HALE

A Web Based Game to Teach Functional Programming

Joe Moore 1917702
Department of Computer Science
University of Warwick
Year of Study: 3rd

Supervisor: Jonathan Foss

Abstract

There exist very few online tools aimed at teaching functional programming. For example commonly used tools for learning programming, Codecademy and Khan Academy offer no functional programming languages. As such the paradigm can be difficult to understand. This is coupled with the fact that functional programming languages can be troublesome to setup. For instance, Stack, which is often used to manage Haskell projects is complicated to install and use. This report details the design, implementation and deployment of Hale. Hale is a tool to teach functional programming to people with some imperative programming knowledge, but no familiarity with the functional programming paradigm. After investigation into teaching methods, it was decided that gamification would be used to aid the teaching tool. The results of such investigations into gamification and teaching tools were used to guide the direction of development. Hale's success was evaluated through user feedback and property-based testing.

Keywords:
Functional Programming, Teaching, Lambda Calculus, Gamification, Interpreter, API

Acknowledgements

I would like to thank Jonathon Foss, for his advice and help during the development of this project, and for the functional programming students at Warwick and elsewhere who tested Hale. I am also grateful to Michael Gale for his initial work in supervising the project.

Contents

λ.1

Introduction

1.1 Motivation

With programming being such a fundamental skill in the field of Computer Science, there exist numerous of tools to teach it. These range from beginner to advanced lessons. These tools are most commonly in the form of interactive websites that provide the ability for students to solve problems by typing code into one box and then seeing an output to its side. They almost always allow the student to see the correctness and less commonly, but still regularly, the efficiency of their solution. However there is a lack of these tools for functional programming, for example Codecademy does not currently offer any functional languages. In addition there are few tools that are engaging for children. Most of them simply ask for the user to complete an arbitrary and often boring task, such as computing the n^{th} number in a complicated mathematical sequence.

Some research has shown that teaching functional programming to children from a younger age has clear benefits to their overall programming ability [1] [2]. It allows the explanation of an algorithm with fewer distracting elements. It excludes complicated syntax, order of evaluation and exceptions. However, there is a steep learning curve and Haskell, alongside other functional languages, have a reputation in the academic community for being intellectual due to them relying heavily on abstract mathematical concepts. This is why there exists the need for a fun teaching tool aimed at making teaching dynamic and exciting for a simple functional programming language. The language will be a halfway between Haskell and λ-calculus with its syntax taking inspiration from both. The benefits are that the simplicity of λ-calculus can be used to impart basic principles whilst the data structures, such as lists, and types alongside the way they are handled in Haskell, are more beneficial. Functional programming has benefits and can often lead to better programming ability in the long run if learnt early [3].

1.2 Current Solutions

Codecademy is the most well known tool for teaching programming and has many courses. These however are all for imperative languages - none are for functional programming languages. Scratch is a good example of an implementation of teaching through a game. The solutions users create can be visualised almost immediately in game form. They can take the solutions and algorithms they create and see how they directly affect the environment [4]. Scratch makes use of a purely constructive approach. It gives way for experiments. This means that despite its strengths it has some gaping weaknesses for teaching new languages. The main weakness being that students learn through trial and error without guidance. Research shows this is more likely to lead to bad habits developing and also a hampered ability to grasp complex concepts [5]. Whilst a constructive approach has many benefits and this project builds on the principles of constructivism, it is key that this is paired with some direction to achieve stronger results [6] [7].

There also exist other block based teaching methods that implement a more traditional teaching style, such as Reduct, a block based puzzle game to teach JavaScript. However studies showed this block approach (even when applied in a less creatively free setting) had serious negative side effects. The students' knowledge transfer was very poor. And students struggled to write good JavaScript code once the blocks were removed [8].

1.2.1 Functional Programming Teaching Resources

Interactive functional programming teaching tools are not particularly common. In the majority of instances it is still taught through traditional teaching methods. Often the first interaction students will have with functional programming is at university. As such, functional programming has developed a reputation as an academic form of programming, and one that cannot be easily understood by a layman, due to it being too abstract.

Code World [9] is an example of one of the few online functional programming teaching tools. It is designed to teach concepts such as pattern matching and lambda expressions but it lacks a sense of direction. This is because there are no missions or specific tasks to complete.

Another example is ScalaQuest [10] which is still in development. ScalaQuest is a game that allows puzzle solving by writing small Scala programs. Unfortunately the beta of the game suffered from a number of imperfections and bugs. It is also aimed at people with a fundamental understanding of Scala rather than no functional programming

knowledge at all. Therefore it does not fully fit the needs of the problem Hale is designed to solve.

1.3 Goals and Objectives

The general aim of this project is to create a tool to teach children and young adults (unfamiliar with functional programming) a pure form of functional programming. This should hopefully provide a foundation of understanding such that if they wanted to pick up a functional language at a later stage, they would have the knowledge to do so, without feeling intimidated. The tool will use a game-based approach due to its significant advantages particularly for younger users. Whereas learning new skills through games has been shown to boost enthusiasm among researchers and lecturers [11]. Child programmers who learnt the skill initially through games show a stronger aptitude for the skill years later [12].

The problems should increase in difficulty to provide a sense of progression. For example, the first problem will be to code the main character's move command. This is a simple incrementation problem, whereby the solution is to increment the character's x and y coordinates. These problems will give way to more advanced problems. It is also necessary that some of these later challenges rely on information from the initial levels, this will reinforce knowledge.

The tool will teach a custom functional programming language, Hale. It will be an extension on λ-calculus, to include types and primitives. This use of a version of λ-calculus will allow focus on teaching the key concepts of pure functional programming. Below is an example of a program written in Hale to compute the third power of a number:

$$\texttt{def}\ \ \textit{cube} ::= \lambda x : \texttt{int}.\ \ x * x * x$$

The syntax will use the lambda to define the start of a function with parameters, followed by a colon type and then a full stop to signify start of the function. As seen below:

$$\texttt{def}\ \ \textit{[function name]} ::= \lambda\ \textit{[function parameters]} : \textit{[function type]}\ .\ \textit{[function body]}$$

1.4 Requirements

Requirements were developed with project management principles of software development in mind.

Functional Requirements

1. The backend of the website should successfully interpret and run all of the code written by the user. The scope of code Hale should be able to interpret is:
    (a) Lambda expressions
    (b) Types:
        i. Integers
        ii. Bools
        iii. Strings
        iv. Lists
    (c) Function application
    (d) List operations
    (e) Higher order functions
    (f) Recursion
2. Errors should be properly handled and feedback given to the user. No code should be able to break the website.
3. The game should work if the user inputs correct code and the use of mouse etc should execute the code they have written in order to play the game.
4. The game should only work as intended if the user completes the challenges.
5. If a user writes code that can be interpreted, but does not fulfil the expected purpose, this should be represented by the game playing according to the user's incorrect code.
6. The type-level system designed will use an Embedded Domain-Specific Language in order to make it accessible. It should use the type-level model to type-check inputted code and provide feedback specifically about type issues.
7. The user should be able to see what challenges they have completed.
8. Tangible and tactile in-game feedback should be given whenever:
    (a) A program runs
    (b) A program fails to run
    (c) A program runs but does not give an expected result

    Tactile and tangible feedback means clear in-game behaviours and changes to the users environment to indicate all three of the above situations.

9. Success of a challenge should factor in:
    (a) Length of code
    (b) Program efficacy
    (c) Program result
10. The website should adhere to cyber security principles in order to protect users' passwords and personal information.

Non-functional Requirements

1. The teaching tool should allow students to fulfil the *MUSIC* [13] model in their learning. This is down by:
    (a) Presenting the user with problems with multiple solutions so as to provide them with significant choices and place them in control of their learning.
    (b) Content should increasing in difficulty with later tasks being possible but fairly tricky. This should provide a challenging learning environment and improve knowledge whilst allowing the user to be certain they have built up sufficient skills in previous tasks to attempt the current one.
    (c) Explanations as to the usefulness and applications of different aspects of functional programming should be explained with each task.
2. The total learning experience should result in the creation of an overarching project, so as to facilitate *project based learning*.
3. Students should feel that their understanding of functional programming has improved by engaging with the teaching tool.
4. The code feedback should be helpful and understandable to students who are unfamiliar with functional programming.
5. The UI should be clean and intuitive.

The elements of the language that are to be required of Hale were chosen specifically as they are usually studied early on in functional programming textbooks. They appear in chapters 1-6 of *Learn You a Haskell for Great Good!* [14]. They are covered also in part 1 of Hutton's *Programming in Haskell* [15]. The gamification element of the project is set to be satisfied by functional requirements 3-8. Requirement 10 is in place, since by nature of being a teaching tool, it is likely that this project will be used by legal minors. It is therefore critical that users and their data are protected accordingly.

$\lambda.2$

Background and Literature Review

This chapter details the research conducted during the completion of this project. It is split into 3 sections, Learning Processes, Functional Programming and Interpreter Design. The former contains research into the complexity of the learning process. This details the factors affecting engagement and performance. This leads onto gamification and how this principle can solve and fulfil conditions set by such research. The latter half of the chapter details the research done into functional programming languages. The research into languages was naturally then used to influence the design of Hale and its interpreter. Finally research into the fundamentals of interpreter design lays the groundwork of the structure of the interpreter for Hale.

2.1 Learning Processes

2.1.1 Constructivism

Constructivism is a learning theory that places its emphasis on active interaction [6]. In constructivism a student infers knowledge as a result of direct feedback from an environment. This environment needs to be affected by their actions. Another key feature of constructivism is the importance placed on instruction. It is expected that the user will need to interpret instructions [7] and this differs from more memorisation techniques. Such memorisation based learning methods were noted to be less effective in long term gain [16].

In addition to these benefits over other learning techniques, constructivism has been shown to be the most compatible learning process with gamification and game-based teaching tools [17].

Constructivism has two key concepts, assimilation and accommodation. Assimilation is the process of taking new information and applying it to existing circumstances. Whilst accommodation is when new information is used to adjust an existing schema.

2.1.2 Deep Focus in Constructivism

Deep focus is hard to maintain in constructivist projects if material is too challenging. However, if objectives are easy to meet, focus can be maintained and anxiety can be avoided. Furthermore, it is also possible for onset boredom to occur if tasks are consistently easy and do not progress with the ability of the user. As such Csikszentmihalyi describes the idea of a flow channel [18]. This is a progressing increase of difficulty to accompany the increase in user skill.

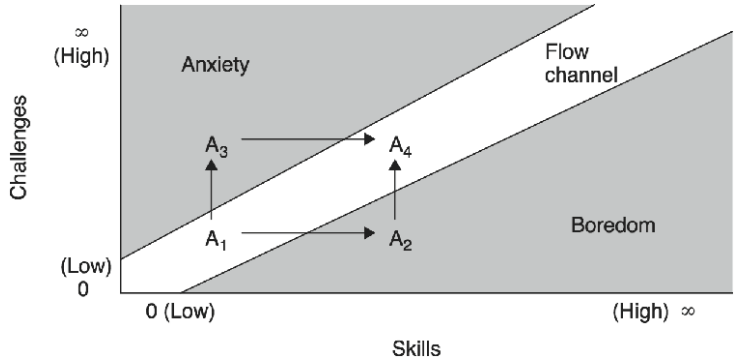


Figure 2.1: Flow channel, from Jesse Schell's Art of Game Design: A Book of Lenses. *A* represents the player [19].

As such it is clear that the employment of constructivism, with instructions requiring interpretation, is itself only effective with clear goals and tangible feedback. In addition it is also apparent that the difficulty of the challenges needs to increase within the range of the flow channel. As each flow channel is unique for each user, it is expected that some users will reside in either the anxiety or boredom regions whilst using the tool. The emphasis of the design therefore needs to be placed on ensuring that x% of users remain within the flow channel for the duration of using the teaching tool, where x is an arbitrarily chosen threshold value.

2.1.3 Gamification

Gamification is the process of using attributes of video games as design elements, as well as the basis of other applications [20] [21]. The most common of such attributes are [22]:

- Levels (challenges that users must complete, often in a particular order)
- Missions (sets of behaviours that might compose part of a level)
- Notifications/Feedback (user should see direct consequences of their actions in game)

Research has shown gamification to be effective. The tangible feedback of seeing effects in game helps to cement knowledge gained. It was shown that students showed little interest in learning the languages C and C++ through traditional methods of teaching, and became more interested when gamification elements (levels, stages etc.) were incorporated [22]. Tangible long term benefits are also aided by gamification principles. For instance, child programmers who learnt the skill initially through games show a stronger aptitude for the skill years later [23].

Levels are often incorporated by similar existing solutions. For example Codecademy splits up coding into levels with a description of the current task on the left, with a code editor and resulting window on the right.

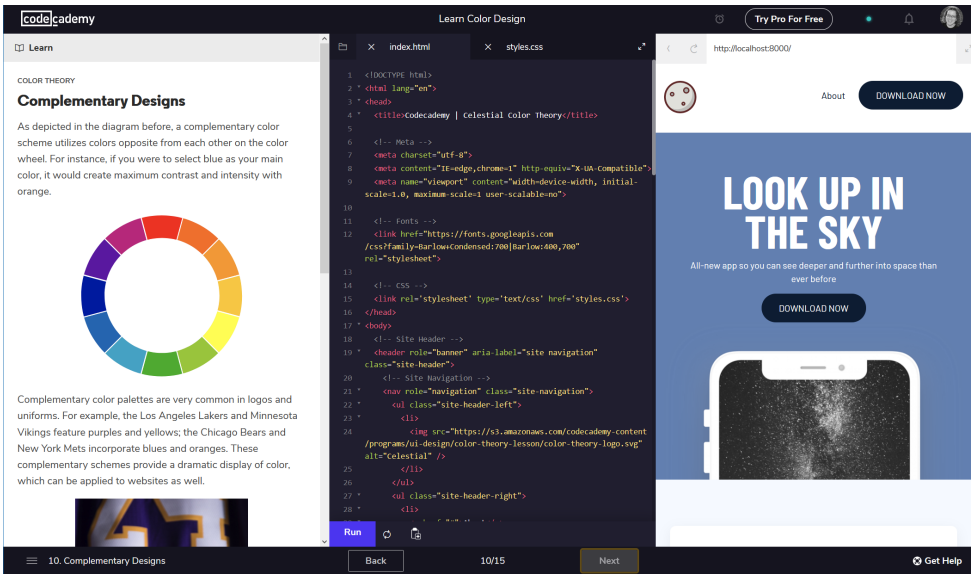


Figure 2.2: A level in Codecademy

Codecademy also uses notifications which pop up to denote achievement.

2.1.4 Gamification Combined with Constructivism

The above sections have shown the merits of both gamification and constructivism. It is therefore useful to analyse how best to combine the principles of both these concepts.

Constructivism	Gamification
Knowledge is constructed. Knowledge is built up by use of previous knowledge. Elements that are taught can be combined in a unique way by the student. This can allow each student to come up with a unique solution. Previous knowledge and solutions to past problems are used as foundations for tasking going forward. [24]	**Players can visibly see how they are doing.** Actions taken need to indicate to a user that they are moving closer to a goal. Visual indicators and sensory/tactile feedback can be used to demonstrate to players the results of their actions. These can be used by the players to judge their success. [25]
Learning is not effective when done in a passive manner. Students need to use sensory input to engage with an environment that provides tactile feedback. The effects guide their learning. [17] [24]	**Difficulty must increase.** A constant difficulty will lead to boredom. An easy first level can engage users with initial success, whilst harder later levels are a way to provide challenging settings for experimentation. These challenging settings provide the context for feelings of accomplishment upon completion. [23]
The contextual nature of learning is important. Facts and principles are not isolated. They relate to one another and can be built on top of one another to unlock further possibilities and by extension avenues of learning. [16]	**Players need to be rewarded.** It is important for users to notice positive affirmations of their actions. They should be rewarded with unlocking moves and further avenues of play through the use of games. Rewards do not necessarily have to be medals and trophies they can come in the form of changes to gameplay, visual enhancements, or advancements to the story. [22]
Task requires exploration. Tasks involve a degree of vagueness. They encourage thinking time rather than the explicit execution of an instruction. [16] [24]	

Table 2.1: The key concepts of both gamification and constructivism

From this it is clear that the inclusion of principles from both gamification and constructivism will aid the effectiveness of the teaching tool. It is important to combine the benefits of both. Whilst they have very different scopes and agendas it is clear they have aspects that compliment one another. Rather than favouring a fully gamification based approach or a fully constructivism based one (such as Scratch), benefits can be unlocked through the combination of both principles.

The inclusion of levels seems almost compulsory to gain the advantages the gamification

offers and it seems sensible that these be split into sub missions or checkpoints. This offers a superb opportunity to combine with constructivism. To do this the levels will be made slightly vague and will build on one another. This will allow the user to utilise exploration and intuition to deduce solutions to tasks. In addition it will help to promote a greater understanding as to how the elements of Hale fit together and can be used in conjunction in order to create more complex solutions.

2.1.5 MUSIC Model

The *MUSIC* model of learning [13] is one that is hugely compatible with gamification. It has shown five key conditions for successful student learning [13].

1. Feel empowered by having the ability to make decisions about some aspects of their learning,
2. Understand why what they are learning is useful for their short or long-term goals,
3. Believe that they can succeed if they put in the effort required,
4. Are interested in the content and activities,
5. Believe that others involved in their learning, such as the teacher and other students, care about their learning and them as a person.

The inclusion of gamification techniques can help fulfil these conditions of the MUSIC model and help boost student engagement.

2.2 Functional Programming Theory

2.2.1 Lambda calculus

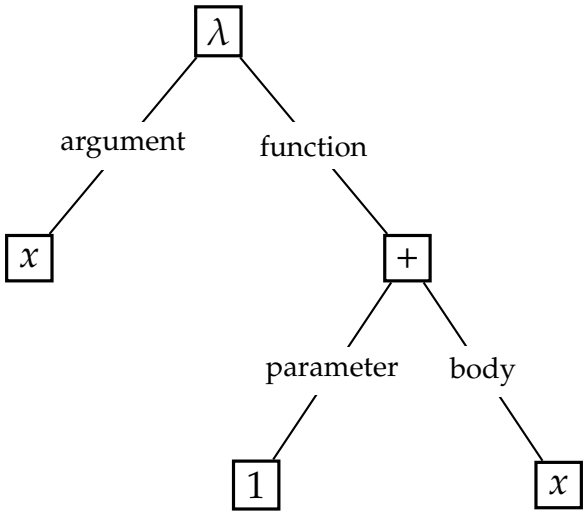
λ-calculus is a formal system of computation. In λ-calculus computation is seen by the definition of functions which are applied to arguments. The style of λ-calculus devised by Church [26] has only 3 forms an expression can take. These production rules remain unchanged and remain the only 3 production rules for terms in pure λ-calculus [27]:

$$
\begin{array}{rcll}
expression & ::= & x & \text{(variable)} \\
 & | & e_1 e_2 & \text{(function application)} \\
 & | & \lambda x.e & \text{(function abstraction)}
\end{array}
$$

For example let us take the function:

$$(\lambda x. + 1x)$$

Here the function takes a parameter, described as x and performs the action of applying it with the second argument 1 to the + function. This can be seen in the following abstract syntax tree:



λ-calculus allows functions to be returned by other functions. This therefore means they are able to take functions as arguments also.

2.2.2 Types in λ-Calculus

The typing rules of λ-calculus form the basis of the research conducted during this project into type evaluation in functional programming languages. Since λ-calculus forms the basis of almost every functional language it is the most logical foundation for a new one, especially one whose goal is simplicity.

λ-calculus allows functions to return and take values of any type. There is no type checking. As such it has been adapted to incorporate types, which is called *simply typed lambda calculus* [28].

Whilst there remain many similarities between pure and simply-typed λ-calculus the main exception lies in abstractions. Because abstractions define functions, that take an argument in a modified λ-calculus, the type of such an argument is explicitly outlined. For example in an abstraction,

$$\lambda x : \tau.e$$

τ is used to represent the expected type of the argument. T original syntax is therefore modified to create new simply-typed production rules [29].

$$
\begin{array}{lcll}
\textit{expression} & ::= & x & \text{(variable)} \\
& | & e_1 e_2 & \text{(function application)} \\
& | & \lambda x : \tau . e & \text{(function abstraction)} \\
\textit{types} & ::= & \textit{int} & \\
& | & \textit{bool} & \\
& | & \tau_1 \to \tau_2 & \text{(type conversion)}
\end{array}
$$

The type of a term in simply-typed λ-calculus can be deduced by applying *typing rules*. The typing relation $\Gamma \vdash e : \sigma$ is a way of denoting that in the context of Γ the term e has type σ. Simply typed lambda calculus has the following 4 rules:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \ (1) \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c : T} \ (2)$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma e) : (\sigma \to \tau)} \ (3) \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e2 : \tau} \ (4)$$

These rules are as follows:

1. Given x is of type σ in the context, it is true that x has type σ
2. Term constants have appropriate base types
3. If x has type σ means e has type τ in a given context then in the same context with no x, $\lambda x : \sigma . e$ has type $\sigma \to \tau$
4. If e_1 has type $\sigma \to \tau$ and e_2 has type σ in a context, then in that same context $e_1 e_2$ has type τ

Then it is true that given a typing environment Γ and expression e, if $\exists \tau$ such that $\Gamma \vdash e : \tau$, then it is true that e is *well typed* (in that context) [30] [31].

Therefore the judgement as to the type of a variable e in the context of Γ is inductively defined [31].

$$\frac{}{\Gamma \vdash n : \text{int}} \ \text{T-INT} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \ \text{T-ADD} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \ \text{T-VAR}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \to \tau'} \ \text{T-ABS} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e2 : \tau'} \ \text{T-APP}$$

Here it is seen that the expression $e_1 + e_2$ has type int if both e_1 and e_2 have type int.

In these type inferences the $\vdash$ is used to represent *entailment*, i.e. some truth or known type entails another. The top half of the inference (above the line) is an assumed premise and the information below, the conclusion.

2.3 Interpreter Design

An interpreter is split most commonly into [32] [33]:

1. *The lexer*, which turns the sequences of characters into a sequence of tokens.
2. *The parser*, which takes such a sequence of tokens and constructs an abstract syntax tree for the program.
3. *The interpreter*, which interprets an AST and computes its result.

The lexer can be constructed using tools such as ANTLR [34] [35]. This can generate the tedious parts of interpreter design from a grammar. It does however leave the evaluation and AST design still to the user.

If the lexer is written from scratch then it starts with an empty list of tokens. It goes through the string (the program) and add tokens to the list as they are identified (whitespace is ignored). A lookahead character is implemented. This allows us to decide the type of the next token [36] [37].

The parser is used to change a list of tokens into a structure that can be easily interpreted [37]. A tree is chosen due its natural ability to denote nesting and ordering. Rules specified by the grammar are used to convert the tokens into a tree. The most common type, recursive descent, uses one function for each non terminal in the grammar. It starts at the first non terminal and works its way to the lowest level. This returns an AST which represents the initial expression [38]. Function calls etc. are replaced with the function itself.

The interpreter navigates through the AST assigning variables to store results of function calls. It uses these to work up the tree, evaluating expressions as they go until the total tree is evaluated. Encounters to things like `printlines` result in the interpreter outputting the value of certain variables at that stage of the tree traversal. The interpreter is used to implement context, as variables are deleted and initialised throughout traversal. At any stage of the traversal the available variables to the interpreter denote the current context [32] [33].

λ.3

Design

Having conducted research into the background of relevant areas and completed a literature review, it was possible to begin to design Hale. The design chapter is split into 4 sections. A whole system overview details the basics of the site and how the components fit together. Then Hale's parser is examined more closely (the more technically complicated elements of the parser are focussed on in the Implementation chapter). The back end of the website is then more clearly examined in the third section, focusing on elements such as database design and calling the interpreter. Finally the design of the front end is reviewed to complete the whole system design overview.

This chapter deals with the interaction of the components and design of the system. Further specifics into their workings, such as the Hale language itself, are detailed to a greater degree in chapter 4 - *Implementation*.

3.1 Whole System

The flow of data throughout the system and the interaction between the components is detailed in figure 3.1. The user interacts with the Hale website by sending login details to the web server. These are then validated by the SQLite database. If validation is successful the web server will respond with the main webpage. Solutions to the challenges are also stored in the database when the user saves their work.

Evaluation of Hale code occurs in two scenarios:

1. User clicks run to see the results of the current contents of the editor in the console.
2. User attempts to interact with an element of the game environment conditional on Hale code.

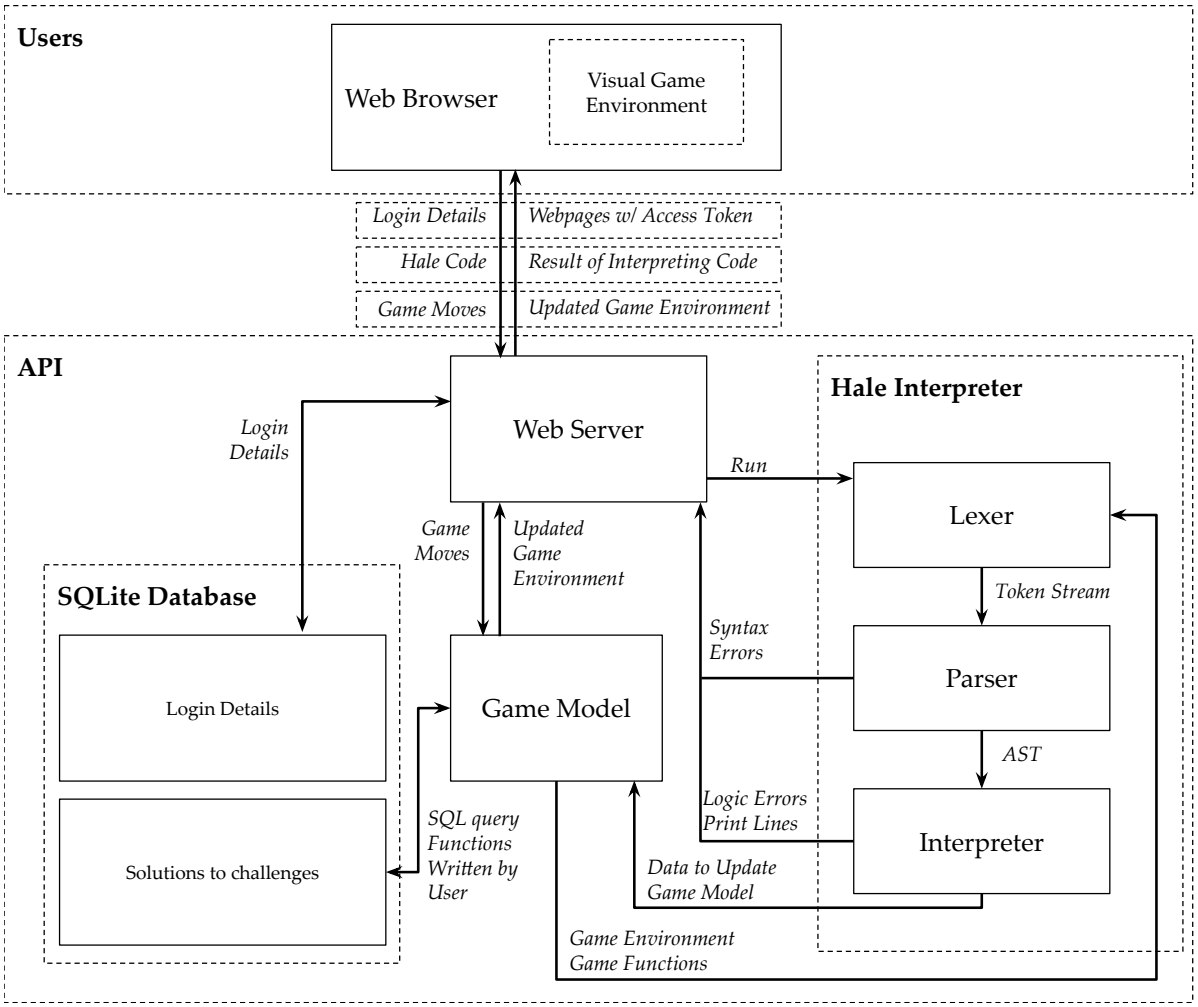


Figure 3.1: Whole system data flow diagram

Both situations are shown by the two arrows leading directly from the web server to the lexer (labelled *RUN*) and from the game model to the lexer (labelled *Game Environment & Fame Functions*)

As seen from the diagram the code is then parsed through three stages. The code is lexed into a series of tokens. It is then parsed in order to generate an abstract syntax tree. It is at this stage that syntax errors are detected and reported. If no syntax errors are found then the code is interpreted to produce either an error output or a result. The result may include either a return function, a series of `printlines` or both. In the instance where this process occurs as the result of a chess move, the new positions are returned to the chess game. The chess game then updates the board and the result is sent (via the web server) back to the user.

Otherwise (when the code is interpreted as part of a *RUN*) the result is returned as an output to the terminal. Naturally this can be null in the instance that the code contains no errors, but also no `printlines`. If any errors are present they are returned and

outputted to the terminal.

Whilst the DFD in figure 3.1 clearly demonstrates the flow of data within the API it does not clearly convey the split between the components. This is more clearly conveyed in figure 3.2. Here we see that the Flask back end communicates with a separate database and the ReactJS front end. The whole system is hosted on a reverse proxy (using Nginx). Such a proxy is also how the Yarn docs and end-to-end tests are incorporated.

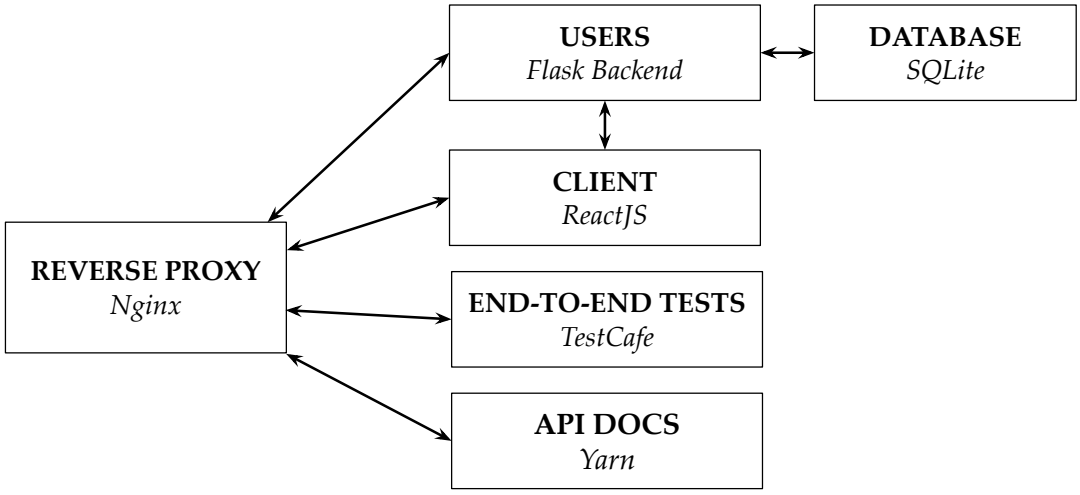


Figure 3.2: The structure of the website

Data is sent between the front and back ends via Json objects. The Json objects contain the relevant information, for example here is the Json packets used at login to send the username and hashed password to the backend where they can be verified.

```
{
    "username": "john@gmail.com",
    "hashed password": "195139e1f2363eea89dfee1dd"
}
```

In the instance that verification occurs, an access token is sent back to the front end. All such communication between the constituent parts of the API use Json.

```
{
    "username": "john@gmail.com",
    "accessToken": "AccessToken1"
}
```

3.2 The Hale Interpreter

The Hale interpreter follows a simple interpreter design with some additional modularisation. It consists of a lexer, parser and interpreter like many interpreters. It does also modularise some of these components into smaller chunks for ease of debugging. The lexer and parser are called by a main file which acts as the interpreter. A listener (which catches and logs errors) as well as a visitor (which visits nodes in an AST) are passed as parameters to these calls. The file structure is as follows:

```
/Interpreter
├── /Hale
│   ├── HaleLexer.py
│   ├── HaleListener.py
│   ├── HaleParser.py
│   └── HaleVisitor.py
└── HaleMain.py
```

3.2.1 Lexer

The lexer tokenises Hale code to return a stream of accepted tokens. Syntax errors are caught at this stage. Much of the lexer is hard coding values for tokens, the decision was taken to automate this process. Therefore ANTLR [34] was used to generate part of the lexer code. ANTLR takes a defined grammar and returns part of the lexer. It groups the tokens into symbols literals and rules:

```
literalNames = [ "<INVALID>", "'def'", "'::='", "'.'", "'\u03BB'",
 ↪  "'+'", "'-'", "'*'", "'/'", "'('", "')'", "'**'", "'<'", "'<='",
 ↪  "'>'", "'>='", "'=='", "','", "'\"'", "'?'", "':'", "'='", "';'",
 ↪  "'|'", "'&'", "'^'", "'~'", "'!'", "'['", "']'", "'{'", "'}'"]

symbolicNames = ["BOOL", "DEF", "TYPE", "DEFEQ", "FULLSTOP", "LAMBDA",
 ↪  "INT", "PLUS", "MINUS", "MULT", "DIV", "LPAR", "RPAR", "POW", "LT",
 ↪  "LTE", "GT", "GTE", "CEQ", "COMMA",  "ID", "DQUOTE", "QUESTION",
 ↪  "COLON", "EQ", "SEMICOLON", "OR",  "AND", "XOR", "NOT", "LNOT",
 ↪  "LBR", "RBR", "LCBR", "RCBR", "DOLLAR", "STRING", "COMMENT"]
```

```
ruleNames = [ "BOOL", "DEF", "TYPE", "DEFEQ", "FULLSTOP", "LAMBDA",
 ↪  "INT", "PLUS", "MINUS", "MULT", "DIV", "LPAR", "RPAR", "POW", "LT",
 ↪  "LTE", "GT", "GTE", "CEQ", "COMMA", "ID", "DQUOTE", "QUESTION",
 ↪  "COLON", "EQ", "SEMICOLON", "OR", "AND", "XOR", "NOT", "LNOT",
 ↪  "LBR", "RBR", "LCBR", "RCBR", "DOLLAR", "STRING", "COMMENT", "WS"]
```

The Hale lexer inherits from the ANTLR lexer class. When the Hale lexer is initialised the available rules of the language are then converted to a DFA. Whereby the DFA created indicates the validity of Hale code. If some Hale code reaches an accepting state when passed into the DFA, it is deemed acceptable. A simple example of such a DFA for one Hale function is seen in figure 3.3.

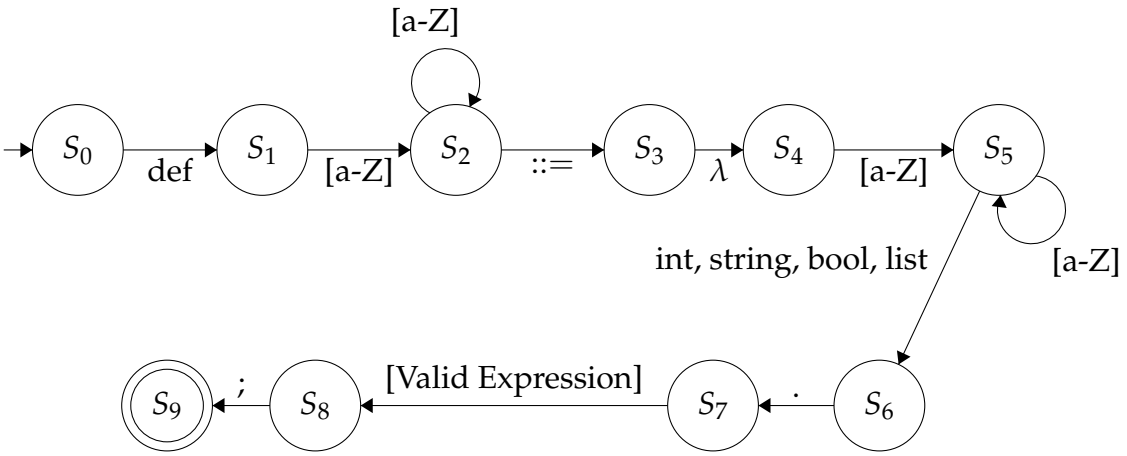


Figure 3.3: A simple DFA for a Hale function. The transition between states S_7 and S_8 has been abstracted away for simplicity. Recall a Hale function takes the form: `def` *[function name]* ::= λ *[function parameters]* : *[function type]* . *[function body]*

The DFA class is implemented with a dictionary representing the set of states. Each state in the dictionary is represented by the key (the state name) and a list of transitions to various other states. The DFA class is initiated with an initial decision and initial state.

```
class DFA(object):
    __slots__ = ('atnStartState', 'decision', '_states', 's0',
    'precedenceDfa')
    def __init__(self, atnStartState:DecisionState, decision:int=0):
        self.atnStartState = atnStartState
        self.decision = decision
        self._states = dict()
```

```
        self.s0 = None
        self.precedenceDfa = False
```

The lexer can then use the DFA for the Hale language to tokenise the code. The stream of characters can be fed into the DFA. Each transition completed can be converted to its appropriate token. For instance, in the example in figure 3.3, successfully transitioning between S_0 and S_1 will result in `"def"` being added to the token stream.

3.2.2 Listener

The listener identifies syntax errors during lexing. Therefore an instance of the listener class is passed to the lexer as a parameter. Whilst possible for the lexer to categorise and store errors, modularising this component allows for easier development. The HaleListener is an extension of the ErrorListener class

```
class ErrorListener(object):
def syntaxError(self, recognizer, offendingSymbol, line, column, msg,
 ↪  e):
    pass
def reportAmbiguity(self, recognizer, dfa, startIndex, stopIndex,
 ↪  exact, ambigAlts, configs):
    pass
def reportAttemptingFullContext(self, recognizer, dfa, startIndex,
 ↪  stopIndex, conflictingAlts, configs):
    pass
def reportContextSensitivity(self, recognizer, dfa, startIndex,
 ↪  stopIndex, prediction, configs):
    pass
```

The HaleListener is only used to identify syntax errors. Thus only a definition for syntax errors needs to be identified. The rest of the ErrorListener class is not used. This is because context errors, ambiguities etc. are all dealt with in the Hale interpreter. Thus the following class definition is given for the ErrorListener:

```
class MyErrorListener(ErrorListener):
def syntaxError(self, recognizer, offendingSymbol, line, column, msg,
 ↪  e):
    temperror = ("SYNTAX ERROR: when parsing line %d column %d:
     ↪  %s\n" % \
```

```
            (line, column, msg))
        errors.append(temperror)
```

The lexer utilises the custom listener by calling its procedures when an unexpected token is encountered. This is identified by a symbol that cannot leave a state of the DFA. To revisit the example in figure 3.3, if the letter x were encountered in state S_3 this would trigger such a procedure.

The reason for the listener adding the errors to a list, `errors`, is that the errors all need to be sent back to the front end of the webpage in a single Json object, to be displayed in the console. For example:

```
{
    "prints":["hello world", "Hale is great fun"],
    "errorbool":true,
    "errors":["SYNTAX ERROR: when parsing line 1 column 18: missing
    ↪  ';' at '<EOF>'", "SYNTAX ERROR: when parsing line 3 column 20:
    ↪  extraneous input ')' expecting ';'"]
}
```

In this example the list, `errors`, is the list to which the listener appends the syntax errors.

3.2.3 Parser

The parser takes the stream of tokens after lexing and returns an abstract syntax tree. Like the lexer and listener, the parser is a class of which the interpreter creates an instance. The syntactic analysis, which checks tokens are in an order that represents a permitted expression, is done with reference to the Hale grammar. The grammar is detailed more heavily in chapter 4. The Hale parser is a top-down parser. This means that it first looks to the highest level of the parse tree and attempts to match expressions by working its way down. It uses the rewriting rules of the Hale grammar to move downwards. The Hale parser uses a lookahead of 1, making it formally an $LL(1)$ parser.

The parser attempts to find the leftmost derivations of an input stream. It uses a statement class to construct a tree of statements. In Hale, each function has to be either a simple statement or a function definition. More simply, each line must either call a function or define a new function. As such it is easy when parsing each statement to use the `def` keyword to identify the latter.

```
def statement(self):
try:
    self.state = 41
    self._errHandler.sync(self)
    token = self._input.LA(1)
    if token in [HaleParser.BOOL, HaleParser.INT, HaleParser.LPAR,
    ↪  HaleParser.ID HaleParser.LBR, HaleParser.LCBR,
    ↪  HaleParser.STRING]:
        self.simplestmt() #code is not a function definition
        pass
    elif token in [HaleParser.DEF]:
        self.funcdef() # code is a function definition
        pass
except RecognitionException as re:
    self._errHandler.reportError(self, re)
    self._errHandler.recover(self, re)
return localctx
```

In the case of a simple statement a function call is made to the `simplestmt()` function and in the case of a function definition a similar call to the `funcdef` function.

The function definition function makes sure that the function definition matches the form required by Hale. After the initially required tokens it reaches the parameter section. It then adds the parameter of the function to the current context:

```
if _la==HaleParser.ID:
    self.match(HaleParser.ID)
    self._errHandler.sync(self)
    _la = self._input.LA(1)
    while _la==HaleParser.ID:
        self.match(HaleParser.ID)
        self._errHandler.sync(self)
        _la = self._input.LA(1)
```

After this the parser makes a few simple checks for the type of the function, full stop, parameters etc. before calling the simple statement function. This is because a simple statement is the same as the expression class of a function.

The simple statement parser attempts to match the leftmost derivation each time. In the case that this leads to a dead end it recursively backtracks and then follows the leftmost

unvisited path of the grammar. Thus it is true that the Hale parser can be considered a recursive descent parser.

A successful parse is considered when a function definition or simple statement is successfully and fully parsed to the lowest non-terminal level. At each successful parse the statements are added to the local context. In this manner after a full parse `parser.statements()` will return the full program context in the form of an AST. Code below is abridged to focus on key lines.

```
try:
    self._errHandler.sync(self)
    _la = self._input.LA(1)
    while True: #while there remain unparsed tokens
        self.statement() #the statement function is where the Hale
        ↪  code is parsed in figure 3.4
        self.match(HaleParser.SEMICOLON) #end of either func def or
        ↪  simple statement
        self._errHandler.sync(self)
        _la = self._input.LA(1)
return localctx #local context is a full AST after the full parse
```

3.2.4 Visitor

Similar to the listener, the visitor is a component of the parser that has been extracted for modularisation. The visitor evaluates a tree by visiting its nodes. It is called by the main interpreter and generates classes for functions, variables and statements. The built-in functions of Hale are automatically added to the list of available functions. The tree in figure 3.4 is the parse tree for the following Hale code:

```
def sumxy λ x y : int . x + y;
sumxy(2, 3);
```

This code will have no visible effect since the result is not printed out or stored anywhere, it is useful though to demonstrate the process of the visitor, since all the same evaluations occur. Here the visitor will follow down the left hand node to arrive at the function definition. Therefore it will need to initialise a new instance of function class. The function object takes the name, parameters, return type and a simple statement object (initialised similarly to the function) as parameters. These are then assigned as attributes. This function object is added to the list of functions.

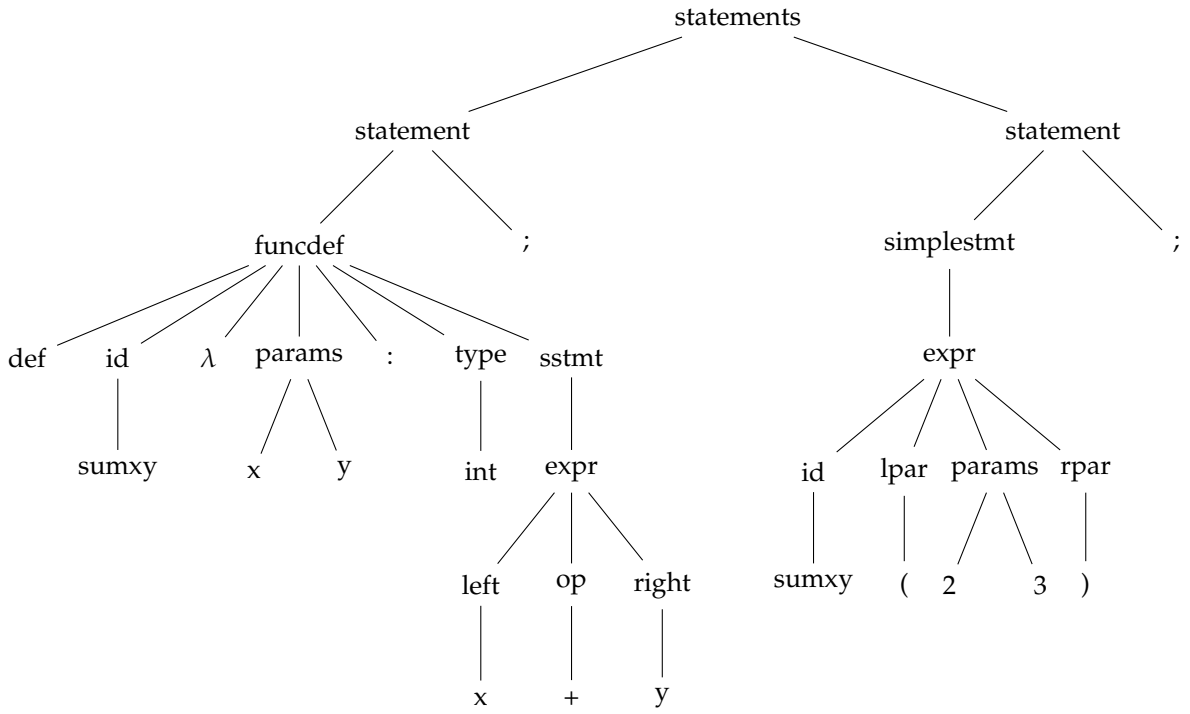


Figure 3.4: A parse tree for the `sumxy` Hale code snippet. (*sstmt* = *simplestmt*)

```
def _add_function(self, function):
    self._functions[function.name] = function
```

The visitor has then encountered the semicolon node and traverses back up to the original statements node. It then starts to visit the nodes of the right hand side of the tree. Here a simple statement needs to be evaluated. The simple statement is reduced to a single expression. This expression starts with an id. Here the visitor checks the list of functions, which now has the new function declared in it. It applies the parameters in the order they are received in the tree, x becomes 2 and y becomes 3. The result of the function's simple statement is then computed with the parameters replaced for their new values. This then leads to the result being set at the result of the expression.

3.2.5 Calling Process

The calling process is the same as the order of the subsections. The code can be seen here:

```
def parse_file(code):
    input_stream = FileStream(code)
```

```
    tree = get_tree(input_stream)
    visitor = Visitor()
    return visitor.visit(tree)

def get_tree(input_stream):
    error_listener = HaleListener()
    lexer = HaleLexer(input_stream)
    lexer.removeErrorListeners()
    lexer.addErrorListener(error_listener)
    tokens = CommonTokenStream(lexer)
    parser = HaleParser(tokens)
    parser.removeErrorListeners()
    parser.addErrorListener(error_listener)
    return parser.statements()
```

The function `get_tree` is initially called to return the tree. This takes the input stream of characters as a parameter. This function initialises the lexer and listener. The lexer takes the listener as an input and the tokens are returned. These tokens are then fed into the parser which returns the tree in the form of its statements variable. The visitor then calls its `visit` function with the tree as a parameter, which results in the evaluation of the Hale program.

3.3 Back End Design

The interpreter for Hale is written in Python. The decision to use Python was taken because of the familiarity with the language that has been built up from various course-works and projects during the last two years. It was therefore logical for the entirety of the back end to be written in Python for ease of integration.

The API's back end is built with Python Flask [39]. Being a microframework, Flask offers the basic features of a web application, however requires supplementary libraries to add plenty of functionality [40]. This enables it to be tailored to its specific purpose. Flask is more compatible with a greater number of small sub-services [41]. Whilst traffic to Hale is not expected to be large, the website does involve a series of smaller services (lexer parser interpreter etc.) and thus is more suited to Flask. Flask's greater support for libraries also led to it being chosen in this instance, since numerous of tools needed to be imported to complete the project.

There are many options for creating web servers in Python compatible with Flask. The industry standard of which is Gunicorn3 [42]. The benefit of Gunicorn3 is its ability (as

a separate entity) to manage workers. It is also lightweight with limited resources. Its second main advantage is its compatibility with NGINX [43] and Flask. This lightweight and ease of connection means it is very suitable for small-medium Flask based projects [44]. In addition the routing provided by Flask combined with Nginx and hosted by Gunicorn3 is faster than most other python frameworks for small levels of traffic [44]. As such it was chosen for the development of this project.

3.3.1 Running Hale Code

When a user runs their code using the run command (with no effect to the game). The following steps occur:

1. The user types their Hale code into the website editor. When they are happy with it, they hit the RUN button.
2. This triggers a POST request which sends the current value of the editor to the back end alongside the username for validation.
3. The backend receives a Json object containing the code and username. After it validates the user it passes the Hale code into the interpreter. This returns three values. The first two are lists (the print lines and errors). The third is the return value (if any).
4. The backend sends a Json object back to the React front end containing both lists and return values.
5. The front end displays the information. If there are any errors these are displayed to the console, otherwise the printlines are all displayed. (This process is detailed more clearly in the front end section).

```
@app.route('/write', methods = ['POST'])
def write_file():
    code = request.get_json()
    code_to_write = code['code']
    code_to_write = code_to_write.replace(";", ";\n")
    return, prints, errors = HaleMain.runInterpreter(code_to_write)
    errorbool = False
    if(len(errors)):
        errorbool = True
```

```
    return {'return': str(test), 'prints':prints,
     ↪  'errorbool':errorbool, 'errors':errors}
```

3.3.2 Saving Code for the Challenges

When the user hits save, it is expected that the code is saved so that its effects influence the game. The process is as follows:

1. Similarly to running Hale code, when the button is pressed the contents of the editor and username are sent via Json to the back end.
2. Received code is extracted from a Json object, split into lines and stored in a list.
3. The backend then iterates through the list. Upon encountering one that starts with *def* + *x*, where *x* is one of the challenge function names, the following step occurs:
   (a) The backend executes an UPDATE function to the table of the task relating to the challenge. This causes the new function to be stored in that cell of the database.
4. The backend sends a boolean back in a Json packet - True indicating success False indicating an error occurred.

```
task1 = ["spaceForward", "equalPos", "pieceAt", "pawnMoveForward"]
for code in array_of_code:
    found = False
    for function in task1:
        if (not found) and code.startswith('def ' + function):
            found = True
            with sqlite3.connect("APIData.db") as con:
                cur = con.cursor()
                query = "UPDATE TASK1 SET "+ function + " = '" + code
                 ↪  + "' WHERE Username = '"+ username +"';"
```

3.4 Database

A database is necessary in the development of this API, because an API is inherently *stateless*. Therefore each request needs access to a database in order to allow the service

to fetch the data required by the context of the request. For example the context would be the user's unique identifier, which would allow the user to submit code, so that it can be later identified as belonging to them.

SQLite [45] was chosen for the database. This is because of SQlite's high performance [46]. SQLite offers $O(\log n)$ query complexities as opposed to $O(n)$ offered by more traditional database systems [47]. Alongside this, data retrieval and queries are more robust in SQLite than other traditional file storages [45] [46] .

Another reason for using SQLite is that the traffic for the website is not expected to be very large. This makes SQLite the appropriate choice since it has been shown to work effectively for small levels of traffic [48].

3.4.1 Database Structure

The database needs to facilitate storing and registering new users as well as their solutions to each of the tasks' problems.

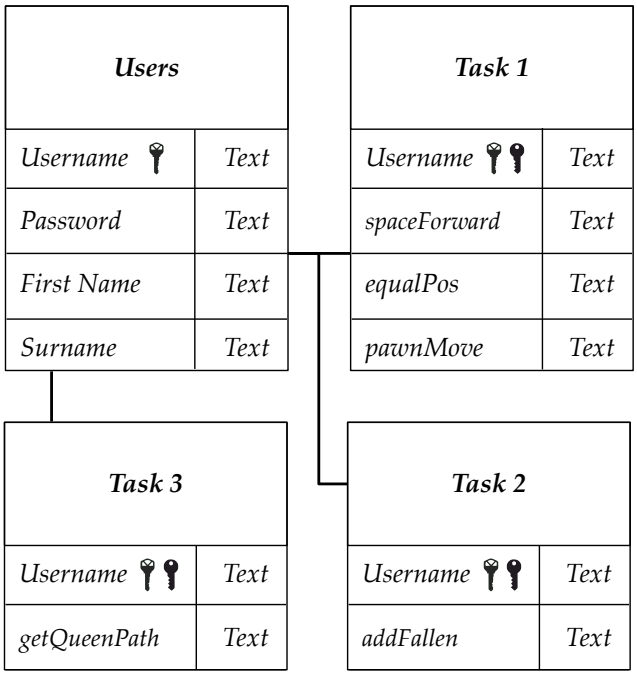


Figure 3.5: The structure of the database

Figure 3.5 shows the database design. The *Users* table contains the unique identifier (*The Username*) which is the primary key. It also acts as the foreign key in the other 3 tables linking the solutions of a user. The *Password* field is used to store the hashed password which is compared to the hashed entry of the password field on login. The

passwords are hashed in order to substantially increase (in the region 4000-20,000 times larger) the cost of a brute force attack [49].

The other tables all store the solutions to each function in the game. The database is fully normalised to the degree of the third normal form.

3.5 Front End Design

The decision was taken to build the front end with React. It was clear that a LAMP stack web framework was not sensible since the combination of unrelated languages and weaknesses of PHP make it a challenging decision for development [50]. The use of Virtual DOM by React means that it has a far less complex lifestyle, better scalability and less UI lag when compared to MEAN stack frameworks [51]. The DOM of a website refers to its Document Object Model, or structured text. Each component of an HTML webpage is represented by a node of this DOM. The Virtual DOM is effective at boosting the performance of a web framework since it is detached from the browser. This means it is possible to alter and adjust it without any need to render the page again after such a change. After a set of manipulations the result can be transferred to the main DOM which causes a re-render. This means that re-renders happen less frequently and thus improves overall performance [51].

3.6 Pages

The user needs to be greeted with a login page. This must be redirected to from every page given the lack of a valid authentication cookie. It is critical the user authenticates themselves and login properly. This is to protect users data as well as to allow the website to function properly.

After the user logs in they should be redirected to their dashboard. The dashboard is where they write Hale code and see its effects as well as play the game. A preliminary design layout for this is shown in figure 3.6.

The website requires a documentation page that includes all the information about how to use Hale. This needs to be fairly basic from the start and increase in complexity for more challenging topics.

The profile page needs to show the user their progress. This is in line with the MUSIC model of learning [13] since it promotes the user receiving tangible feedback for their actions. From the profile page a user is able to either sign out or return to their dashboard. A design for the profile page is seen in figure 3.7.

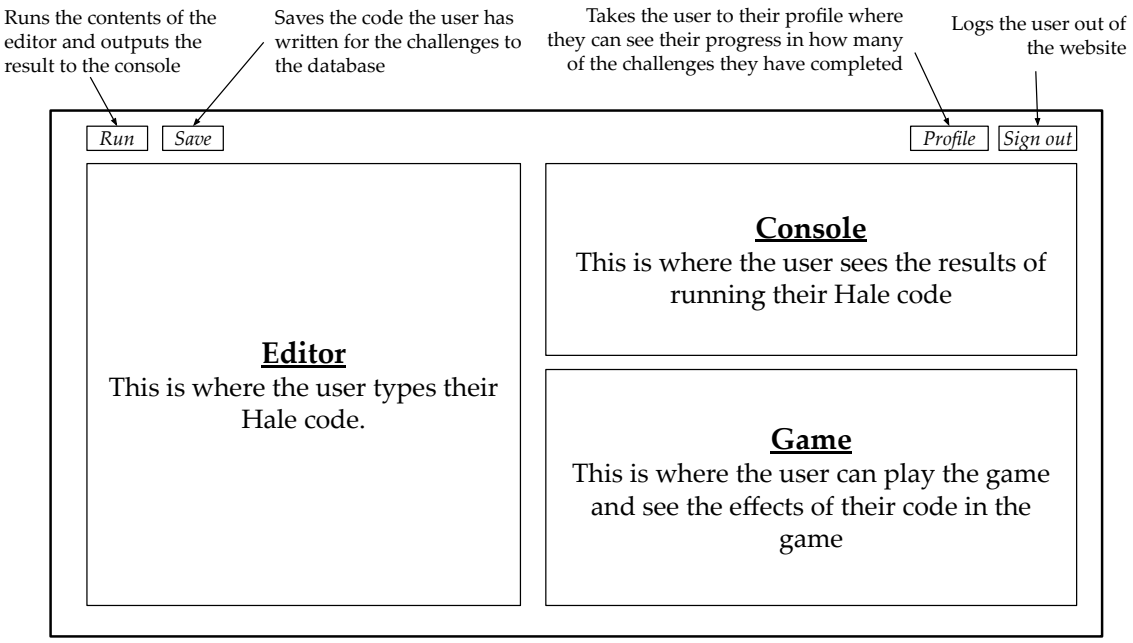


Figure 3.6: An initial design for the layout of the dashboard page

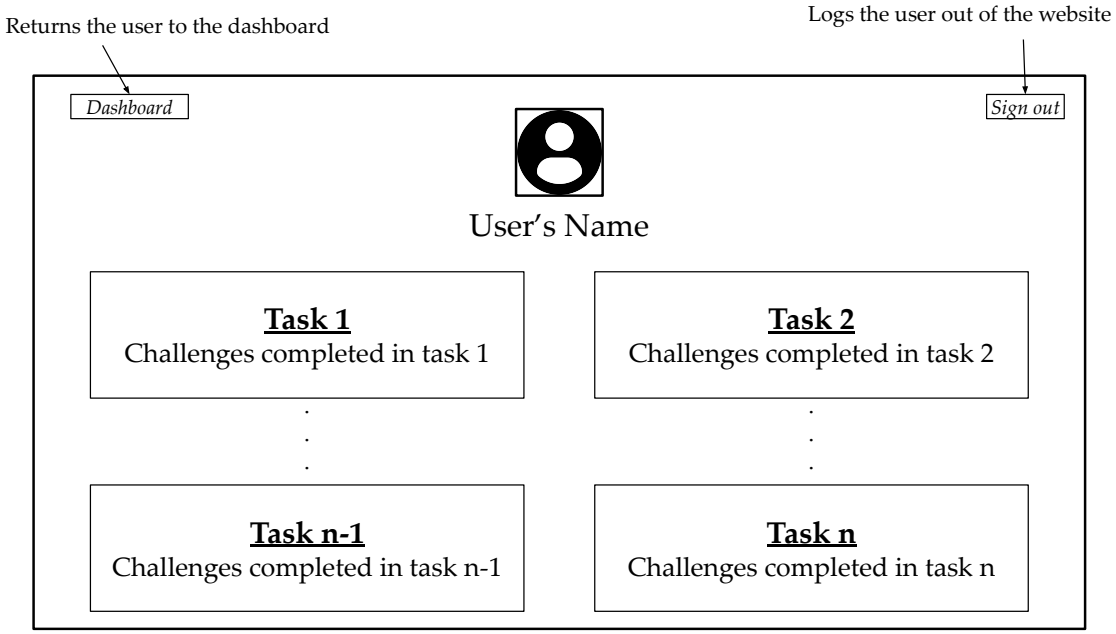


Figure 3.7: An initial design for the layout of the profile page

3.7 Game Design

Initially development used a game that worked in real time, for instance Snake. However latency issues arose from this.

Chess was therefore chosen as the game which would be used to teach the students. The tool was designed to teach the students by requiring them to code the rules of the game of chess. This meant that initially the pieces would not all behave as expected and through completion of a series of challenges and levels the user would be able to piece together a fully working game of chess. The decision to use chess was taken, since the game needed to be turn based due to the need to avoid latency issues. Each time a user wanted to interact with part of the game that they had altered, the Hale code they had written had to be interpreted.

3.7.1 Constructivism

In order to benefit from the research conducted, it was necessary that the game adhere to constructivism principles. It was very important that a reasonable balance between constructivism and gamification was struck.

Constructivism might at first seem at odds with the idea of chess. Chess follows strict rules and does not allow for much exploration and self learning. The first clear decision made here is that the game would not tell a user if their code did not achieve the desired effect. Instead the chess game would be playable with the code they had written. This immediately turned the chess environment into more of a sandbox. Experimentation was more widely encouraged and the user could play around with the effects in game. They could follow the challenges strictly but they could also use the information within the challenges to mess around and see what creative rules they could add to the chess game.

This creative freedom, whilst a strength as far as constructivism was concerned, also had to factor in gamification. Therefore, whilst experimentation was allowed, the user would also be able to see their tangible progress with regards to the actual challenges on the profile page.

In order to allow for proper creative freedom, should the user wish to use it, the challenges needed to provide the user with a certain degree of support and understanding as to how the chess game works or how their Hale code influences it. One of the key predicates for successful constructivism is familiarity with the workings of the environment [6][7].

Gamification also benefits from sub levels to aid the user along the way [21] [22]. Therefore the first level was split up into small chunks. However, this is the opposite of constructivist policies and the area they disagree most on. Therefore in order to balance both, later levels were not split down for the user. They could be solved by being broken down but were left to the user to work out how to do that. This decision meant that

users were guided through their first few Hale functions but left to explore when it came to writing complicated algorithms.

3.7.2 Challenge 1

Challenge 1 introduces the user to the game of chess and how their Hale programs are to be interpreted. In order to provide the necessary understanding for successful exploration and experimentation at a later stage, it starts by introducing how the pieces' positions are counted. The grid squares are numbered from 1 to 64. A piece's position is documented as one of these numbers. Therefore a direct move forward for a piece is an increase of 8 to its position variable.

It is therefore necessary for the first challenge to familiarise the user with this movement system before later harder challenges can be introduced. The piece with one of the simplest movement conditions is the pawn. As such this first challenge is to move the pawn forward a single space.

In line with gamification principles this is split into sub levels to allow for more rewarding learning. The first is to define a function that can return an int 8 less than one passed as a parameter. This represents the position one in front of the pawn.

The next step will be to ask the user to define a function that takes two positions and returns true if they are equal and otherwise false. This introduces the user to booleans and conditionals in Hale.

The third step is to write a function that takes a pawns position and a boolean. It returns a list of the available spaces the pawn can move to. The user is encouraged to use their first function here in the definition.

The website will use all these functions to actually influence the pawns. The program also uses their equivalence function to generate the boolean fed into the final function. The game allows the pawn to move to any square returned in the list from this function.

3.7.3 Challenge 2

Challenge 2 is to add the functionality of fallen soldiers. This means the pieces that appear on the side of the board once they are taken. For this challenge, now that the user has been introduced to the program, gamification gives way to a more constructivism approach. The user is encouraged to use print lines to test their code first and to break up their solution but they are not specifically told in what manner.

The task requires the user to identify if two pieces occupy the same square. Their solution is triggered each time a piece is moved. If this is the case a new function needs

to be defined, which appends the piece id to a list of fallen pieces. This task introduces the user to lists in preparation for the next task.

3.7.4 Challenge 3

The first two challenges introduced the semantics of Hale as well as lists - a key feature of functional programming. The third challenge aims to show how functional programming approaches a task that would be solved with a for or while loop in an imperative language.

The function is to plot the path a queen takes between 2 squares. It takes a start and end space as well as increment and a current path (originally the empty list). It needs to add all the spaces to that list that the queen would travel until it reaches the end destination. This requires the use of recursion. The program suggests that recursion should be used but does not dictate in what manner. This promotes a more constructivist approach to the problem.

3.7.5 Future Challenges

There is a large scope for future challenges. These are detailed in the future work section of the evaluation.

3.8 Software Development Methodology

Agile methodology is implemented in the development of this project. Agile methods lend themselves to continuous improvement and as such are more responsive to changes [52]. A goal of the CS310 module is to encourage innovation and creativity. As such allowing for the project scope and plan to change with time is in line with these principles. More specifically, SCRUM techniques were used since they focus on and benefit from iterative development. Projects developed with SCRUM often use team leaders and team sprints to make the most of the methodology, however this was adapted to include individual sprints. No stand up or team meetings were required, which saved time.

Tools used for development were chosen to make the process streamlined and as clean as possible. Git was used for version control, since it was familiar and well known. Version control is essential to allow the rollback of broken or vulnerable features. This was synced with a Github repository. Records of changes made could be seen in the commit logs. Visual Studio Code was used to edit and write code, since it also benefits

from a strong familiarity. Its multi-purpose nature allowed LaTex files to bed edited in it alongside the code. An initial timetable of the project was made. This was designed to be slightly ambitious in order to provide a safety buffer in case development fell behind schedule. This is in accordance with the risk assessment in table 3.1. The timetable is found in appendix C.

3.8.1 Risk Assessment

Due to the scope and scale of the project it was necessary to identify and mitigate all potential risks. Table 3.1 shows the risk assessment conducted as part of the project management side of development.

Summary	**Risk Level**	**Mitigation**	**Residual Level**
A third-party cloud service being used becomes unavailable	Low	Offline backups regularly made with clear commit titles	Low
Underestimation of task complexity and scope or debugging ease	High	Internal deadlines earlier than necessary to allow for leeway	Medium
Personal illness or an unforeseen personal event encountered	Low	Timetable is more ambitious but also therefore features more leeway	Low
Outage of cloud based server of choice	Medium	Website can be simulated locally and development still continue	Low
Lack of data due to small number of test subjects. Therefore inability to perform statistical analysis on data	High	Request for message to be put out to functional programming labs	Medium

Table 3.1: Risk assessment conducted at project outset

$\lambda.4$

Implementation

This chapter details the further specifics of the implementation of the design choices. For example the design and grammar of Hale or the database schema and precise queries.

4.1 Language Design

The first step to design Hale required a formalised grammar. This was created using a top down approach. This meant the starting node chosen was *statements*. This is because in its simplest form any language is a series of statements. Therefore *statements* is defined as one or more *statement*. Each separated by a semi-colon.

$$\textit{statements} \quad ::= \quad (\textit{statement}\ \textbf{semicolon})+$$

Then it was necessary to begin to define what was meant by a statement. Hale allows two types of statement, simple statements and function definitions. Function definitions are (as suggested by their name) the declaration and definition of a new function. A simple statement is a call made to zero or more functions. This definition of simple statement will be explored further shortly.

$$\begin{array}{lcl} \textit{statements} & ::= & (\textit{statement}\ \textbf{semicolon})+ \\ \textit{statement} & ::= & \textit{simplestmt} \\ & | & \textit{functiondef} \end{array}$$

The function definition is then next. This is because it contains a form of expression. The definition of the function is in essence any expression that can call a function etc. Therefore the expression of a function is also a simple expression. Since most of function definition in Hale follows a similar pattern,

`def` *[function name]* ::= λ *[function parameters]* : *[function type]* . *[function body]*

it can be easily defined in a grammar, with the function body represented by a simple statement. The function parameters are 0 or more identifiers. Thus represented by *id**.

statements	::=	(*statement* **semicolon**)+
statement	::=	*simplestmt*
	|	*functiondef*
functiondef	::=	**def id eq lambda id* colon type fstop** *simplestmt*

Things become more complicated as simple statements begin to be defined. To begin we categorise all simple statements into two categories, expressions and assignments. An assignment is identical to other languages. It is one or more id followed by an '=' and then any valid expression.

statements	::=	(*statement* **semicolon**)+
statement	::=	*simplestmt*
	|	*functiondef*
functiondef	::=	**def id eq lambda id* colon type fstop** *simplestmt*
simplestmt	::=	*expression*
	|	*assign*
assignment	::=	**id** (**comma id**)* **eq** *expr*

Digressing further it is essential to formally define an expression. Like simplestmt this starts simple and becomes more complicated. Expressions are categorised into two groups, comparisons and conditionals. Conditionals are Hale's equivalent of if statements. They take the form:

[condition] ? *[expression if condition met]* : *[expression if condition not met]*

The conditions are themselves simple statements since they can be any valid expression besides a function definition.

statements	::=	(*statement* **semicolon**)+
statement	::=	*simplestmt*
	|	*functiondef*
functiondef	::=	**def id eq lambda id* colon type fstop** *simplestmt*
simplestmt	::=	*expression*
	|	*assign*
assignment	::=	**id** (**comma id**)* **eq** *expr*
expr	::=	*comp*
	|	*conditional*
conditional	::=	*comp* **qmark** *simplestmt* **colon** *simplestmt*

Comparisons are slightly more complicated. The reason that the other form an expression can take is a comparison is that comparative operators (<, > $\geq$, $\leq$ etc.) have the lowest precedence. Boolean operators come next, followed by plus and minus, then multiply and divide, and finally exponential. Since Hale's parser is a recursive descent parser, it is necessary that the operations with highest precedence appear as far down the tree as possible. For example the expression,

$$x - 1 > 2 + 3^2 * 4$$

needs to be evaluated as

$$((x - 1) > (2 + ((3^2) * 4)))$$

This is seen in figure 4.1, which illustrates why the lower order preferences need to appear higher in the grammar definitions in order to be evaluated last by a recursive descent parser.

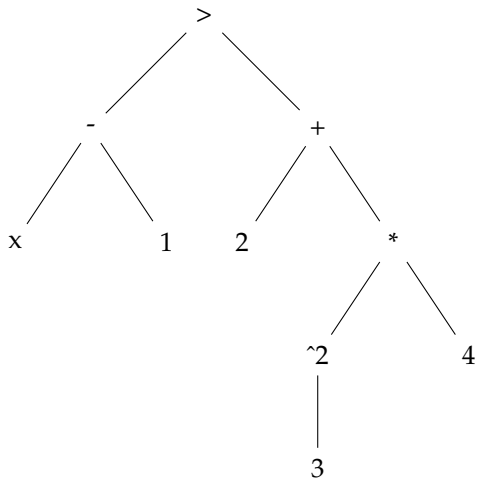


Figure 4.1: Precedence tree for expression $x - 1 > 2 + 3^2 * 4$

Based on this precedence we can define a comparison as a right expression and then 0 or more comparative operators followed by a left expression. In this way the following are all valid comparisons:

$$x + 1$$

$$x + 1 > y$$

$$x + 1 > y \geq z * 2$$

NOTE: Not all of these expressions are evaluable. The third isn't since the result of the first comparison will return a boolean which cannot be compared to the second equation. But they will all lex into a valid token stream.

In a similar way to the fact that the comparison operators needed to go first in the grammar, we treat the left and right expressions in a comparison. We pick the next object by order of precedence, which are boolean operators.

statements	::=	(*statement* **semicolon**)+
statement	::=	*simplestmt*
	|	*functiondef*
functiondef	::=	**def id eq lambda id* colon type fstop** *simplestmt*
simplestmt	::=	*expression*
	|	*assign*
assignment	::=	**id** (**comma id**)* **eq** *expr*
expr	::=	*comp*
	|	*conditional*
conditional	::=	*comp* **qmark** *simplestmt* **colon** *simplestmt*
comp	::=	*boolean* ((**lt** | **lte** | **gte** | **gt** | **gte** | **ceq**) *boolean*)*

Boolean expressions are defined identically - a left expression (this case a sumsub - an expression containing + or -) and then 0 or more boolean operators followed by sumsubs. This repeats for multiply and divide expressions (multdivs) and exponential expressions (exps):

statements	::=	(*statement* **semicolon**)+
statement	::=	*simplestmt*
	|	*functiondef*
functiondef	::=	**def id eq lambda id* colon type fstop** *simplestmt*
simplestmt	::=	*expression*
	|	*assign*
assignment	::=	**id** (**comma id**)* **eq** *expr*
expr	::=	*comp*
	|	*conditional*
conditional	::=	*comp* **qmark** *simplestmt* **colon** *simplestmt*
comp	::=	*boolean* ((**lt** | **lte** | **gte** | **gt** | **gte** | **ceq**) *boolean*)*
boolean	::=	*sumsub* ((**or** | **and** | **xor**) *sumsub*)*
	|	**not** *sumsub*
sumsub	::=	*multdiv* ((**plus** | **minus**) *multdiv*)*
multdiv	::=	*exp* ((**mult** | **div**) *exp*)*
exp	::=	*atom* ((**pow**) *atom*)*

Having defined expressions down to atom, it is necessary to determine what form an atom can take. The most common in a functional language should be a function call. A function call can be pattern matched to the sequence

[function name](parameter$_1$, parameter$_2$, $\cdots$, parameter$_n$)

It could also be more simply just a *value* or in other words an int, bool, string, or variable name (id). It could be a bracketed statement since these would take precedence over the order of set out by the grammar. Or it could be list, which is defined as,

(expression$_1$, expression$_2$, $\cdots$, expression$_n$)

statements	::=	(*statement* **semicolon**)+
statement	::=	*simplestmt*
	|	*functiondef*
functiondef	::=	**def id eq lambda id* colon type fstop** *simplestmt*
simplestmt	::=	*expression*
	|	*assign*
assignment	::=	**id** (**comma id**)* **eq** *expr*
expr	::=	*comp*
	|	*conditional*
conditional	::=	*comp* **qmark** *simplestmt* **colon** *simplestmt*
comp	::=	*boolean* ((**lt** | **lte** | **gte** | **gt** | **gte** | **ceq**) *boolean*)*
boolean	::=	*sumsub* ((**or** | **and** | **xor**) *sumsub*)*
	|	**not** *sumsub*
sumsub	::=	*multdiv* ((**plus** | **minus**) *multdiv*)*
multdiv	::=	*exp* ((**mult** | **div**) *exp*)*
exp	::=	*atom* ((**pow**) *atom*)*
atom	::=	*value*
	|	**lpar** *simplestmt* **rpar**
	|	**id lpar** *params*? **rpar**
	|	*list*
params	::=	*expr*(**comma** *expr*)*
list	::=	**lpar** (*expr* (**comma** *expr*)*)? **rpar**
value	::=	**int**
	|	**string**
	|	**bool**
	|	**id**

Figure 4.2: The final Hale Grammar, excluding the definition of **literals** (e.g. **gt** ::= '>'). The complete grammar with literals can be found in appendix A.

4.2 Language Capabilities

4.2.1 Higher Order Functions

Higher order functions are one of the biggest advances to programming introduced by functional programming [53]. They have proved so useful that they have been introduced to Python, C and other major imperative programming languages [53] [54].

At their most basic level in typed λ-calculus, higher order functions take a function as argument and can be described as having the type $(\tau_1 \to \tau_2) \to \tau_3$.

It was necessary to include the capability to use higher order functions in Hale. This is because it is a key benefit of the functional programming paradigm and one of the main ways to create complicated and more useful programs. Since Hale's job is to introduce a user to functional programming, it seemed critical to include higher order functions in Hale.

To examine higher order functions in Hale further, consider the following program where the function call, `printFibs(fib, n)` prints the first n fibonacci numbers.

```
def fib ::= λ x : int . x <= 1 ? x : fib (x-1) + fib (x-2);
def printFibs ::= λ f x : int . x > 1 ? printFibs(f, (x-1)) &
↪  printline(f(x)) : printline(f(x));
printFibs(fib, 10);

> 1
> 1
> 2
> 3
> 5
> 8
> 13
> 21
> 34
> 55
```

As seen `printFibs` takes `fib` as an input. Looking at line 1:

```
def fib ::= λ x : int . x <= 1 ? x : fib (x-1) + fib (x-2);
```

It is clear `fib` is a function. It is type $int \to int$, since it takes an integer input x and returns the result of a series of addition operations leading down to 1. `fib(x)` will return the x^{th} fibonacci number.

If $x \leq 1$ the if condition results in the function returning x, otherwise a recursive call is made to return the addition of `fib(x-1)` and `fib(x-2)`.

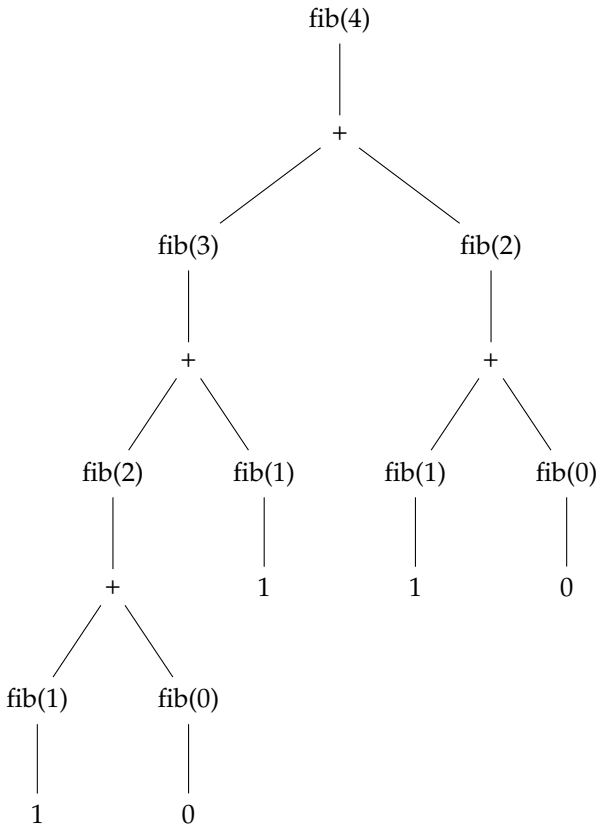


Figure 4.3: The sequence of calls for the code `fib(4)`

The higher order function `printFibs` is then defined:

```
def printFibs ::= λ f x : int . x > 1 ? printFibs(f, (x-1)) &
 ↪  printline(f(x)) : printline(f(x));
```

`printFibs` takes a function and an integer x as parameters. It does not return anything but it does print out the results of calls. In Hale we use type `do` to represent functions that have no return type, similar to void in Java.

Therefore we can describe this function as having type $((int \rightarrow int) \rightarrow int) \rightarrow do$.

Higher order functions are implemented in Hale at the atom stage of the program evaluation. When the current context is of type id the interpreter knows that it is either a variable name or a function name, since both are represented by the same id terminal in the grammar. Therefore to determine the difference, the program attempts to evaluate it first as if it were a variable. If this isn't possible it knows the id needs to be evaluated as

a function. To do this the name is extracted from the id and the parameters from the local context. Then the result of the function call is returned by the expression. This is seen by the code below:

```
func_name = context.ID().getText()
params = self.visit(context.params())
return self.get_var(func_name).call(self, params)
```

The call here on the second line evaluates the parameters by visiting the terminal node. The parameters of the current context are used to evaluate the state of the passed parameters. Then the function can be called. The variable storing the function is retrieved using the `get_var` function. This has a call procedure which is executed with the values of the parameters that were just evaluated, resulting in fully working higher order function capabilities.

4.2.2 Recursion

As demonstrated in the fibonacci sequence used in section 4.2.1, Hale is capable of recursively defined functions. Recursion is key in functional programming as an absence of for and while loops often require the use of case-based recursive iteration. Another example of recursion in Hale is demonstrated by the following code which prints the elements of a list one by one.

```
def plines ::= λ list : do . list == [] ? printline("") :
↪  printline(listhead(list)) & plines(listtail(list));
plines([1,2,3,4,99,5]);

> 1
> 2
> 3
> 4
> 99
> 5
```

Recursion is evaluated similarly to function calls. The recursion depth of Hale is the same as the recursion depth of Python.

4.2.3 Dynamic Typing

Hale is dynamically typed. The code used in section 4.2.2 can be slightly modified to demonstrate this. It relies on the fact that the function `printline` can take either a string

or an integer. Therefore if a list of both types is fed to the `plines` function, the result would appear as such:

```
def plines ::= λ list : do . list == [] ? printline("") :
 ↪  printline(listhead(list)) & plines(listtail(list));
plines([1,"string2",3,"string4",99,"string5"]);

> 1
> string2
> 3
> string4
> 99
> string5
```

This is dynamically typed because a list of multiple types can be fed into a function and since that function can take either type as an input no error is triggered. However, if the function were to be modified to include an arithmetic operation such that the result could not be evaluated with a string, an error would trigger.:

```
def plines ::= λ list : do . list == [] ? printline("") :
 ↪  printline(listhead(list)/2) & plines(listtail(list));
plines([2,6,24,"string4"]);

> 1
> 3
> 12
> Logical Error: Division can only be applied to type int. You have applied
 ↪  it to 'string4' which is type String
```

In this sense type errors are only evaluated during the execution of operations rather than in the passing of parameters or definition of variables. Therefore Hale is fully dynamically typed.

4.3 Base Functions

Hale has a number of base functions which can be used initially and without declaration or definition. No function can be defined with these names.

Base functions are implemented in classes. Each base function has its own class, which inherits from a generic `baseFunction` class. The class assigns the name and call of the

Function	Effect
`printline`	Prints the parameter passed to the console. In the instance where the parameter is an expression, it prints the result of its evaluation.
`listsplit`	Splits a list into its first element and the tail of the list.
`listhead`	Returns the first element of a list.
`listtail`	Returns a list without its first element.
`listinit`	Returns a list without its last element.
`listend`	Returns the last element of a list.
`listlength`	Returns the number of elements in a list.
`map`	A higher order function that takes a function of type $A \rightarrow B$ and applies it to every element in a list of type A to return a list of type B. It is possible for $A = B$ in this context.

Table 4.1: The list of base functions in Hale

function. The `call` procedure is to be overwritten by each inheritance of the class, with the procedure for that specific base function:

```
class BaseFunction:
    def __init__(self, name):
        self._name = name
    @property
    def name(self):
        return self._name
    def call(self, context, param_values):
        return 1
    def __str__(self):
        return f"<function {self._name}>"
    def __repr__(self):
        return str(self)
```

4.3.1 PrintLines

Section 4.4 shows how the program displays either errors or printlines. Printlines are displayed when no errors are encountered.

When `printline` is called its parameter is evaluated. For instance in the code `printline(4*3)` the expression $4 * 3$ is evaluated. This evaluated parameter is then appended to a list. This list is returned at the end of program evaluation.

4.3.2 Listsplit

`listsplit` is implemented by taking the list (the parameter), confirming it is of type list and that it is of minimum length 1. Then the first element and the tail of the list are returned:

```
if isinstance(p, list) and p:
    return (p[0], p[1:])
errors.append("LOGICAL ERROR: Can't use 'listsplit' on empty list")
```

4.3.3 Listhead and Listend

Both functions are very similar. `listhead` retrieves the first element of a list and `listend` the last. Both return a single element. As long as the list length is greater than 0 both functions return a list.

```
if p:
    return p[0]
errors.append("LOGICAL ERROR: Can't use 'listhead' on empty list")
```

4.3.4 Listtail and Listinit

Similar checks are used for these two functions. The list needs to be of length 1 or more. If it is 1 the result will always be the empty list. They both however return a list not an element.

```
if p:
    return p[1:]
errors.append("LOGICAL ERROR: Can't use 'listtail' on empty list")
```

4.3.5 Listlength

The `listlength` function takes a parameter p. It checks if it is of type list and returns its length. Otherwise an error is appended.

```
if isinstance(p, list):
    if p:
        return len(p)
    if p == []:
```

```
            return 0
    errors.append("LOGICAL ERROR: Can't use 'listlength' on a non list")
```

4.3.6 Map

Mapping is key principle in functional programming. It is an example of a higher order function (one that takes a function as an input). It applies a function of type $A \rightarrow B$ to each element of a list of type A to return a list of type B. The idea of mapping, whilst not necessarily complicated at a basic level, is a key method for optimising more complicated fully functional code [55]. Therefore it is a necessary addition to Hale in order to teach a base understanding for critical functional programming concepts.

Map in Hale is implemented through the use of a function call on all elements from a list, within the bounds of the current context.

The function is isolated, since we know it is the first parameter.

```
f = param_values[0]
```

Following this, the list that the function is being applied to is extracted. The return type is a list where the function call is applied to each element.

```
return [f.call(context, [e]) for e in listParam]
```

Hale is dynamically typed, which means type errors are evaluated during expressions where variables are actually used rather than at calls or declaration. For instance a list may not contain only one type. It is also possible to use a map of type $A \rightarrow B$ on a list of type C. An error will be triggered at the first call of $f(C[0])$ rather than at the call of map. The implementation does not ever need to trigger or store an error since any potential error will be caught on the first function application.

4.4 Running Programs

The Monaco Editor [56] was imported to the website. This was because it provided suggestions based on previously typed code. When a user wishes to run a program they click the run button.

```
<button onClick={buttonpress} class="btn btn-lg btn-success
 ↪  float-right" type="button">Run</button>
```

This triggers the `buttonpress` function, indicated by the code snippet above. The `buttonpress` function creates a request of type post with the body containing the current value of the Monaco editor.

```
const requestOptions = {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ title: editorRef.current.getValue() })
}
```

This is then sent using a `fetch` request to the `/write` app route. The data received back is then interpreted. After the POST request is received at the back end, the code is extracted from the Json. It is then passed into the Hale interpreter. This action clears the lists storing the print lines and errors. The program is interpreted, appending any print lines, return values and errors to their respective lists. These lists are then returned from the interpreter.

To detect if there are any errors, the program checks the length of the errors list. If so the `errorbool` is also set to true.

```
if(len(errors)):
    errorbool = True
return {'prints':prints, 'errorbool':errorbool, 'errors':errors}
```

The Json object is received by the front end. If the `errorbool` is true then the `result` variable is set to the list of errors, else it is set to the list of printlines. `result` is what is printed to the terminal.

```
fetch('/write', requestOptions)
.then(res => res.json()).then(data => {
    if(data.errorbool == false){
        setResult(Object.keys(data.prints).map((key) =>
        ↪  data.prints[key]))
    }
    else{
        setResult(Object.keys(data.errors).map((key) =>
        ↪  data.errors[key]))
    }
});
```

The contents of `result` (either errors if there are any, otherwise print lines) are displayed line by line in the console using the `map` function in javascript.

```
<div>
{result.map(value => <p class="line4
 ↪  terminalp">{value.toString()}</p>)}
</div>
```

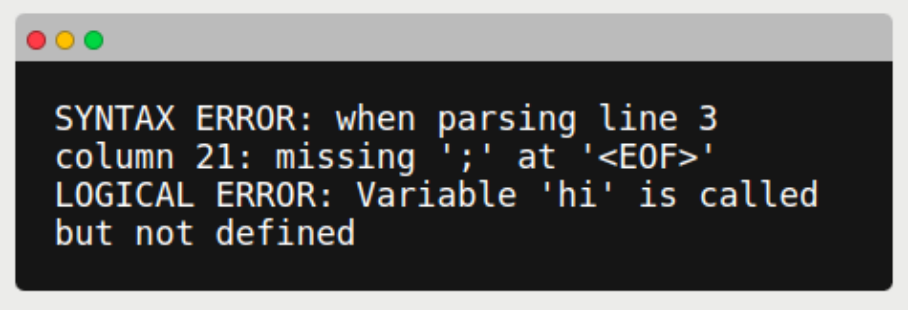


Figure 4.4: The terminal displaying the contents of `result` line by line - in this case 2 error messages

4.5 Error Reporting

Syntax errors are detected through a listener (detailed in 3.2.2). Logic errors are dealt with in the interpreter, which had to be hard coded.

In line with the requirements, error reporting had to be comprehensive and clear. It was therefore necessary to categorise errors and provide clear error messages.

4.5.1 Type Errors

Since Hale is dynamically typed it means that types are only evaluated when variables are used. This means types are checked each time an operation is carried out. The Hale interpreter makes use of Python's `try/except` clauses. The `typerror` is used to check for type errors:

```
if context.PLUS(i):
try:
    right += left
except TypeError:
    value = right
    if(type(left) is not int):
        value = left
```

```
    errors.append("Logical Error: Addition can only be applied to type
    ↪  int and string. You have applied it to '" + value + "' which
    ↪  is type " + str(type(value)))
```

In the code above a potential type error is spotted. If the current symbol in the tree is a '+' then the interpreter tries to compute compute right + left. If this isn't possible an exception is triggered by a type error, and error is created. It is then appended to the list of errors. This process is repeated for all mathematical operations.

If a conversion can avert a type error then the Hale interpreter performs that conversion, for example converting integers to floats.

4.5.2 Key Errors

Key errors refer to errors regarding variable names, for example calling variables that don't exist or initiating variables multiple times.

The `get_var` function is used to take the name of a variable from an AST and to return its value back to the tree. This is done by checking if the name exists in the functions dictionary. If so the value is returned. Otherwise a new error must be added:

```
if len(self._context_stack) > 0:
    value = self._context_stack[-1].get(name)
    if value is not None:
        return value
if name in self._functions:
    return self._functions[name]
errors.append("LOGICAL ERROR: Variable '"+  name +"' is called but not
↪  defined")
```

Similarly when a variable is initiated its name and value are placed into the dictionary. First a check happens to ensure its name doesn't already exist in the dictionary:

```
if name in self._context_stack[-1]:
errors.append("LOGICAL ERROR: Variable '"+  name +"' is defined
↪  twice")
```

4.5.3 Recursion Errors

A recursion call will be made during an evaluation of an **atom**. Therefore the call is attempted. But if a recursion error occurs errors are appended with *maximum recursion*

depth exceeded. The built in Python error `RecursionError` is used to identify this in a `try/except`.

4.5.4 List Errors

If, when evaluating list operations, no value can be returned, an error is appended to the list of errors.

List Func	Try Case	Except Case
`listsplit`	If list can be split on the index supplied	*Cannot use 'listsplit' on empty list*
`listhead`	If list length > 0 return the first element of the list	*Cannot use 'listhead' on empty list*
`listtail`	If list length > 0 return the list without the first element	*Cannot use 'listtail' on empty list*
`listinit`	If list length > 0 return the list without the last element	*Cannot use 'listinit' on empty list*
`listend`	If list length > 0 return the the last element	*Cannot use 'listend' on empty list*
`listlength`	If passed parameter is of type list	*Cannot use 'listlength' on non-list*

Table 4.2: The conditions and exception cases of the available list errors in Hale

4.6 React Web API

The React web application is responsible for managing the different number of packets and components that make up the API. The React application sets `\dashboard` as the home node of the application. If no token is detected, or a bad token is detected however, then the user is instead redirected to `\login`. Components are stored as Javascript functions. This is more secure and better practice than using classes [50] [51]. The use of constant functions, with set and use states, is also implemented to store values such as usernames and tokens. This adheres to React software development principles [50].

4.7 Chess

The chess game utilises code from the React Chess Github repository by Talha Awan [57]. Use of this repository was limited to the board sprites and layout as well as the basic moves (before Hale was included). Use of this library is done in accordance with the license [58], which required acknowledgment of use and inclusion of the licence in the folder where code is used. Both conditions have been met.

The square class is the most terminal and lowest level of them. It takes a shade, and a value. The squares are numbered 1 to 64. Therefore moving forward one space is an increase or decrease of 8. The squares are buttons since the pieces need to be able to be dragged onto them.

```
export default function Square(props) {
  return (
    <button className={"square " + props.shade}
      onClick={props.onClick}
      style={props.style}
      key={props.keyVal}>
    </button>);
}
```

The fallen soldiers class is an array of ids of the pieces that have been taken by the opposite player.

The pieces all exist as separate classes that contain the rules for their movement. Each has an `isMovePossible` procedure which takes a source square and destination square. It returns if the move is valid for that piece. It does not consider the positions of other pieces. This is left for the game file to factor in.

The board uses composition to construct an array of squares of alternating shades. The `game.js` file initialises this board and the fallen soldiers array, alongside all the pieces.

The game file contains the if conditions and other loops that allow certain pieces to move. For instance, only white pieces can move first. When a piece is selected a call is made to their `canMove` function which returns a boolean as to whether the move is valid within the rules of that piece. If the move is theoretically valid it checks to see if any piece exists in the path of that piece as well as applying other situational rules, for example is the king in check.

This code is mostly taken from the React Chess repository and as such detail into its workings is not as detailed as other sections. Section 4.7.1 details the way Hale influences the chess game which is all original work.

4.7.1 Chess and Hale

For the sections where Hale influences the chess game, a multi step process takes place. The front end sends a POST request containing the information of the piece to the backend. The backend then takes the information received. It retrieves the code the user has written from the database with an SQL query. This code is then edited with the information from the board as a parameter. The result is posted back to the front end which is what the new position or game environment is set to. In this way the game will work but behave oddly if a user codes a correct function but one that does not fulfil the challenge requirements. Sections 4.7.2, 4.7.3 and 4.7.4 detail this process more specifically with regards to the website's 3 challenges.

4.7.2 Challenge 1

Challenge 1 requires the user to direct the movement of a pawn. Once they have saved their code to the database and wish to react with the game environment the following steps take place:

The POST request is made to send the current position - stored in the variable `srcc` - of the pawn to the backend:

```
async movepawn(srcc, username) {
    const requestOptions = {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ username: username, src: srcc })
    }
    const response = await fetch('/task1', requestOptions)
    const data = await response.json();
    return data.srcminuseight;
}
```

The backend receives this request, takes the username of the signed in user out of the Json container and fetches their commands for challenge 1 tasks:

```
query = "SELECT spaceForward, equalPos, pawnMoveForward FROM TASK1
↪  WHERE username = '" + profile + "';"
cur.execute(query)
data = cur.fetchall()
```

It interprets the code within the context of the board by appending the `srcc` parameter to the end of the code:

```
file_to_run += "return(pawnMoveForward(" + str(src) + ", True));"
test, prints, errors = haleMain.runInterpreter(file_to_run)
return {'finallist':test}
```

The results are then returned in a Json container to the front end with the updated pawn position. This is done by setting the position equal to the result of the `movepawn` function:

```
let srcc = await this.movepawn(src, username);
```

For an example walkthrough, suppose user x has written the following commands for challenge 1:

```
def spaceForward ::= λ pos : int . pos+8;
def equalPos ::= λ x y : bool . ((y-x == 8)) ? True : False ;
def pawnMoveForward ::= λ x boolean : bool . boolean == True ?
↪  spaceForward(x) : x;
```

Having clicked save these functions are committed to the database. Then user x begins by moving a pawn. This triggers the `movepawn` function in the chess diagram:

```
async movepawn(srcc, username) {
    const requestOptions = {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ username: username, src: srcc })
    }
    const response = await fetch('/task1', requestOptions)
    const data = await response.json();
    return data.srcminuseight;
}
```

Therefore the Json object sent to the back end will include their username and the position of the pawn (say 11):

```
{
    "username": "userx",
    "src": 11
}
```

The api back end then extracts the 2 values from the Json object and stores them in variables:

```
json = request.get_json()
profile = json['username']
src = json['src']
```

The backend then retrieves the functions of that user from the database with a simple SELECT request:

```
cur = con.cursor()
query = "SELECT spaceForward, equalPos, pawnMoveForward FROM TASK1
 ↪  WHERE username = '" + profile + "';"
cur.execute(query)
data = cur.fetchall()
```

These functions are then appended to a string variable one by one with a new line between them:

```
file_to_run += data[0][0]
file_to_run += "\n"
file_to_run += data[0][1]
file_to_run += "\n"
file_to_run += data[0][2]
file_to_run += "\n"
file_to_run += "return(pawnMoveForward(" + str(src) + ", True));"
```

This means that `file_to_run` now holds the following value:

```
def spaceForward ::= λ pos : int . pos+8;
def equalPos ::= λ x y : bool . ((y-x == 8)) ? True : False ;
def pawnMoveForward ::= λ x boolean : bool . boolean == True ?
 ↪  spaceForward(x) : x;
return(pawnMoveForward(11, True));
```

This Hale "program" is then evaluated by passing it into the Hale interpreter. The return value is then returned in a Json object as it holds the result of calling the user's function:

```
returnval, prints, errors = haleMain.runInterpreter(file_to_run)
return {'srcminuseight':returnval}
```

The pawn class on the front end then uses this to update its position by returning the value of the request:

```
const response = await fetch('/task1', requestOptions)
const data = await response.json();
return data.srcminuseight;
```

4.7.3 Challenge 2

Challenge 2 asks the user to code the fallen soldiers command by asking the user to append the id of a taken piece to the list of fallen soldiers:

```
async getFallen(username) {
    console.log("start33");
    const requestOptions = {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ username: username, newFallen: id})
    }
    const response = await fetch('/task2', requestOptions)
    const data = await response.json();
    // console.log(useUserNameCurr());
    return data.finallist;
}
```

This, similar to challenge 1, loads the user's code from the database, and then runs it with the passed parameters:

```
file_to_run += "return(addFallen(" + listFallen + "," + newID + "));"
test, prints, errors = haleMain.runInterpreter(file_to_run)
return {'finallist':test}
```

The returned list contains the ids of the pieces that are displayed on the side of the board.

4.7.4 Challenge 3

The final challenge requires the calculation of the array values of the queen's path between two squares. For this the parameters passed are the start, stop and increment:

```
async moveQueen(start, stop, increment, username) {
    console.log("start");
    const requestOptions = {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ username: username, start:start,
        ↪  stop:stop, inc:increment})
    }
    const response = await fetch('/task3', requestOptions)
    const data = await response.json();
    console.log(data);
    return data.queenlist;
}
```

Similarly to the previous two challenges, the backend then runs the Hale code fetched from the database. This then returns the list of positions via a Json container:

```
file_to_run += "return(getQueenPath("+str(start) + ", " + str(stop) +
↪  ", [], " + str(increment) + "));"
print(file_to_run)
test, prints, errors = haleMain.runInterpreter(file_to_run)
return {'queenlist':test}
```

4.7.5 Chess as a Sandbox

The tangible response from the game in accordance with users code, as well as the splitting up of the challenges into subsections, adheres to gamification principles.

Constructivism's benefits are accessed through the feedback of incorrect solutions being given through incorrect gameplay rather than directly informing the user of mistakes. They can use the game to experiment and figure out how Hale works and how to functionally code, using the game environment as a sandbox.

$\lambda.5$

Evaluation

With work on Hale concluded it was important to evaluate its strengths and weaknesses and consider how many of the requirements were met. This was achieved through automated testing of the functional programming language in Python as well as statistical analysis of survey data collected.

5.1 Unit and Automated Testing

Testing each component of a system is called unit testing. This is done to ensure each element works as intended before fitting them together. The biggest element was the back end interpreter. Thus this was heavily tested, as an error here would render most of the project unusable. Pytest [59] is a Python framework which allows small readable tests to be evaluated. Pytest was used due to its simplistic nature and ease by which simple tests can be implemented. An example of such a test for the Hale interpreter is:

```
def add1test(x):
    code = "def add1 ::= "+ lambda + " x : int . x + 1"
    code += "\n"
    code += "return add1(" + str(x) + ");"
    returnVal, prints, errors = haleMain.runInterpreter(code)
    return returnNal

def test():
    for i in range 1000:
        assert add1test(i) == i + 2
```

This will display a result of the test (in this case a fail) when run as follows:

```
==================== FAILURES ====================
___________________ test_answer ___________________

    def test():
        for i in range 1000:
>           assert 0 == 1
E        +  where 0 = add1test(0)

test_sample.py:6: AssertionError
============== short test summary info ==============
FAILED test_sample.py::test_answer - assert 0 == 1
================= 1 failed in 0.12s =================
```

Unit tests like this were conducted to check the validity of the compiler interpretation. A full table of unit tests conducted can be found in Appendix D. Tests were all conducted 1000 times with different inputs as seen above. No test case failed in any situation. Therefore it is correct to assume that Hale's interpreter is likely implemented correctly.

TestCafe [60] was used to test the API as a whole. It was used to verify the size and length of lists, strings etc being passed between the front and back ends. It was also used to ensure that Javascript functions were returning values within an expected range. This was achieved by use of the `expect` function which is appended to await function calls. Only a few of these tests were conducted due to time limitations. An unfamiliarity with TestCafe meant time had to be used getting to grips with the system. Therefore important features were prioritised such as user authentication functions and challenge data. All of the tests conducted were passed, however given more time a more formal testing regime would be a welcome addition.

5.2 Solution Statistics

23 students used and tested Hale. All of them studied a STEM degree with at least some level of programming involved. Most of them had limited functional programming knowledge. Users were asked to fill out a survey. It asked the users to rate their programming ability (functional and imperative) before using the tool. This was then compared to the rating they were asked to give after using the tool. In addition users were asked to rate the difficulty of all three challenges. Each user submitted a valid response, that being one with a valid answer to all compulsory questions.

The success rate of the three challenges and their subparts were evaluated manually. As expected, success decreased with each challenge. The reason for manual evaluation

was that it facilitated insight into reasons for mistakes, whilst automated testing could only determine the binary success or failure of a program.

Figure 5.1 shows the percentage of users who completed each challenge successfully, whilst figure 5.2 shows the percentage of users who gave no solution for each challenge. The difference given by these indicates the number of incorrect attempts made to solve each.

Both of these followed the trends expected when the levels were designed. Tasks were supposed to become more difficult as the user progressed. Therefore a drop in the number of correct solutions and increase in the number of users unable to approach the problem was expected.

Challenges 2 & 3 had large numbers of incorrect attempts. For challenge 2, the most common reason was that the user attempted to concatenate an id to a list. A correct solution needed to encompass the id with square brackets to turn it into a list. Line 1 below shows a correct solution for the challenge, with line 2 indicating a mistake commonly made:

```
def addFallen ::= λ x y : list . x + [y];
def addFallen ::= λ x y : list . x + y;
```

Challenge 3 required the implementation of a recursive program. Every single incorrect attempt here was due to a lack of a base case. In the example solution on line 1 below, a check is made to ensure `start` and `stop` are equal. If this is the case the list is returned, whereas if it is not, a further increment is added. All incorrect solutions resemble the code on line 2, where no if statement is used and no base case to the recursion results in a recursion error.

```
def getQueenPath ::= λ start stop list inc : list . (start == stop) ? (list)
↪  : getQueenPath((start+inc), stop, (list+[start]), inc);
def getQueenPath ::= λ start stop l plus : list .getQueenPath((start+plus),
↪  stop, (l+[start]), plus);
```

5.3 Questionnaire

Each Hale user was asked to fill out a questionnaire after use to draw further conclusions about the program's strengths and weaknesses. The online flexibility of the questionnaire allowed students at multiple universities to participate widening the sample size and range.

Full graphs for all the survey questions can be found in appendix B.

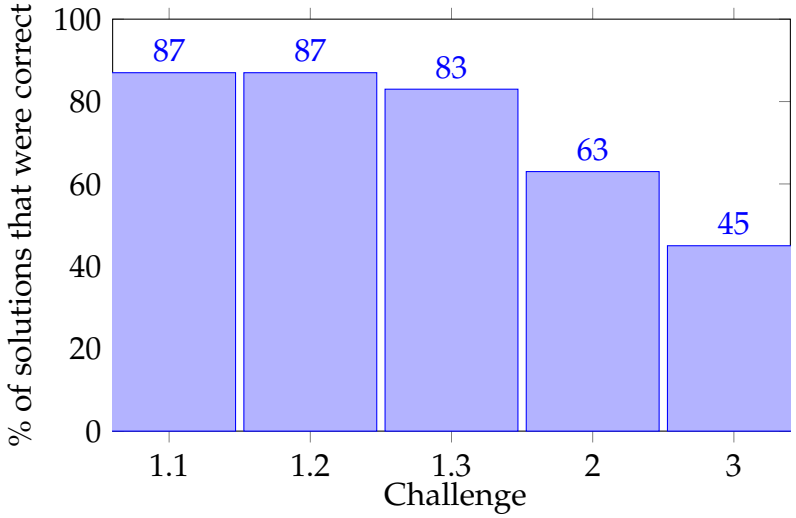


Figure 5.1: Percentage of submissions which were correct for each of the challenges

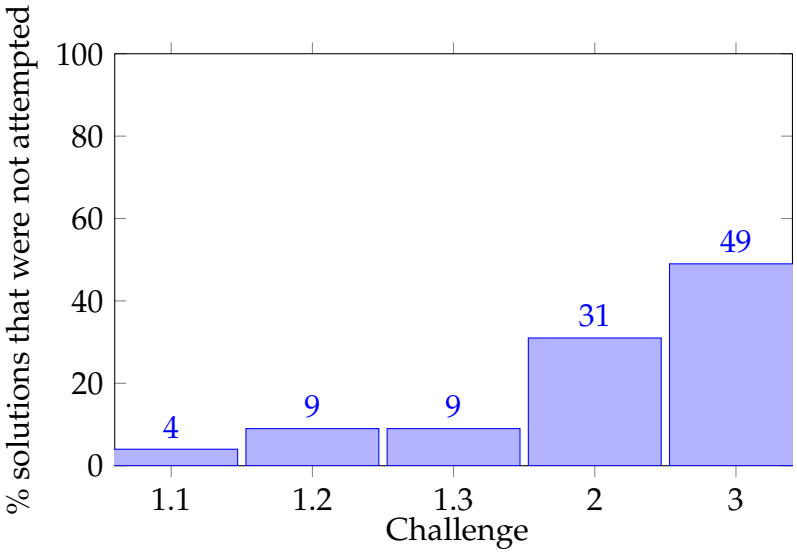


Figure 5.2: Percentage of users who did not provide a solution to each challenge

5.3.1 Ethics

The questionnaire included the requirement for each user to consent to their data being used in an anonymous capacity to indicate general trends regarding the project. Every person who filled out the survey agreed. Each user was given a contact email address and if they wished it was confirmed that their survey data would be deleted.

5.3.2 Initial Questions

The questionnaire gathered initial details regarding user age and asked them to rank their functional programming ability before and after using Hale. For every single user this increased after using Hale. These results are seen in figures 5.3, 5.4 and 5.5.

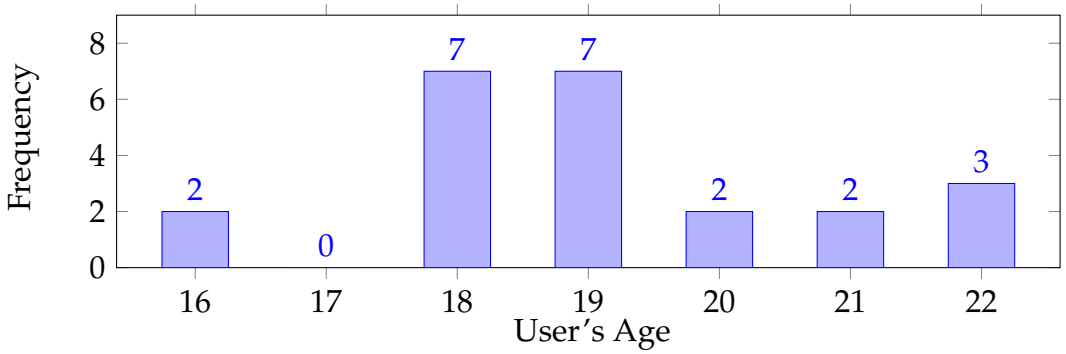


Figure 5.3: Age frequency graph of Hale testers

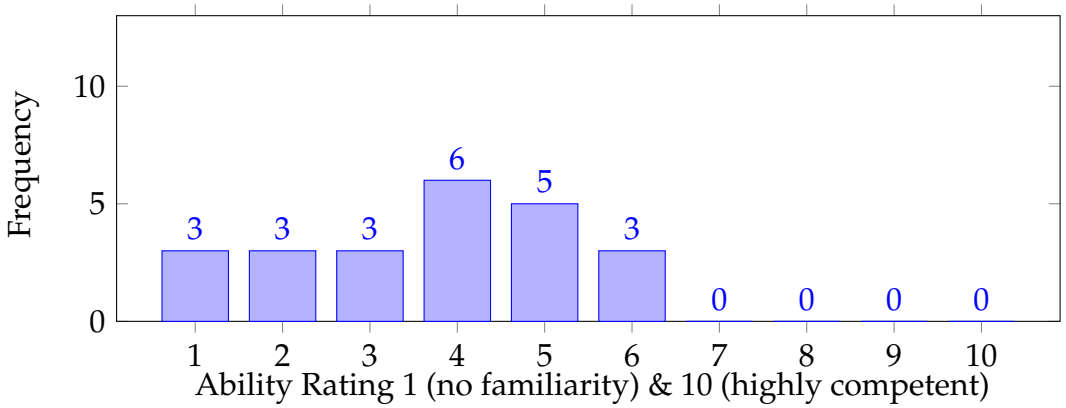


Figure 5.4: Graph depicting user responses to the question where they were asked to rate their functional programming ability before using Hale

The clear indication from each user of the increase in their perceived functional programming ability is a strong mandate of success. This shows the teaching tool assists in the understanding of the fundamental principles of functional programming. However this does not mean issues were not raised through the questionnaire, nor does it mean Hale is without its limitations. A full analysis of the responses can expose the things Hale succeeds in, as well as the things which still require work going forward. Percentages given in further sections are rounded to the nearest whole number.

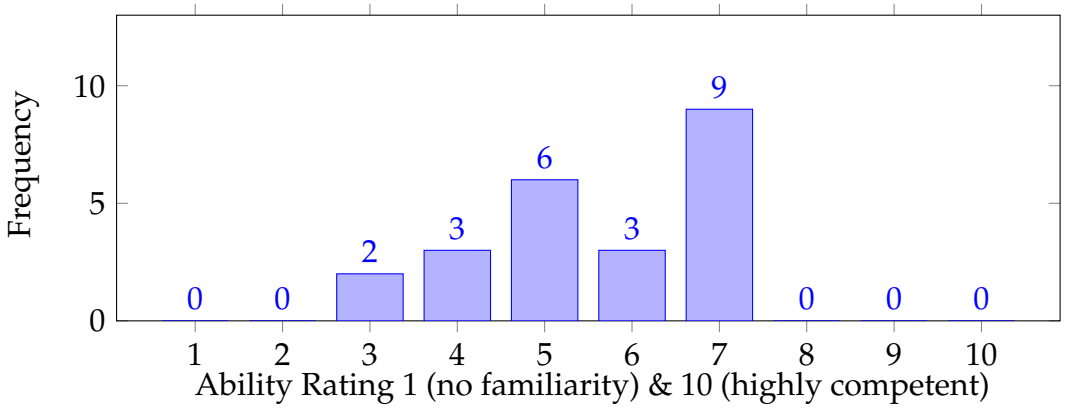


Figure 5.5: Graph depicting user responses to the question where they were asked to rate their functional programming ability after using Hale

5.3.3 Challenges

The first set of questions revolved around the challenges. When asked if they had solved all parts of challenge 1, 91% of users said they had managed it, with 9% saying they had not. 91% is higher than the 87% given in figure 5.1. Upon inspection this was put down to 1 user who's code produced a functioning, but incorrect result. This is the downside of the constructivist approach, and future work should aim to mitigate any such further cases.

This percentage dropped further to 63% with challenge 2, however all those who wrote correct code were able to successfully identify their code as correct. The reason for this is that the challenge was easier to acknowledge as correct due to the display of fallen pieces on the side of the board.

Similarly, all those who submitted correct code for challenge 3 were able to identify it as correct. However, this percentage was far lower, at only 45%. This can be attributed to the complex nature of the task which required case base recursion, a tricky concept for those unfamiliar with the functional paradigm.

5.3.4 User Interface

The responsiveness of the website was rated very highly with 22 of 23 participants saying the website was *"totally responsive"*. All users either agreed or strongly agreed that the website's UI was clean and easy to understand. Furthermore, 20 users either agreed or strongly agreed that the website was easy and intuitive to navigate (figure B.3). 21 users felt that the profile page made it easy to see which tasks had been completed and which had not (figure B.4).

5.3.5 Gamification

18 of the 23 users agreed or strongly agreed with the statement, *"Seeing effects in the game of chess makes understanding what your functions do and why they are working / not working easier"* (figure B.5). Users also made comments with regards to the worded questions, which supported the game as a successful medium of teaching, although room for improvement and future adjustment was also identified.

5.3.6 Worded Questions

The final section of the questionnaire invited written feedback on what Hale's strengths were and what had room for improvement. The questions used were:

- What are the things you liked about the project?
- Are there any limitations of the project you think could be improved?
- Do you have any final comments?

These questions had a different type of usefulness. They were less open to statistical analysis but did allow students to express a much wider range of opinions. Determining what the user liked about the project was a useful mechanism to identify what components were working well and useful. Meanwhile, asking the users about the limitations highlighted areas that warranted adjustment. These questions were optional and as such not every question received 23 responses.

The first question received 17 responses. These ranged from verbose paragraphs detailing user experience to concise one sentence responses. The following themes were extracted and tallied up:

The layout and design of the webpage stuck out as one of the things the users liked most about Hale. One user said, *"The website is designed really well, the responsiveness of everything was top tier."* and another commented that *"The UI and design are superb, really clean"*. The number of comments regarding responsiveness is to be expected with 22 out of 23 users rating the responsiveness 5/5 and the last rating it 4. 26% of users made a direct comment about the game aspect of the project being fun. However, opinions on the game were less clear cut in the questionnaire with 2 people disagreeing that the game was helpful to their learning. 9 strongly agreed it helped, and another 9 agreed to some extent, whilst 3 were neutral. This is seen in figure B.5.

4 people commented specifically on the language being fun to learn and use, with one saying *"The language Hale was interesting and fun to use, I enjoyed getting to grips with it"*.

Frequency	Praise
11	Website is very responsive
10	Clean UI
8	Having code make the game work is fun
7	The errors provided were helpful
6	Terminal interface is very clean
5	The fact the editor, tasks and game are all on one page
5	The ability to change between light and dark mode
4	The language was fun to use
1	The stylish layout of the info page

Table 5.1: The tallies of responses to the first worded questions

One of the requirements of the project was to have comprehensive error messages to indicate mistakes, therefore it was helpful to see that 7 people had commented that they found the error messages clear and easy for debugging. 1 comment mentioned the stylish layout of the info page. This is contrasted with several issues raised about the info page. Meanwhile in response to the second question, users also commented about limitations of Hale, seen in table 5.2.

Frequency	Limitation
13	Lack of λ copy and paste button
12	Confusion over what *save* button does
8	Improvement to documentation required
5	Info page would be better as a pop-up
2	Does not inform you if your functions are correct
1	Boolean datatype is confusing
1	Opening one task doesn't close the others, meaning the chess board easily gets pushed out of view

Table 5.2: The tallies of responses to the second worded questions

The lack of a copy and paste button for λ stood out. This was something raised during development of the project, had more time been available for development this would have been completed. It was also viable to remove the λ from the language syntax, and replace it with a backslash as Haskell does. There seemed to be a lot of confusion over the save button. One user noted, *"It was very unclear what save did, to me it seemed intuitive that if I hit save I would return to my work unchanged - this was not the case"*. There were multiple other users who felt similar confusion. A save and submit button

might help to distinguish between this, where save stored the code you had written whilst submit would cause code to affect the game. Documentation improvements were widely desired, with one user appropriately noting that *"There was no explanation of how to use recursion even though the task said you should use it"*. Again time limitations affected this. 5 users noted they would like to see the info page as a pop up with chapters that could be jumped to. This would help finding documentation about specific issues easier, as well as preventing the instances where code was lost by accessing the info page.

5.4 Project Management Evaluation

The SCRUM approach to iterative development helped to ensure that elements of the project progressed at similar levels. Sprints allowed the process ot be less stressful due to increased organisation. Meetings were used as "SCRUM meetings" whereby a sprint would be completed before a project meeting. In the meeting, a roadmap for project progress over the next week would be laid out. The risk assessment shown in table 3.1 allowed for issues and setbacks to be easily mitigated against. The timetable shown in appendix C was largely adhered to, however its ambitious nature did allow for some flexibility which was utilised when the project fell behind. Github was used to revert to a previous commit when an unidentified bug could not be located.

5.5 Self-Assessment

What is the (technical) contribution of this project?

This project set out to produce to a web based teaching tool, using gamification and constructivism to teach functional programming. The technical contribution this entailed was the creation of an interpreter for a custom made language, Hale. Hale supported key functional programming concepts including lambda expressions, function applications, recursion, and higher order functions. This was built into an API which used the functions written by the user to influence a game environment. Progress was displayed in a profile page and the website adhered to strict and up to date cyber security benchmarks.

Why should this contribution be considered relevant and important for the subject of your degree?

This project built on several fields from my degree. The language design, with typing rules and evaluation style were directly built on the work done in CS141. Meanwhile the design and implementation of the interpreter, from the lexical analysis to error

reporting were directly influenced and aided by the research and work conducted as part of CS325. Knowledge gained from CS263 was critical in ensuring a safe system, and CS348 strongly aided website design and layout.

How can others make use of the work in this project?

Others can use the project to get to grips with the basics of functional programming. They can use Hale as a stepping stone to introduce them to the paradigm instead of facing the complexities and overwhelming nature of Haskell or other functional languages. They use the game environment to experiment with its capabilities as well as reinforce their learning by observing their mistakes play out in game.

Why should this project be considered an achievement?

The creation of a brand new Turing complete language, alongside the implementation of a working interpreter was a big achievement and proved tricky to implement. The functionality of the language extending to things such as higher order functions, which are not possible in many mainstream imperative languages, was also an achievement. The consideration of cyber security and social informatic principles in web design alongside successful creation of a sandbox game environment influenced by user functions are all achievements.

What are the limitations of this project?

The balance between gamification and constructivism posed some limitations. This was because gamification required direct feedback as to the correctness of moves whilst constructivism encouraged creativity and exploration. This meant the full benefits of both were not fully unlocked. In future work (section 6.1) mitigations are proposed. Time limitations also restricted the number of challenges that could be devised. With more time, further challenges and concepts would be taught.

$\lambda.6$

Conclusion

Overall the project is a success. Hale, a web based tool using gamification and constructivism to teach functional programming was successfully implemented. The overall consensus from users was positive. Hale is a Turing complete functional programming language able to implement the core principles of functional programming. The language is capable of implementing λ-expressions, integers, bools, strings, lists, function applications, list operations, higher order functions and recursion as presented in requirement 1 (section 1.4). Errors are grouped into logical and syntactic categories. They are presented to the user with clear sentence descriptions of what is causing the problem. Users on the whole were pleased with error reporting, satisfying requirement 2. The gamification and constructivism techniques employed resulted in a pseudo-sandbox environment, whereby the user can either follow a set of challenges or experiment themselves, meeting requirements 3, 4, & 5. Being dynamically typed, Hale reports type errors during expression evaluation. At this level, types are checked for compatibility, conversions made if necessary and if not possible a type error reported, as per requirement 6.

The challenges presented increased in difficulty. All three relied on key functional programming principles. The game environment provides feedback when a program is correct (whether it achieves the intended purpose or does not). The console provides feedback if a function does not run, but the game does not do this. Therefore requirement 8 is partially fulfilled with opportunities for improvement. Requirement 9 also remains only partially fulfilled, with the program result being the only thing considered. This is due to time constraints and challenge design. The nature of the game environment may need to be changed to provide an environment where this can be completely fulfilled. This is discussed in section 6.1.

Users widely agreed that the profile page made it simple and easy to see what challenges they had completed so far, satisfying requirement 7. Access token and password hashing,

making use of up to date and cryptographically secure libraries and principles, meant requirement 10 was also satisfied. Overall of the 10 functional requirements, 8 were fully met and 2 were partially met.

The choice to use chess was strong way to balance the ideas of gamification and constructivism. The fixed rules of chess promoted gamification principles, however the game did not tell users when their code gave undesirable effects. Instead it allowed them to use the game environment to deduce if there were any issues. This promoted exploration and followed the principles of constructivism. Overall this balance worked well. In future work (section 6.1) further discussion is conducted into how to best balance these two fields. Whilst there were some issues, this balance seemed to work well with the majority of users saying the game aspects boosted their learning experience.

The principles of the MUSIC model of learning were implemented. The challenges presented opportunities for multiple working correct solutions, with each challenge increasing the difficulty from the last. The challenges provided insight into the ways in which aspects of functional programming are used and can be useful. The project encouraged the users to create a fully working game of chess which acted as a general overreaching project, in line with project based learning. The users agreed that the project aided their learning of functional programming and that errors were reported helpfully. The UI was considered clean and intuitive. Therefore all of the non functional requirements have been fully satisfied. Thus the project can be deemed a success due to its strong user feedback alongside fulfillment of all major criteria.

6.1 Future Work

An initial first step is to create the λ copy and paste button as users have mentioned that this would be a welcome feature.

It would also be prudent to use additional time to design and lay out a more comprehensive testing regime for the front end. The back end Pytest unit tests were very comprehensive but due to time limitations front end testing with TestCafe were less thorough.

Additional future work could extend the functionality and capabilities of the system by introducing more features. For instance balancing gamification and constructivism principles led to some of the benefits of each being missed. Therefore a key additional feature would be a toggle switch to swap between a challenge orientated mode and a sandbox mode. In sandbox mode there are no challenges, but the user is instructed to experiment around with Hale in order to interact with a game environment. This

means that the game mode of the program could aim to follow more closely the gamification principles. As such challenges could be marked as passed or failed, with the user getting feedback on whether or not their code is correct. Hale could be further extended to include even more gamification principles such as badges, achievements and leaderboards.

Other games could also be incorporated in order to keep challenges seeming fresh and exciting. The sandbox element could also allow the users to create new rules for the game or even their own games.

Bibliography

[1] Simon Thompson and Steve Hill. Functional programming through the curriculum. In *International Symposium on Functional Programming Languages in Education*, pages 85–102. Springer, 1995.

[2] Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3(1):49–65, 1993.

[3] Manuel M T Charkravarty and Gabrielle Keller. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, 14:113–123, 01 2004.

[4] Mustafa Serkan Gunbatar and Halit Karalar. Gender differences in middle school students' attitudes and self-efficacy perceptions towards mblock programming. *European Journal of Educational Research*, 7:925–933, 10 2018.

[5] Filiz Kalelioglu and Yasemin Gulbahar. The effects of teaching programming via scratch on problem solving skills: A discussion from learners' perspective. *Informatics in Education*, 13:33–50, 04 2014.

[6] Peggy A. Ertmer and Timothy J. Newby. Behaviorism, cognitivism, constructivism: Comparing critical features from an instructional design perspective. *Performance Improvement Quarterly*, 26(2):43–71, 2013.

[7] Vishal Dagar and Aarti Yadav. Constructivism: A paradigm for teaching and learning. *Arts and Social Sciences Journal*, 7, 01 2016.

[8] Ian Arawjo, Cheng-Yao Wang, Andrew C. Myers, Erik Andersen, and François Guimbretière. Teaching programming with gamified semantics. pages 4911–4923, 2017.

[9] Code World. `https://www.https://code.world/`. [Online; accessed 28-March-2022].

[10] Alejandro Lujan. Scalaquest - the game to learn scala. `https://www.kickstarter.com/projects/andanthor/scalaquest-a-game-to-learn-scala`. [Online; accessed 28-March-2022].

[11] Ming-Chaun Li and Chin-Chung Tsai. Game-based learning in science education: A review of relevant research. *Journal of Science Education and Technology*, 22(6):877–898, 2013.

[12] Maja Pivec. Editorial: Play and learn: potentials of game-based learning. *British Journal of Educational Technology*, 38(3):387–393, 2007.

[13] Brett D. Jones. Engaging second language learners using the music model of motivation. *Frontiers in Psychology*, 11:1204, 2018.

[14] Miran Lipovaca. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, USA, 1st edition, 2011.

[15] Graham Hutton. *Programming in Haskell*. Cambridge University Press, USA, 2nd edition, 2016.

[16] Ahmed Alanazi. A critical review of constructivist theory and the emergence of constructionism. 03 2019.

[17] Ming-Chaun Li and Chin-Chung Tsai. Game-based learning in science education: A review of relevant research. *Journal of Science Education and Technology*, 22(6):877–898, 2013.

[18] Mihaly Csikszentmihalyi and Mihaly Csikzentmihaly. *Flow: The psychology of optimal experience*, volume 1990. Harper & Row New York, 1990.

[19] Jesse Schell. *The Art of Game Design: A Book of Lenses*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[20] Sebastian Deterding. Gamification: Designing for motivation. *Interactions*, 19(4):14–17, jul 2012.

[21] Gabe Zichermann and Christopher Cunningham. *Gamification by Design: Implementing Game Mechanics in Web and Mobile Apps*. O'Reilly Media, Inc., 1st edition, 2011.

[22] Parul Khurana and Balraj Kumar. Gamification in education - learn computer programming with fun. 2017.

[23] Maja Pivec. Editorial: Play and learn: Potentials of game-based learning. *British Journal of Educational Technology*, 38:387–393, 05 2007.

[24] Abed H Almala. Applying the principles of constructivism to a quality e-learning environment. *Distance Learning*, 3(1):33, 2006.

[25] Karen Robson, Kirk Plangger, Jan H. Kietzmann, Ian McCarthy, and Leyland Pitt. Is it all a game? understanding the principles of gamification. *Business Horizons*, 58(4):411–420, 2015.

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

[27] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[28] Ralph Loader. Notes on simply typed lambda calculus. Technical report, 1998.

[29] Clem Baker-Finch. An introduction to the lambda calculus. In *Beyond Classical Logic*, 2013.

[30] Didier Rémy. Type systems for programming languages. *MPRI lecture notes*, page 64, 2013.

[31] Programming Languages and Logics Lecture 18: Simply Typed λ-calculus. `https://www.cs.cornell.edu/courses/cs4110/2018fa/lectures/lecture18.pdf`. [Online; accessed 28-March-2022].

[32] M Anton Ertl and David Gregg. Building an interpreter with vmgen. In *International Conference on Compiler Construction*, pages 5–8. Springer, 2002.

[33] Gregory John Michaelson. Interpreter prototypes from formal language definitions. 1993.

[34] ANTLR (ANother Tool for Language Recognition). `https://www.antlr.org/`. [Online; accessed 28-March-2022].

[35] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[36] Danyang Cao and Donghui Bai. Design and implementation for sql parser based on antlr. In *2010 2nd international Conference on Computer engineering and technology*, volume 4, pages V4–276. IEEE, 2010.

[37] Scott Owens, Matthew Flatt, Olin Shivers, and Benjamin McMullan. Lexer and parser generators in scheme. 2004.

[38] Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. A fast abstract syntax tree interpreter for r. *ACM SIGPLAN Notices*, 49(7):89–102, 2014.

[39] Flask Web framework. `https://flask.palletsprojects.com/en/2.1.x/`. [Online; accessed 29-March-2022].

[40] Miguel Grinberg. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[41] Nuruldelmia Idris, Cik Feresa Mohd Foozy, and Palaniappan Shamala. A generic review of web technology: Django and flask. *International Journal of Advanced Science Computing and Engineering*, 2(1):34–40, 2020.

[42] J Ellingwood and K Juell. How To Serve Flask Applications with Gunicorn and Nginx on Ubuntu 16.04, 2020.

[43] Rahul Soni. *Nginx*. Springer, 2016.

[44] Vivek Thoutam. A study on python web application framework. *Journal of Electronics, Computer Networking and Applied Mathematics (JECNAM) ISSN: 2799-1156*, 1(01):48–55, 2021.

[45] Grant Allen and Mike Owens. Sqlite design and concepts. In *The Definitive Guide to SQLite*, pages 125–152. Springer, 2010.

[46] Lv Junyan, Xu Shiguo, and Li Yijie. Application research of embedded database sqlite. In *2009 International Forum on Information Technology and Applications*, volume 2, pages 539–543. IEEE, 2009.

[47] SQLite ltd. SQLite Query Planning. `https://www.sqlite.org/queryplanner.html#searching`. [Online; accessed 02-April-2022].

[48] SQLite ltd. Appropriate Uses For SQLite. `https://www.sqlite.org/whentouse.html`. [Online; accessed 02-April-2022].

[49] George Hatzivasilis, Ioannis Papaefstathiou, and Charalampos Manifavas. Password hashing competition - survey and benchmark. *IACR Cryptology ePrint Archive*, 2015:265, 2015.

[50] Matthew David. Get off and running with these tools - The 5 best web application frameworks: How to choose. `https://techbeacon.com/app-dev-testing/five-best-web-application-frameworks-how-choose`. [Online; accessed 04-April-2022].

[51] Sanchit Aggarwal and Jyoti Verma. Comparative analysis of mean stack and mern stack. *International Journal of Recent Research Aspects*, 5(1):127–32, 2018.

[52] David Cohen, Mikael Lindvall, and Patricia Costa. Dacs state-ofthe-art / practice report agile software development. 2003.

[53] John Hunt. Higher order functions. In *A Beginners Guide to Python 3 Programming*, pages 157–166. Springer, 2019.

[54] Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 280–293, 2005.

[55] Eric Kidd. Map fusion: Making Haskell 225 `http://www.randomhacks.net.s3-website-us-east-1.amazonaws.com/2007/02/10/map-fusion-and-haskell-performance/`. [Online; accessed 13-April-2022].

[56] Microsoft. Monaco Editor. `https://microsoft.github.io/monaco-editor/`. [Online; accessed 11-April-2022].

[57] Talha Awan. React Chess. `https://github.com/TalhaAwan/react-chess`. [Online; accessed 15-April-2022].

[58] Talha Awan. React Chess License. `https://github.com/TalhaAwan/react-chess/blob/master/LICENSE`. [Online; accessed 15-April-2022].

[59] pytest. pytest: helps you write better programs. `https://docs.pytest.org/en/7.1.x/`, 2015.

[60] TestCafe. TestCafe. `https://testcafe.io/`, 2012-2021.

Appendix A

Full Hale Grammar

statements ::= (*statement* **semicolon**)+
statement ::= *simplestmt*
| *functiondef*
functiondef ::= **def id eq lambda id* colon type fstop** *simplestmt*
simplestmt ::= *expression*
| *assign*
assignment ::= **id** (**comma id**)* **eq** *expr*
expr ::= *comp*
| *conditional*
conditional ::= *comp* **qmark** *simplestmt* **colon** *simplestmt*
comp ::= *boolean* ((**lt** | **lte** | **gte** | **gt** | **gte** | **ceq**) *boolean*)*
boolean ::= *sumsub* ((**or** | **and** | **xor**) *sumsub*)*
| **not** *sumsub*
sumsub ::= *multdiv* ((**plus** | **minus**) *multdiv*)*
multdiv ::= *exp* ((**mult** | **div**) *exp*)*
exp ::= *atom* ((**pow**) *atom*)*
atom ::= *value*
| **lpar** *simplestmt* **rpar**
| **id lpar** *params*? **rpar**
| *list*
params ::= *expr*(**comma** *expr*)*
list ::= **lpar** (*expr* (**comma** *expr*)*)? **rpar**
value ::= **int**
| **string**
| **bool**
| **id**
def ::= 'def'

bool ::= 'True'
| False
fstop ::= '.'
eq ::= '::='
type ::= 'int'
| 'bool'
| 'string'
| 'do'
| 'list'
lambda ::= 'λ'
int ::= [0-9]+
plus ::= '+'
minus ::= '-'
mult ::= '*'
div ::= '/'
lpar ::= '('
rapr ::= ')'
pow ::= '**'
lt ::= '<'
lte ::= '<='
gt ::= '>'
gte ::= '>='
ceq ::= '=='
comma ::= ','
id ::= [a-zA-Z][a-zA-Z0-9]*
dquote ::= '"'
qmark ::= '?'
colon ::= ':'
eq ::= '='
semicolon ::= ';'
or ::= '|'
and ::= ''
xor ::= '^'
not ::= ' '
lnot ::= '!'
lbr ::= '['
rbr ::= ']'
lcbr ::= '{'
rcbr ::= '}'
string ::= **dquote** ["\r \n]* **dquote**
comment ::= '#' ~ [\n]* -> skip

Appendix B

Questionnaire Graphs

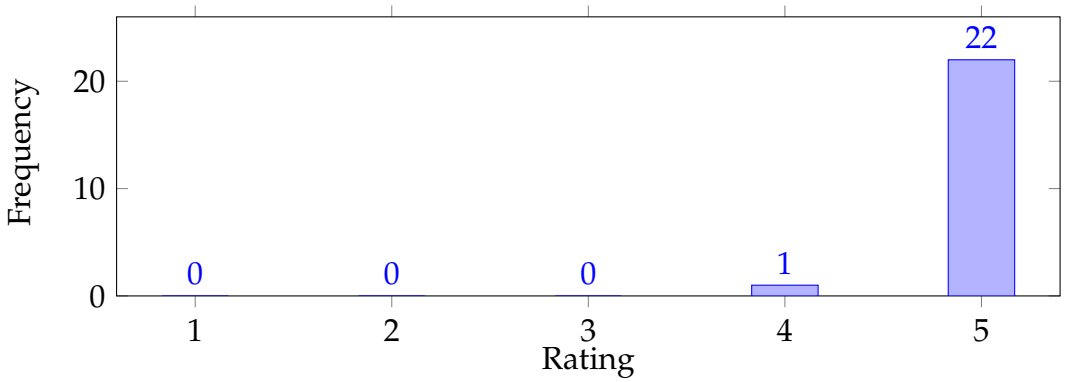


Figure B.1: Graph depicting user ratings for the responsiveness of the website from 1 (not responsive at all) to 5 (totally responsive)

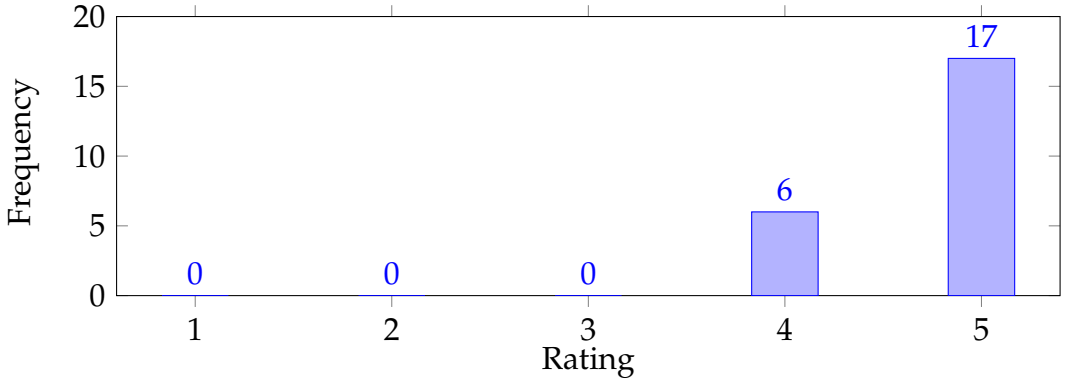


Figure B.2: Graph depicting user ratings for the cleanness of the user Interface from 1 (cluttered and difficult to use at all) to 5 (clean and easy to understand)

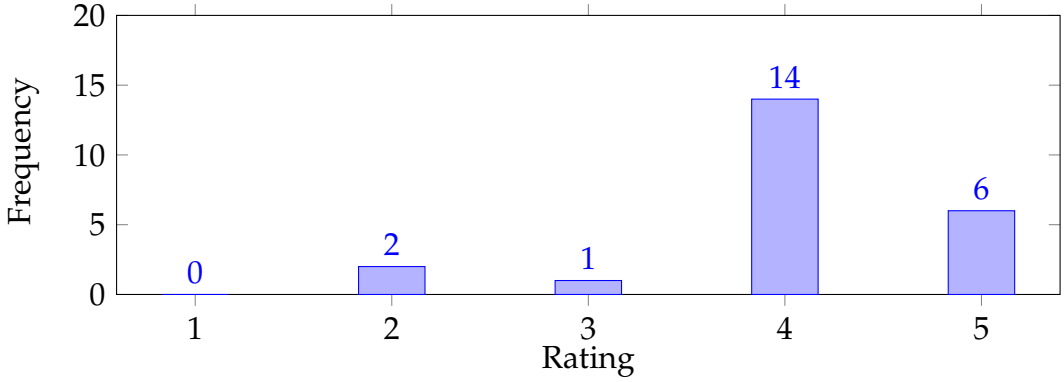


Figure B.3: Graph depicting user ratings for how easy the website was to navigate from 1 (not very easy at all) to 5 (very easy)

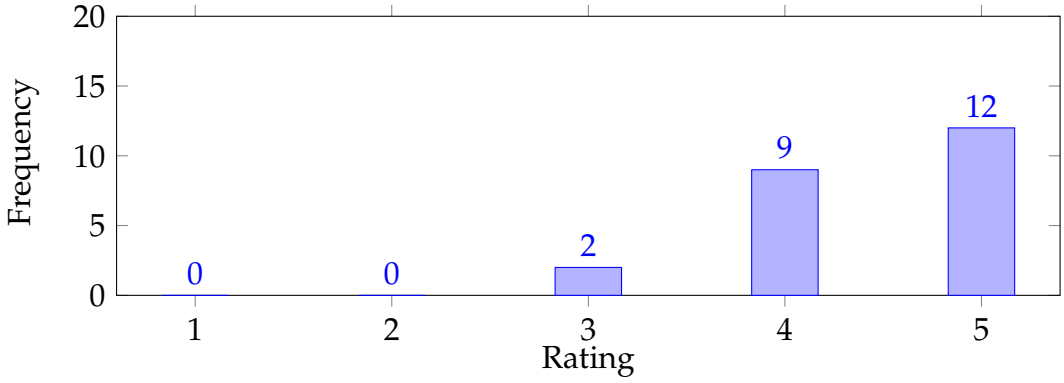


Figure B.4: Graph depicting user ratings for how easy the profile page makes it to see what functions have already been written from 1 (not very easy at all) to 5 (very easy)

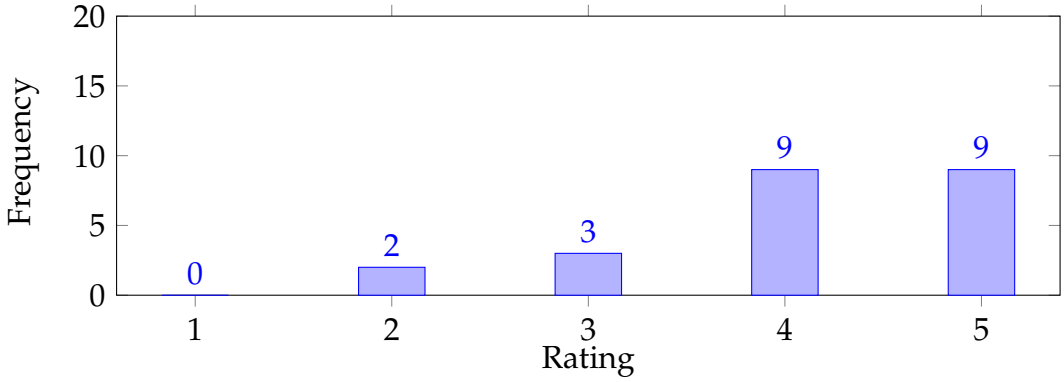


Figure B.5: Graph depicting results of asking users to respond to the statement *"Seeing effects in the game of chess makes understanding what your functions do and why they are working / not working easier"* from 1 (strongly disagree) to 5 (strongly agree)

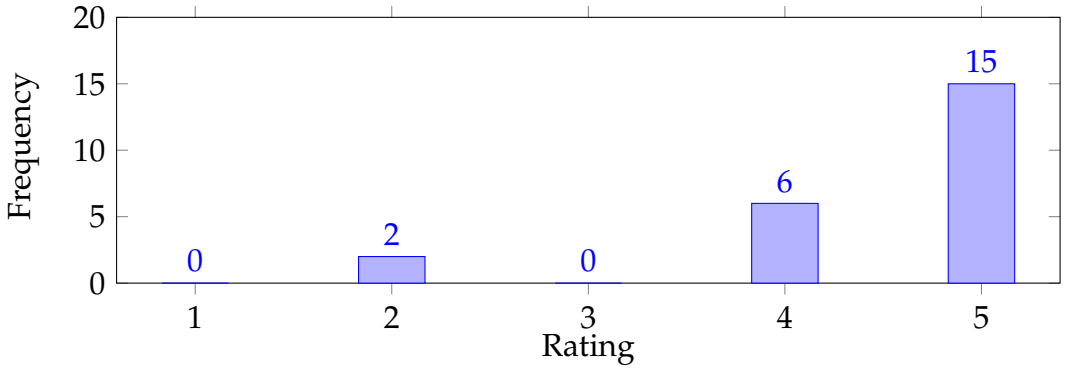


Figure B.6: Graph depicting results of asking users to respond to the statement *"The challenges get progressively more difficult"* from 1 (strongly disagree) to 5 (strongly agree)

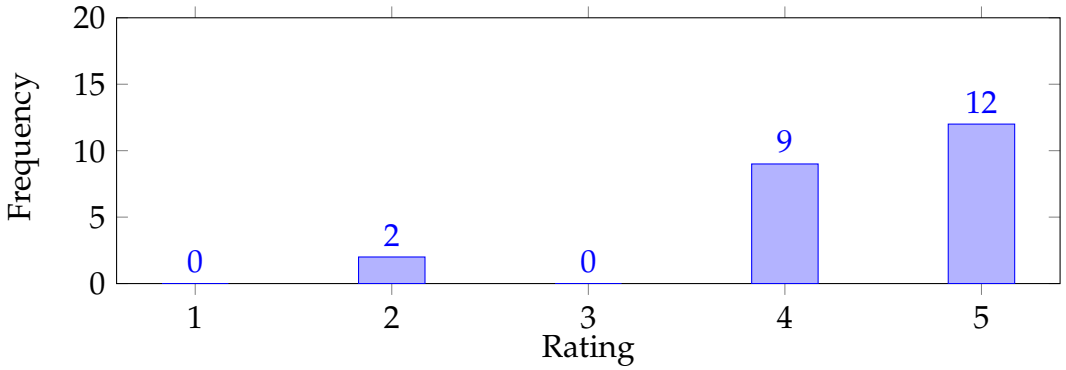


Figure B.7: Graph depicting results of asking users to respond to the statement *"The first challenge is easy enough to get you started with Hale"* from 1 (strongly disagree) to 5 (strongly agree)

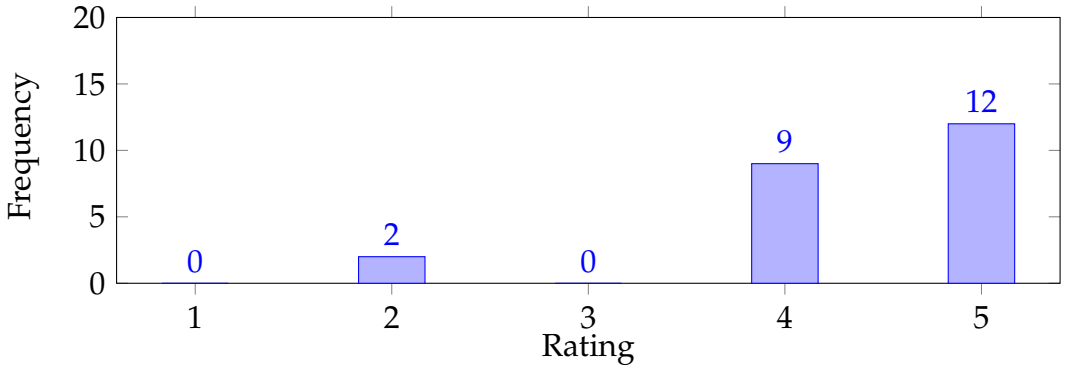


Figure B.8: Graph depicting results of asking users to respond to the statement *"The last challenge is challenging enough to stretch you"* from 1 (strongly disagree) to 5 (strongly agree)

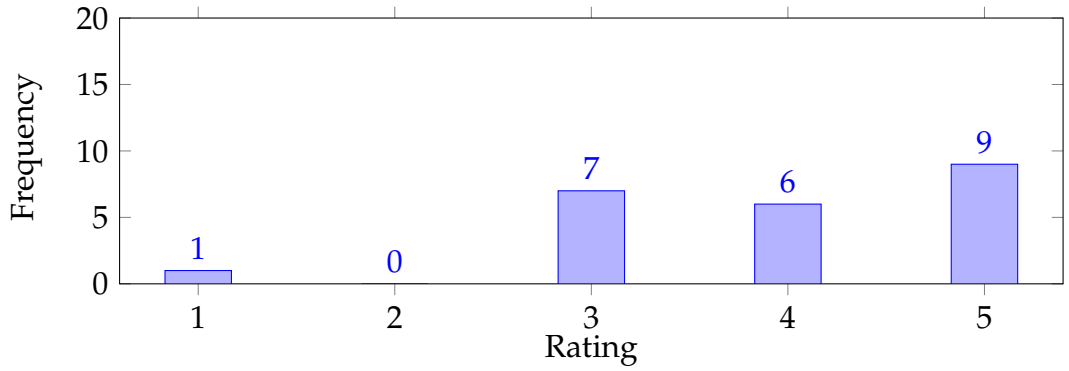


Figure B.9: Graph depicting results of asking users to respond to the statement *"You found it clear what you needed to do when first loading the website"* from 1 (strongly disagree) to 5 (strongly agree)

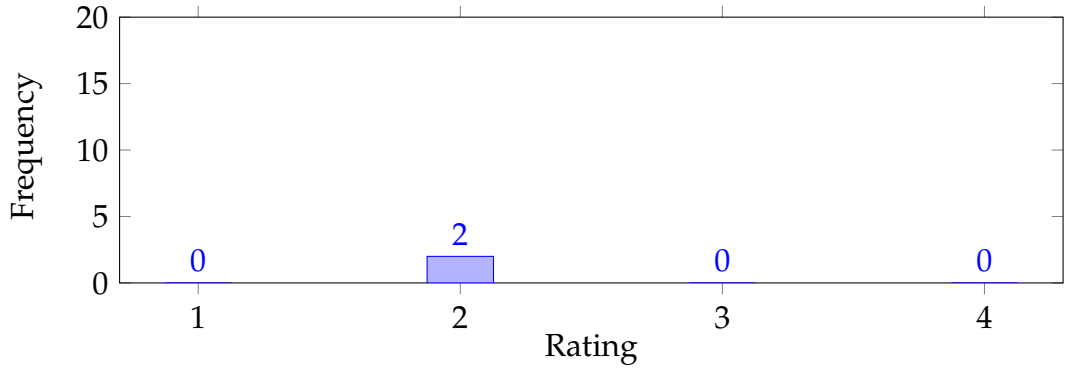


Figure B.10: Graph depicting how many times users reported the website crashing

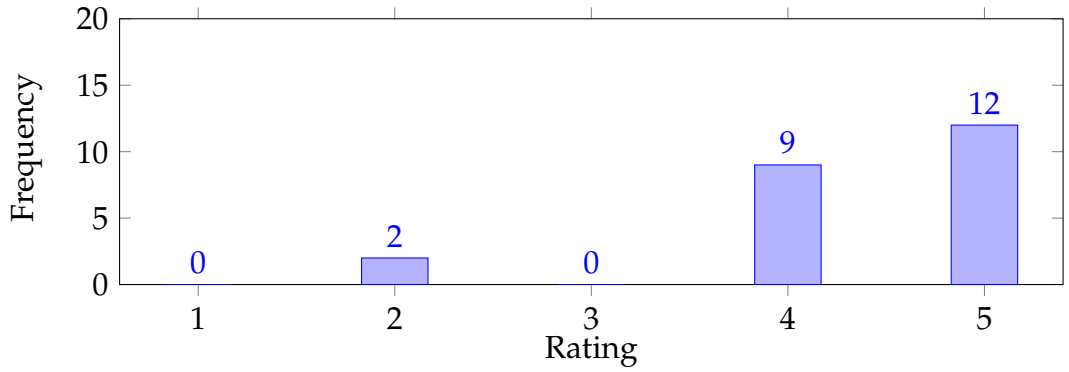


Figure B.11: Graph depicting results of asking users to respond to the statement *"The website has improved my understanding of functional based languages"* from 1 (strongly disagree) to 5 (strongly agree)

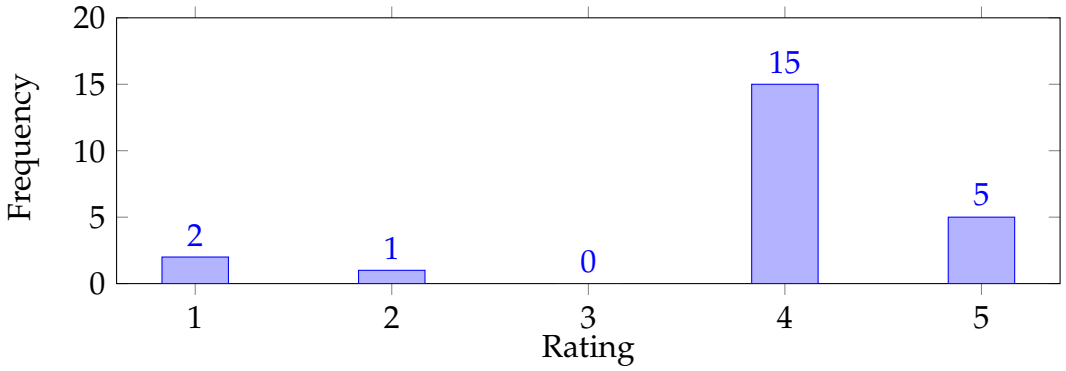


Figure B.12: Graph depicting results of asking users to respond to the statement *"I found this teaching tool fun to use"* from 1 (strongly disagree) to 5 (strongly agree)

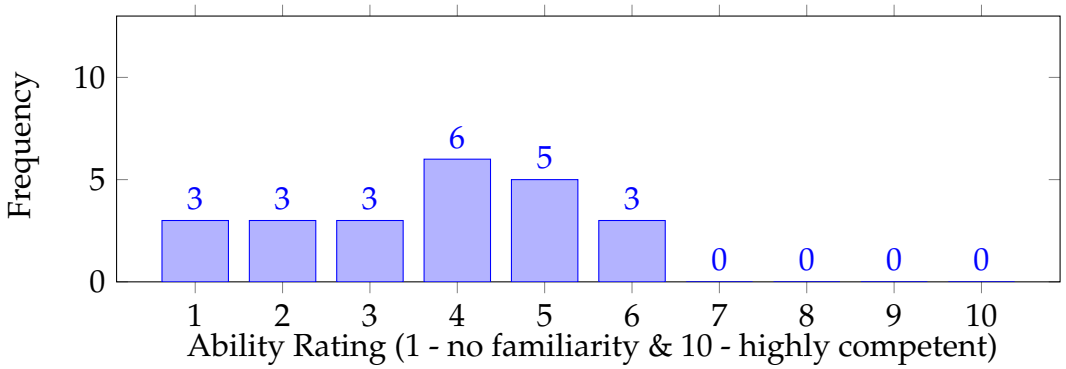


Figure B.13: Graph depicting user responses to the question where they were asked to rate their functional programming ability before using Hale

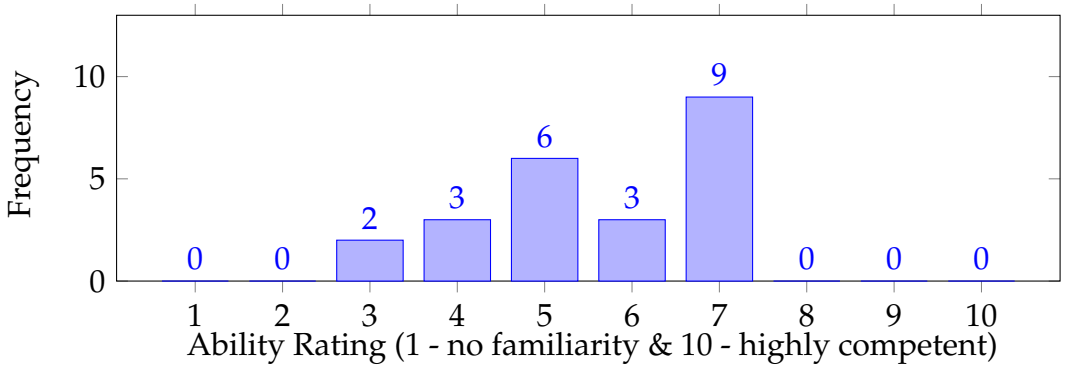


Figure B.14: Graph depicting user responses to the question where they were asked to rate their functional programming ability after using Hale

Appendix C

Timetable

	Weeks 1-2
4 October	Draft and write project specification. Research further into learning techniques and start to put together an initial basic React website.
	Weeks 3-4
18 October	Work further on the website creating the UI etc. such that the user can type code but without a backend.
	Weeks 5-6
1 November	Flesh out the game by developing the custom functional programming language that will be used and design the levels on paper.
	Weeks 7-8
15 November	Create the interpreter for the language designed. This will be done in Python
	Weeks 9-10
29 November	Start work on code feedback algorithm.
	Christmas Break
13 December	Catch up on any of the tasks that I fell behind on to ensure all the work that was meant to be completed by the end of week 10 was completed.

10 January
Weeks 1-2
Finish the code feedback code in Python.

24 January
Weeks 3-4
Connect frontend to backend, bug test and ensure the website is fully working.

7 February
Weeks 5-6
Start writing the initial report. This will focus on the introduction and the details of development stages.

21 February
Weeks 7-8
Finish writing the development sections of the report and start on the conclusion.

7 March
Weeks 9-10
Finish the report and analyse the success of the project from the tests and interviews.

21 March
Easter Break
Review report and make any necessary changes.

25 April
Weeks 1-2
Finish and submit the final report.

Appendix D

Unit Tests

Mathematical Unit Tests

Test	Result
Implements + - / * operators	Pass
Implements comparison operators	Pass
Implements logical operators	Pass
Implements comparison operators	Pass
Implements exponential operator	Pass

Typing Unit Tests

Test	Result
Preserves base case for literal types	Pass
Converts type int to string for print lines	Pass
Converts type bool to string for print lines	Pass
Reports type error when mathematical operators are applied to non-integers	Pass
Reports type error when logical operators are applied to non-bools	Pass
Reports type error when comparison operators are applied to non-ints	Pass
Reports type error when list functions are applied to non-lists	Pass
Converts conditional expressions to type bool for conditionals	Pass

List Unit Tests

Test	Result
Implements listhead	Pass
Implements listtail	Pass
Implements listinit	Pass
Implements listlength	Pass
Implements listend	Pass

Higher Order Functions Unit Tests

Test	**Result**
Implements map	Pass
Map can apply function of type $A \rightarrow B$ to list of type A to return list of B	Pass
Functions can be passed as arguments	Pass
Function of type $A \rightarrow B$ can be passed to a function as an argument and applied to an argument of type A	Pass

Recursion Unit Tests

Test	**Result**
Functions can call themselves	Pass
Maximum recursion depth prevents programs from never ending	Pass

Conditionals Unit Tests

Test	**Result**
Conditional first case triggers when expression is True	Pass
Conditional second case triggers when expression is True	Pass
Conditional clause can evaluate logical expressions such as 'not True' and 'not not False'	Pass
Conditional clause can evaluate conditional expressions such as 'x == 3' and 'animal == "dog"'	Pass