SCHOOL OF COMPUTER SCIENCE

Implementing a Step by Step Evaluator for a Simple Functional Programming Language

Kiran Sturt

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering **worth 40CP**.

Saturday 19th April, 2025

Abstract

Students often find functional programming languages more difficult to learn than imperative languages, and they may struggle to gain an intuitive understanding of how functional languages are evaluated. I have created a tool SFL Explorer, available at https://functional.kiransturt.co.uk, which aims to help build intuitive understanding of how functional programming languages work. The primary use case this project has been designed and tested for is for use as a demonstration tool in lectures, particularly in the University of Bristol's own combined imperative and functional programming unit COMS10016. My client, Samantha Frohlich, a lecturer on this unit plans to integrate this system into the unit in future.

The system includes my own functional programming language. SFL is a very minimal language, but it includes many standard functional programming features, including polymorphism, pattern matching and user definable algebraic data types. This language is type checked, using a modified version of Dunfield and Krishnaswami's bidirectional type checking algorithm [6], modified to include SFL's extended type system.

All functionality for the language was written in Rust, compiled to Web Assembly, and included into a React app that acts as the frontend. This functionality is therefore available entirely client side. The app is a Progressive Web App (PWA) and is therefore able to be installed and used offline.

As the system is designed to be a teaching tool, I have done user testing in the form of 3 focus groups at various points throughout the project who are at various stages in the process of learning functional languages. Their feedback ensured that the project stayed on track and remained as useful as possible to potential users of a wide variety of skill levels.

Dedication and Acknowledgements

My supervisors, Jess Foster and Sarah Connolly, have been unwaveringly helpful, supportive and kind throughout this project, as well as my university journey as a whole. I would like to thank them for all of their help, without which this would not have been possible.

I would like to thank Samantha Frohlich for being a really great client, whose enthusiasm for what I was creating really inspired me to do my best work. Likewise, I would also like to thank Dr. Steven Ramsey for being a fantastic lecturer in programming languages; this project would not have been anywhere near as good without the knowledge and inspiration I gained from his lectures.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others including AI methods, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Kiran Sturt, Saturday 19th April, 2025

AI Declaration

I declare that any and all AI usage within the project has been recorded and noted within Appendix A or within the main body of the text itself. This includes (but is not limited to) usage of translators (even google translators), text generation methods, text summarisation methods, or image generation methods.

I understand that failing to divulge use of AI within my work counts as contract cheating and can result in a zero mark for the dissertation or even requiring me to withdraw from the University.

Kiran Sturt, Saturday 19th April, 2025

Contents

List of Figures

List of Tables

Ethics Statement

Supporting Technologies

- I used React to develop the website for this project.
- The bindings for the web assembly interface to the library for the language were generated by using macros from the wasm-pack rust crate.
- I used GitHub Copilot to help assist with generating unit tests.

Notation and Acronyms

SFL Simple Functional Language

FP Functional Programming

WASM Web ASseMbly

CLI Command Line Interface

MVP Minimum Viable Product

NPM Node Package Manager

AST Abstract Syntax Tree

Chapter 1

Introduction

In this dissertation I present SFL-explorer: a tool to demonstrate how functional programming languages are evaluated, allowing users to gain a valuable intuition of these languages. It is an open source web based tool, available for download and offline use as a PWA (Progressive Web App, see 2.4).

SFL-explorer takes the form of a functional language (Simple Functional Language (SFL)), packaged with two interfaces that allows users to observe the process of evaluation of a term as a series of step by step or multi-step reductions, and control the order that sub-terms are evaluated. These interfaces are a Command Line Interface (CLI) and a web application. The ultimate goal of this project was to make a tool that makes learning and teaching the basics of functional programming easier. There are two groups of people the project is designed to be of interest to:

- Those involved in learning functional languages. These could be students of a university course, or anyone interested in the topic.
- Those involved in teaching functional languages, as part of a university course or otherwise.

1.1 The Language

The language itself is not meant to be the main interest for the users of this system. It is designed to be fairly generic, with syntax and semantics similar to popular functional languages, so that users can take their understanding from using SFL-explorer and apply it to these languages. 1.1 is an example program in the language, to find the factorial of 2. The relevant prelude functions are included for clarity.

```
1  if :: Bool -> a -> a -> a
2  if cond then_branch else_branch = match cond {
3    | true -> then_branch
4    | false -> else_branch
5  }
6
7  fac :: Int -> Int
8  fac n = if (n <= 1) (1) (n * (fac (n - 1)))
9
10 main :: Int
11 main = fac 2
```

Figure 1.1: An example SFL program. Evaluation is shown 1.2

1.2 is a table showing the evaluation of this function in lazy mode by the system. The 'Prompt' column shows what the user is presented with as a button to make progress. The first prompt entry at row 0 is empty, as it represents the starting program state. This table is generated dynamically as the user progresses through the given program.

The user is provided with messages telling them what the next step that they can make is. Additionally, there is a 'free choice' mode where users are presented with the options for progress, and they can choose which one is taken.

1.2 COMS10016: Imperative and Functional Programming at the University of Bristol

In the first year of most computer science programs at the University of Bristol, students take the module COMS10016, a combined imperative and functional programming module. This is many students first encounter with both of these types of programming. In the functional part of this unit, students are taught Haskell. The unit material is presented to students through a very effective lecture series, supplemented by weekly worksheets that students have the opportunity to work through in labs attended by the lecturers, as well as some teaching assistants. Two of the lecturers in this unit are Jess Foster and Samantha Frohlich.

> 'The aim [of the functional portion of the unit] is to introduce types and functions. Important principles include datatypes, evaluation order, higher-order functions, and purity' [17]

I acted as a teaching assistant in the labs for two academic years. My role was to answer students questions about functional languages or the worksheets they were given. The inspiration for this project came from my experience struggling to explain key functional programming concepts.

1.3 Agile Development Lifecycle

The project followed a development lifecycle inspired by Agile principles[3], structured into four iterative phases. Each phase built upon the last, integrating evaluation and feedback to continuously and rapidly refine the features and the UI/UX of the system.

Each phase was further subdivided into four phases:

- **Requirements Gathering**
- **Design and Research**
- **Implementation**
- **Evaluation**

This iterative methodology helped manage complexity and uncertainty. Getting frequent feedback from focus groups

Step	**Prompt**	**Main Expression Afterwards**
0		`fac 2`
1	Apply function 'fac' to 2	`if (2 <= 1) 1 (2 * (fac (2 - 1)))`

2	Apply function if to (2 <= 1), 1 and (2 * (fac (2 - 1)))	`match (2 <= 1) {`
		`  | true -> 1`
		`  | false -> 2 * (fac (2 - 1))`
		`}`

3	Apply inbuilt <= to '2' and '1'	`match (false) {`
		`  | true -> 1`
		`  | false -> 2 * (fac (2 - 1))`
		`}`

4	Match to pattern 'false'	`2 * (fac (2 - 1))`
5	Apply function fac to (2 - 1)	`2 * (if ((2 - 1) <= 1) 1 ((2 - 1)` `  * (fac ((2 - 1) - 1))))`

6	Apply function if to ((2 - 1) <= 1), 1 and ((2 - 1) * (fac ((2 - 1) - 1)))	`2 * match ((2 - 1) <= 1) {`
		`  | true -> 1`
		`  | false -> (2 - 1) * (fac ((2 - 1) - 1))`
		`}`

7	Apply inbuilt - to 2 and 1	`2 * match (1 <= 1) {`
		`  | true -> 1`
		`  | false -> 1 * (fac (1 - 1))`
		`}`

8	Apply inbuilt <= to 1 and 1	`2 * match (true) {`
		`  | true -> 1`
		`  | false -> 1 * (fac (1 - 1))`
		`}`

9	Match to pattern true	`2 * 1`
10	Apply inbuilt * to 2 and 1	`2`

Figure 1.2: A table showing how the system leads a user through the step by step evaluation of the program shown at 1.1

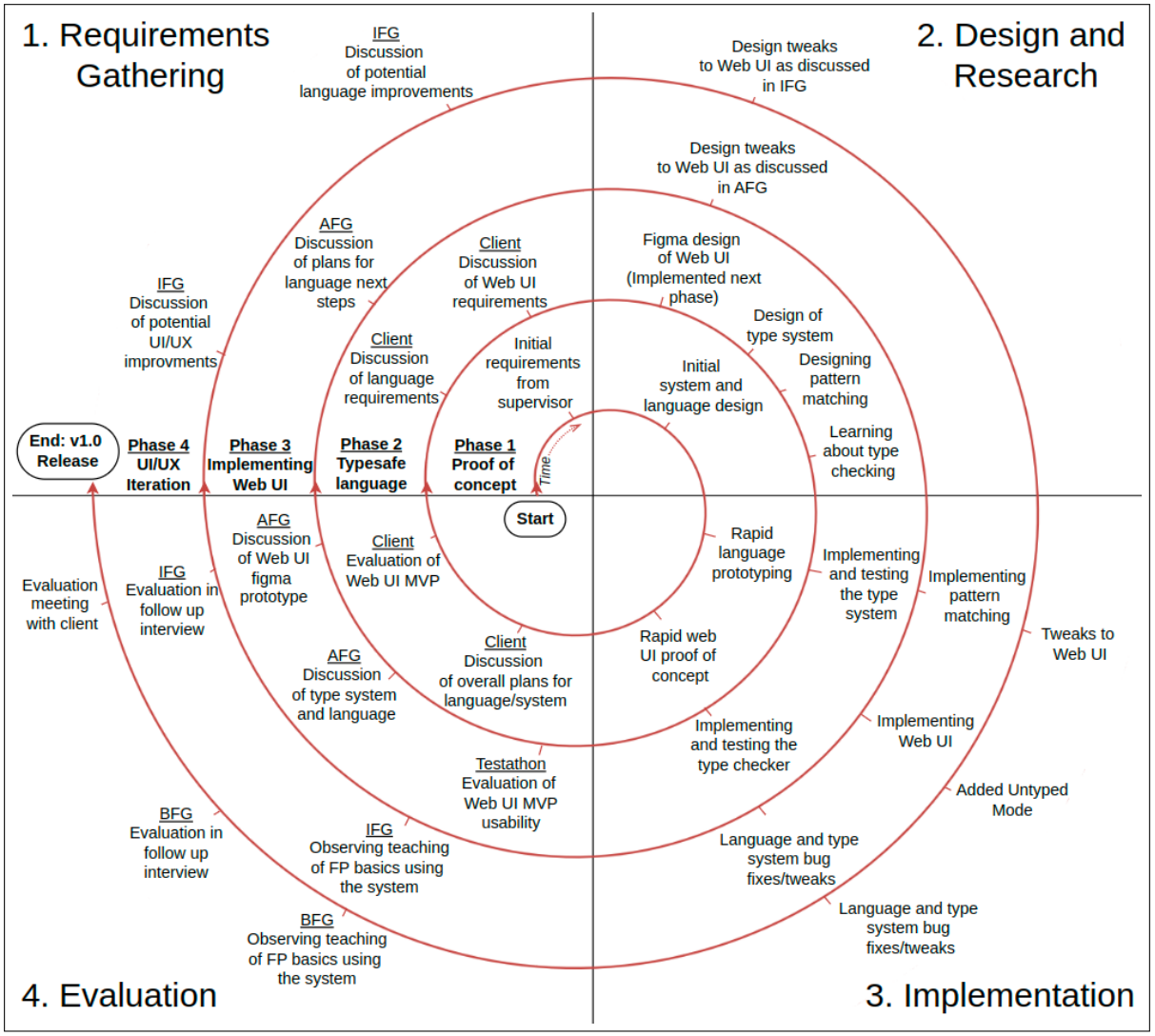

Figure 1.3: A spiral representation of the project lifecycle, showing the 4 iterations, and the work done in each part of each phase.

Chapter 2

Background

2.1 The Lambda Calculus

The lambda calculus (λ-calculus) was described by Alonzo Church in 1936 [5]. [Sam: too terse: what is the lamda calc? (model of computation). Why is it important / interesting to this project?] Below is a definition of the λ-calculus [2].

[Sam: you can't jump straight into this]

Definition 1 (The λ-calculus)
The set of λ-terms, notation Λ, is built up from an infinite set of variables $V = v, v', v'', ...$ [Sam: here you say v, but below you use x] using application and (function) abstraction: [Sam: functions? application? abstraction? what are these? I recommend getting Sarah to have a look a this BG to see what a non-PL person needs to know]

$$\begin{array}{lcl} x \in V & \implies & x \in \Lambda \\ M, N \in \Lambda & \implies & (MN) \in \Lambda \\ M \in \Lambda, x \in V & \implies & (\lambda x.M) \in \Lambda \end{array}$$

Using abstract syntax one may write the following:

$$\begin{array}{l} V ::= v \mid V' \\ \Lambda ::= V \mid (\Lambda\Lambda) \mid (\lambda V \Lambda) \end{array}$$

[Sam: I dont think the way you have done variables is standard, i know you dont wanna plagurise, but this is such a basic thing that its better to cite where you got the AST from than tweek to get your own]

We shall also use the following conventions:

Definition 2 (The λ-Calculus Conventions)

1. *$x, y, z, ...$ denote arbitrary variables.*
2. *We may omit outermost parenthesis.*
3. *Nested abstractions can be grouped. For instance, if we were to write the term $\lambda x\ y.M$, we would mean $(\lambda x.(\lambda\ y.M))$.*
4. *We assume that the body of an abstraction extends as far to the right as possible. For instance, the term $\lambda x.M\ N$ means $(\lambda x.(M\ N))$ and not $((\lambda x.M)\ N)$. [Sam: is this case you have used the standard thing, but I'd cite where you got these conventions from]*

2.1.1 Reduction

The λ-calculus is evaluated by β-reduction. This is where an abstraction is applied to a value. The result of applying an abstraction to a term is the body of the abstraction, with the all free [Sam: what do you mean by free?] instances of the abstracted variable are substituted with the term the abstraction was applied to. Below is the definition of how this substitution works, followed by the definition of β-reduction.

Definition 3 (Substitution of free variables in terms of the λ-calculus)

$$\begin{array}{lcl} x[x ::= N] & \equiv & N \\ y[x ::= N] \text{ where } y \neq x & \equiv & y \\ (M_1 M_2)[x ::= N] & \equiv & (M_1[x ::= N])(M_2[x ::= N]) \\ (\lambda y.M)[x ::= N] & \equiv & \lambda y.(M[x ::= N]) \end{array}$$

Definition 4 (β-Reduction of the λ-calculus)

$$\begin{array}{lcl} x & \rightarrow_\beta & x \\ \lambda x.M & \rightarrow_\beta & \lambda x.M \\ (\lambda x.M)N & \rightarrow_\beta & M[x ::= N] \end{array}$$

Definition 5 (Normal Form)
A term is said to be in normal form if it cannot be β-reduced

[Sam: these are all standard, so you should cite where they come from and explain them. Spend more time on reduction, because that will be key later] [Sam: this section is very dry. why do we want to learn about lambda calc? well we are writing an evaluator, so to understand what that means we need to understand syntax and evaluation, so lets looks at the lambda calc as a small example. It makes the intro of lambda calc more active and motivated, rather than a bunch of dry definitions, instead its a small example of what we will do later, that is background cos its pre-defined and not implemented we are just exploring the ideas we will implement]

2.1.2 Types

[Sam: para generally motivating types]

The λ-calculus that we have discussed so far is untyped. This means that any λ term can be applied to any other term. When a term can no longer be β-reduced (i.e. no further reductions are possible), we say it is a value. This is why β-reduction is often referred to as *evaluation* — it is the process by which terms are reduced to their final, fully evaluated form (i.e. values).

[Sam: i havent read further of lambda calc, cos i think my above advise applies here too. 1. you are burdened with too much knowledge so you keep jumping into the middle of an explanation 2. it is unclear why we are learning about types 3. remember lambda calc is a whole 3rd year course, and you've seen how long stevens notes are, we cannot replicate that so we need to highlight to the user what is important to understand about the lambda calc for just this project]

If we were to extend our λ-calculus with a new sort of term, an integer literal (something commonly done, especially when building up to discussing practical functional languages)

$$\ldots, -2, -1, 0, 1, 2, \ldots$$

we could say that these values are members of a set of values Int. It would be useful for us to be able to assert that a term eventually evaluates to one of these Ints. The λ-calculus terms that evaluate to a value in the set of Ints can all be said to have 'type' Int. More generally:

> 'Saying that 'a term t has type T' (or 't belongs to T,' or 't is an element of T') means that t 'obviously' evaluates to a value of the appropriate form - where by 'obviously' we mean that we can see this statically, without doing any evaluation of t' [20]

Functions

We want to be able to express the types of functions. The term $\lambda x.x$ can be said to have type $T \rightarrow T$, as it takes in a term of type T and returns the same term, which still has type T.

A more complex term $\lambda x\ y.x$ can be said to have type $T \rightarrow (U \rightarrow T)$; If we give it a term M of type T, it would return the function $\lambda y.M$ which takes whatever is given to it (represented by U) and returns M which has type T.

By convention, $\rightarrow$ is right associative so way may omit the right most parenthesis.

Typechecking: Well typed programs do not go wrong

In this extended value of the lambda calculus, the set of valid values V is:

$$V ::= \lambda x.M \mid \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid ...$$

The evaluation of an λ-calculus expression is said to have 'gone wrong' if it gets to a normal form that is not a valid value.

Let us consider the expression and its reduction

$$(\lambda x.x\ x)\ 1 \rightarrow_\beta 1\ 1$$

The reduction is not a valid value.

We will now attempt to derive the type of the parameter x, in order to show that it is untypeable. As x here is applied to itself, it must be some kind of function that takes a term x of with type T and then applies it to itself. This means that $T = T \rightarrow U$ which is absurd. This means that this is 'untypeable'. Indeed, it is clearly never possible to type an expression where a term is applied to itself. If this was a real programming language, when it got to the normal form 1 1, it would be some form of runtime error. We can see that only allowing typeable terms would have prevented us from creating a term that does not evaluate in this case.

In general, **well typed programs do not go wrong** [15]. Therefore, if we are able to exclude all terms in our functional language that are untypeable, we will be able to guarantee that it does not go wrong, and thus prevent these runtime errors.

The system that looks at a program to decide whether it is well typed is called the **typechecker**. Types can often also be inferred without specific assignments, which is called **type inference**.

2.2 Haskell: A Functional Programming Language

[Sam: why are we talking about Haskell? we need to talk about SFL!] Functional programming languages are programming languages where 'computation is carried out entirely through the evaluation of expressions' [8]. Functional programming languages are based on the lambda calculus. In this section, we will only discuss one example: Haskell.

Haskell is a very prominent functional programming language that is widely taught. It is a programming language specifically designed to be suitable for teaching [10]. This dissertation involves the development of a programming language with some similar features to Haskell, so the corresponding Haskell features and ideas will be introduced here.

2.2.1 Declarations

Haskell, along with most other languages, provides the facility to name functions and values for reference elsewhere in the program. These can be typed, but the types can almost always be inferred.

Some examples of these declarations, all typed for clarity, are below. For instance, the top level declaration

```
x :: Int
x = 5
```

means that x is equal to 5. We can also name lambda functions:

```
add :: Int -> Int -> Int
add = \x y -> x + y
```

This can be shortened to:

```
add :: Int -> Int -> Int
add x y = x + y
```

2.2.2 Polymorphic functions

In Haskell, functions can be written that operate on values of various types. A simple argument is

```
id :: a -> a
id x = x
```

which simply returns its argument. 'a' in the type signature represents any type, and can be substituted for any type. The two 'a's must be the same however, mandating that the argument and the return value must be the same type.

2.2.3 User Defined Algebraic Data Types

Many languages, including Haskell, have Algebraic Data Types allowing us to 'Compose' other data types. The set of all values of an algebraic data type is isomorphic to an expression involving the sets of values of their constituent types combined using 'set algebra' operations. Haskell allows for 'union' and 'product' types.

Haskell allows users to define their own algebraic data types using the `data` keyword. For instance, booleans can be defined as

```
data Bool = True | False
```

This data definition creates a type *Bool* with two data constructors, `True` and `False`. These data constructors are zero-ary. We can also have data constructors that have arguments.

An example of a union type in Haskell is the tagged union

```
data Shape = Circle Int | Rectangle Int Int
```

which is isomorphic to the type $Int \cup (Int \times Int)$.

An example of a product type is the tuple $(Int, Bool)$: the set of all possible values of this type is isomorphic to the Cartesian product of the set of all values of Int and the set of all values of $Bool$. Most languages have product types, which often take the form of structs or tuples.

2.2.4 Polymorphic Types and Kinds

Haskell includes polymorphic types. These are 'types that are universally quantified in some way over all types' [9].

One example is the type constructor $Maybe$, written as $Maybe\ a$. This is first-order polymorphism, as opposed to higher-order polymorphism where a type can be an 'abstraction over type constructors'. [24]. Here, $Maybe$ is not a type in itself, but it represents a constructor that takes a type, and returns a concrete type.

The 'Type of a Type' is its *kind* [20]. For example, the type constructor $Maybe$ has the kind $* \to *$. This notation looks similar to how functions over values are defined, indicating that it behaves like a function, but at the type level rather than the value level. If we were to apply the constructor $Maybe$ to the concrete type Int, the resulting type would be the concrete type $Maybe\ Int$.

'Either' is a type constructor with the kind $* \to * \to *$, meaning it takes two concrete types and returns a concrete type.

2.2.5 Pattern Matching

Haskell allows us to do pattern matching, allowing conditional execution based on whether a term matches a given form. The following function would have different results depending on whether the input value was 0 or another number

```
isZero :: Int -> Bool
isZero 0 = true
isZero _ = false
```

The underscore '_' represents a wildcard pattern that matches anything. In this case, it matches any Int that is not 0. We can match more complicated expressions, and assign variables throughout the pattern.

[TODO: ADD SYNTAX HIGHLIGHTING]

```
data SomeValues a = One a | Two a a | Three a a a | Four a a a a

valuesToList :: SomeValues a -> [a]
valuesToList (One x) = [x]
valuesToList (Two x1 x2) = [x1, x2]
valuesToList (Three x1 x2 x3) = [x1, x2, x3]
valuesToList (Four x1 x2 x3 x4) = [x1, x2, x3, x4]
```

In the first match case of `valuesToList`, we assign the variable x when matching the pattern. In the second, we assign `x1` and `x2` etc.

2.3 Rust

This project is written in rust. [Sam: why? motivate] Some of the decisions made, particularly in the implementation of the AST, require an understanding of Rust, especially the memory management model.

'Ownership' is an important concept. The rules of ownership [12]:

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

If a value is owned in one scope, but another scope needs to read/write it, we may use a reference to the value. The rules of references [12]:

- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid.

These rules ensure that immutable references are to things that don't change, and all references are always to things that exist.

2.4 Frontend Technologies

- Vite
- React
- NPM
- PWAs

2.5 Web Assembly

This project runs entirely within the browser, despite being written in Rust. This is due to the fact that it compiles to web assembly. Automated tools exist for the generation of JavaScript bindings around Rust functions/types, but this process places certain restrictions around their arguments and return type, or attributes. We will discuss this here to allow us to refer to these restrictions, and also to explain the process of compiling and using Rust code in a modern web browser.

Web Assembly 2.0 is a 32 bit target [22]. This means we only have 4 GB of addressable memory. The Rust compiler is based on LLVM, which provides a web-assembly compilation target. The Rust compiler has a toolchain around this compilation target [REFHERE: rust WASM toolchain], that allows for easy compilation to web-assembly. However, this only creates a binary blob, which requires more work to make interoperable with our JavaScript build system (Vite). We must do two things to achieve interoperability:

- Incorporate it into our build so it can be served with it.
- Load the WASM package in a way that allows for us to call the functions.

Producing an NPM package with some JavaScript functions that call the WebAssembly functions would achieve both of these goals. However, if we wish to use TypeScript, we must create a separate type definition file that contains the types of all of the JavaScript wrapper functions around the WASM functions. This would be difficult to maintain manually as we would have to update it every time we made a change to the public interface of our rust library.

Fortunately, the rust crate `wasm-bindgen` provides macros that generate a whole NPM package, including TS bindings, automatically. This package can then be added as a dependency to an NPM app that provides a website, and the functions within it can [TODO: WASM-bindgen vs WASM pack?]

wasm-pack

2.6 Existing systems

2.6.1 WinHIPE

2.6.2 Ask Elle

Chapter 3

Phase 1 — Proof of Concept

The goal of Phase 1, which spanned approximately the first month of my project, was to arrive at a proof of concept. This phase started at the beginning of the project with a discussion my supervisor, and the identification of my client. I proceeded by analysing the project requirements using autoethnographic methods, designing the system, designing SFL and the Explorer, and then implementing the proof of concept. I then evaluated the merits and drawbacks of the proof of concept by speaking to my client.

3.1 Requirements Analysis

3.1.1 Autoethnography

> 'Autoethnography is an ethnographic method in which the fieldworker's experience is investigated together with the experience of other observed social actors. [21]'

In this phase, I took an autoethnographic approach to requirements analysis and to design. As the 'fieldworker', I drew on my own experience being involved in teaching Haskell for the last two academic years. This experience was very valuable to this project, and it allowed me take the initial brief from my supervisor and effectively design a solution, and then quickly implement a proof of concept of this solution.

3.1.2 The Brief

This project was proposed by my supervisor, Jess Foster. In our initial meeting, we discussed how she wanted a tool that would help build intuition for how functional languages are evaluated, that she could use to supplement her explanation of otherwise difficult to intuit functional language concepts. We also discussed the benefits of the tool being accessible to students to use themselves during labs or at home. Jess helped me to identify an appropriate client: Samantha Frohlich. Jess and Samantha are both lecturers on COMS10016. It was necessary to identify a client other than Jess, as her existing role as my supervisor/primary marker could limit guidance she would be able to give me if she were also my client.

Following this meeting, I broke down this brief into smaller parts. Taking an autoethnographic approach, I used my own experience teaching functional languages to consider solutions and come up with requirements for each part.

Building Intuition

This is the key to an effective solution. [Sam: MOST students OF] Many students to the first year Functional Programming (FP) unit do not have any experience with functional programming. [TODO: REWORK THIS P and find it difficult to gain an intuition from passively watching lectures, and do not attend or engage with labs.

During phase 2 [Sam: but this is the phase 1 section?] [REFHERE: Testathon], to confirm my belief that students found functional languages harder to learn, and harder to build an intuition about, I performed a survey. There were 15 participants, who were a mixture of undergraduate and postgraduate computer scientists, all of whom had taken the first year FP unit. In one section of the survey, they were presented with a series of statements designed to gauge their feelings towards functional and imperative

languages. A Likert scale[13] was used to measure the attitudes of participants towards the statements. See 3.1a for imperative results, and 3.1b for functional results.

(a)

(b)

Figure 3.1: The results of a survey performed during the phase 2 [REFHERE: Testathon], where a Likert scale was used to gauge 15 participants feelings towards imperative (a) and functional (b) programming languages

For imperative languages, 80% of respondents agreed/strongly agreed that they can program in them, similarly 80% of respondents agreed/strongly agreed that they had an intuitive understanding of them. For functional languages, 66.7% of respondents strongly/agreed that they knew them, but only 47.6% of respondents would agree/strongly agree that they had an intuitive understanding of them. The number of people who claim to know how to program in functional languages is less than imperative, but more striking is the difference in reported 'intuition'.

In my experience teaching FP, a very effective way to build intuition for functional programming languages is to demonstrate evaluation step by step. I frequently wrote out evaluations on paper for student during the COMS10016 labs. I would also ask students to complete sections themselves. Others have also found that encouraging stepwise evaluation on paper is an effective way to get 'a feeling for what a program does'. [4] [TODO: theres more things to cite here]

Thus, a tool to perform these step by step evaluations in an interactive manner would be very valuable. The tool should have an interface that allows progress to be made step by step, showing the history of past steps as well as giving information about the step about to be taken. This would allow students to understand and interact with a stepwise evaluation, without anyone having to undertake the long process of writing it out, and without risk of incorrectness. Furthermore, the effects of changing the input program could be seen quickly, providing instant feedback.

Use as a Lecture Tool

The tool should be suitable for use in lectures. It should provide an interface that facilitates quality explanation of functional programming languages. The interface must be understandable, for both FP 'experts' (lecturers, advanced users) as well as people who have never seen a functional language before. The tool must also be portable, and not require a complex installation process.

Use as a Self Teaching Tool

The tool should be 'self-explanatory' enough for people to use it on their own without expert help. It should be fairly intuitive, and should have all the information required to use it presented to users. The tool must also be portable, and not require a complex installation process. The less complex this tool is to use, the more people will use it.

Demonstrating FP languages

The tool must contain, at its core, a functional language in order to demonstrate how they work. A language that is similar to Haskell would make for easier evaluation of the project, as this would match the language taught in COMS10016, and therefore more people at the University of Bristol will be able to engage with the language. The most Haskell-like programming language that exists (as far as I am aware) is Haskell.

Haskell could be included in the system/required to be installed on the host machine, however creating a demonstration tool around Haskell would be difficult due to the sheer size of the language, the number of features, and the complexity of the type system. It would be better to create Haskell-like language with a strictly limited size and maximum clarity, and include this in the system. The programming language should be designed with simplicity and clarity at its heart.

3.1.3 SFL Explorer

The requirements extracted from autoethnographic methods, as well as from my initial supervisor meeting, came together to form the idea for SFL Explorer.

The system should be a website to maximize portability. The system should include a functional programming language, as well as some sort of UI that allows a functional program in the language to be entered, and evaluated step by step in a visual manner.

The Simple Functional Language

The language was given a name reflecting its core design principle: Simplicity. More precisely, the programming language should be designed with the following design principles in mind:

1. **It should be simple and easy to understand**. This requires that the language should not have features that users might find difficult to understand why they work. This means that the language should have very few inbuilt functions, all of which should be easy to understand why they work.
2. **It should be similar to existing functional languages**. This would allow users to be able to transfer their intuition to other languages. It should be similar in syntax (it should have similar tokens and structures), as well as semantics (it should work similarly).
3. **It should be powerful enough to explain key concepts.**

The features that should be selected for the language are the features that maximize these goals for the minimum implementation complexity. Out of our design goals, 2 and 3 have the potential to be in conflict, as more expressive power often requires more complex syntax. We must ensure a sensible compromise between all of our goals, while accounting for implementation complexity. When adding features for the language, we must prioritize the features that allow explanation of the 'core' features of functional languages, and de-prioritise features that are not so 'core' to the understanding of functional languages.

The Explorer

The website (the explorer) should include a code editor for people to enter programs. Including the functionality for the language inside the website rather than requiring complex client/server communication would simplify the system, as well as improving responsiveness.

[TODO: Finish this bit, summarize requirements]

3.2 Language Design

In this section, I will discuss the design of SFL with respect to the requirements. This is iteration 1 of the design, and it was the proof of concept.

3.2.1 Definitions

Definition 1 (Lowercase and Uppercase ID syntax as regular expressions)
(Lowercase Identifier): id ::= $[a..z][a..zA..Z0..9\_]*$
(Operator): op ::= $+ \mid - \mid \times \mid / \mid > \mid \geq \mid < \mid \leq \mid == \mid ! =$
(Uppercase Identifier): Id ::= $[A..Z][a..zA..Z0..9\_]*$

[Sam: im so sorry but this looks ugly. I think it just needs more space, it is very crampt, this is a minor thing and can wait till the very end]

3.2.2 Basic Syntax

Lambda calculus is the basis of modern functional programming languages. As discussed in the background, lambda calculus consists of 3 structures: identifiers, application, and abstraction. [Sam: sadly this effectly says in prose the same as the BG, the only thing the BG adds is the formal definition, which not everyone will be able to read] One common extra structure that functional languages implement is an assignment. This is where we label an identifier with a certain meaning, such that all references to the assignment henceforth are identical to a reference to the meaning assigned. For instance:

```
1 f = (\x.x)
2 main = f y
```

Is identical to [Sam: dont do this, cos it gives latex the chance to mess up your formatting. Always give your listings a name and use the name in a full sentence e.g. "For instance, listing 1 and listing 2 are semantically equivalent"]

```
1 main = (\x.x) y
```

Note the use of `"\"` instead of λ as it is the closest character available on most keyboards. A program is then defined as a set of assignments, and we pick one specific label name to mark the 'entry-point' expression in the program. Haskell, as well as many other languages, uses 'main' to represent a programs' entry point, so we may use main.

We must [Sam: why must? (because we want to run our code), this is also a common extension of the STLC] also add a way to represent values, such as integers and booleans, to our language. Most programming languages, including functional ones, at least support integers. Booleans are also often supported to represent the results of integer comparison. Without literal values, programs would have to use complicated encodings (such as church numerals) to represent these values, making programs look more complicated. [Sam: bring this why forward] [Sam: this is a very typical para for you: i encourage you to structure your paras as because of x we need y, instead of we need x because of y] [Sam: no new line] These two features massively shorten and simplify programming in this language.

[Sam: this is not referenced anywhere. Above you say we need x and then just dump x here. Instead motivate, introduce and explain. What additions have been made to execute x. Explain your definitions like you would explain your code]

Definition 2 (The basic syntax of SFL)
(Expression) $E, F ::= [-][0, 1, ..] \mid E\ op\ F \mid true \mid false \mid id \mid \backslash id.E \mid E\ F$
(Assignment) $A ::= id = E$
(Module) $M ::= A\ M \mid End$

3.2.3 Reduction and Progress

As discussed in the background, functional programs progress via reduction. SFL programs can reduce when we have an abstraction applied to a term.

We may also want to replace variables with their assigned values. This is not reduction, however it is still progress

[TODO: FINISH, will be easier when the bg on reduction is improved]

3.3 Implementation

3.3.1 The Abstract Syntax Tree

The tree structure of SFL requires the following different types of tree nodes:

```
struct AST {
    vec: Vec<ASTNode>,
    root: usize,
}

enum ASTNodeType {
    Identifier,
    Literal,
    Pair,
    Application,
    Assignment,
    Abstraction,
    Module,
}

struct ASTNodeSyntaxInfo {...}

struct ASTNode {
    t: ASTNodeType,
    token: Option<Token>,
    children: Vec<usize>,
    line: usize,
    col: usize,
    type_assignment: Option<Type>,
    additional_syntax_information: ASTNodeSyntaxInfo
}
```

Figure 3.2: The Rust code listing for the definition of the AST, with lifetime specifiers, accessibility modifiers, and the syntax information (see 3.3.2) removed for conciseness.

- Identifier
- Literal
- Pair
- Application
- Abstraction
- Match
- Assignment
- Module

[Sam: bullets look silly, if you have spare time draw a pictures, otherwise just say that it is a tree structure, listing the nodes adds nothing]

Initially, the approach taken when implementing this tree structure was to have each node 'owning' its child nodes (see 2.3). However, it will be frequently necessary to be able to find nodes based on certain conditions (for example, the condition that this node is a valid redex) and then provide a value that represents the location of this node within the tree. Even if each of the tree nodes had a unique ID, locating a node from this value representing its location will require some sort of tree search.

Rather than this solution, which would have a non-constant node lookup time, a secondary structure can be used to store the tree nodes with constant time lookup, and then each node can store a value enabling constant time lookup of its children within this structure. In the implementation, these types were labelled as Abstract Syntax Tree (AST) and ASTNode, where AST was an array of ASTNodes, and each ASTNode stored their children's indices in this array. The position in the array of an ASTNode will be referred to as its index.

See 3.2 for the code listing for the AST definition. In this implementation, `Vec` was used for the array, as it is growable, resizeable, and facilitates constant-time lookup of its elements. The AST stores and

owns all of the nodes, as well as storing the index of the root node rather than requiring it to be at a specific index.

The node indices in the `children` vector represent different things depending on what kind of node it is.

- If it is an abstraction, the first node represents the variable (or pair of variables) abstracted over, and the second node represents the expression.
- If it is an application, the first node is the function, and the second is the argument.
- If it is a pair, the first node is the first in the pair, and the second is the second in the pair.
- If it is a match expression, the first node represents the matched value, then after this it consists of the case followed by the resulting expression. Match expressions will always therefore have an odd number of children.
- If it is a module, then each of the children is an assignment.
- If it is an assignment, then the first child is the variable being assigned to, and the second is the expression.

`Literal` and `Identifier` nodes store the tokens that defined them, so the strings can be accessed. `Identifier` nodes used as abstraction arguments. These types can either be specified in the source program, or inferred later. Nodes also store their positions (line and column) in the source program, which can be used for error messages.

In order to effectively explain the structure of a parsed program going forwards, the following structure will be used to give a written representation of an AST:

- Nodes are represented as one line each, where, with the name of the node type, followed by its value for `Literal`s and `Identifier`s.
- The children of a node are all of the nodes with an indentation level one deeper than the node in question listed directly below it, until a shallower or equal depth node is listed.

For instance,

```
main = (\x.1) 2
```

would be represented as:

```
Module:
  Assignment:
    Identifier: main
    Application:
      Abstraction:
        Identifier: x
        Literal: 1
      Literal: 2
```

With the Benefit of Hindsight

[Sam: very nice evaluation, love this section] This project was my first major project using Rust. Below is a discussion of some Rust features which were not fully taken advantage of in this definition of syntax trees, followed by a discussion of the combination of these features that would have been more optimal.

Tagged Unions An alternative implementation could have involved `ASTNodeType` being a tagged union, with different node types being associated with different children and data items. For instance, application could be represented by `Application(f: usize, x: usize)`, and identifiers could be `Identifier(String)`. This would be more space efficient, as each node requires different data. It would also more elegantly represent the fact that each type of node is a different thing, and de-obfuscate the meaning of each of the different fields of a node.

```
struct AST<'a> {
    vec: Vec<ASTNode<'a>>,
    root: &'a ASTNode<'a>,
}

enum ASTNodeType<'a> {
    Identifier{name: String},
    Literal{value: String, _type: PrimitiveType},
    Pair{first: &'a ASTNode<'a>, second: &'a ASTNode<'a>},
    Assignment{to: String, expr: &'a ASTNode<'a>, type_assign: Type},
    Abstraction{var: String, expr: &'a ASTNode<'a>, type_assign: Type},
    Module{assigns: Vec<&'a ASTNode<'a>>},
    Match{expr: &'a ASTNode<'a>, cases: Vec<&'a ASTNode<'a>>}
}

struct ASTNodeSyntaxInfo { ... }

struct ASTNode<'a> {
    t: ASTNodeType<'a>,
    info: ASTNodeSyntaxInfo
}
```

Figure 3.3: An alternative implementation with a few advantages over the actual implementation.

References This definition of the AST [Sam: i dont understand the rest of the sentence] and the nodes has a parent object owning all of the nodes. As previously discussed, this was done to enable constant-time lookup of nodes from their indices. However, all things in a program already have such a reference enabling constant time lookup: a pointer, represented in rust by a reference. This was not used, as there were concerns about ensuring validity of each reference, and avoiding use-after-free bugs. These concerns were unfounded, as one of Rust's major features is that it provides safety guarantees ensuring that these problems are never encountered [12]. An object can only store a reference to another object if it can be guaranteed that it exists, and it will continue to exist for at least as long as the object storing the reference will. This is achieved via lifetime checking, using either inferred or explicitly stated specifiers of how long the two objects will exist relative to each other.

A Better Implementation [Sam: numbers alone look odd, especially at the beginning of a sentence. An easy solution is to always give references titles e.g. Figure bla or Table bla]3.3 shows an implementation that uses tagged unions to store information that is different for different node types, and pointers to the nodes directly rather than list indices. This avoids the possibility of referencing nodes that don't exist. It is also easier to understand what is common between nodes (syntax info) and what is uncommon. It is also more space efficient as it only stores the information that each type requires. The size of the improved implementation is 88 bytes, and the size of the original implementation is 128 bytes. The improved implementation is subjectively more elegant and readable. Objectively, it also takes up less space. It also forces memory safety, without the need for carefully implemented getter and setter functions.

Despite this, the decision was made not to update the implementation for several reasons. The AST is so central to the implementation, that it would take a long time to switch properly. Memory and speed are not major constraints for this project, but implementation time is. Furthermore, as long as all indices used are either produced by a helper function, or the AST root, there should not be a problem with memory safety.

3.3.2 Methods on the AST

Below are a selection of the more important or interesting methods implemented on the AST and its nodes. [Sam: i think I want to see the code as well]

Adding new nodes We will frequently want to add new nodes to the tree. Where they are inserted is not important, so the tree will add them to the end, and return their index. These methods are needed

extensively for the parser.

Getting children of nodes As the interpretation of the `children` array for each node changes depending on what type of node it is, a series of getters are implemented, such as '`get_func`' to get the function of an application. These methods are needed extensively for the type checker, and the redex finding system.

Substitute variable Substitutes all instances of a variable in an expression with a given expression. This is needed for applying abstractions. For instance, the process of reducing `(\x.x) 1`, is:

- Get the name of the variable abstracted over: `x`.
- Replace all instances of x in the abstraction expression with the right hand side of the application: `1`.
- Replace all references to the abstraction with references to the abstractions expression.

Note that this orphans the node for the abstraction, and the node for the abstraction variable `x`. This is hard to rectify as deleting any nodes will shift the whole list, which would invalidate indies of nodes, which will break many of the references. This is rectified by cloning the AST, as described below.

Clone The AST, or just a subsection of the AST from a given node, can be cloned by starting from the desired new root, and cloning each nodes children recursively. The new indices of each node may not be the same, as they may be moved in the list, but they will all be in the same place relative to each other. This also removes orphaned nodes, as they will never be cloned as they have no parents.

To String Programs can also be effectively transformed back into strings. This requires a few other pieces of information to be associated with some tree nodes, to make the output program as similar to the input program as possible. The more similar the output is to the input, the easier it is to understand. Some examples include:

- Whether the application was generated by using the right associative `$` operator in order to avoid parenthesis, for instance `id $ 1 + 1`.
- Whether the assignment, where the expression is an abstraction, was generated using the syntax `x = \a.e` or the syntax `x a = e`.

We must also take into account whether a binary infix operator was used to generate a function call, and if so we must place it in the middle of its arguments.

3.3.3 Finding Redexes

[TODO: do once the background is fixed] In SFL, a redex is an application [Sam: unfinished?]

3.3.4 The Parser

The parser needs to consume a program, and return the following things:

- The AST.
- The 'Label Table': The types of all labels defined, including both those defined explicitly (assignments) or implicitly (constructors). This is implemented as a struct '`LabelTable`' which is a wrapper around a `HashMap<String, Type>` with some useful methods.
- The 'Type Table': All type constructors and concrete types defined, stored with their arities. This is implemented as: `HashMap<String, usize>`.

For instance, from the program: [Sam: examples are good!]

```
data List a = Cons a (List a) | Nil
double x = x * 2
main :: List Int
main = Cons (double 1) (Cons (double 2) Nil)
```

We should extract the following data:

- The AST:

```
Module:
  Assignent
    Identifier: double
    Abstraction
      Identifer: x
      App
        App
          Identifier: +
          Identifier: x
        Literal: 2
```

- All the known type assignments (excluding inbuilts)
  - Cons: $\forall a.a \Rightarrow List\ a \Rightarrow List\ a$
  - Nil: $\forall a.List\ a$
  - main: $\forall a.List\ a$
- The names of all known types (excluding inbuilts)
  - List, with an arity of 1

The parser will also store a set of all bound variables at each location. This will allow us to disqualify some invalid programs while generating the tree, rather than having to traverse it after generation to catch these issues. For instance, we must the following assignments:

- `x = (\x. e)` where e is a valid expression, as x is ambiguous during the expression e. This would be disqualified when attempting to parse the abstraction as x is already bound.
- `x = y` where y is undefined.

Lexical Analysis

Lexical analysis is the process splitting a program into its constituent tokens (Lexemes). For instance, the program `main = (\x.x) 1` is the following stream of tokens:

$$[Id : main, Assignment, LeftParen, Backslash, Id : x, Dot, Id : x, RightParen, Literal : 1]$$

See E for the code listing of the definition of the tokens output by the lexical analysis.

The lexer loads the entire string into memory at once. This is not typical, as this can lead to problems with large files. The approach discussed in [1] relies on a system of two buffers only holding individual pages of the file from disk. However, this system will not be loading files from disk; the program string is already in memory as it comes from the UI. Therefore, there would be no benefit to a more traditional lexer optimised to reduce memory usage.

The lexer provides a `next_token` function that returns the next token, and advances the pointer to the start of the token after. The lexer keeps track of line and column information, which is stored in the token to then be stored in the AST.

Expression Parsing

Expressions are parsed using recursive descent parsing. Some of the techniques used for this part of the parser were inspired by the discussion of top down parsing in [1].

At the top level, the expression parsing method is `parse_expression`. A variable `left` stores what is currently the index of the expression. It is called `left` as if we encounter a token that denotes that `left` is applied to whatever comes next, it becomes the left hand side of the application. `left` is originally set to be the output of parsing a primary (see 3.3.4), and then progresses differently based on the next token. Below are some of the ways that `parse_expression` could proceed.

- If the next token is an open bracket, we consume the token and then parse an expression. We then expect a closing bracket. We set `left` to the application of `left` to the expression

- If the next token is a dollar sign, we consume the token and then parse an expression. We do not expect a closing bracket, and we error if we receive one. We set `left` to the application of `left` to the expression.
- If the next token is a token denoting the start of a primary expression structure:
  - A backslash, indicating the start of a lambda
  - An identifier, indicating a variable.
  - A literal

  We parse a primary, and set `left` to the application of `left` to our primary.
- If the next token is:
  - A closing bracket
  - EOF
  - A newline
  - An opening brace (indicating the end of parsing the matched expression of a match statement)
  - A double colon, indicating a type assignment follows

  We return `left`.

Primary Parsing A primary is a less complex structure than an expression. In this system, a primary is any expression structure other than applications. The primaries are:

- Literals
- Identifiers
- Lambdas
- Expressions in brackets

Each of these have their own specific parsing algorithms, which may include calling `parse_expression`.

Literal and Identifier Parsing Literals and identifiers are turned trivially into their respective AST Nodes. For instance, the token:

```
Token {
    line: 0,
    col: 0,
    tt: TokenType::IntLiteral
    value: "2"
}
```

Is turned into this ASTNode:

```
ASTNode {
    t: ASTNodeType::Literal,
    token: Some({the token}),
    children: [],
    line: 0,
    col: 0,
    type_assignment: Option<Primitve::Int>,
    additional_syntax_information: ...
}
```

Parsing Abstractions Abstractions (in the simple case) are parsed by:

- Consuming a lambda (represented by '\' for ease of typing on standard keyboards)
- Parsing a variable. This variable must be added to our set of 'bound' variables.
- Consuming the dot separator'.'
- Parsing an expression
- Constructing an abstraction node from the variable and the expression

However, the definition of abstractions has a few complicating elements of syntax sugar.

Abstractions May be Assignments The assignment `f x = x` is implicitly `f = \x. x`. This is solved by parsing an argument to `parse_abstraction` representing whether this is an assignment. If it is an assignment, we do not parse the lambda, and expect the assignment operator '=' as our separator rather than the dot. As previously mentioned in3.3.2, in order to output the string in a format that is as close as possible to the input, we set a flag in the `ASTSyntaxInfo`: `assign_abst_syntax` to all abstraction nodes defined like this.

Abstractions May Have Multiple Variables The abstraction `\x y. x` is syntax sugar for `\x. (\y. x)`. Additionally, with the assignment syntax, `f x y = x` is syntax sugar for `f = \x. (\y. x)`. This can be accounted for by continually parsing variables until we encounter '.' or the assignment operator '=', and then producing a series of abstractions over these variables in order.

To parse an identifier, we must also check that the identifier is bound at this location.

3.3.5 Web UI MVP

As part of this section, I also developed the MVP for the Web UI. 3.4 shows the web UI after this stage of the project.

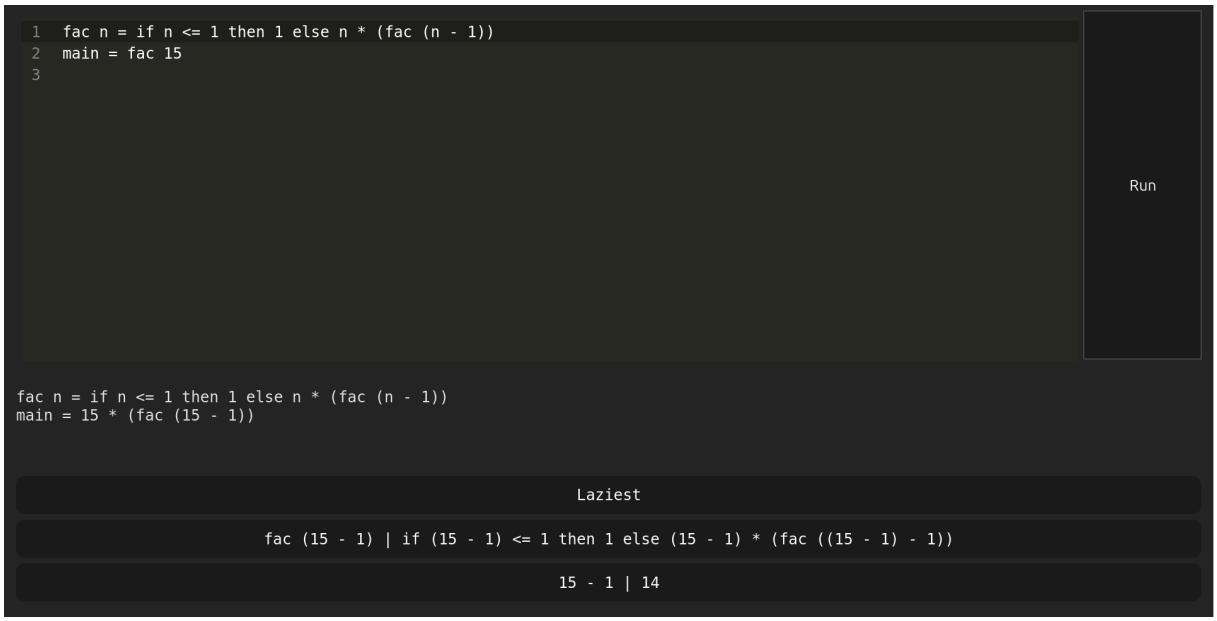

Figure 3.4: The Web UI MVP, as presented to my client at the end of phase 1.

Until this point, development was done in one rust package. This package would be compiled to a binary and run natively, with a basic CLI. This needed to be changed so that it can compile to Web ASseMbly (WASM) and run in the web browser. As I wanted to keep the CLI for debugging, as well as for use later, I did not want to change the whole project to a project with a WASM interface. A solution to keeping both interfaces was to separate the functionality that would be common to the CLI as well as the WASM library into a separate library, and then have the two interfaces as separate packages that depended on this one. The structure of the project became the following 4 packages.

- **libsfl**: All of the language functionality, as this is common to both interfaces
- **sflcli**: All of the original CLI functionality without the language functionality.
- **libsfl_wasm**: a package set up for use with wasm-pack (2.5). It provides a wrapper around **libsfl**, with wrapper functions returning data structures supported by wasm-bindgen**??**. wasm-pack would compile this to an Node Package Manager (NPM) package containing:
  - The WASM binary blob of the compiled rust code
  - A JavaScript file that would load the blob into the browsers' memory, and provides methods that can call the appropriate the methods in the blob
  - A TypeScript file providing the types of all of the packages exported functions.
- The Vite+React frontend (see 2.4) that requires the package that results from compiling **libsfl_wasm**

The WASM library provided functions that could be called from the JavaScript module. Rather than passing the AST around between the WASM library and the JavaScript module, the AST was stored in

[TODO: original approach was to use a set memory region for the AST as i did not wanna pass it to javascript]

[TODO: More stuff about WASM bindings??]

3.4 Proof of Concept Client Meeting: Evaluation and Next Steps

At the end of the phase, I presented the proof of concept project to my client, who was very positive about the project and its potential. The discussion was informal, a friendly conversation rather than a structured interview, to allow the direction of questioning to change depending on the clients answers. The meeting started with me giving my client a demo of the proof of concept using by using the system to evaluate the following program:

```
fac n = if n <= 1 then 1 else n * (fac (n - 1))
main = fac 5
```

Below is a summary of my clients thoughts about various aspects of the proof of concept system and potential future iterations

3.4.1 Usefulness as a Teaching Tool

Below are some notes on what the client thought about the effectiveness of the project as a teaching tool, and how it could be improved in future iterations.

- The project is already very useful as a teaching tool to demonstrate:
  - Evaluation order, and the importance of laziness
  - Currying
  - Recursion
  - The λ-calculus
- On top of this my client wanted to be able to use the system to demonstrate:
  - High Priority
    - List and common list functions such as 'map' and 'fold'. These do not have to by polymorphic, they could be just defined over *Int*s or some other type. These also do not need to be user-definable, they can be built in.
    - Pattern Matching
  - Lower Priority
    - User definable data types, preferably polymorphic. This would mean we could define '*List*' as part of the language which would be good for clarity.

3.4.2 The Existing Language

Positives:

- The language looked similar to Haskell. Particularly, the `if _ then _ else` syntax, and the function assignment shorthand syntax (`fac n = ...` rather than `fac = \n. ...`, even though these are identical)
- The language is minimal and clear
- The factorial function was quite elegant, and it would be understandable to people who did not know Haskell.

Negatives: My client had no specific complaints about the language as it currently stands, however we agreed was lacking many important features. The most difficult things to teach are concepts involving more complex data types.

Requested Features: Below are the specific features my client asked for in order for the system to be able to demonstrate the things she wants to use the system to demonstrate:

- Recursive Types
- Polymorphism
- Type Aliases
- Typechecking
- User Definable Data Types

3.4.3 The Existing UI/UX

Positives:

- The editor, as it feels like a very popular editor: VSCode

Negatives

- It is unstable. This is bad in a teaching tool, as it would waste a lot of time if it constantly broke in the lecture.
- 'laziest' as an option is confusing, as it was unclear if it was referring to one of the other on screen options, or if it was referencing a 'hidden' option
- The vertical bar separating redex from contraction on the progress buttons was not obvious enough. On top of this, the bar was not centred, so it was hard to look through all the redexes at once as they were not aligned with each other.

Requested Features

- Syntax highlighting, to make the language easier to read
- A history of what the expression has been is vital to demonstrate step by step evaluation. I identified this as an important feature at the beginning of the phase (see 3.1.2), but I had not finished it by the client meeting. It was implemented in the next phase (see 4.3.6).
- Sample Programs

3.4.4 Conclusion

At the end of this phase, and going into phase 2, I had a strong proof of concept system and an idea for how the system will look. The meeting with my client yielded many ideas, all of which I sucessfully implemented throughout this project.

Chapter 4

Phase 2 — Types and Pattern Matching

In this phase, I moved away from the autoethnographic (3.1.1) approach, where most of my requirements came from within, to an externally motivated client-led approach.

At the end of this phase, I held a focus group (4.4) to help me evaluate the progress of the project. Because this was the plan from the beginning of the phase,

4.1 Requirements Analysis

The requirements for this phase were motivated by my client meeting (3.4). I wanted to tackle the most technical aspects in this phase to give me the maximum time to complete them, as this was still early in the project lifecycle. The most difficult features out of the client's requests were the ones to do with extending the language, so these were the main focus for this phase.

The client's central idea for what they wanted to use the tool was to demonstrate methods on lists, such as 'map' and 'foldr/l'. This requires lists to be built into the language. Lists in functional programming languages are commonly defined recursively, using `Cons x xs` to represent constructing a list from an element `x` and the rest of the list `xs`. `Nil` represents an empty list. This recursive construction of lists comes from Lisp [14]. Similarly, in Haskell, lists are defined as `data [a] = [] | a : [a]`

[TODO: finish yapping about lists and talk about why that means we need ADTs]

This definition of lists is as example of a polymorphic data type. It also implicitly defines two polymorphic constructors, '[]' also known as Nil which has type $\forall a.[a]$, and ':' also known as Cons which has a type $\forall a.a \to [a] \to [a]$.

4.2 Design

4.2.1 Language Changes

The focus of this project phase is mainly to upgrade the language SFL. We have already identified what features we would like to add the language. This section will go into detail about the design for the extension for the language enabling these new features.

Type System

If we are to effectively represent the type of expression containing integers and booleans, we must have types *Int* and *Bool*. We also want our type system to be able to express functions, as our language support functions.

We also want polymorphism in our type system, as rewriting functions many times for different data types makes programs more verbose.

Allowing for algebraic user defined data types similarly to Haskell would make the language much more expressive and much more powerful, as well as bringing it closer to Haskell. Supporting tagged unions and tuples in the SFL type system would massively increase the ease of writing complex programs. It would also allow for complex data structures such as trees and lists.

$$\text{Types} \quad A, B, C \quad ::= \quad \text{Int} \mid \text{Bool} \mid \alpha \mid \forall \alpha.\, A \mid A \to B \mid (A, B) \mid \text{Name}[A_1, \ldots, A_n]$$

Figure 4.1: The SFL type system

Type names, as well as constructor names, start with uppercase letters in Haskell. This allows them to be easily differentiated from type variables, as well as regular variables.

First-order polymorphic type constructors would be useful to have in SFL, with one example of their utility being defining the polymorphic function '`length :: List a -> Int`' which should work regardless of what type the list is over.

Figure 4.5 shows the type system in SFL. Note that the definition of type constructors here is more permissive than is correct, as it does not enforce that we apply our type c

User Definable Algebraic Data Types

In Haskell, we can create algebraic types using the `data` keyword (see 2.2.3). Replicating this syntax for SFL's user defined data types would be good, allowing people already familiar with Haskell to use the system, as well as viva versa.

As an example, the SFL (and Haskell) data declaration:

```
data Either a b = Left a | Right b
```

creates a tagged union type called *Either* with two constituent type parameters a and b. In our type system (4.2.1) this would be represented as $Either[a, b]$. The Name`Either` uniquely identifies this type, this must be enforced by the parser. It also creates two data constructors: `Left` which has the type $\forall a\ b.a \to Either[a, b]$, and `Right` which has the type $\forall a\ b.b \to Either[a, b]$.

Type aliases allow us to make code more readable and expressive. For instance, if we were to define playing cards like this:

```
data Suit = Hearts | Clubs | Spades | Diamonds
data Rank = Num Int | Jack | Queen | King | Ace
type Card = (Suit, Rank)
```

having the type alias `Card` for `(Suit, Rank)` allows us to very easily, and more readably, create functions on Cards, as well as values with that type.

To summarize, we will implement type aliases and algebraic data types to work similarly to Haskell with similar syntax.

Match

See 2.2.5 for more information about Haskell pattern matching. A basic example of pattern matching in Haskell:

```
1 fac :: Int -> Int
2 fac 0 = 1
3 fac n = n * factorial (n - 1)
```

Here, the definition of the 'fac' function is different depending on if is it applied to 0 or to any other *Int*. If it is applied to an *Int* other than 0, n is substituted for this value in the expression.

Pattern matching at the top level like this would be difficult to implement, as it would require significantly changing how abstractions are represented. It would be easier to create a new syntax structure: a match expression. This could look like:

```
1 fac :: Int -> Int
2 fac n = match n {
3     | 0 -> 1
4     | _ -> n * (fac (n - 1))
5 }
```

This syntax was fairly arbitrary, as syntax is quite easy to change. However, this syntax proved to be fairly popular with all three focus groups, so it did not change between this stage and the end of the project.

The 'fac' function takes an *Int* n, and proceeds differently with different values of n. If the value is 0, the value of the whole expression becomes 0, otherwise it becomes `n * (fac (n - 1))`. We can use literals in our pattern to differentiate between different values of literals. Inspired by Haskell, we can use a variable (which is a lowercase identifier) to match anything, a 'wildcard' pattern. All instances of the variable in the pattern's corresponding expression with the term that the variable matches. '_' is a special case wildcard, where no variable is bound, but it still matches anything,

We should also be able to match more complex structures including Algebraic Data Types. For instance, we can write the following function to figure out whether a list has length 2 or grater

```
1 lenIsAtLeastTwo :: List a -> Bool
2 lenIsAtLeastTwo list = match list {
3     | Cons _ (Cons _ _) -> true
4     | _ -> false
5 }
```

In this example, it is important that we evaluate the term 'list' enough to *know for sure* that it does not match the first pattern before moving on to the second, as the second pattern is irrefutable.

4.2.2 Next UI Iteration

At this phase of the project, the current version of the web UI is a proof of concept. See 3.4 for the current state. The UI requires a total redesign

I completely redesigned the UI based on the clients' feedback, as well as based on other requiremenets identified during the autoethnographic phase of the project. See 4.2 and 4.3 for screenshots of the new design. These designs were done using Figma.

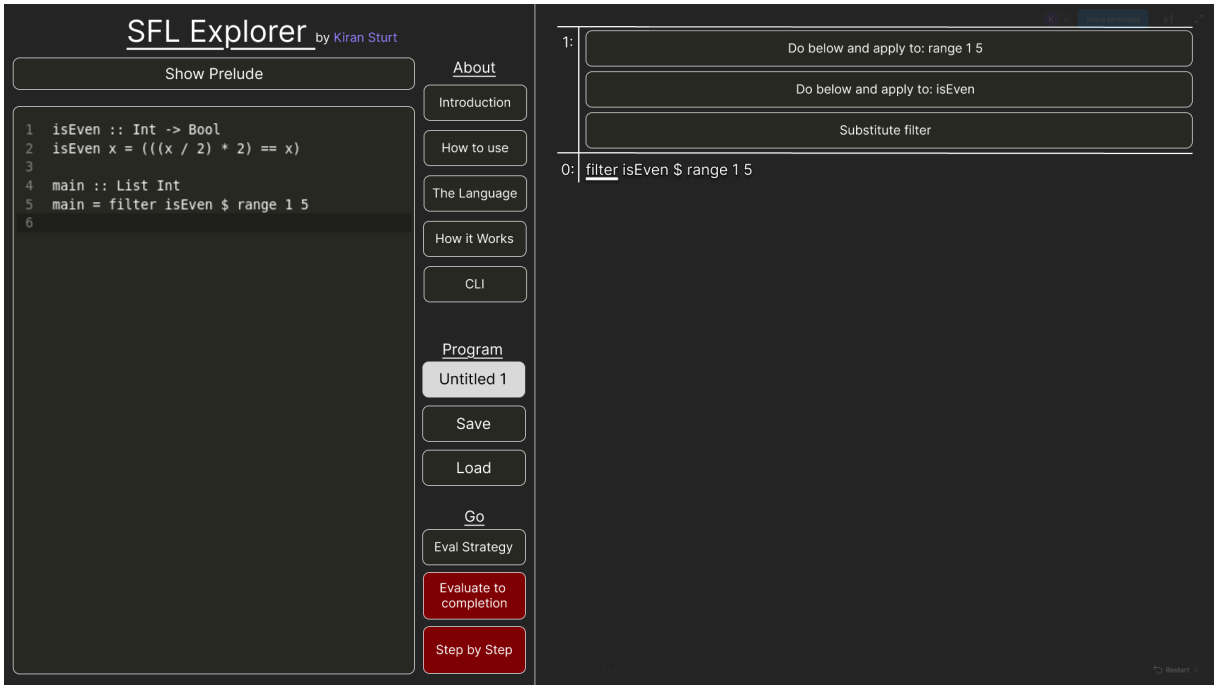

Figure 4.2: Screenshot 1 of the Figma design of the web UI

This design was meant to be a work in progress, but it looks quite similar to the final release of the product (Screenshots F.4, F.5, F.6 and F.7). Before implementing this design, I discussed this design with the Advanced Focus Group (see 4.4) and they were much more positive about this UI than the existing one [TODO: Discuss revert] [TODO: Design principle: simplicity, speed, minimalism, feeling like vscode.]

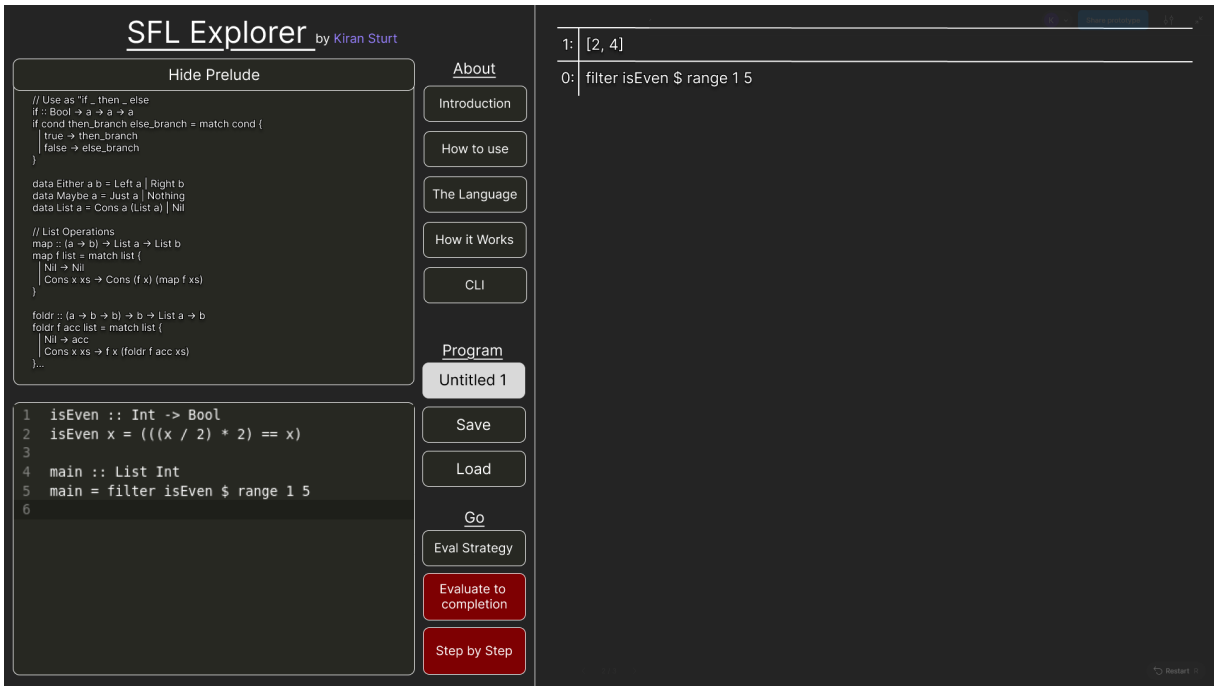

Figure 4.3: Screenshot 2 of the Figma design of the web UI. This version shows the prelude extended

4.3 Implementation

4.3.1 Parser Changes

We must make some changes to the parser to include these new features.

Parsing Match Statements

An example of using a match statement follows:

```
lengthIsAtLeast2 list = match list {
  | Cons x (Cons y xs) -> true
  | _ => false
}
```

The algorithm used for parsing match statements is:

- Consume the 'match' keyword.
- Parse the expression matched over
- Consume an open brace
- While the next token isn't a close brace:
  - Parse a pattern (4.3.1).
  - Consume a right arrow
  - Parse an expression
- Consume a close brace

Following this, a match node is created, where the `children` vector is set appropriately with the pattern and expressions.

Patterns A pattern must be a value **??**; a pattern must not contain anything that can be reduced. It would be nonsensical to have a situation where we had a pattern not in normal form such as `1 + 1` and the expression to be matched was `2`.

To parse a pattern, we may use the same techniques as parsing an expression, with a few differences:

- Disallowing abstractions
- Identifiers must be either
  - Unbound lowercase variables
  - Underscore (`_`) representing a wildcard pattern
  - A bound uppercase variable (a constructor)

4.3.2 Types

Rust allows us to represent our types (see 4.5 for the definition of the type system), quite easily using Enums. Rust's Enums are an example of algebraic data types, and are therefore very useful for defining our own algebraic data type system. See 4.4 for the listing.

```
pub enum Primitive {
    Int64,
    Bool,
}

pub enum Type {
    Unit,
    Primitive(Primitive),
    Function(Box<Type>, Box<Type>),
    TypeVariable(String),
    Forall(String, Box<Type>),
    Product(Box<Type>, Box<Type>),
    Union(String, Vec<Type>),

    Alias(String, Box<Type>),
    Existential(usize),
}
```

Figure 4.4: The Rust code listing for the definition of types. Existential and Alias are separated as they are more of an implementation detail than a part of the type system

We must use `Box<Type>`, which represents a pointer to a heap allocated object, otherwise it would be impossible to calculate the size of `Type`, as it could be infinitely large with it containing another `Type` recursively. `Box<Type>` however has known size: the size of a pointer in the target architecture.

We also define Existential, as an implementation detail needed for the type checker.

Aliases are defined here with their name, and the type they are an alias for. Aliases could simply be implemented by replacing all occurrences of the string on the left with the string on the right, but defining them here allows us to use their names to generate type errors making them easier to understand.

Methods on Types

Below are a selection of the more important or interesting methods implemented on Types.

Substitution of type variables We may wish to set a type variable to another type. For instance, if given the type expression $T\ U$ where T and U are types, and we know that one of the constructors of T is of generic type $\forall a.a \to T\ a$, the type of the constructor for this type should be $U \to T\ U$. We have 'instantiated' the type variable a to be U by substituting a with U throughout the expression, and removing the $\forall a$. This is required for the type checker.

To String We will frequently wish to display types as strings for debugging purposes.

4.3.3 The Type Checker

The type checker will be bidirectional, and will follow and algorithm largely based on the one in [6]. The quote that follows from this paper, describes bidirectional type checking and its merits:

> 'Bidirectional typechecking, in which terms either synthesize a type or are checked against a known type, has become popular for its scalability …its error reporting, and its relative ease of implementation' [6]

It was the 'relative ease of implementation' that attracted me to bidirectional type checking. After running the algorithm by hand to convince myself the algorithm works on checking `id` ($\lambda x.x$) against type $\forall a.a \to a$, I modified their algorithm to add my extra types (the inbuilt types Int, $Bool$, as well as the $\bigcup$ and $\times$ types) and my extra expression syntax structures (literals, match, pairs). This does not include assignment and modules as these are not part of expression syntax. 4.5 shows the type system, including the typechecker implementation details, as well as the unmodified context structure, which keeps track of the state of the typechecker as it progresses recursively through the type system. B shows some example derivations using this algorithm including some of my rules. 4.6 shows the modified algorithm for substituting all of the information in a context into a type.

$$\begin{array}{llll} \text{Types} & A, B, C & ::= & \text{Int} \mid \text{Bool} \mid \alpha \mid \hat{\alpha} \mid \forall \alpha.\, A \mid A \to B \mid (A, B) \mid \text{Name}[A_1, \ldots, A_n] \\ \text{Contexts} & \Gamma, \Delta, \Theta & ::= & \cdot \mid \Gamma, \alpha \mid \Gamma, x : A \mid \Gamma, \hat{\alpha} \mid \Gamma, \hat{\alpha} = \tau \mid \Gamma, \blacktriangleright_{\hat{\alpha}} \end{array}$$

Figure 4.5: Syntax of types, monotypes, and contexts as seen by the typechecker. The definition of types differ slightly from the definition offered in figure 4.1, as we include existential type variables ($\hat{\alpha}$) that can not actually be created by users. They are an implementation detail required for the type checking algorithm

$$\begin{array}{ll} [\Gamma]\text{Int} & = \text{Int} \\ [\Gamma]\text{Bool} & = \text{Bool} \\ [\Gamma]\alpha & = \alpha \\ [\Gamma[\hat{\alpha} = \tau]]\hat{\alpha} & = [\Gamma[\hat{\alpha} = \tau]]\tau \\ [\Gamma[\hat{\alpha}]]\hat{\alpha} & = \hat{\alpha} \\ [\Gamma](A \to B) & = ([\Gamma]A) \to ([\Gamma]B) \\ [\Gamma](\forall \alpha.\, A) & = \forall \alpha.\, [\Gamma]A \\ [\Gamma](A, B) & = ([\Gamma]A, [\Gamma]B) \\ [\Gamma]\text{Name}[A_1, \ldots, A_n] & = \text{Name}[[\Gamma]A_1, \ldots, [\Gamma]A_n] \end{array}$$

Figure 4.6: Applying a context, as a substitution, to a type

The full typechecking algorithm is listed in figures 4.7, 4.8, 4.9. 4.9 shows the main algorithm for checking and synthesizing the types of various expression structures. 4.7 shows the algorithm for how we verify that a type is a subtype of another type. For instance, our typechecking rule `Sub` synthesizes the type, and the uses the algorithmic subtyping rules to check that the synthesized type is a subtype of the expected type.

Most of these rules are untouched, the ones that I added or modified are highlighted.

$\boxed{\Gamma \vdash A <: B \dashv \Delta}$ Under input context Γ, type A is a subtype of B, with output context Δ

$$\frac{}{\Gamma[\alpha] \vdash \alpha <: \alpha \dashv \Gamma[\alpha]}\ <:\mathsf{Var} \qquad \frac{}{\Gamma \vdash \mathrm{Int} <: \mathrm{Int} \dashv \Gamma}\ <:\mathsf{Int} \qquad \frac{}{\Gamma \vdash \mathrm{Bool} <: \mathrm{Bool} \dashv \Gamma}\ <:\mathsf{Bool}$$

$$\frac{}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} <: \hat{\alpha} \dashv \Gamma[\hat{\alpha}]}\ <:\mathsf{Exvar} \qquad \frac{\Gamma \vdash B_1 <: A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 <: [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \to A_2 <: B_1 \to B_2 \dashv \Delta}\ <:\to$$

$$\frac{\Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]A <: B \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Theta}{\Gamma \vdash \forall \alpha.\, A <: B \dashv \Delta}\ <:\forall\mathsf{L} \qquad \frac{\Gamma, \alpha \vdash A <: B \dashv \Delta, \alpha, \Theta}{\Gamma \vdash A <: \forall \alpha.\, B \dashv \Delta}\ <:\forall\mathsf{R}$$

$$\frac{\hat{\alpha} \notin \mathrm{FV}(A) \qquad \Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\leqq A \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} <: A \dashv \Delta}\ <:\mathsf{InstantiateL} \qquad \frac{\hat{\alpha} \notin \mathrm{FV}(A) \qquad \Gamma[\hat{\alpha}] \vdash A \leqq: \hat{\alpha} \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash A <: \hat{\alpha} \dashv \Delta}\ <:\mathsf{InstantiateR}$$

$$\frac{\Gamma \vdash A_1 <: A_2 \dashv \Theta \qquad \Theta \vdash B_1 <: B_2 \dashv \Delta}{\Gamma \vdash (A_1, B_1) <: (A_2, B_2) \dashv \Delta}\ <:\times$$

$$\frac{(\Gamma_i \vdash A_i <: B_i \dashv \Gamma_{i+1}) \text{ for } i \text{ in } [1, 2, \ldots, n]}{\Gamma_1 \vdash \mathrm{Name}[A_1, A_2, \ldots, A_n] <: \mathrm{Name}[B_1, B_2, \ldots, B_n] \dashv \Gamma_{n+1}}\ <:\bigcup$$

Figure 4.7: Algorithmic subtyping. The rules with highlighted names are my additions, the rest are unchanged from [6]

$\boxed{\Gamma \vdash \hat{\alpha} :\leqq A \dashv \Delta}$ Under input context Γ, instantiate $\hat{\alpha}$ such that $\hat{\alpha} <: A$, with output context Δ

$$\frac{\Gamma \vdash \tau}{\Gamma, \hat{\alpha}, \Gamma' \vdash \hat{\alpha} :\leqq \tau \dashv \Gamma, \hat{\alpha} = \tau, \Gamma'}\ \mathsf{InstLSolve} \qquad \frac{}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash \hat{\alpha} :\leqq \hat{\beta} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]}\ \mathsf{InstLReach}$$

$$\frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash A_1 \leqq: \hat{\alpha}_1 \dashv \Theta \qquad \Theta \vdash \hat{\alpha}_2 :\leqq [\Theta]A_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\leqq A_1 \to A_2 \dashv \Delta}\ \mathsf{InstLArr}$$

$$\frac{\Gamma[\hat{\alpha}], \beta \vdash \hat{\alpha} :\leqq B \dashv \Delta, \beta, \Delta'}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} :\leqq \forall \beta.\, B \dashv \Delta}\ \mathsf{InstLAllR}$$

$\boxed{\Gamma \vdash A \leqq: \hat{\alpha} \dashv \Delta}$ Under input context Γ, instantiate $\hat{\alpha}$ such that $A <: \hat{\alpha}$, with output context Δ

$$\frac{\Gamma \vdash \tau}{\Gamma, \hat{\alpha}, \Gamma' \vdash \tau \leqq: \hat{\alpha} \dashv \Gamma, \hat{\alpha} = \tau, \Gamma'}\ \mathsf{InstRSolve} \qquad \frac{}{\Gamma[\hat{\alpha}][\hat{\beta}] \vdash \hat{\beta} \leqq: \hat{\alpha} \dashv \Gamma[\hat{\alpha}][\hat{\beta} = \hat{\alpha}]}\ \mathsf{InstRReach}$$

$$\frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash \hat{\alpha}_1 :\leqq A_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \leqq: \hat{\alpha}_2 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash A_1 \to A_2 \leqq: \hat{\alpha} \dashv \Delta}\ \mathsf{InstRArr}$$

$$\frac{\Gamma[\hat{\alpha}], \blacktriangleright_{\hat{\beta}}, \hat{\beta} \vdash [\hat{\beta}/\beta]B \leqq: \hat{\alpha} \dashv \Delta, \blacktriangleright_{\hat{\beta}}, \Delta'}{\Gamma[\hat{\alpha}] \vdash \forall \beta.\, B \leqq: \hat{\alpha} \dashv \Delta}\ \mathsf{InstRAllL}$$

Figure 4.8: Instantiation. These rules are unmodified from [6]

$\boxed{\Gamma \vdash e \Leftarrow A \dashv \Delta}$ Under input context Γ, e checks against input type A, with output context Δ

$\boxed{\Gamma \vdash e \Rightarrow A \dashv \Delta}$ Under input context Γ, e synthesizes output type A, with output context Δ

$\boxed{\Gamma \vdash A \bullet e \Rrightarrow C \dashv \Delta}$ Under input context Γ, applying a function of type A to e synthesizes type C, with output context Δ

$$\frac{}{\Gamma \vdash \mathit{IntLiteral} \Leftarrow \mathrm{Int} \dashv \Gamma}\ \mathsf{IntLit}\Leftarrow \qquad \frac{}{\Gamma \vdash \mathit{IntLiteral} \Rightarrow \mathrm{Int} \dashv \Gamma}\ \mathsf{IntLit}\Rightarrow$$

$$\frac{}{\Gamma \vdash \mathit{BoolLiteral} \Leftarrow \mathrm{Bool} \dashv \Gamma}\ \mathsf{BoolLit}\Leftarrow \qquad \frac{}{\Gamma \vdash \mathit{BoolLiteral} \Rightarrow \mathrm{Bool} \dashv \Gamma}\ \mathsf{BoolLit}\Rightarrow$$

$$\frac{\Gamma \vdash e_1 \Leftarrow A \dashv \Theta \qquad \Theta \vdash e_2 \Leftarrow B \dashv \Delta}{\Gamma \vdash (e_1, e_2) \Leftarrow (A, B) \dashv \Delta}\ \mathsf{Pair}\Leftarrow \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta \qquad \Theta \vdash e_2 \Rightarrow B \dashv \Delta}{\Gamma \vdash (e_1, e_2) \Rightarrow (A, B) \dashv \Delta}\ \mathsf{Pair}\Rightarrow$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A \dashv \Gamma}\ \mathsf{Var} \qquad \frac{\Gamma \vdash e \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A <: [\Theta]B \dashv \Delta}{\Gamma \vdash e \Leftarrow B \dashv \Delta}\ \mathsf{Sub}$$

$$\frac{\Gamma, \alpha \vdash e \Leftarrow A \dashv \Delta, \alpha, \Theta}{\Gamma \vdash e \Leftarrow \forall \alpha.\, A \dashv \Delta}\ \forall\mathsf{I} \qquad \frac{\Gamma, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]A \bullet e \Rrightarrow C \dashv \Delta}{\Gamma \vdash \forall \alpha.\, A \bullet e \Rrightarrow C \dashv \Delta}\ \forall\mathsf{App} \qquad \frac{\Gamma, x : A \vdash e \Leftarrow B \dashv \Delta, x : A, \Theta}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B \dashv \Delta}\ \to\mathsf{I}$$

$$\frac{\Gamma, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} \vdash e \Leftarrow \hat{\beta} \dashv \Delta, x : \hat{\alpha}, \Theta}{\Gamma \vdash \lambda x.\, e \Rightarrow \hat{\alpha} \to \hat{\beta} \dashv \Delta}\ \to\mathsf{I}\Rightarrow \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \dashv \Theta \qquad \Theta \vdash [\Theta]A \bullet e_2 \Rrightarrow C \dashv \Delta}{\Gamma \vdash e_1\, e_2 \Rightarrow C \dashv \Delta}\ \to\mathsf{E}$$

$$\frac{\Gamma[\hat{\alpha}_2, \hat{\alpha}_1, \hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2] \vdash e \Leftarrow \hat{\alpha}_1 \dashv \Delta}{\Gamma[\hat{\alpha}] \vdash \hat{\alpha} \bullet e \Rrightarrow \hat{\alpha}_2 \dashv \Delta}\ \hat{\alpha}\mathsf{App} \qquad \frac{\Gamma \vdash e \Leftarrow A \dashv \Delta}{\Gamma \vdash A \to C \bullet e \Rrightarrow C \dashv \Delta}\ \to\mathsf{App}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow A \dashv \Theta_1 \qquad (\Theta_i \vdash c_i \Leftarrow A \dashv \Theta_{i+1}) \text{ for } i \text{ in } [1, 2, \ldots, n] \\ \Delta_1 = \Theta_{n+1}, \hat{\alpha} \qquad (\Delta_i \vdash e_i \Leftarrow \hat{\alpha} \dashv \Delta_{i+1}) \text{ for } i \text{ in } [1, 2, \ldots, n]\end{array}}{\Gamma \vdash \mathrm{match\ e}\ \{c_1 \to e_1 \mid c_2 \to e_2 \mid \cdots \mid c_n \to e_n\} \Rightarrow \hat{\alpha} \dashv \Delta_{n+1}}\ \mathsf{Match}\Rightarrow$$

Figure 4.9: Algorithmic typing. The rules with highlighted names are my additions, the rest are unchanged from [6]. Note that the checking rules IntLit⇐, BoolLit⇐, Pair⇐ are not actually necessary, as they could be caught by the Sub rule. They are included as they remove the unnecessary steps that using the Sub rule in this manner creates, speeding up/simplifying the algorithm

4.3.4 Pattern Matching

We must update the redex finding system with the ability to match patterns.

As discussed in the design (4.2.1), patterns are checked in order from first to last. We can check an expression against a pattern recursively, and get all variables bound, using the following algorithm:

```
// We return a list of
enum MatchResult {
    Sucess{bindings: Vec<(String, Term)>},
    Unknown,
    Refute
}

fn match_against_identifier(expr, pattern) -> MatchResult {
    if (pattern is "_") {
        // Suceed but dont bind
        Sucess([])
    }
    if (pattern is a lowercase identifier) {
        // We suceed as a lowercase ID is a wildcard, and we must add to our list
        // of bindings the fact that the named wildcard now has a value, which is the
        // matched term
        Sucess([(pattern.string, expr)])
    }
    if (pattern is a constructor (i.e. is uppercase)) {
        if (expr is also a constructor with the same name) {
            return Sucess([])
        }
        if (expr is an application) {
            // `Head' refers to the recursive front of an application
            if (the head of expr is a constructor) {
                // We can refute, as constructors never evaluate, so the structure of
                // the expression will never be the same as the pattern
                return Refute
            }
        }
    }
}
```

[TODO: split description beteen what type of node pattern is]

We must be careful to distinguish betweens situations where we know we cannot match a pattern

```
1 match list {
2     | Cons _ (Cons _ _) -> true
3     | _ -> false
4 }
```

We must be careful not to move on to matching against a pattern until we have thoguhrougly checked that it does not match any pattern before it. Not only do we need to check that it does not match before moving on, we must check that it *can not* match the expression i.e. we must refute the pattern.

4.3.5 The Prelude, and 'if e then a else b'

Most programming languages come with functionality packaged that is included by default, and is written in the language. In Haskell, this is referred to as the Prelude. There is also the standard library which is more extensive and is not imported by default.

As our language does not need extensive extra functionality, we do not need a whole standard library. However, a basic prelude with common functionality would be useful. C shows the SFL prelude. I

included 'if' in the prelude to show that it is based on a match statement, rather than being a mysterious inbuilt:

```
if :: Bool -> a -> a -> a
if cond then_branch else_branch = match cond {
    | true -> then_branch
    | false -> else_branch
}
```

In order to make the language more like Haskell, I also added syntax sugar that allowed you to use it using the '`if e then a else b`' syntax. The parser would ignore the '`then`' and the '`else`' keywords, and it would be equivalent to '`(((if e) a) b)`' internally. However, this was unpopular with the advanced focus group, who said that this was confusing (see 4.4.4).

4.3.6 Changes to the Proof of Concept UI

In this phase, I made some changes to the proof of concept web UI
- Initial approach to diff - separate lazy and free choice mode

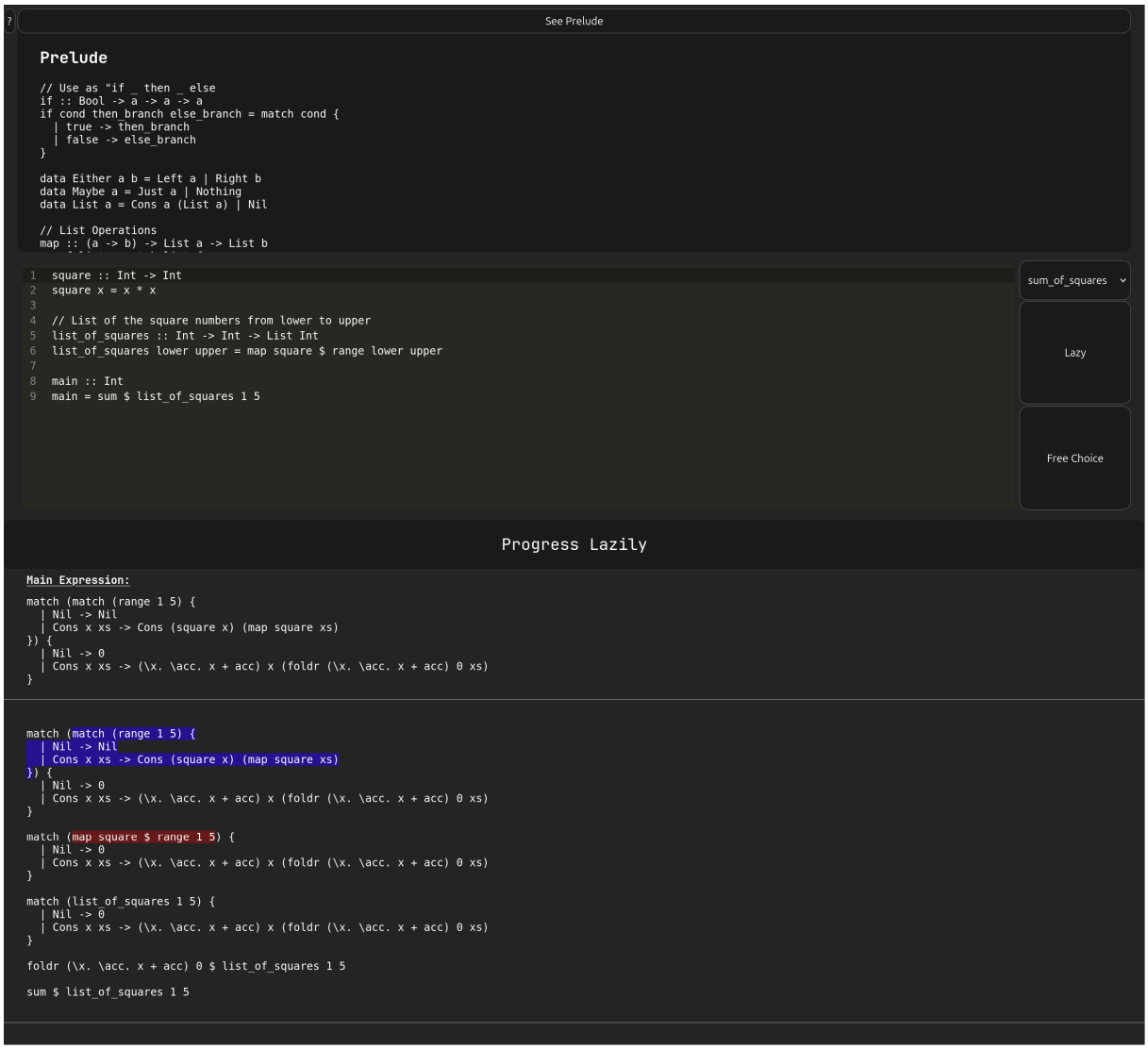

Figure 4.10: The product at the end of phase two during lazy evaluation of the 'sum of squares' sample program, with the prelude dropdown extended

4.4 The Advanced Focus Group: Evaluation and Next Steps

The aims of this phase were to develop the language as well as some other more technical features of this project. To discuss the language, I held a focus group with students who were very knowledgable and interested in functional programming languages.

This was my first of three focus groups, the most advanced of the three. As the UI/UX was not polished at this stage, I wanted to find people who would be able to discuss the parts that I had already implemented to a reasonable level of completion: the language. However, I also wanted to discuss future steps for the system as a whole. Because of this, I wanted to find people who had learned functional languages as a part of a university course fairly recently and within memory, so they would have an insight into what is required for the system to be useful for use in this setting.

The transcript from this focus group is included in the additional submitted materials (see D)

4.4.1 Selection

For this focus group, I recruited four students in their fourth year of studies here at the University of Bristol. They had all taken the first-year FP unit 3 years prior, and they had all taken units specializing in programming language theory since, including:

- The second year Programming Languages and Computation unit COMS20007, where they learnt to (among other things) 'Understand the interplay between the design and implementation of programming languages' [18]
- The third year optional Types and Lambda Calculus unit COMS30040 where they learnt (among other things): [19]
  - 'Type systems: types, judgements and rules'
  - 'Syntax and semantics of an untyped lambda calculus'
- The fourth year optional Advanced Topics in Programming Languages, where the unit outcomes were that they should be able to (among other things): [16]
  - 'Specify the dynamics of program evaluation for a variety of programming constructs'
  - 'Specify static typing rules for a variety of programming constructs'

These people I selected for this focus group were the closest to 'subject experts' that I could find while still being students.

4.4.2 Format

This focus group started with me briefly presenting SFL explorer. 4.10 shows how the system looked at this stage of the project. We also discussed the next UI iteration (see 4.2.2).

4.4.3 Outcomes

Below is the summary of outcomes from the discussion with this focus group. The assertions about what they thought are backed up with quotes, marked with timestamps of where these quotes can be found in the transcript.

4.4.4 The Existing Language

Positives:

- They liked the explicit match statements: 'Stick with the match expressions because it's very clear that matching has happened when you have the word match there' [24:11]

Negatives:

- They were confused about if-then-else syntax. They said it could be confusing to have the parser act differently for one specific function type. 'The issue I was having is just the fact that there is a function in the prelude which has the same name as some syntactic sugar that is a parser construct' [42.55]

Requested Features: Below are the specific features my client asked for in order for the system to be able to demonstrate the things she wants to use the system to demonstrate:

- Recursive Types
- Polymorphism
- Type Aliases
- Typechecking
- User Definable Data Types
- They liked the revert: 'Something I had not thought of, very good' [54:59]
- They wanted syntax highlighting
- They wanted an indication of which direction evaluation was going so I added numbers: 'Because the reduction steps generate bottom-up, it might be good to have some sort of indication about the direction things are going in'. This was already in the new UI, which that had not seen by this point in the transcript.
- They really appreciated its utility for what it was designed for. 'I think this is very good … I wish I'd had this in the functional labs' [1:00:09]
- They liked the horizontal split: ' It's easier to have everything on screen and it's more akin to what people may have experienced' 'Its like compiler explorer'. [52:38] 'I think immediately not having to scroll is a massive plus' [52:58]

Chapter 5

Phase 3 — Improving the UI/UX

The main focus of this phase is to implement the next UI iteration, as well as to improve the language.

5.1 Requirements Analysis

The motivations for this phase come mainly from the advanced focus group, however requirements from the autoethnographic phase of the project, as well as the proof of concept client meeting continue to be relevant.

The advanced focus group was generally very positive about the language, but they had many thought about the Proof of Concept UI they were presented with. During phase 2, I created a Figma prototype for the next UI (see 4.2.2). Many of their thoughts about the Proof of Concept UI were things that were already addressed with the new design. This prototype was presented to the advanced focus group, who much preferred it. The advanced focus group had no criticism of the new UI, so it should be implemented as designed for now.

The prototype for the new UI also included the functionality to 'undo progress', by clicking on a previous program state in the table to make this the current version of the program. The advanced focus group appreciated this functionality.

5.2 Implementation

Implementing the new UI mostly consisted of time-consuming React and CSS tweaks which are not worth mentioning here. However, there were some more challenging aspects that required some more interesting considerations and changes to be made.

Diff

Our frontend requires the ability to see what has changed between two program states. Highlighting these changes make understanding the changes in the users program in the frontend easier. This function generates the strings for the two trees simultaneously, producing the similarities and differences. 5.1 shows a subsection of this algorithm, showing how it works for IDs, Literals and Pairs.

Reduction Messages

Rather than presenting the user with simply the before and after of the reduction, this design calls for presenting the user with a message describing what will happen. While generating the options for reduction (see 3.2.3), we can keep track of information relevant to how it was generated to inform the message displayed. For instance, if a reduction is generated from the application of a named function with name A to two arguments B, C, we can convert those arguments to strings and then broadcast the message 'Applied function A to B and C'.

If B or C are large pieces of syntax, this may generate a very large unintelligible string. To solve this, we can modify our stringification algorithm to do certain things different to normal:

- Do not show the cases of a match statement, as the condition should be enough differentiate it
- We can truncate the output to a fixed length

In past iterations, redex-contraction pairs were passed to the front end as two strings. We can make it three strings instead, where one of the strings is the reduction message which can be displayed before the reduction. The other two strings, the redex and contraction, can be displayed after the reduction in the history.

Revert Progress Functionality

We may wish to undo progress. This was functionality designed into the new UI that the advanced focus group specifically mentioned liking.

Undoing progress requires that previous AST states must be stored. Before now, the most recent AST state was stored at a known memory address so any of the functions in the binary could know where to find it. This was done to avoid having to pass the AST to the JavaScript module. If we wanted to store the history of all ASTs, one approach could be to store all the ASTs in a pre-allocated memory region in a stack, and then allow the JavaScript module to refer to each of the ASTs in the history by their stack index. However, pre-allocating enough memory for any potential program execution logs would be misguided, as it would cause accessibility problems for computers with less memory. Instead, we should employ dynamic allocation.

The issue with dynamic allocation of memory for the ASTs as they are added to our history is that we no longer know exactly where they will be located, meaning this information must be stored such that it will not be erased between calls to WASM library functions. One method of doing this is passing a pointer to where in memory the AST is located to the JavaScript module so that it can refer to it later, and use library functions on it. At first glance, this sounds like a bad idea, as when pointers are returned from a function for which wasm-bindgen (see **??**) is used to make a JavaScript binding, the pointer is represented as a JavaScript *number* type [23], which is a double-precision IEEE-754 value [7]. Storing pointers as floating point values, and then attempting to dereference them, sounds like a recipe for memory mismanagement. However, this is safe because WebAssembly 2.0 has 32 (see 2.5), and thus has 32 bit pointers, and a double precision floating point number has a 52 bit [11] mantissa meaning it can safely store the 32-bit memory location without issue.

In our JavaScript module, we can then store a stack of pointers to the ASTs, and display the options for reducing the one at the top. When an option is selected, we can apply the reduction and then store the new AST on the top of our stack and recalculate reduction options. If the user decides to start evaluating a new program, all the ASTs with pointers in this list are freed to avoid memory leaks.

5.2.1 Major Bugfixes

- Typechecker

5.3 The Intermediate Focus Group: Evaluation and Next Steps

Evaluation: - They liked explicit match: they liked it more than haskell for learning about how pattern matching works - Really Really needed light mode - Horizontal overflow bug

```
enum DiffElem {
    Similarity(String),
    Difference(String, String)
}

type Diff = Vec<DiffElem>

// rust-like psuedocode, not valid rust
// ast1 and 2 are the two ASTs, and expr1 and 2 are the indices
// of the terms we are considering for our diff.
fn diff(ast1, ast2, expr1, expr2) -> Diff {
    node1, node2 = ast1.get(expr1), ast2.get(expr2)
    diff = Diff::new();
    match (node1, node2) {
        // IDs and Lits are compared based on their string "values"
        case (ID, ID)
        case (Lit, Lit) {
            if node1.value == node2.value {
                diff += Similarity(node1.value)
            } else {
                diff += Difference(node1.value, node2.value)
            }
        }

        case (Pair {first1, second1}, Pair {first2, second2}) {
            // As both are pairs, their opening brackets,
            // commas, and closing brackets are in common.

            // We get the diff of the first and second
            // element to find the diff of the whole pair
            diff += Similarity("(")
            diff += diff(ast1, ast2, first1, first2)
            diff += Similarity(",")
            diff += diff(ast1, ast2, second1, second2)
            diff += Similarity(")")
        }

        ...
    }

    return diff;
}
```

Figure 5.1: rust-like psuedocode listing for the type of the output of the AST::diff function, as well as a small section of the algorithm. There is also (not shown) a wrapper around the Diff type, to allow for conversion into JavaScript (see 2.5), as well as the some logic for combining diffs.

Chapter 6

Phase 4 — Further UI/UX Iteration

Chapter 7

Conclusion

The aims of this project were to create a system to help to build an intuitive understanding of functional programming languages

7.1 Strengths

7.1.1 The Language Achieves Its Design Aims

To remind the reader, the design aims for the language were:

1. **It should be simple and easy to understand**. This requires that the language should not have features that users might find difficult to understand why they work. This means that the language should have very few inbuilt functions, all of which should be easy to understand why they work.
2. **It should be similar to existing functional languages**. This would allow users to be able to transfer their intuition to other languages. It should be similar in syntax (it should have similar tokens and structures), as well as semantics (it should work similarly).
3. **It should be powerful enough to explain key concepts.**

7.2 Limitations

7.2.1 The Expressions Balloon During Evaluation

I believe that the languages lack of inbuilts is one of the languages best 'features'. However, it is also a curse: as everything is defined with match expressions, the expression balloons vertically with match statements during evaluation. For instance, in the provided 'square_sum' example:

```
square :: Int -> Int
square x = x * x

// List of the square numbers from lower to upper
list_of_squares :: Int -> Int -> List Int
list_of_squares lower upper = map square $ range lower upper

main :: Int
main = sum $ list_of_squares 1 5
```

Despite their being no match expressions in sight, the 'main' expression balloons to 3 match statements deep within 6 lazy steps:

```
match (match (match (infiniteFrom 1) {
    | Nil -> Nil
    | Cons x xs -> if ((5 - 1) > 0) (Cons x (take ((5 - 1) - 1) xs)) Nil
}) {
    | Nil -> Nil
```

```
6       | Cons x xs -> Cons (square x) (map square xs)
7   }) {
8       | Nil -> 0
9       | Cons x xs -> (\x. \acc. x + acc) x (foldr (\x. \acc. x + acc) 0 xs)
10  }
```

The outer one comes from 'sum', the middle one comes from 'map', and the inner one comes from 'range', all prelude functions. Unfortunately, this is only really solvable to an extent, as pattern matching is a key concept in functional programming languages. Furthermore, a conclusion of the intermediate focus group was that the explicit match syntax, where it was obvious where/how pattern matching was occurring, made understanding pattern matching much easier. Indeed, they agreed that they would have liked to have SFL to learn about pattern matching rather than Haskell (see 5.3).

This situation could be improved by being able to select which functions we are interested in seeing the expansion of, and which ones we are not. See 7.3.1

7.3 Future Work

7.3.1 Selective Skipping

We are not always interested in all the functions involved in our program. For instance, if a lecturer is attempting to demonstrate `foldr` over a list, they may not be interested in the expansion of how `range` works in order to generate their list they are going to fold over. They may want the evaluation of some things to be skipped over. This is something that I have considered doing from the start, however I have not had time to properly investigate how this could be done.

We could mark certain expressions as 'uninteresting', and evaluate them as much as we can immediately. For instance, if the syntax for an uninteresting expression looked like '[e]':

```
1  main :: Int
2  main = sum $ [range 1 4]
```

We could immediately evaluate '`range 1 4`' to '`Cons 1 (Cons 2 (Cons 3 Nil))`'. However, consider:

```
1  fix f = f $ fix f
2
3  id x = x
4
5  main = if true 1 [fix id]
```

The evaluation of '`fix id`' will never terminate. If we were to attempt to evaluate this, it would run forever. If we were to provide a mechanism that forces full evaluation of a term, we would be providing functionality that the user could use to 'shoot themselves in the foot'. This would need to be clearly communicated to the user, and a mechanism of stopping this evaluation should be provided if the user judges it has been too long.

7.3.2 More Extensive User Testing

The agile approach taken during this project allowed for 5 different testing opportunities:

- The Proof of Concept client meeting
- The Advanced Focus group
- The Intermediate Focus Group
- The Beginner Focus Group
- The final client meeting

However, the

7.3.3 Extensions to the language

7.3.4 Improvements to the UI

Bibliography

[1] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Alternative eText Formats Series. Addison-Wesley, 2007. URL: https://books.google.co.uk/books?id=WomBPgAACAAJ.

[2] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[3] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001. Accessed: 2025-04-06. URL: https://agilemanifesto.org/.

[4] MANUEL M. T. CHAKRAVARTY and GABRIELE KELLER. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, 14(1):113–123, 2004. doi:10.1017/S0956796803004805.

[5] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[6] Jana Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Int'l Conf. Functional Programming*, September 2013. arXiv:1306.6032[cs.PL].

[7] Ecma International. Ecmascript language specification. https://tc39.es/ecma262/#sec-ecmascript-language-types-number-type, 2025. Section 6.1.6: The Number Type.

[8] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[9] Paul Hudak and Joseph H Fasel. A gentle introduction to haskell. *ACM Sigplan Notices*, 27(5):1–52, 1992.

[10] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1, 2007.

[11] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic, 2008. IEEE Std 754-2008, IEEE, New York, NY, USA.

[12] S. Klabnik and C. Nichols. *The Rust Programming Language, 2nd Edition*. No Starch Press, 2023. URL: https://books.google.co.uk/books?id=a8l9EAAAQBAJ.

[13] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[14] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[15] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. URL: https://www.sciencedirect.com/science/article/pii/0022000078900144, doi:10.1016/0022-0000(78)90014-4.

[16] University of Bristol School of Computer Science. Unit information: Advanced topics in programming languages (teaching unit) in 2024/25, 2025. URL: https://www.bris.ac.uk/unit-programme-catalogue/UnitDetails.jsa?ayrCode=24%2F25&unitCode=COMSM0067.

[17] University of Bristol School of Computer Science. Unit information: Imperative and functional programming in 2024/25, 2025. URL: https://www.bristol.ac.uk/unit-programme-catalogue/UnitDetails.jsa?ayrCode=24%252F25&unitCode=COMS10016.

[18] University of Bristol School of Computer Science. Unit information: Programming languages and computation in 2024/25, 2025. URL: https://www.bristol.ac.uk/unit-programme-catalogue/UnitDetails.jsa?ayrCode=24%252F25&unitCode=COMS20007.

[19] University of Bristol School of Computer Science. Unit information: Types and lambda calculus (teaching unit) in 2024/25, 2025. URL: https://www.bristol.ac.uk/unit-programme-catalogue/UnitDetails.jsa?ayrCode=24%2F25&unitCode=COMS30040.

[20] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002. URL: https://books.google.co.uk/books?id=hPL6DwAAQBAJ.

[21] Amon Rapp. *Autoethnography in Human-Computer Interaction: Theory and Practice*, pages 25–42. Springer International Publishing, 06 2018. doi:10.1007/978-3-319-73374-6_3.

[22] Andreas Rossberg. WebAssembly Core Specification. Technical report, W3C, 2022. URL: https://www.w3.org/TR/wasm-core-2.

[23] Rust and WebAssembly Working Group. `wasm-bindgen` guide. https://rustwasm.github.io/wasm-bindgen/, 2024. Accessed April 16, 2025.

[24] Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In *International Symposium on Functional and Logic Programming*, pages 119–135. Springer, 2014.

Appendix A

AI Usage

I did not directly prompt any Large Language Models, or any other AI model, to assist with the writing of my dissertation or implementation. However, as listed in the Supporting Technologies list, I used GitHub Copilot to help with writing some tests for the parser and type checker. I used it via the VS Code extension, which uses the context of your file, to provide advanced AI autocompletion.

Appendix B

Some Example Derivations Using the Type Checking Algorithm

B.1 Typechecking the Pair Function

The pair function $\lambda x\ y.\,(x, y)$ takes two arguments, and returns a pair of the two values. Here, we type check it against its correct type $\forall\alpha\ \beta.\,\alpha \to \beta \to (\alpha, \beta)$. Type checking/synthesis is done recursively from bottom up, so read [1] upwards.

$$\Gamma = \alpha, \beta, x : \alpha, y : \beta$$

$$\Delta = \alpha$$

$$\dfrac{\dfrac{(x : \alpha) \in \{\Gamma\}}{\Gamma \vdash x \Rightarrow \alpha \dashv \Gamma}\ \mathsf{Var}\ [8] \qquad \dfrac{}{\Gamma[\alpha] \vdash \alpha <: \alpha \dashv \Gamma[\alpha]}\ <:\mathsf{Var}\ [9]}{\Gamma \vdash x \Leftarrow \alpha \dashv \Gamma}\ \mathsf{Sub}\ [6]$$

$$\dfrac{\dfrac{(y : \beta) \in \{\Gamma\}}{\Gamma \vdash y \Rightarrow \beta \dashv \Gamma}\ \mathsf{Var}\ [10] \qquad \dfrac{}{\Gamma[\beta] \vdash \beta <: \beta \dashv \Gamma[\beta]}\ <:\mathsf{Var}\ [11]}{\Gamma \vdash y \Leftarrow \beta \dashv \Gamma}\ \mathsf{Sub}\ [7]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{[6]\Gamma \vdash x \Leftarrow \alpha \dashv \Gamma \qquad [7]\Gamma \vdash y \Leftarrow \beta \dashv \Gamma}{\Gamma \vdash (x, y) \Leftarrow (\alpha, \beta) \dashv \Gamma}\ \mathsf{Pair}\Leftarrow\ [5]}{\alpha, \beta, x : \alpha \vdash \lambda y.\,(x, y) \Leftarrow \beta \to (\alpha, \beta) \dashv \Gamma}\ \to\mathsf{I}\ [4]}{\alpha, \beta \vdash \lambda x\ y.\,(x, y) \Leftarrow \alpha \to \beta \to (\alpha, \beta) \dashv \Delta, \beta, \{x : \alpha, y : \beta\} \qquad (= \Gamma)}\ \to\mathsf{I}\ [3]}{\alpha \vdash \lambda x\ y.\,(x, y) \Leftarrow \forall\beta.\,\alpha \to \beta \to (\alpha, \beta) \dashv \{.\}, \alpha, \{.\} \qquad (= \Delta)}\ \forall\mathsf{I}\ [2]}{. \vdash \lambda x\ y.\,(x, y) \Leftarrow \forall\alpha\ \beta.\,\alpha \to \beta \to (\alpha, \beta) \dashv .}\ \forall\mathsf{I}\ [1]$$

1. Here, we begin typechecking with the $\forall\mathsf{I}$ rule to introduce $\forall\alpha$. We do this by adding α to the initially empty context. We then check the function against the type without the $\forall\alpha$: $\forall\beta.\,\alpha \to \beta \to (\alpha, \beta)$. The output of this checking is then split into 3 parts: everything before the α, the α itself, and the bits after the α. Our output context is everything in Δ before the alpha, which is nothing.

2. We apply the same rule as above, but we are introducing $\forall\beta$. We then check the function against $\alpha \to \beta \to (\alpha, \beta)$ Our output context for this rule is everything in Γ before the β: only α.

3. We then start to unwrap the abstractions. We strip the abstraction over x from the expression, leaving us with $(\lambda y.\,(x, y))$. We then add $(x : \alpha)$ to our context, and then progress by checking the remaining part of the expression against $\beta \to (\alpha, \beta)$. Our output context is Γ.

4. Same as above, with y against β. We unwrap the abstraction over y to give us (x, y). We then check this against (α, β). The output context is Γ.

5. We check (x, y) against the type (α, β). To check this, we check x against α and y against β. The output context is Γ.

6. To check x against type α we synthesise the type of x ([9]: trivial, as its in the context). We then check this against α, and a check of α against α ([10]) trivially passes.

7. Same as above with y against β. The output context is Γ

B.2 Typechecking an Expression Involving Lists

We shall attempt to use the algorithm to check the type of Cons 1 x against $List\ Int$. This derivation should serve as a demonstration of how more complex checking works. This derivation assumes $Cons$ and Nil are defined over Ints only. The reason the context is never changed is as we do not have any abstractions or foralls, so no variables or type variables are introduced.

$$T_{Nil} = List\ Int \qquad T_{Cons} = Int \to List\ Int \to List\ Int$$

$$\Gamma = T_{Cons}, T_{Nil} \qquad \Gamma = \Gamma_0 = \Gamma_1$$

$$\frac{\dfrac{}{\Gamma_0 \vdash \text{Int} <: \text{Int} \dashv \Gamma_1}\ \text{<:Int}\ [11]}{\Gamma_0 \vdash List[Int] <: List[Int] \dashv \Gamma_1}\ \text{<:}\bigcup\ [10]$$

$$\frac{\dfrac{\dfrac{(Nil : T_{Nil}) \in \Gamma}{\Gamma \vdash Nil \Rightarrow T_{Nil} \dashv \Gamma}\ \text{Var}\ [9] \qquad [10]\Gamma \vdash List[Int] <: List[Int] \dashv \Gamma}{\Gamma \vdash Nil \Leftarrow List\ Int \dashv \Gamma}\ \text{Sub}\ [8]}{\Gamma \vdash List\ Int \to List\ Int \bullet Nil \Rightarrow\!\!\!\Rightarrow List\ Int \dashv \Gamma}\ \to\text{App}\ [7]$$

$$\frac{\dfrac{(Cons : T_{Cons}) \in \Gamma}{\Gamma \vdash Cons \Rightarrow T_{Cons} \dashv \Gamma}\ \text{Var}\ [4] \qquad \dfrac{\dfrac{}{\Gamma \vdash 1 \Leftarrow \text{Int} \dashv \Gamma}\ \text{IntLit}\Leftarrow\ [6]}{\Gamma \vdash T_{Cons} \bullet 1 \Rightarrow\!\!\!\Rightarrow List\ Int \to List\ Int \dashv \Gamma}\ \to\text{App}\ [5]}{\Gamma \vdash Cons\ 1 \Rightarrow List\ Int \to List\ Int \dashv \Gamma}\ \to\text{E}\ [3]$$

$$\frac{\begin{array}{c}[3]\Gamma \vdash Cons\ 1 \Rightarrow List\ Int \to List\ Int \dashv \Gamma \\ [7]\Gamma \vdash List\ Int \to List\ Int \bullet Nil \Rightarrow\!\!\!\Rightarrow List\ Int \dashv \Gamma\end{array}}{\Gamma \vdash Cons\ 1\ Nil \Rightarrow List\ Int \dashv \Gamma}\ \to\text{E}\ [2]$$

$$\frac{[2]\Gamma \vdash Cons\ 1\ Nil \Rightarrow List\ Int \dashv \Gamma \qquad [10]\Gamma \vdash List[Int] <: List[Int] \dashv \Gamma}{\Gamma \vdash Cons\ 1\ Nil \Leftarrow List\ Int \dashv \Gamma}\ \text{Sub}\ [1]$$

1. To check $Cons\ 1\ Nil$ against $List\ Int$ [2], we first synthesise the expression type, and check the synthesised type is as subtype of $List\ Int$ [10].

2. To synthesise the type of the expression $Cons\ 1\ Nil$ (implicitly $((Cons\ 1)\ Nil)$) we synthesise the type of the left hand side of the application $Cons\ 1$ to be $List\ Int \to List\ Int$ [7] and then we synthesise the type of Nil under the application of that type, which gives us $List\ Int$.

3. To synthesise the type of $Cons\ 1$, we synthesise the type of the left hand side of the application $Cons$ to be $T_{Cons} : Int \to List\ Int \to List\ Int$ [4], and then synthesise the type of 1 under the application of that type, which gives us $List\ Int \to List\ Int$ [5].

4. We synthesise the type of $Cons$ to be T_{Cons} from the context.

5. We synthesise the type of 1 under the application of T_{Cons}, by first checking the type of 1 against the type Int which is the left hand side of the applied type[6]. This allows us to synthesise the right hand side of the applied type: $List\ Int \to List\ Int$.

6. 1 checks against the type Int, as it is an Int literal.

7. To synthesise the type of Nil under the application of $List\ Int \to List\ Int$, we check Nil against the type $List\ Int$ which is the left hand side of the applied type[8]. This allows us to synthesise the right hand side of the applied type: $List\ Int$

8. To check Nil against the type $List\ Int$, we first synthesise the type of Nil resulting in $T_{Nil} : List\ Int$[9]. We then check that this is a subtype of $List\ Int$[10].

9. We synthesise the type of Nil to be T_{Nil} from the context.

10. We apply the $<:\bigcup$ rule to check that $List\ Int$ is a subtype (non strict) of $List\ Int$. The first check is that the names are the same, as the names uniquely identify these types. We then iterate over the list of the arguments to the type constructor. The name of the context increments to reflect this iteration, but the context is unchanged during this check. There is only one type in the list

Appendix C

The SFL Prelude

```
if :: Bool -> a -> a -> a
if cond then_branch else_branch = match cond {
  | true -> then_branch
  | false -> else_branch
}

data Either a b = Left a | Right b
data Maybe a = Just a | Nothing
data List a = Cons a (List a) | Nil

// List Operations
map :: (a -> b) -> List a -> List b
map f list = match list {
  | Nil -> Nil
  | Cons x xs -> Cons (f x) (map f xs)
}

foldr :: (a -> b -> b) -> b -> List a -> b
foldr f acc list = match list {
  | Nil -> acc
  | Cons x xs -> f x (foldr f acc xs)
}

filter :: (a -> Bool) -> List a -> List a
filter pred list = match list :: List a {
  | Nil -> Nil
  | Cons x xs -> if (pred x) (Cons x (filter pred xs)) (filter pred xs)
}

repeat :: a -> List a
repeat n = Cons n $ repeat n

length :: List a -> Int
length xs = foldr (\_ i. i + 1) 0 xs

infiniteFrom :: Int -> List Int
infiniteFrom x = Cons x (infiniteFrom (x + 1))

take :: Int -> List a -> List a
take n list = match list {
  | Nil -> Nil
  | Cons x xs -> if (n > 0) (Cons x (take (n - 1) xs)) (Nil)
}
```

```haskell

range :: Int -> Int -> List Int
range lower upper = take (upper - lower) $ infiniteFrom lower

sum :: List Int -> Int
sum = foldr (\x acc. x + acc) 0
```

Appendix D

Additional Materials

File Name	Description	How to Open
`Figma_Design.fig`	The Figma Prototype of the design	Using Figma Desktop or Web
`afg_transcript.pdf`	The AI generated transcript from the audio recording of the advanced focus group	Using a PDF viewer
`c[2, 3, 4]_end.zip`	The built application as it was at the end of phases 2, 3, 4 respectively	Unzip, and then serve the 'list' folder. An easy way is to run `python3 -m http.server 3000` in the 'list' folder to serve on port 3000, and then go to `localhost:3000` in the browser. Unfortunately, I was not able to package the end of phase 1 product in a way that was as simple to serve, however 3.4 shows what it looked like

Appendix E

Tokens for Lexical Analysis

Below is the code for how tokens outputted by lexical analysis are defined.

```
enum TokenType {
    EOF,
    Newline,

    Id,
    UppercaseId,

    If,
    Then,
    Else,

    Match,
    LBrace,
    RBrace,

    IntLit,
    FloatLit,
    StringLit,
    CharLit,
    BoolLit,

    DoubleColon,
    RArrow,
    Forall,
    KWType,
    KWData,

    LParen,
    RParen,

    Lambda,

    Dollar,
    Dot,
    Comma,
    Bar,

    Assignment,
}

struct Token {
    tt: TokenType,
```

```
    value: String,
}
```

Appendix F

UI Screenshots

Figure F.1: The UI as tested in the testathon

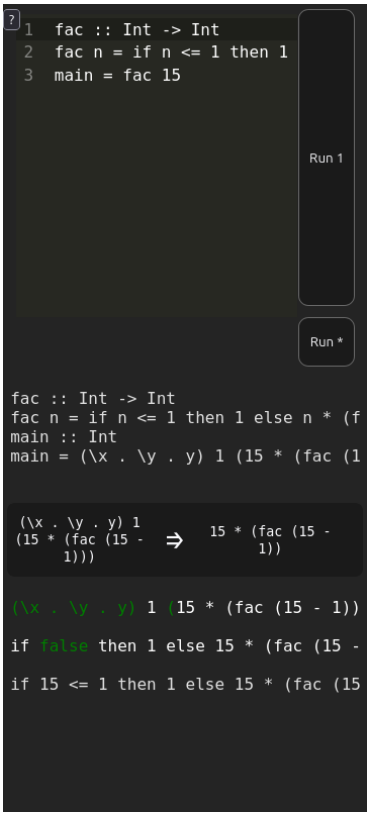

Figure F.2: The UI as tested in the testathon, as it would have appeared on a Samsung Galaxy S20

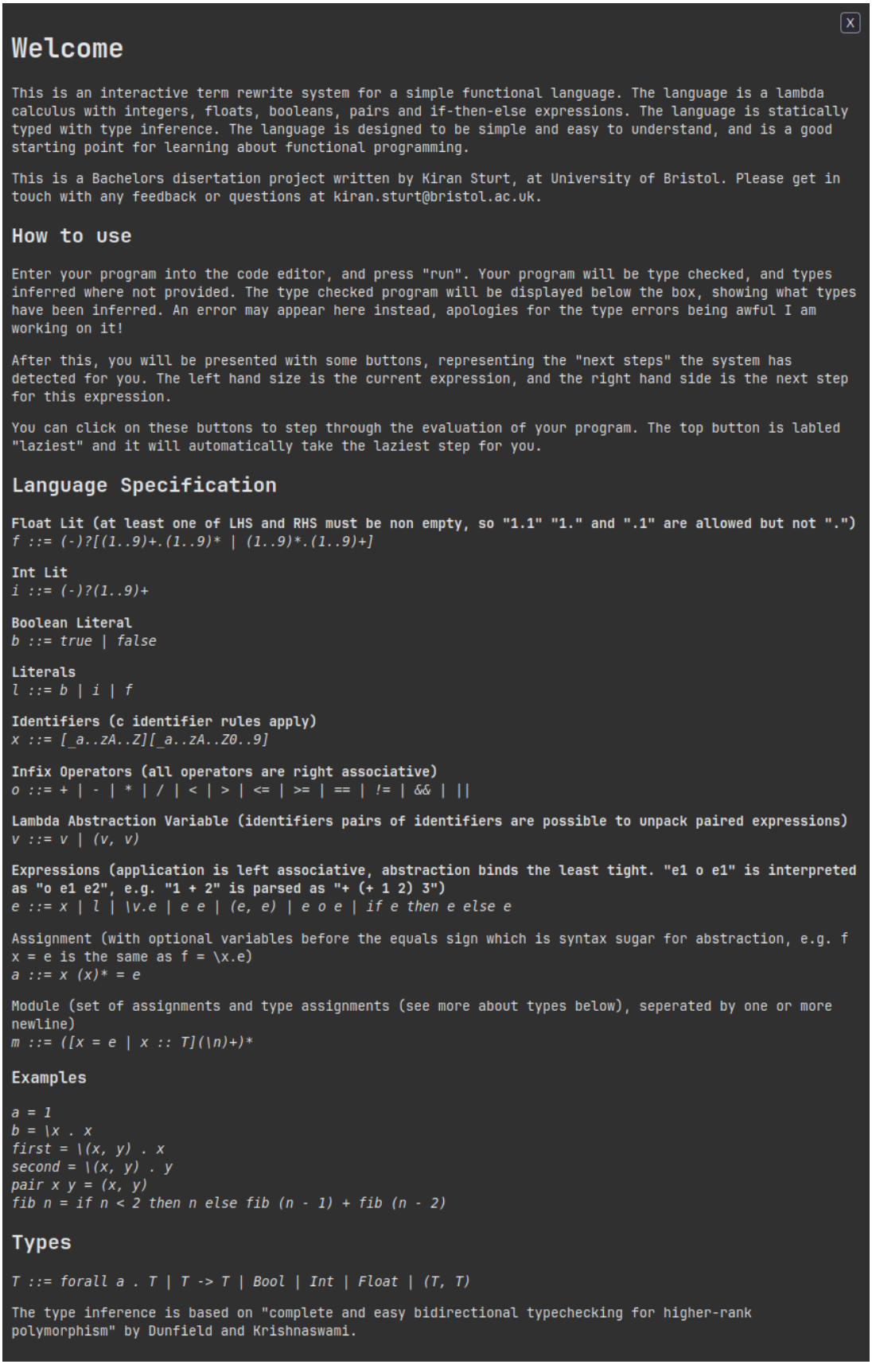

Figure F.3: The 'Help menu' in the testathon. This was spawned by pressing the '?' button in the top left of the UI, and dismissed by pressing the 'X' button, or clicking outside the box

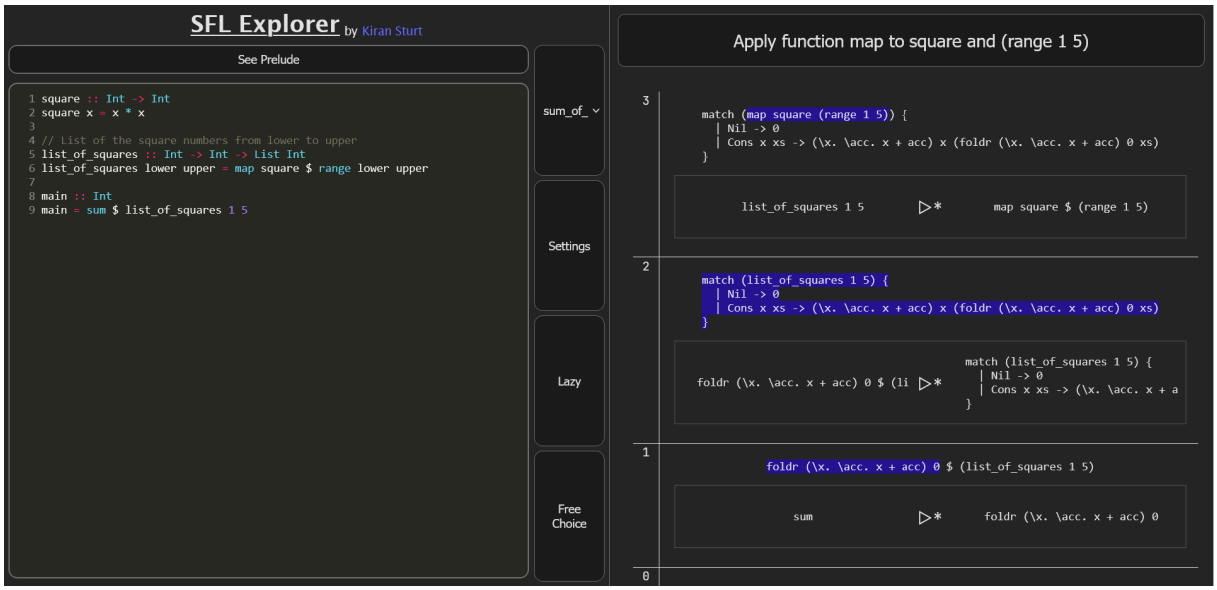

Figure F.4: The final product during lazy evaluation of the 'sum of squares' sample program

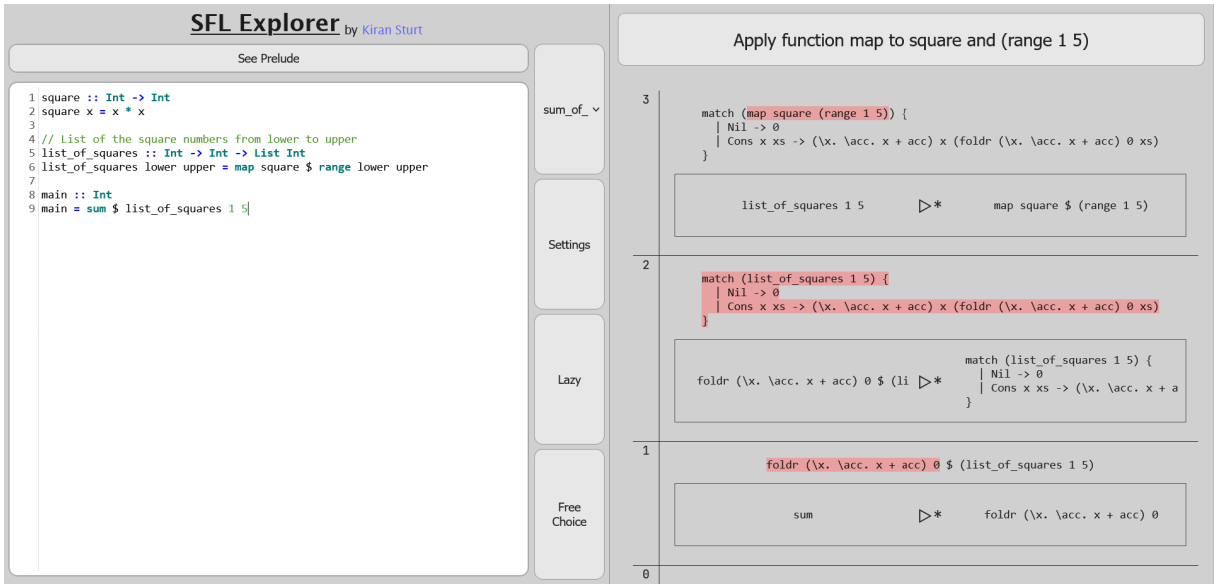

Figure F.5: The final product during lazy evaluation of the 'sum of squares' sample program in light mode

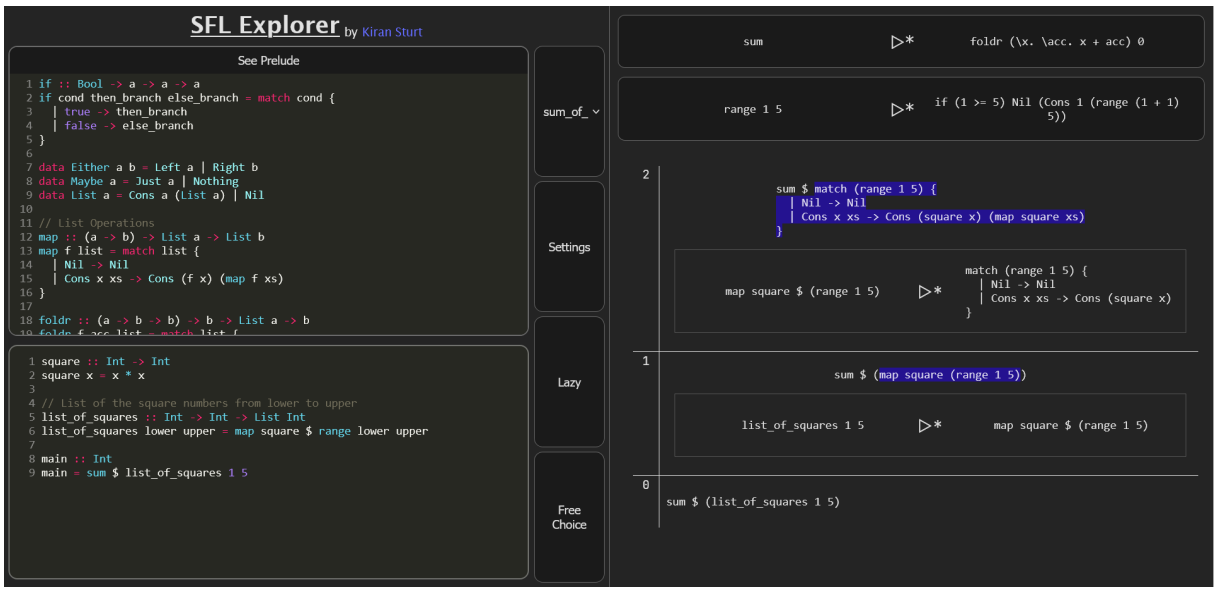

Figure F.6: The final product during free choice evaluation of the 'sum of squares' sample program, with the prelude visible

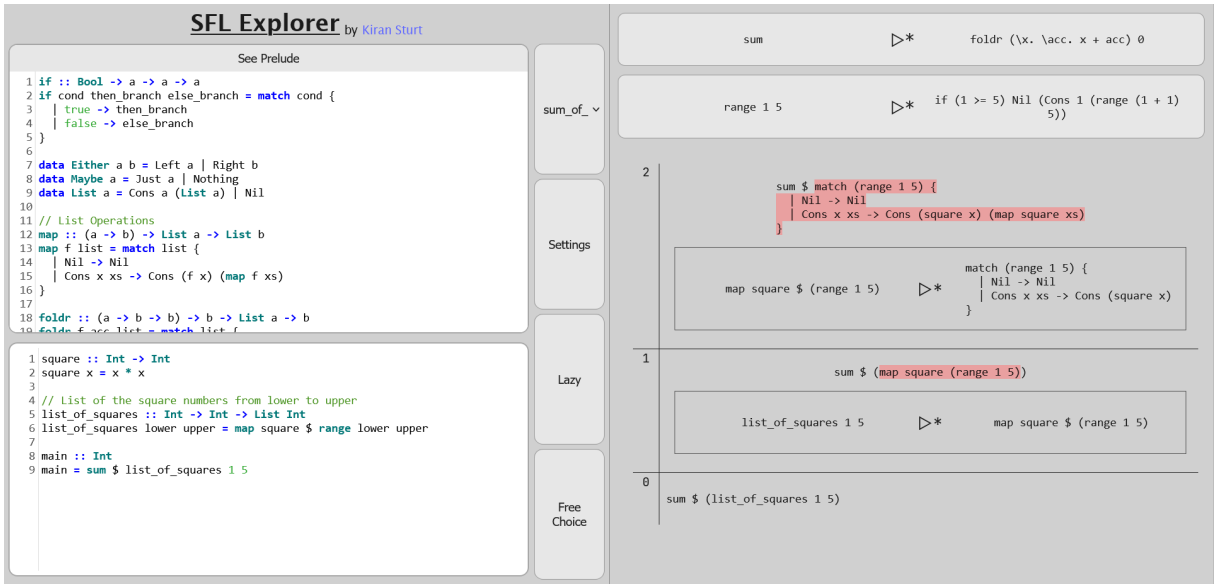

Figure F.7: The final product during free choice evaluation of the 'sum of squares' sample program, with the prelude visible in light mode

Appendix G

Language Grammar