old Macintosh operating systems). This would also make it difficult to allow the user to add their own languages and modifications.

The more extensible and futureproof strategy would be to host the source code online and instruct the user to clone it and use local build tools to compile and run it. This should work on a wider range of operating systems and not be likely to become outdated. However, it would be a massive barrier to entry for most users.

### 4.5.2 Website & Server

This approach involves a separate front-end and back-end, with a shared API.

The backend would be a server application written in Scala. It would be responsible for parsing trees and the requested action, evaluating or type-checking them as requested, and then replying with the result in a form that the website can display.

The frontend would be a webpage set up to make requests for actions based on the user's inputs. It would wait for the server's response and then display the new tree.

In this design, all the information that the server needs is provided by each request from the webpage, so nothing would need to be stored. This would make it a stateless server.

The issues with this approach are simply the issues with running a server. It would have upkeep costs, require regular attention, be potentially vulnerable to malicious attacks, and have the potential to be slow or overloaded.

### 4.5.3 Serverless Website

Since the Website & Server design was stateless, it is possible to eliminate the need for the server. However, the challenge with this is that Scala compiles to Java bytecode which is meant to be executed by a JVM Java runtime, not a web browser.

Conveniently, there is a perfect solution to this problem. ScalaJS is a mature Scala Build Tool (SBT) plugin which compiles the program to JavaScript instead of Java bytecode. This would mean that instead of having a server written in Scala, the Scala code would be compiled into JavaScript, and then that code would be included in the webpage.

The webpage could then be hosted on GitHub's free Pages service.

The main limitations with this approach are that ScalaJS has some restrictions on the Scala code it can use and that testing the Scala code becomes much more difficult, as ScalaJS prevents compilation to Java bytecode, which would be needed to test the Scala code directly.



Every library used in the Scala code needs an up-to-date ScalaJS version of the library. Also, Java's useful Reflection library, which is used to examine properties of objects at runtime, is not supported by ScalaJS.

#### 4.5.4 Chosen Approach

To begin with, the Website & Server approach was used. This was opted for since it made developing the systems for languages easier and allowed the use of Java's reflection library.

It was also selected so that the option of users being able to upload Scala files containing their languages was possible.

The server was developed to be self-contained so that it would be easy to switch to another approach later.

### 4.6 Purpose

There are multiple possible approaches for what the interface should achieve.

#### 4.6.1 Displaying Derivation

The simplest approach would be to have the user input an expression and then display the resulting evaluation tree. This is simple but does not facilitate learning through the use of the tool.

#### 4.6.2 Displaying Derivation with Challenges

This is similar to the approach taken by the previous project by Valentus Dicevicius and makes the displaying derivation approach more interactive. At each step of the derivation, the user is prompted to enter what the expression at that point should be which is then compared to the actual expression.

This improves on the basic approach by challenging the user and engaging them. Users can tell whether they have understood each rule.

#### 4.6.3 Constructing Derivation

This approach is fundamentally different from the aforementioned implementations. Instead of starting from a complete expression and building up the corresponding evaluation or type-checking tree, this instead invites the user to build the expression by constructing the tree.

### 4.7 Interface Functions

#### 4.7.1 Tree View

The most important part of the interface is that it matches the structure that students need to become familiar with. This means that the interface needs to represent the derivation using a tree.

The classical text editor view used by other tools does not meet the needs of this tool.

#### 4.7.2 Entering Expressions

There are various approaches to how the user could input their expressions and complete the derivation tree. Some are easier for new users, while others are more powerful for advanced users. In general, while a balance would be preferable, the priority is to accommodate students.

### Plain Text Input

if (53 == 87) then (true + 5) else ((6 + 1) \* 2)

Figure 9: Mock-up for text input with example expression

This would consist of asking the user to enter the expression text, following the syntax of the language.

This is similar to how the code for most programming languages is edited.

#### Advantages

- Proficient users can quickly and efficiently input the intended expression.
- Can be shared between users by copying and pasting the expression text.

#### Disadvantages

- It requires users to be proficient with the syntax for each language, which is not beginner-friendly. This conflicts with the goal of accessibility.
- It does not suit the tree structure well, since the entire expression would need to be entered at once at the root. There is no simple way to use a bottom-up approach to make it so that the user only needs to enter part of an expression.
- It would also require a custom parser and meaningful error messages, which would make adding new languages much more difficult. This conflicts with the goal of extensibility.
- Some language features such as lambda functions or poly-types typically use symbols not found on the keyboard.

### Tree With Dropdowns

This would consist of the tree interface having a dropdown element for each incomplete node, as in Bob Atkey's interface. For an expression, the dropdown would list each of the expressions in the language, and for a type, the available types would be displayed.

#### Advantages

- Available options are clearly visible.
- Syntax errors due to typos are not possible.
- Much easier for new users to approach.

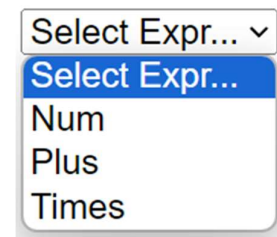


Figure 10: Expression selector with dropdown menu

#### Disadvantages

- With larger languages, there may be far too many expressions to show in a single dropdown menu, meaning the list will not fit on the screen.
- The dropdown menus necessitate the use of the mouse, while the literal inputs require keyboard inputs. This means the user would need to constantly switch between typing on their keyboard with both hands and moving their mouse with one hand.
- When first using a language, the names of new expressions presented in the dropdown may be confusing.

Tree With Drag & Drop

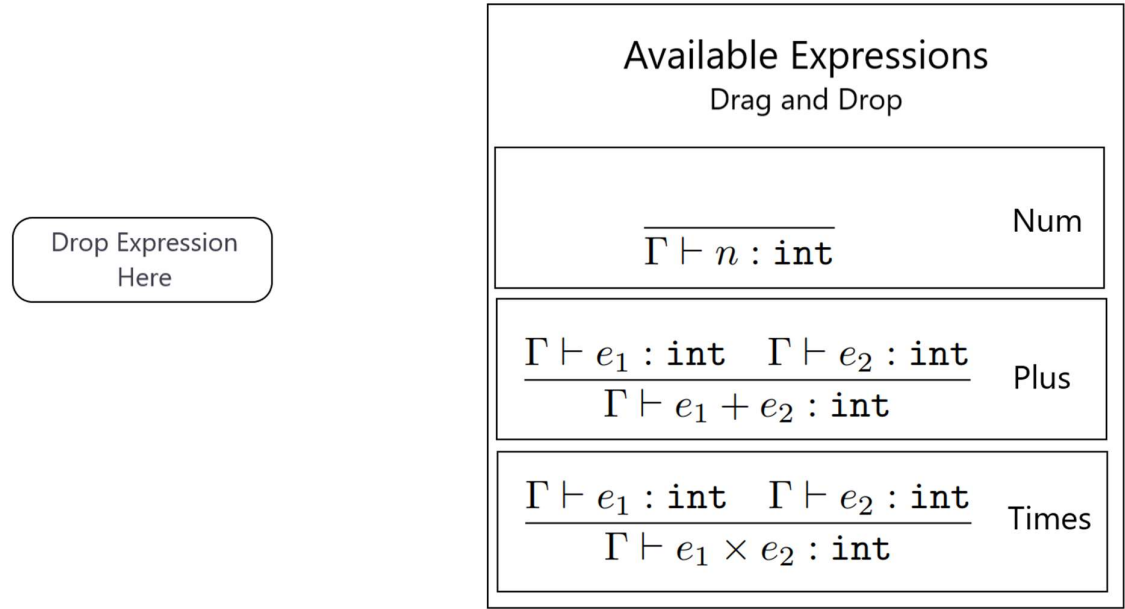


Figure 11: Mock-up of drag-and-drop menu

This would be similar to the tree with dropdowns, but instead of the dropdown elements, there would be a list of available expressions next to the tree interface. The user could then drag and drop the expressions into empty spaces.

Advantages

- A permanent UI element for displaying expressions can implement filtering to make finding a particular expression in a large language possible.
- More space to display expressions.

Disadvantages

- Similarly to the tree with dropdowns approach, this requires the user to frequently switch between typing and using the mouse, making using the interface tedious and possibly quite slow.
- Occupies a lot of screen space reducing the space available for the tree interface.
- Requires more effort to implement. More scope for issues due to being more complex and specialised than other options.

Tree With Custom Selectors

This would be similar to the tree with dropdowns, but instead of a dropdown menu, there would be a more advanced interface element.

It would have a text box where the user could enter the expression name. While they are typing, a dropdown menu displays the expressions which could match that for which they are searching. Once they have found the right expression, they press Enter and it is selected.



Figure 12: Search input with results in dropdown

### Advantages

- All available expressions are presented to the user and the user only needs to consider one subexpression at a time.
- In languages with a large number of expressions available, the searching functionality makes it easier to find the desired expression.
- With a custom interface element, it is possible to change the focusing functionality, so that the TAB key switches focus between elements.
- This would remove the need for the mouse while editing the tree since the keyboard can be used for selecting expressions, editing literals, and switching focus between branches. This would make using the interface more comfortable.

### Disadvantages

- The user needs to know the names of available expressions in a language, otherwise they cannot search for their name.
- For users unfamiliar with using TAB to switch element focus, they are likely to still end up using the mouse.
- More development time and effort are required to implement. Potential for being unintuitive if it doesn't resemble familiar interface elements.

### Conclusion

Overall, I believe the plain text input is not suitable for newer users, so I do not consider it a viable candidate. A tree interface with one of the described methods for selecting expressions is much more sensible. The tree with dropdowns approach is simpler and quicker to implement than the other two options but is unsuitable once languages with lots of expressions are introduced. Since it can eliminate mouse usage, I believe the custom selectors are the best input method.

Therefore, the chosen approach was to first implement the tree with dropdowns, so that an early prototype could be quickly developed. Later, once larger languages were implemented, custom selectors replaced the simple dropdowns.

## 4.8 Languages

There are multiple languages presented in the Elements of Programming Languages course. They generally follow a linear hierarchy, with each new language building on top of the previous one.

A student at the beginning of the course would likely be overwhelmed by seeing every kind of expression when first using the tool, so it makes sense to allow the user to pick which language to use.

Furthermore, in the case of users developing their own language features, it will be much easier for them to develop and share them if they are self-contained.

There needs to be an interface element which presents a list of all the languages which the user can choose from.

Each language needs to define its expressions and type-checking/evaluation rules and any associated types or values. The language can then simply be added to the list of available languages in the tool and then used.

## 4.9 Type Inference

A common feature in many popular programming languages is the ability for the types of certain variables or functions to be inferred by their usage. This can make code slightly more concise. A technique such as the Hindley-Milne type inference algorithm can be employed.

Two of the key features of ClickDeduce are the ability to view the type-checking result at any point and to be able to easily implement new expressions (including their type-checking rule). Needing to be compatible with a type inference algorithm would conflict with both of these goals. It would add an additional layer of complexity while the tool strives to be as simple and accessible as possible. This complexity may also result in unpredictable or inconsistent results, requiring additional work to debug.

Instead, with the expressions in the “Elements of Programming Languages” course, it is possible to enforce explicit type declarations to avoid the need for type inference. This is relevant for the following expressions:

- Lambda functions: The type of the input parameter needs to be included.
- Recursive functions: The type of both the input parameter and the function output are required.
- Union expressions: For both “left” and “right” expressions, the type of the other half of the union needs to be stated.

## 5 Implementation

### 5.1 Overview

The complete version of ClickDeduce [1] consists of two web pages hosted on GitHub's Pages service [9]. There is the main page for the tool and a separate guide page.

All of the language features are implemented in Scala, alongside the internal node tree structure, the ability to perform actions on a tree, and the conversion from the tree to HTML or LaTeX. ScalaJS is used to convert the Scala code into JavaScript code for web pages.

Both of the web pages are written in HTML. The scripts for the web page (primarily focused on interface functionality) are written in TypeScript. The styling for the site is written using Sass [10]. The HTML, TypeScript, and Sass contents are bundled using Webpack [11].

### 5.2 Language Features

There are four distinct kinds of terms which are used for languages in ClickDeduce. Each of them implements a common “Term” trait that allows them to be used elsewhere in the program. This trait also allows some functionality – particularly related to being displayed – to be overridden by individual terms.

All expressions, types, values, and literals in ClickDeduce are implemented as case classes – a special kind of class in Scala which have inherent support for pattern-matching.

#### 5.2.1 Expressions

Expressions only need four main attributes to be defined:

1. Parameters – (any combination of subexpressions, literals, and types)
2. Evaluation rule
3. Type-checking rule
4. Text conversion – (how the expression should be displayed inline)

These closely match what an expression is defined as when using abstract inference rules, meaning that implementing expressions does not require much beyond what they are supposed to mimic.

The parameters for expressions are limited to only other terms (except values). This is because the expression's parameters are used extensively in the structure of the tree and the tree structure cannot recognise other types of parameters.

#### 5.2.2 Types

Types are generally the simplest terms to define since they are typically static and don't normally have any special behaviours. Some types, such as functions and pair types, have types as parameters, but a type cannot have an expression as a parameter.

Types can appear in expressions, but typically only play a role in type-checking, not evaluation. They are used to avoid the need for type inference, such as lambda functions requiring the type of the variable to be declared.

The most complex types, of those present in the languages in the EPL course, are the polymorphic types and type variables. Ignoring these, every type is the same regardless of the

current environment. For type-checking, a type variable needs to look at the current environment to check what type it is.

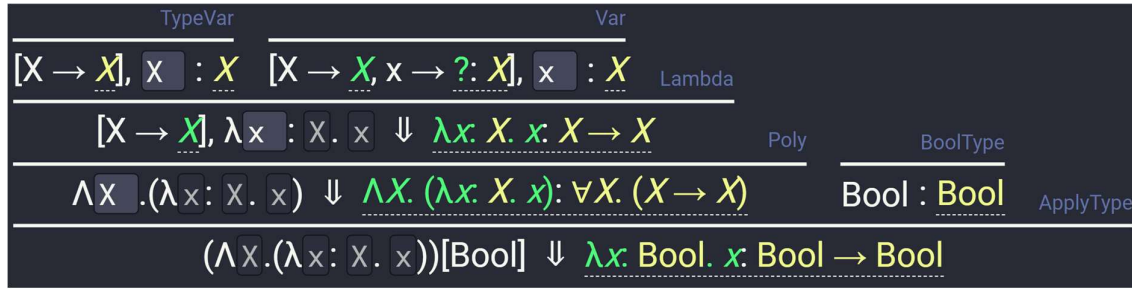


Figure 13: Evaluation of Boolean type being applied polymorphic identity function

In this expression, the identity function is defined ( $\lambda x : X. x$ ), with the parameter type being the polymorphic type variable  $X$  ( $\Lambda X. (\lambda x : X. x)$ ). The Boolean type is then applied to the polymorphic type, binding the value of the type variable  $X$  to the Boolean type ( $(\Lambda X. (\lambda x : X. x))[\text{Bool}]$ ), which evaluates to the identity function for Booleans ( $\lambda x : \text{Bool}. x$ ).

Like the parameters for expressions, the parameters for types are limited to only terms. Additionally, a type cannot have an expression as a parameter (a type cannot be evaluated) – only types and literals are allowed as parameters.

### 5.2.3 Values

Similarly to types, values are easy to implement when creating a new language. They need three attributes:

- Parameters – (any data type, can include other terms)
- Corresponding type (e.g. Boolean value corresponds to Boolean type)
- Text conversion

All values need a corresponding type to be displayed wherever they appear as an evaluation result.

The parameters for values contrast with the parameters for expressions and types in how they are not limited to only terms. This is because values are not part of the tree structure and only appear as evaluation results, so the rest of the program does not need to be able to handle their parameters.

### 5.2.4 Literals

Literals are used extensively by other terms but are themselves simple. The only pieces of functionality they require are the ability to be parsed from a user-entered string and how to be displayed. To be useful, they also need some way to be used by other expressions, typically by storing something from what they parse (e.g. a number or a variable name).

The following kinds of literals are defined by default in ClickDeduce:

- LiteralInt: Stores an integer (of arbitrary size)
- LiteralBool: Stores a Boolean
- LiteralString: Stores a string

- **LiteralIdentifier:** Stores a valid identifier for a variable or function name (only allows alphanumeric characters, underscore, and dollar sign, and cannot be empty or start with a number)
- **LiteralAny:** Simply stores the literal input provided, used when the input does not match any other kind of literal

### 5.3 Actions API

The interface between the frontend and backend had the potential to have a large number of very similar connections. For example, selecting options from expression and type dropdowns would need two different endpoints, despite the parameters being nearly identical.

This is not necessarily problematic, but I opted to create a common “action” endpoint for all the ways that a user can edit an expression tree.

The action endpoint has the following parameters:

1. Language Name (e.g. “LArith” or “LData”)
2. Mode Name (e.g. “edit”, “type-check”, or “eval”)
3. Action Name (e.g. “SelectTypeAction” or “DeleteAction”)
4. Node String (the format for storing an expression tree)
5. Tree Path (indices through a tree to the node which is being acted on)
6. Extra Arguments (any additional arguments that the action requires, such as the select expression action needing the name of the expression to use)

This is all the information that is needed to make any changes to any expression tree. With this, the selected language can perform the change, and then return the updated node string and HTML representation of the tree.

The possible actions are:

- **Select Expression:** Replace a blank expression selector dropdown with the chosen expression, each of its subexpressions/types/literals is initially empty. Additional argument: expression name.
- **Select Type:** The same as “Select Expression” but for a type (such as for the argument type of a Lambda function). Additional argument: type name.
- **Edit Literal:** Update the value of a literal. Additional argument: literal value
- **Delete:** Replace an expression or type with a blank dropdown (as appropriate), also removes any subexpressions/subtypes.
- **Paste:** Replace an expression with an expression tree, or a type with a type tree. Additional argument: node string to use as the replacement.
- **Identity:** No change to tree structure. Included so that the user can change the language or view mode without needing to modify the tree.

### 5.4 View Modes

Each expression has rules for type-checking and evaluation, which are used when determining the type or value of an expression. ClickDeduce presents 3 options of view modes to the user. The first 2 are the evaluation and type-checking modes, matching the two kinds of derivation trees that commonly appear.



There is a third mode called the “edit” mode, which the interface defaults to. This becomes relevant when lambda functions are used.

The main purpose of the “edit” mode is to allow the user to modify the tree with a consistent structure since some expressions can have evaluation and type-checking rules that have varying premises (for example the evaluation rule for the apply expression has a third tree if the left expression evaluates to a lambda function).

In “edit” mode, the evaluation result is displayed where possible, otherwise the type-checking result is displayed instead. This is performed per node of the tree, so there can be a mix of evaluation and type-checking results in the tree.

Attempting to evaluate inside a lambda function itself may not be possible, since the variable is not bound to a value – only a type. As shown in Figure 14, the Var expression inside the Plus, inside the Lambda, cannot be evaluated because the value of  $x$  is unknown, which also means that  $x + 1$  cannot be evaluated. In these cases, the type-checking result is displayed instead.

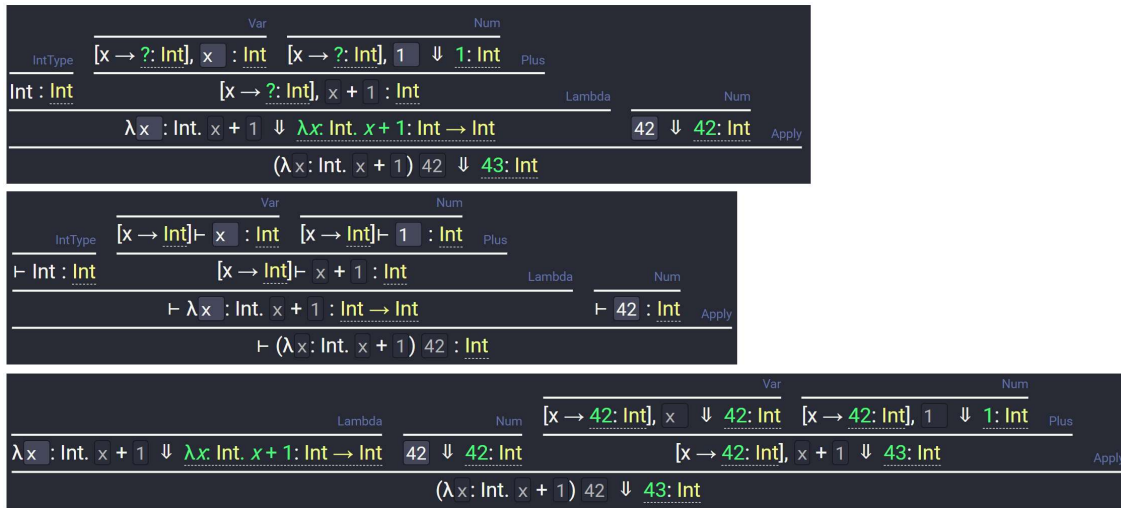


Figure 14: Comparison of available view modes with the same expression. From top to bottom: edit mode, type-checking mode, evaluation mode.

By default, expressions do not have any special behaviour for different view modes. The subexpressions (and subtypes, if relevant) are used as the children nodes, and the evaluation or type-checking environment is unchanged. When implementing expressions, these behaviours can be overridden per view mode to accommodate a wide range of possible language features.

## 5.5 Tree Structure

Expressions inherently have a tree structure where each node is an expression or type. Some nodes are literals, but they are always leaves (though not all leaves are literals).

While it would be possible to only use expressions for the internal tree representation, it limits the scope for additional features. It would be impossible to have a tree which does not perfectly match the expected expression structure.

It would also make adding new expressions more difficult since they may need to consider conversion to and from different formats.

Considering this, an additional tree structure was introduced. It uses “inner” and “outer” node interfaces. Each expression or type is an “outer” node. Each outer node has any number of inner nodes as arguments. An inner node optionally has an outer node as an argument. The origin of the tree is always an outer node. This results in a tree structure which alternates between outer and inner nodes.

Here is the context-free grammar for the *LArith* language node tree structure:

```

treeroot := nodeexpr
nodeexpr := kindexpr argsexpr
nodetype := kindtype argstype
nodeliteral := string
nodesubexpr := nodeexpr
nodesubtype := nodetype
argsexpr := nil | argexpr : argsexpr
argstype := nil | argtype : argstype
argexpr := nodeliteral | nodesubexpr | nodesubtype
argtype := nodeliteral | nodesubtype
kindexpr := "Num" | "Plus" | "Times"
kindtype := "Int"

```

Extending this in other languages only requires adding more options for kind<sub>expr</sub> and kind<sub>type</sub>.

### 5.5.1 Tree Paths

It is important to be able to identify each node within a tree, particularly when the user makes an edit to a specific node. Tree paths are used as part of the Actions API.

This is accomplished using a tree path, using the following algorithm:

1. Begin at the selected node, initial tree path is empty.
2. If the node has no parent, return the current tree path.
3. Otherwise, find the index of this node in the parent's arguments.
4. Prepend that index to the tree path.
5. Go back to step 2, updating the current node to be the parent.

## 5.6 Phantom Trees

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad e[v_2/x] \Downarrow v}{e_1 e_2 \Downarrow v}$$

Figure 15: Type-checking and evaluation rules for applying to a lambda function

Compare the type-checking and evaluation rules for applying a value to a lambda function. Notice that they have different numbers of subtrees, with the evaluation rule having an additional subtree, where the value ( $v_2$ ) of the bound variable ( $x$ ) is substituted into the expression ( $e_1$ ).

The internal tree structure is designed to map one-to-one to the visible output, while also being agnostic to the output mode. The evaluation and type-checking modes having different tree structures contradict this design, requiring an alternative solution which accommodates this while also being generic and extensible.

To resolve this, a system for “phantom” tree nodes was implemented. These are not part of the internal tree, but appear when rendered.

As part of the expression class, they can override which subexpressions (and variable scopes) should be displayed. Subexpressions which do not have a corresponding child node as part of the original tree are marked as “phantoms”.

Phantom trees appear to the user like any other tree, but since it does not correspond to the internal tree, allowing the user to edit it does not make sense. All inputs in phantom subtrees need to be disabled.

## 5.7 Command History

One essential feature in most popular editors is the ability to undo recent changes. This was added to ClickDeduce in a simple manner, the HTML and node strings resulting from each change are stored in memory and retrieved when the user undoes an action.

It would be more efficient to store the difference between the state for each action, reducing memory usage. This wasn’t implemented because constructing and editing a tree takes relatively few actions (typically less than 50), and each string is at most a few kilobytes. Implementing more memory-efficient algorithms has a greater scope for logic errors.

## 5.8 Transition from Server-Based to Serverless

Development began with the website and server approach mentioned in the design chapter. This made configuring the project easy since it was the same as a typical Scala project with the addition of another directory containing the HTML and TypeScript files.

Tests were written separately for the Scala and TypeScript portions of the code. This made testing the languages portion of the codebase easy since it could be tested in its native environment.

Later in development, after the project was stable, a transition was made to the serverless website approach. The web server portion of the Scala code was replaced with an entry point for the ScalaJS linking.

ScalaJS changes the behaviour of the project it is enabled in to no longer compile to run on the JVM, instead creating JavaScript code. This means that the Scala tests can no longer be run in that project. The solution to this was to create multiple sub-projects:

1. **Shared:** Contains the source code for the main project.
2. **ScalaJS:** Only contains the ScalaJS entry point. Extends the shared project.
3. **Test:** Contains the tests for the source code. Extends the shared project.

## 5.9 Stack Overflow

$$\frac{e_1 \Downarrow \text{rec } f(x). e \quad e_2 \Downarrow v_2 \quad e[\text{rec } f(x). e / f, v_2 / x] \Downarrow v}{e_1 e_2 \Downarrow v}$$

Figure 16: Evaluation rule for applying a value to a recursive function

The new expression added by the *LRec* language implements recursive functions. The recursion is arbitrary and it is easy to write an expression which will never terminate, such as  $(\text{rec } f(x). f(x)) 1$ .

Attempting to evaluate this will result in an infinitely deep evaluation tree of phantom nodes, invariably resulting in a runtime error.

It is not possible to arbitrarily determine whether an expression will have infinite recursion, as per the halting problem. Some other mechanism for either detecting the infinite recursion or handling the resulting runtime error is required.

Handling the runtime error is an option, but depending on the way stack overflow exceptions work during runtime this may result in slow responses.

The chosen option was to track the depth of each node and apply a maximum allowed node depth before throwing a custom exception.

## 5.10 Saving and Loading

Owing to the tree structure mentioned above, saving, and loading trees was easier to implement. The “node string” is simply the string representation of the origin node instance. Since trees use Scala’s case class, there is a standard string format (name of the class followed by the arguments, separated by commas, and surrounded by brackets). A custom parser was written that mostly follows the context-free grammar described in the tree structure section.

A file is saved using JSON format, with the following properties:

- **nodeString**: The node string for the origin node.
- **lang**: The name of the language used.
- **mode**: The name of the view mode.

To load the file, the user either selects the file using the file load dialogue or by dragging and dropping the file into the web page. The JSON inside the file is parsed, and then the paste action is used to overwrite the root node of the tree with the node string from the file. This efficiently reuses the existing action API.

If the file being loaded is invalid, then an error occurs.

## 5.11 Error Handling

There are some cases where user actions make an error unavoidable, primarily related to stack overflows and loading invalid files. It is also possible that other potential language features could cause errors.

When an error occurs, the error message is displayed in a red box in the bottom-right corner of the page. The message uses the print method of the error object caught, meaning that any part of the code can throw a meaningful and specific error message. The message is displayed for 5 seconds before disappearing.

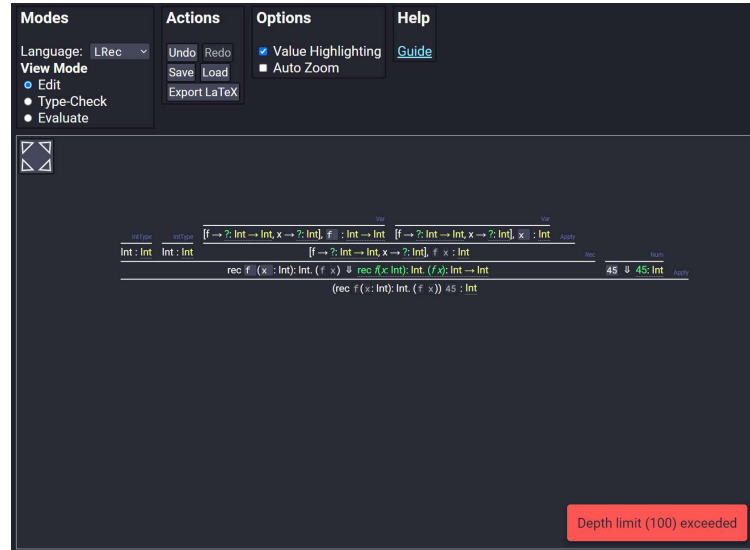


Figure 17: ClickDeduce interface, error message resulting from stack overflow is displayed in the red box in the bottom-right corner.

Since an error is caused by an action, the current tree state is assumed to be unusable. To resolve this, the tree state is reverted to the state from the previous action (using the command history functionality).

## 5.12 Converting to HTML

The internal tree structure is not important to the user, but rather what is displayed on the webpage. A converter interface is defined which converts a node tree into an export format, in this case, HTML.

This works in tandem with the styling rules for the webpage, allowing a simple HTML structure to be displayed in the desired bottom-up tree structure. Every tree action the user makes results in the displayed HTML being updated.

The output HTML has the following structure:

1. Each part of the tree is a “subtree” div.
2. Each “subtree” div has its “node” div and “args” div.
3. The “node” div contains the expression or type at that point in the tree, as well as the type-checking/evaluation result and the scoped variables, if present.
4. The “args” div contains a “subtree” div for each of the arguments that the internal node had. Additionally, it contains the name of the expression used.



Figure 18: Example HTML output

Most of the conversion to HTML is done in the Scala code, but the webpage itself makes a few changes to the HTML it receives:

- The simple HTML select inputs for expression selection are replaced by the more advanced inputs with search boxes.
- Event listeners for hovering over parts of the tree are added.
- Phantom inputs are disabled.
- The initial values for each of the literal inputs are stored.

## 5.13 Converting to LaTeX

One useful feature for teaching staff and academics is the ability to export trees they have created to LaTeX since it is popular for creating documents and slideshows. Since it is not important for students, it is not a feature that should be prominently focused or be difficult to implement.

An “Export LaTeX” button was added to the interface which opens a window displaying the tree in LaTeX text form. The user can copy and paste it into an existing LaTeX document and use the `bussproofs.sty` package [12] to render it.

This package was selected because it has a simple syntax and displays the tree in the expected manner with a bottom-up tree structure.

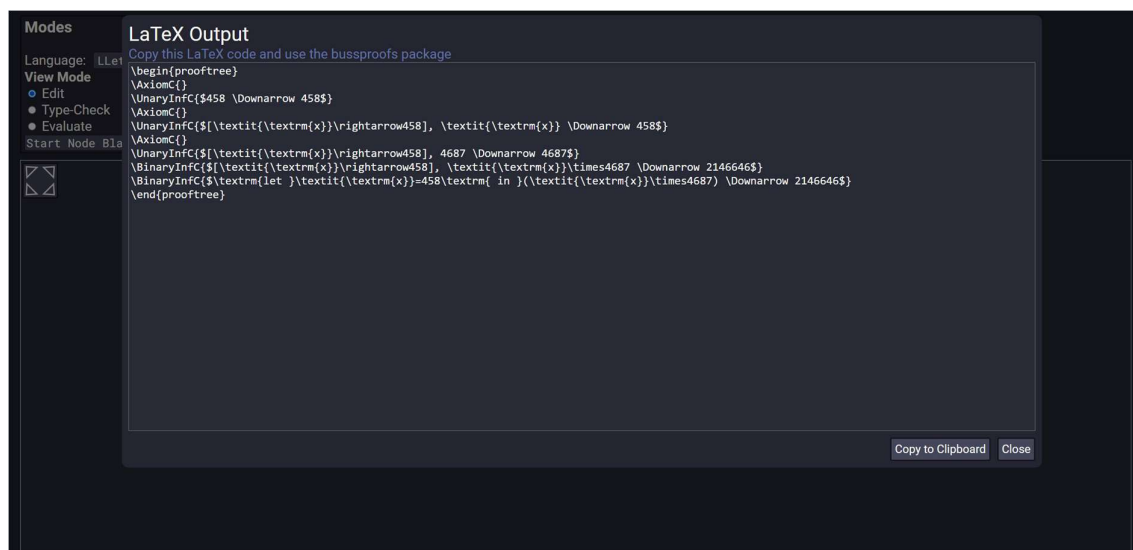


Figure 19: LaTeX output dialogue element.

Examples of how trees are rendered can be viewed in Appendix 1: LaTeX Output Examples.

Due to the similarities between the conversion of the tree structure to HTML and LaTeX, it became apparent that a standard interface for converting these was possible in case another export format was needed in the future.

## 5.14 Interface

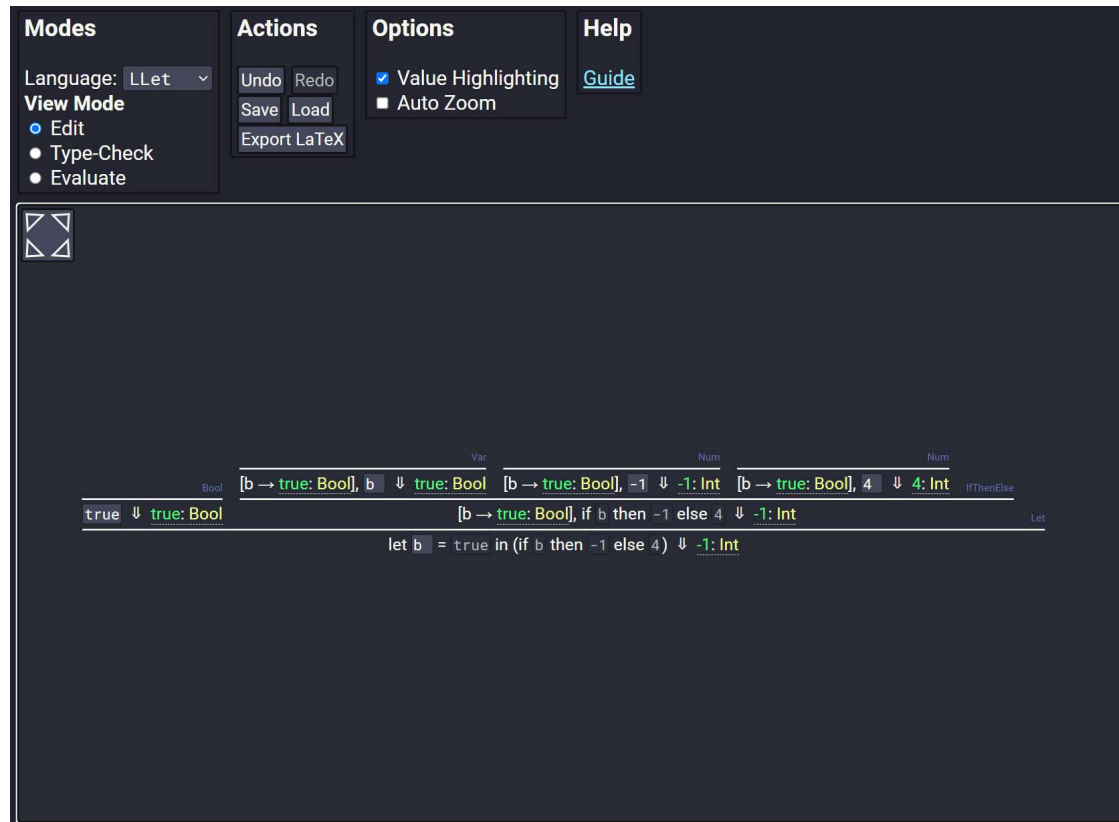


Figure 20: The interface of the main page of ClickDeduce

### 5.14.1 Controls

There are multiple groups of controls presented to the user.

#### Modes

This section includes the controls for selecting the language and view mode. The language selector is a dropdown menu which lists all of the languages that are implemented. The view mode is a trio of radio buttons which switch the current view mode between the edit, type-check, and evaluation modes.

#### Actions

This group includes the buttons for undoing and redoing actions, saving, and loading files, or exporting the current tree to LaTeX. The undo and redo buttons are only enabled when it is possible to undo/redo an action.

#### Options

This group is a list of checkboxes for toggling some features that different users might prefer to be enabled or disabled. The value highlighting option causes the interface to use distinct

colours for values (green) and types (yellow), to help make it clear which is which. The auto zoom option causes the tree to zoom to fit every time an action is performed, to keep the tree fully visible even if it is bigger after a change.

### Help

This section simply contains a link to the guide page, as described in the Guide section.

## 5.14.2 Tree Canvas

The tree canvas occupies the rest of the screen space. Initially, the tree appears with only a single root node asking the user to select the first expression.

The user can click and drag to pan around the tree and scroll to zoom in and out. This is implemented using the PanZoom library [13], as this is a common functionality for a website and a custom implementation would likely result in more issues and significant development costs.

Finally, there is a zoom-to-fit button that always appears in the top left corner of the canvas.

Hovering over any subtree will highlight that subtree, including its children. This makes it immediately clear which children a node has. The user can then right-click on the highlighted tree to open the context menu.

## 5.14.3 Context Menu

After the user right-clicks on a highlighted subtree, the context menu appears, focusing on that node. As long as the context menu is open, that subtree remains highlighted, even if the user hovers over other subtrees. Clicking on any other part of the interface closes the context menu and defocuses that subtree.

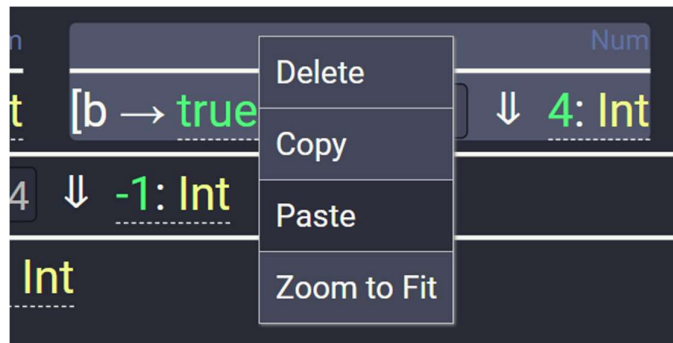


Figure 21: Appearance of context menu. The paste option is disabled because nothing has been copied yet.

The following options are available in the context menu:

- **Delete:** Clear the selected subtree, replacing it with a blank expression selector.
- **Copy:** Copy the selected subtree, so that it can be pasted later. This stores the node string in memory.
- **Paste:** Only enabled if a subtree has been copied already. Replace the selected subtree with the previously copied subtree.
- **Zoom to Fit:** Zooms the entire tree to fit into the tree canvas. Same functionality as the zoom-to-fit button in the top-left corner of the tree canvas.

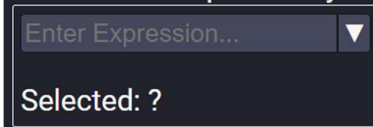
## 5.15 Guide

As part of making the interface beginner-friendly, there is a concise guide offered, with the link visible in the Help section. It contains a brief introduction to what kinds of terms there are and how to use the interface. It also includes example interactive elements for expression selection and literals to help introduce them in an isolated manner.



### Selecting Expressions

To select an expression, click on the "Enter Expression..." text box and type in the name of the expression you want to select, then press Enter.



Enter Expression... ▼

Selected: ?

Figure 22: Guide section with example expression selector.

### Editing Literals

Some expressions have *literal* fields that can be edited. For example in the Num, Bool, and Var expressions.

To edit a literal, select the text box and type in the new value, then press Enter or TAB to confirm the change and move to the next field.



Num

↓ !

Figure 23: Guide section with example Num node, where the user can edit the literal input.

## 5.16 Expression Selection

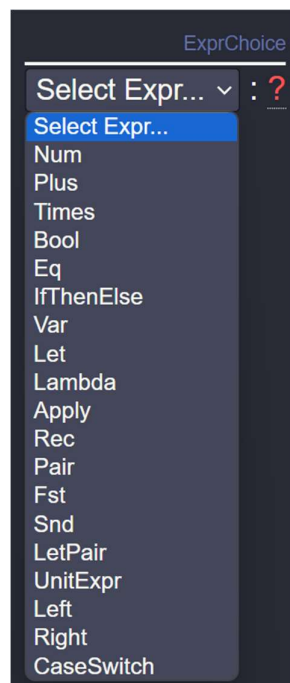


Figure 24: Dropdown selector with all options from LData language visible.

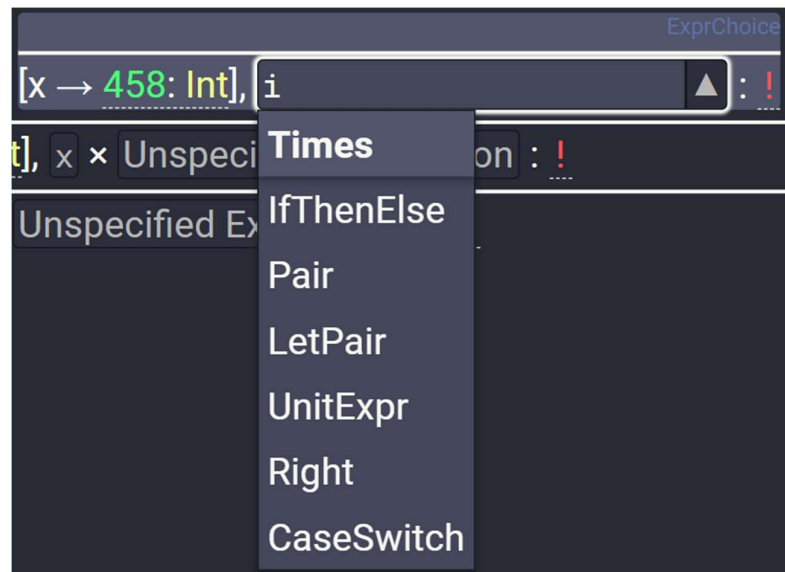


Figure 25: Custom expression selector interface, with all expressions containing "i" from LData

As discussed in Entering Expressions, the initial approach was Tree With Dropdowns. This worked up until the LData language was added, at which point the dropdown menu had far too many options, often spilling off the bottom of the screen.

After implementing LData, the Tree With Custom Selectors replaced the simpler dropdowns. Since this requires a lot of event listeners to be added by the webpage, it made more sense for the webpage itself to replace the old selectors in the HTML with the new selectors. There is no change in functionality from the Scala code's perspective.

## 5.17 Nested Expression Elements

The user enters parts of expressions one at a time. What appears in the subexpressions will (in most cases) appear in the parent expression. Literal inputs and expression selectors have distinct appearances, beyond just plain text. They instead appear as text boxes and custom expression selectors as shown in the previous section.



Figure 26: Demonstration of nested expression elements, selector in the right subexpression is highlighted because the user is hovering over the "Unspecified Expression" element

Read-only copies of subexpressions appear in the parent expressions. For literal inputs, this simply involved disabling any inputs and changing the colour scheme to appear muted (though the contrast with the subtree highlight is stronger).

Originally, the read-only version of expression selectors used the same approach, changing it to be darker and disabled, but this was confusing for users as they still had text prompting the user to enter an expression. These were replaced with elements similar in appearance to the literal inputs, stating "Unspecified Expression". Additionally, if the user hovers over one of these then the active selector above it is highlighted, as shown in Figure 26, to draw their attention to it.

## 5.18 Binding and Scope

Some expressions such as let bindings or lambda functions bind values to identifiers during evaluation. There are two simple approaches to how this should be handled.

1. Substituting variables within the bound scope with the bound value.
2. Keeping track of the environment of bound variables at each point during evaluation.

While the Elements of Programming Languages course opted for the substitution method, the environment approach made more visual sense for the interface. It would be difficult to substitute variables for values while still letting the user edit the expression as normal.

At each expression where there is a non-empty environment, the current environment is displayed before the expression.

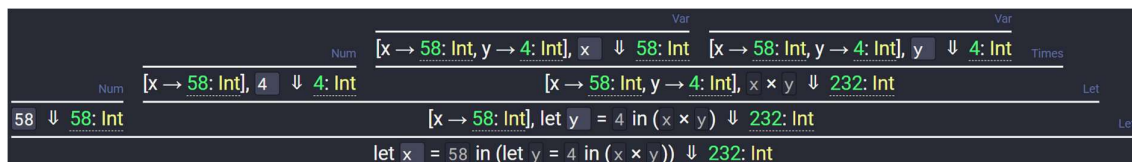


Figure 27: Expression using let-binding, environments appear in square brackets to the left of expressions

The only significant issue with this approach is that when the environment is large, or if there are values which include a lot of text (such as functions), it causes that subtree to become very large. There are some possible approaches to fix this, such as only displaying values on hover or only displaying newly bound values, but this was not a priority during development.



Figure 28: Part of the evaluation tree for a factorial function where the environment occupies a lot of horizontal space

## 5.19 Continuous Deployment and Integration

The repository for ClickDeduce is hosted on GitHub which provides support for automated workflows. The test suites for the Scala and TypeScript parts of the project are automatically run when there is a commit to the main branch.

If the test suites are successful, then the webpage is built and deployed using

GitHub's Pages service. Being able to use this free service means that the website should be much more accessible online for longer, not requiring a separate server to be maintained.

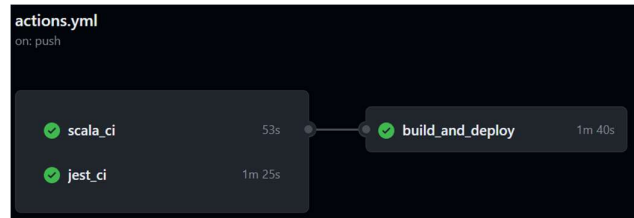


Figure 29: Structure of GitHub Actions workflow. Both Scala and TypeScript tests need to succeed before the page is deployed.

## 6 Evaluation

### 6.1 Automated Tests

While not useful for judging the quality and utility of the interface, automated tests are invaluable for ensuring the correctness of the language feature implementations. ClickDeduce employs an extensive suite of tests for each expression.

While it is not possible to test every expression, as there is a potentially infinite number of them, the test suites are reasonably comprehensive. They aim to ensure that each expression matches the behaviour defined in the original evaluation and type-checking inference rules.

### 6.2 Accessibility

An important aspect of interface design is accessibility. For text, font size and colour contrast are essential parts of this.

Web Content Accessibility Guidelines (WCAG) 2.1 [14] specifies that text needs to have a contrast ratio of at least 4.5:1 with the background or 3:1 for large text. The primary font and background colours in ClickDeduce have a contrast ratio of 13.35:1, which meets the contrast requirement, and even meets the AAA requirement of 7:1.

WCAG also specifies “text can be resized without assistive technology up to 200 percent without loss of content or functionality”. The contents of the tree canvas can be zoomed in or out without limit or loss in quality. The controls above the tree can be zoomed using a web browser’s zoom functionality without issue, though this does reduce the screen space that the tree canvas can occupy.

One guideline that ClickDeduce fails to accomplish is 1.3.4: Orientation, which specifies that the content is presented correctly in both landscape and portrait orientations. This guideline concerns mobile devices, which are not currently supported by ClickDeduce, so this is not an issue currently.

### 6.3 Performance

The original, server-based, implementation would be greatly concerned with performance, for example in terms of network latency, uptime, and performance under load. With the client-side-only implementation, these concerns are eliminated. Instead, the factors to consider become code bundle size and the performance of the site on the web browser itself.

#### 6.3.1 Bundle Size

By default, Webpack recommends that the limit for the entry point (initial bundle) should be 250KB [15]. Meeting this requirement when using ScalaJS is challenging due to the outputted JavaScript code being very large, even with the full optimisation mode, with result optimisation being “typically between 150KB and a few hundreds of KB” [16]. The Scala codebase in ClickDeduce is large, resulting in the optimised output being 492KB.

Combined with the frontend TypeScript code, the final bundle is 568KB – more than double the recommended limit. This can result in long page load times on slower connections. This is difficult to resolve because all of the code in ClickDeduce is for the main page, meaning that it is not possible to split the bundle into multiple modules for specific pages.

Since the website only consists of two pages (which share the same bundled code) this is not a terrible problem, as users will only need to load the bundled code once.

### 6.3.2 Client-Side Performance

After the page has successfully loaded, the performance of the page itself within the browser is the only other area to consider. Removing the server from the picture means that the user's actions will never need to wait for a server response, meaning that the interface has the potential to be very quick.

ClickDeduce is not intended to require a powerful system and should be responsive on a slower device such as a laptop on battery power where CPU performance is limited.

These tests were performed on Google Chrome using its DevTools to profile performance. CPU throttling was set to 6 times slower than normal, to simulate a much weaker device.

In web development, a long task is a main thread operation that takes more than 50ms to complete [17]. Tasks on the main thread block other main thread operations, such as UI actions, while they are executing.

Tested in Google Chrome, using DevTools performance window to capture task time in milliseconds. CPU throttling was set to six times. All tests have a sample size of ten. All tests were performed in the "edit" view mode.

#### *Literal Edits*

In all tests, LITERAL was a Num node where the literal was changed for each test sample. The values for literals were a mix of valid and invalid entries, as well as having varying lengths. The contents of the literal did not have a significant impact on the task times.

- Small: LITERAL
- Medium:  $\lambda x: \text{Int}. \text{LITERAL} + 1$
- Large:  $\text{rec factorial}(n: \text{Int}): \text{Int}. \text{if } n < 1 \text{ then } 1 \text{ else } (n \times \text{factorial}(n + \text{LITERAL}))$

	<i>Mean (ms)</i>	<i>Std. Deviation (ms)</i>
<i>Small</i>	44.7	10.1
<i>Medium</i>	78.6	5.6
<i>Large</i>	179.4	5.4

#### *Expression Selection*

In all tests, EXPR was an expression choice node where a different expression was selected for each test sample.

- Small: EXPR
- Medium:  $(\lambda x: \text{Int}. x + \text{EXPR}) \ 84$
- Large:  $(\text{rec factorial}(n: \text{Int}): \text{Int}. \text{if } n < 1 \text{ then } 1 \text{ else } (n \times \text{factorial}(n + \text{EXPR}))) \ 3$

	<i>Mean (ms)</i>	<i>Std. Deviation (ms)</i>
<i>Small</i>	44.0	9.9
<i>Medium</i>	82.0	16.1
<i>Large</i>	162.7	21.6

### Expression Deletion

In all tests, EXPR was a subexpression (of varying contents per sample) that was deleted. The expression formats used match those from the Expression Selection tests.

	<i>Mean (ms)</i>	<i>Std. Deviation (ms)</i>
<i>Small</i>	23.6	4.9
<i>Medium</i>	56.4	5.0
<i>Large</i>	137.2	16.7

### Conclusion

Performance testing discovered that, in general, larger result trees resulted in tasks that significantly exceeded the 50ms limit, e.g. the mean task time for editing a literal in a large factorial function was 179ms.

While this is a pessimistic case for performance, ClickDeduce would possibly benefit from moving the execution of actions from the main thread to additional threads that do not block further UI actions, to make the interface more responsive.

### Very Large Trees

Since the computational load scales with the tree size, creating an extremely large tree (e.g. 100 or more nodes) results in significantly degraded performance. In an example with the evaluation tree for applying the factorial function to the number 20, panning caused stuttering (tasks of 100-200ms) and evaluating applying the factorial function to 30 took 10 seconds to complete.

This may be possible to alleviate with optimisations in the future, but handling excessively large trees is not a key consideration of ClickDeduce. Trees of this size would never be possible on paper and are impractical to view regardless of the ability to pan and zoom.

## 6.4 Example User Journeys

### 6.4.1 First Time User

In this case, the user is a student who has recently started attending lectures for the “Elements of Programming Languages” course and has completed the first tutorial session. The tutor recommends they use ClickDeduce when preparing for the next tutorial. So far, the LArith and LIf languages have been presented in the lectures, and the second tutorial includes questions about these.

The user follows the link to ClickDeduce provided to them via email or as part of the tutorial notes. The webpage loads, presenting them with the main interface. The student sees the help section and clicks the link to the guide page.

The guide page appears, and the student skims over the descriptions of the website. They notice the interactive element for learning how to select an expression and click on the text box. Underneath it, an element states “Selected: ?”. Their cursor appears in the text box and a dropdown menu appears with three options: “Num”, “Plus”, and “Times”. The student clicks on the “Times” option, causing the dropdown menu to disappear, filling the text box with “Times”, and causing the element underneath to state “Selected: Times”.

Satisfied they understand how to select an expression, the user then continues to the next part of the guide page which includes another interactive element; the literal example. They click on the literal text box and type in “foo”. The evaluation result remains a red exclamation mark, with

a dashed underline denoting that it can be hovered over. The user hovers over the exclamation mark, causing a tooltip to appear; stating “Num can only accept LiteralInt, not foo”. The user clicks back into the text box and types in “67”, causing the evaluation result to become “67: Int”.

At this point, the student decides to return to the main page. They want to try creating an expression which uses the “IfThenElse” expression from LIf. Clicking on the expression selection text box, the same three options from the guide page appear, “IfThenElse” is not an option.

The student sees the language selector, currently containing “LArith”, and clicks on the dropdown menu. The next option is “LIf” which they click on since it was mentioned in the lectures they attended. Clicking on the expression selection text box again, additional options appear; the new options are “Bool”, “Equal”, “LessThan”, and “IfThenElse”.

Now that the option is available, the student clicks on the “IfThenElse” option. This causes the blank node to be replaced by the tree shown in Figure 30. There are now three blank expression selectors that appear above the root node, one for each of the subexpressions of the IfThenElse expression. All three show that the current type-checking result is an error since the expression is incomplete.



Figure 30: Visible tree after selecting “IfThenElse”

Next, the student decides that the condition for the expression should be simply whether the numbers 1 and 2 are equal. They mouse over the “Unspecified Expression” element between the “if” and the “then” text. Their cursor becomes a not-allowed sign and the leftmost expression selector on the line above is highlighted in green, indicating that the student needs to click there instead. They click on the highlighted expression selector, causing the same expression names from before to appear in the dropdown menu. Next, they click on the “Equal” expression name, replacing that selector with a new node with two blank expression selectors above it.

Once again, hovering over the “Unspecified Expression” element in either parent expression (either the Equal or IfThenElse node) highlights the corresponding expression selector. The student selects “Num” in the selector corresponding to the left-hand subexpression of the Equal node. This replaces that selector with a new Num node, as shown in Figure 31.



Figure 31: Visible tree after also selecting “Equal” and “Num”

Recalling how numbers are entered from the example from the guide page, the student clicks on the literal input text box and types “1”. They then perform the same procedure with the other half of the Equal expression, entering “2” for the literal. The Equal node now shows that the evaluation result (for that node) is “false” with the “Bool” type.

The student then decides to fill in the rest of the IfThenElse with simple Num expressions. They repeat the steps for selecting “Num” and entering a value for the other two subexpressions. The student enters “4” for the “then” subexpression and “-8” for the “else” subexpression.

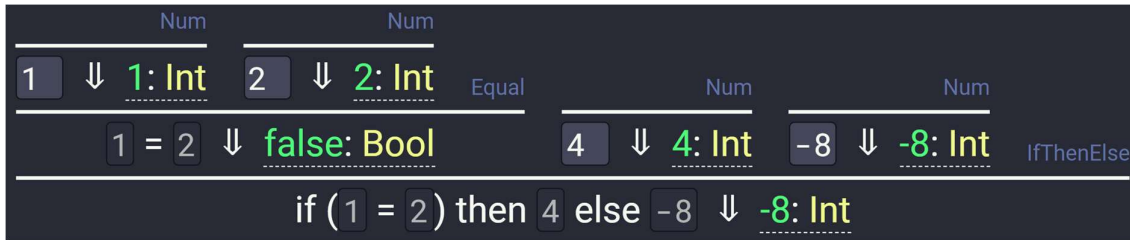


Figure 32: Complete expression tree

### Summary

This student visited the website for the first time, starting by reading the guide. Within the guide, they used the interactive expression selector and literal input to help understand these important interface elements. They then returned to the main site and constructed a simple expression using the IfThenElse, Equal, and Num expressions.

ClickDeduce helps introduce this new user with the guide page it offers. The simplicity of the application, where the student just needs to select which expression they want at each point from a list of available expressions and type in the values of the literals, means there is not a large hurdle to starting to use the application.

A key limitation here is that the student may either not notice the link to the guide page or choose to ignore it. In this case, the interface elements will be a lot more difficult to understand, since there are no instructions on the page itself.

## 6.4.2 Experienced User

This user has used ClickDeduce before. They wanted to use expressions from a language they were unfamiliar with; lambda functions from LLam. They started by switching to the language using the language selector in the controls. They switched to the evaluation view mode since they wanted to create an evaluation tree. In the root node, they selected “Lambda” by typing it in and pressing the Enter key.



Figure 33: Lambda expression with unspecified contents in evaluation mode, it is not possible to specify its contents in this view mode.

They then appeared to be stuck, because there was no way to enter the expression inside the Lambda node. This is because the evaluation rule for Lambda has no premises, meaning there is no way to select the expression. At this point, they would either need to realise that they need to switch view mode (both edit or type-check mode work) or ask for assistance.

Once they resolved this by changing the view mode, they could then enter their expression. To keep it simple, they select Int as the input parameter

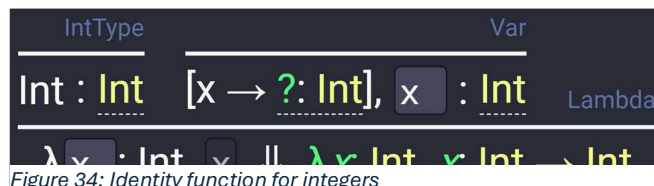


Figure 34: Identity function for integers



type and Var for the expression. In the literal inputs for both the Lambda and Var nodes they enter “x”.

They had a valid expression for the identify function for integers. They now wanted to apply a value to this function. There were two options for this:

1. Reset the tree (by reloading the page or deleting the root node through the context menu), then create the correct tree from scratch, including the lambda function again.
2. Copy the root node (through the context menu), then delete it. Select Apply at the root node, then paste the copied node into the left subexpression of the Apply.

From this example script, two issues are clear. First, it is not immediately clear that the user cannot edit lambda expressions while in evaluation mode. Second, changing parent expressions is not easy.

### 6.4.3 Expert User

In this example, a course organiser attempts to add a new language feature to ClickDeduce, corresponding to course content.

The key issue with this is a lack of documentation. There is currently no guide on how to add new language features to ClickDeduce, so the course organiser would have to look through the repository of code and spend time learning how it works.

Distributing their modified version of ClickDeduce would have to be done by the course organiser (for example by hosting the page on the university’s servers). There is no way to use the new language features on the original version of ClickDeduce, but this makes sense as this would require the ability to load any user-supplied code, which could be dangerous to users.

## 7 Conclusion

Overall, ClickDeduce achieves what it sets out to be: a simple and accessible tool to accompany the Elements of Programming Languages course. It is free and permanently hosted online as long as the GitHub Pages service remains active.

Future user testing is necessary to make improvements to the interface.

It has the potential to be used in a broader scope of programming language design, but some improvements are still needed on this front. In particular, more documentation is required.

### 7.1 Next Steps

This was the first half of the two-year-long Masters project. As such, there is a significant amount of time available next year to improve the application.

#### 7.1.1 User Studies

As a learning tool for the Elements of Programming Languages course, it is most useful to test the application with students only just beginning the course, rather than students that have already completed it.

During the next year of the course, user testing with students taking the course will be invaluable. This will, with appropriate permission, include inviting students at different points in the course to try the application. The feedback from this will heavily influence the changes made to the interface.

#### 7.1.2 Statements

Supporting small-step semantics and statements will require significant changes to the application. The current tree structure and expression architecture are closely linked to big-step semantics, meaning new structures would be required.

#### 7.1.3 Better Expression Extending

While attempts have been made to make ClickDeduce as easy to modify as possible, there are still significant areas for improvement. Critically, comprehensive instructions for how to create new language features are required.

To demonstrate the utility of the tool, it would be beneficial to implement a simple real-world programming language within ClickDeduce.

#### 7.1.4 Interface Improvements

As with any graphical interface, there are almost always ways to make improvements. This will likely be the area most influenced by the results of testing with new users.

- Hotkeys for certain actions, such as undoing an action or deleting a node
- Some options appear when hovering over a node, such as deleting that node, without needing to open the context menu
- Fewer duplicate environments displayed

### 7.1.5 Challenges and Tasks

ClickDeduce currently does not direct the user to complete any particular tasks. While this makes sense as a tool for constructing derivation trees, it would significantly help students if this application also served as a teaching tool itself. This would not necessarily be a huge shift in scope, and could instead manifest as a new selection of interface elements which direct the student to use expressions introduced in the language they select in particular ways.

## 8 References

- [1] R. Holmes, “ClickDeduce,” 21 March 2024. [Online]. Available: <https://clickdeduce.rgh.dev/>.
- [2] The University of Edinburgh, “Course Catalogue - Elements of Programming Languages (INFR10061),” [Online]. Available: <http://www.drps.ed.ac.uk/23-24/dpt/cxinfr10061.htm>. [Accessed 21 March 2024].
- [3] B. Fitelson, “Running MacLogic Under Emulation,” [Online]. Available: <https://fitelson.org/maclogic.htm>. [Accessed 25 March 2024].
- [4] G. Forbes, “A Brief Guide to MacLogic,” [Online]. Available: [https://spot.colorado.edu/~forbesg/pdf\\_files/MacLogic\\_Guide.pdf](https://spot.colorado.edu/~forbesg/pdf_files/MacLogic_Guide.pdf). [Accessed 25 March 2024].
- [5] B. Atkey, “Interactive Natural Deduction,” 24 September 2018. [Online]. Available: <https://bentnib.org/docs/natural-deduction.html>. [Accessed 19 October 2023].
- [6] L. O'Connor and R. Amjad, “Holbert,” 2022. [Online]. Available: <https://arxiv.org/abs/2210.11411>. [Accessed 17 March 2024].
- [7] V. Dicevicius, “Click-Deduce: Interactive Inference Rule Explorer,” 2017.
- [8] J. Robert, “EDUCAUSE QuickPoll Results: Flexibility and Equity for Student Success,” 5 November 2021. [Online]. Available: <https://er.educause.edu/articles/2021/11/educause-quickpoll-results-flexibility-and-equity-for-student-success>. [Accessed 14 March 2024].
- [9] GitHub, “GitHub Pages,” [Online]. Available: <https://pages.github.com/>. [Accessed 21 March 2024].
- [10] Sass, “Sass: Syntactically Awesome Style Sheets,” [Online]. Available: <https://sass-lang.com/>. [Accessed 21 March 2024].
- [11] Webpack, “webpack,” [Online]. Available: <https://webpack.js.org/>. [Accessed 21 March 2024].
- [12] S. R. Buss, “CTAN: Package bussproofs,” 20 August 2012. [Online]. Available: <https://ctan.org/pkg/bussproofs>. [Accessed 14 February 2024].
- [13] A. Kashcha, “anvaka/PanZoom: Universal pan and zoom library (DOM, SVG, Custom),” 4 June 2022. [Online]. Available: <https://github.com/anvaka/panzoom>. [Accessed 8 January 2024].
- [14] World Wide Web Consortium, “Web Content Accessibility Guidelines (WCAG) 2.1,” 21 September 2023. [Online]. Available: <https://www.w3.org/TR/WCAG21/>.
- [15] Webpack, “Performance | webpack,” [Online]. Available: <https://webpack.js.org/configuration/performance/>. [Accessed 21 March 2024].

- [16] Scala.js, “Compilation and optimization pipeline - Scala.js,” [Online]. Available: <https://www.scala-js.org/doc/internals/compile-opt-pipeline.html>. [Accessed 21 March 2024].
- [17] Mozilla Corporation, “Long task - MDN Web Docs Glossary,” 8 June 2023. [Online]. Available: [https://developer.mozilla.org/en-US/docs/Glossary/Long\\_task](https://developer.mozilla.org/en-US/docs/Glossary/Long_task).
- [18] R. Harper, Practical Foundations for Programming Languages, Cambridge University Press, 2016.

## Appendix 1: LaTeX Output Examples

Empty node:

$$\overline{\text{Unselected Expression} \Downarrow \text{error!}}$$

Num node:

$$\overline{8165 \Downarrow 8165}$$

Arithmetic tree:

$$\frac{\frac{45 \Downarrow 45}{45 + 7 \Downarrow 52} \quad \frac{7 \Downarrow 7}{-41658 \Downarrow -41658}}{(45 + 7) \times -41658 \Downarrow -2166216}$$

Boolean tree:

$$\frac{\frac{56 \Downarrow 56}{56 = (2 \times 38) \Downarrow \text{false}} \quad \frac{\frac{2 \Downarrow 2}{2 \times 38 \Downarrow 76} \quad \frac{38 \Downarrow 38}{6 \Downarrow 6}}{\text{if } (56 = (2 \times 38)) \text{ then } (1 + 2) \text{ else } 6 \Downarrow 6} \quad \frac{87 \Downarrow 87}{(\text{if } (56 = (2 \times 38)) \text{ then } (1 + 2) \text{ else } 6) + 87 \Downarrow 93}}$$

Let tree:

$$\frac{\frac{\text{true} \Downarrow \text{true}}{[x \rightarrow \text{true}], x \Downarrow \text{true}} \quad \frac{\frac{[x \rightarrow \text{true}], 1 \Downarrow 1}{[x \rightarrow \text{true}], 1 + 2 \Downarrow 3} \quad \frac{[x \rightarrow \text{true}], 2 \Downarrow 2}{[x \rightarrow \text{true}], 1 + 2 \Downarrow 3}}{\text{let } x = \text{true} \text{ in } (\text{if } x \text{ then } (1 + 2) \text{ else } 6) \Downarrow 3}$$

Lambda tree:

$$\frac{\frac{\lambda x : \text{Int}. x + 1 \Downarrow \lambda x : \text{Int}. x + 1}{42 \Downarrow 42} \quad \frac{\frac{[x \rightarrow 42], x \Downarrow 42}{[x \rightarrow 42], x + 1 \Downarrow 43} \quad \frac{[x \rightarrow 42], 1 \Downarrow 1}{[x \rightarrow 42], x + 1 \Downarrow 43}}{(\lambda x : \text{Int}. x + 1) 42 \Downarrow 43}$$

## Appendix 2: Language Feature Examples

### Expressions

```
case class Num(x: Literal) extends Expr {
  override def evalInner(env: ValueEnv): Value = x match {
    case LiteralInt(x) => NumV(x)
    case _              => UnexpectedArgValue(s"Num can only
accept LiteralInt, not $x")
  }

  override def typeCheckInner(tEnv: TypeEnv): Type = x match {
    case LiteralInt(_) => IntType()
    case _              => UnexpectedArgType(s"Num can only accept
LiteralInt, not $x")
  }

  override def toText: ConvertableText = MathElement(x.toString)

  override val needsBrackets: Boolean = false
}
```

```
case class Plus(e1: Expr, e2: Expr) extends Expr {
  override def evalInner(env: ValueEnv): Value = (e1.eval(env),
e2.eval(env)) match {
    case (NumV(x), NumV(y))      => NumV(x + y)
    case (v1, _) if v1.isError => v1
    case (_, v2) if v2.isError => v2
    case (v1, v2)               => UnexpectedArgValue(s"Plus
cannot accept ($v1, $v2)")
  }

  override def typeCheckInner(tEnv: TypeEnv): Type =
(e1.typeCheck(tEnv), e2.typeCheck(tEnv)) match {
    case (IntType(), IntType()) => IntType()
    case (t1, _) if t1.isError  => t1
    case (_, t2) if t2.isError  => t2
    case (t1, t2)              => UnexpectedArgType(s"Plus
cannot accept ($t1, $t2)")
  }

  override def toText: ConvertableText =
    MultiElement(e1.toTextBracketed,
    SurroundSpaces(MathElement.plus), e2.toTextBracketed)
}
```

## Types

```
case class BoolType() extends Type {  
  override val needsBrackets: Boolean = false  
  
  override def toText: ConvertableText = TextElement("Bool")  
}
```

```
case class PairType(l: Type, r: Type) extends Type {  
  override def typeCheck(tEnv: TypeEnv): Type =  
    PairType(l.typeCheck(tEnv), r.typeCheck(tEnv))  
  
  override def toText: ConvertableText =  
    MultiElement(l.toTextBracketed,  
      SurroundSpaces(TimesSymbol()), r.toTextBracketed)  
}
```

```
case class TypeVar(v: Literal) extends Type {  
  override def typeCheck(tEnv: TypeEnv): Type =  
    tEnv.get(v.toString) match {  
      case None          => UnknownTypeVar(v)  
      case Some(TypeVar(t)) => TypeVar(t)  
      case Some(other)    => other.typeCheck(tEnv)  
    }  
  
  override val needsBrackets: Boolean = false  
  
  override def toText: ConvertableText = v.toText  
}
```

## Values

```
case class BoolV(b: Boolean) extends Value {  
  override val typ: Type = BoolType()  
  
  override val needsBrackets: Boolean = false  
  
  override def toText: ConvertableText = TextElement(b.toString)  
}
```



## Appendix 3: EPL Language Features

$Expr \ni e$	$::=$	$n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 \times e_2$	$L_{Arith}$
		$b \in \mathbb{B} \mid e_1 == e_2 \mid \text{if } e \text{ then } e_1 \text{ else } e_2$	$L_{If}$
		$x \mid \text{let } x = e_1 \text{ in } e_2$	$L_{Let}$
		$e_1 e_2 \mid \lambda x:\tau. e$	$L_{Lam}$
		$\text{rec } f(x:\tau_1) : \tau_2. e$	$L_{Rec}$
		$(e_1, e_2) \mid \text{fst } e \mid \text{snd } e \mid \text{let pair } (x, y) = e_1 \text{ in } e_2 \mid ()$	
		$\text{left}(e) \mid \text{right}(e) \mid \text{case } e \text{ of } \{\text{left}(x) \Rightarrow e_1 ; \text{right}(y) \Rightarrow e_2\}$	$L_{Data}$
		$\Lambda A. e \mid e[\tau]$	$L_{Poly}$
$Type \ni \tau$	$::=$	$\text{int}$	$L_{Arith}$
		$\text{bool}$	$L_{If}$
		$\tau_1 \rightarrow \tau_2$	$L_{Lam}$
		$\tau_1 \times \tau_2 \mid \text{unit} \mid \tau_1 + \tau_2 \mid \text{empty}$	$L_{Data}$
		$A \mid \forall A. \tau$	$L_{Poly}$
$\Gamma$	$::=$	$x_1 : \tau_1, \dots, x_n : \tau_n$	
$Value \ni v$	$::=$	$n \in \mathbb{N}$	$L_{Arith}$
		$b \in \mathbb{B}$	$L_{If}$
		$\lambda x. e$	$L_{Lam}$
		$\text{rec } f(x). e$	$L_{Rec}$
		$(v_1, v_2) \mid () \mid \text{left}(v) \mid \text{right}(v)$	$L_{Data}$
		$\Lambda A. e$	$L_{Poly}$