University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Handwritten Source Code Recognition for Technical Interview Preparation

Solan Manivannan
CID: 00929139

Supervisor: Dr. Mark Wheelhouse

Submitted in part fulfilment of the requirements for the degree of
Master of Engineering in Mathematics and Computer Science

Abstract

Software engineering job interviews usually require developers to handwrite source code that solve small algorithmic problems. The most common approach to practicing such a skill is to handwrite source code, type up the solution on a computer and run against some test cases. However, this approach is both time consuming and frustrating.

This project focuses on streamlining the preparation process by developing an iOS application that recognises handwritten source code using the Apple Pencil, and offers an environment to execute that source code on a local server or third party web application. The main idea is to fix the recognition result output from the current state of the art handwriting recognition system for the English language and mathematical expressions using a pipeline of different techniques.

The project managed to reduce the number of errors by around 50-60% and the time taken to get the correct recognition after manual user fixes by over 50% for 44 handwritten source code samples from 10 students.

Acknowledgements

I would like to thank:

Dr. Mark Wheelhouse for his support and suggestions throughout this project.

Dr. Giuliano Casale for his guidance as a personal tutor over the last 4 years.

My friend **Adrian Nicol**

Microsoft for giving me the opportunity to do my industrial placement in their London office, and providing me with a return offer in Bellevue, WA. My manager, mentor and team who have greatly helped develop my skills outside of University. The overall experience has made me happy to have done this degree and enter this industry.

My **parents** and **brother** for everything.

My **late grandparents** who were with me at the start of this degree, but could not make it to the end. They have all always wanted me to focus on my education, and provided their unwavering support.

Contents

Chapter 1

Introduction

At most of the top tech companies, and more recently many other companies, a series of algorithm and coding interviews are conducted to hire interns and full time employees as software developers. However, these interviews require the interviewee to handwrite their code on paper or on a whiteboard as opposed to typing up their code into an integrated developer environment (IDE). This crucial difference in format requires the interviewee to practice problem solving in a similar manner, and is a completely different skill to develop.

The current approach to preparing for a technical interview is illustrated in the figure below. The interviewee can find a practice question online (e.g. InterviewBit or LeetCode) or in a textbook (e.g. Cracking the Coding Interview). They can then proceed to answer the question by hand on a whiteboard, or if they do not have access to one, then on paper. In order to find out whether their solution 'works', the best way is to type up their code, compile and run against some test cases. Websites like InterviewBit and LeetCode provide online IDEs for the user to type up their code and evaluate their solution in terms of correctness, and time and space complexity.

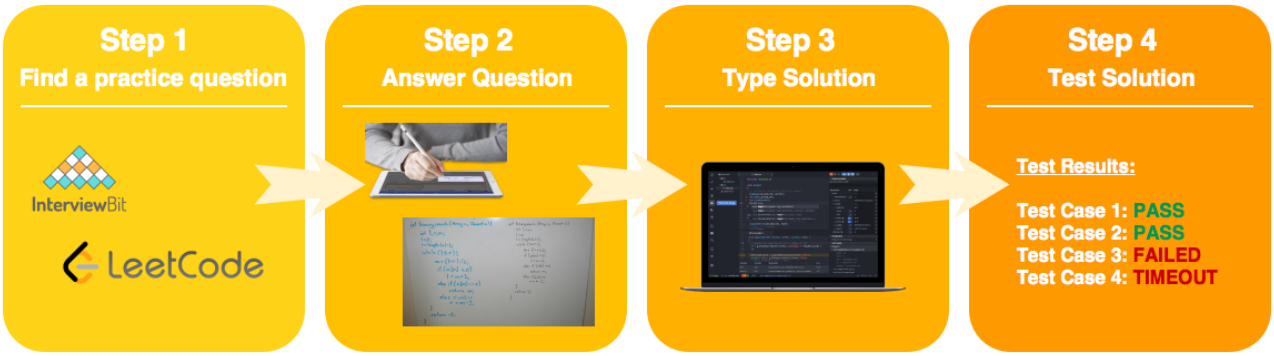
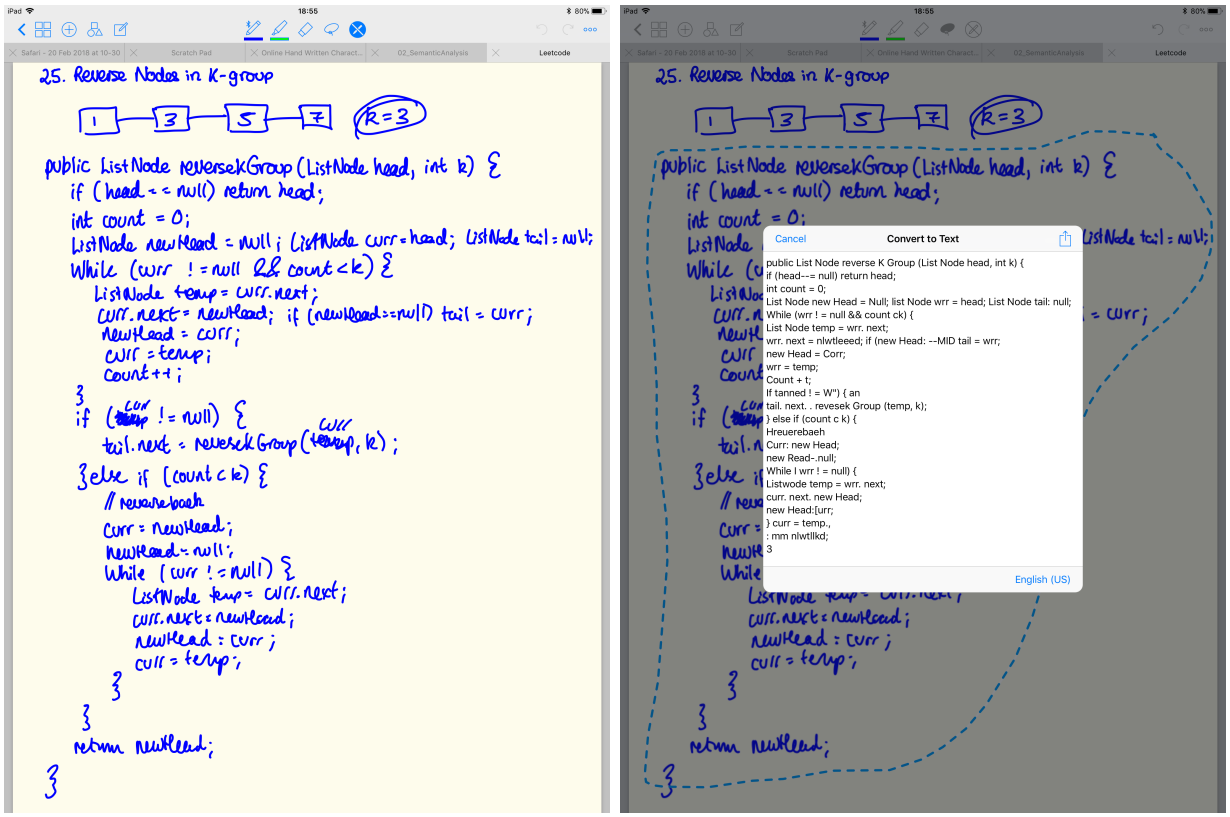


Figure 1.1: Feedback loop for each question

However, there are several problems with the above approach. Step 3 is a real hindrance to preparation and the act of typing does not add any value for the interviewee, yet it needs to be done in order to evaluate the proposed solution and learn how to improve one's problem solving technique. This issue is further amplified considering the number of questions there exist to practice on. Furthermore, if at step 4 your solution does not 'work', the process is even more

tedious having to fix the handwritten code and then correct the typed solution. As a result, many people resort to skipping step 2, and instead go straight to typing up a solution, and debugging the typed up solution. Although this can give the impression of helping the person improve their problem solving technique, it does not necessarily imply that similar execution can happen in an interview environment.

During my techincal interview preparation I decided to continue to adopt the approach of handwriting on my iPad using the Apple Pencil in a note taking application. The majority of the top note taking applications on iOS offer handwriting recognition using the MyScript [1] engine. Below is a picture of one of my own examples using the GoodNotes app [2].



(a) Handwritten source code (b) Recognition output from MyScript engine

Figure 1.2: Handwritten Source Code in the Good Notes app (iOS)

Using the MyScript engine to recognise handwritten source code gives a fair result, but there are still too many errors. Since the result is a block of text, fixing the errors would require copying and pasting this text into a text editor, and then making the fixes via a virtual/physical keyboard. This is still a cumbersome way of approaching every practice question. As a result, during my preparation for technical interviews, handwriting source code on my iPad gave no major benefit aside from being in digital storage.

Why can't state of the art handwriting recognition engines work on handwritten source code?

The majority of research on handwriting recognition in the last couple of decades has been on the English language and mathematical expressions. Source code has a different structure to the English language, and so the problem of recognising handwritten source code is rather different and trips up current state of the art handwriting engines.

The current handwriting recognition systems make use of sophisticated machine learning techniques such as neural networks that have been trained on large (English Language) data sets. However, there is not enough training data (handwritten source code samples) for a company to replicate the same procedure and create a state of the art system for handwritten source code recognition. This training data is not readily available as handwriting source code is only of temporary value e.g. to slowly work through a problem that requires deep thought, or for technical interview practice. As a result, handwritten source code is usually discarded quickly. Furthermore, the large number of programming languages available, and the variety in structure of these programming languages, makes getting training data that conforms to some standard very difficult. Even if we did pick a specific language e.g. Java, people adopt different styles in their code e.g. placing curly braces on new lines. These reasons make it hard to get a large suitable dataset to apply machine learning techniques on.

My solution involves creating an iOS app that works with the iPad Pro and Apple Pencil that will ultimately give the user a more streamlined process for preparing for technical interviews.

The main contributions are listed below:

- A pipeline that post processes the handwriting recognition results from the MyScript Interactive Ink engine by utilising the syntactic structure and semantics of the programming language the source code is written in.
- A touch friendly user interface for manual recognition fixes that requires minimal use of a virtual keyboard. There is no app that delivers a user interface for quick easy manipulation of the handwritten recognition output.
- Compilation and Execution of code on a local server, and results reported in-app.
- Third party integration (LeetCode) with access to questions, and automatic submission of recognised handwritten code. Results of test cases are then reported in-app.
- Loading and saving digital ink files along with recognition fixes in permanent storage. Persistence of recognition fixes between each conversion of the same file. These features complete the app and increases its usability.
- Generation of handwritten source code samples using my handwriting dataset and a given string. Useful in the evaluation.

The creation of such a tool can also help the interviewer in the interview itself. At the moment, it is up to the interviewer to judge the handwritten code by the interviewee. Considering that there are several ways to solve one question (and several programming languages the interviewee can choose to answer in), the ability to judge code without access to a computer can be quite difficult. Furthermore, having access to an app that can save and reload handwritten source code files, and can compile and execute easily, can help the interviewer after the interview has been conducted. Therefore, such an app can benefit both the interviewee and the interviewer.

Chapter 2

Background

Recognition of English handwriting is not a new problem, and proposed solutions have grown over the past few decades. There are two types of handwriting recognition. Offline handwriting recognition uses a raster image of the handwritten text e.g. using a camera to take a photo of a piece of paper or of a whiteboard. Online handwriting recognition uses the data of each stroke the user makes in their handwriting, and so the recognition is able to take place while the user is writing. By capturing the temporal information of the writing, online recognition enhances accuracy over offline. With the strong growth in the number of devices that support input using a stylus, many more people can benefit from an online handwriting recognition system.

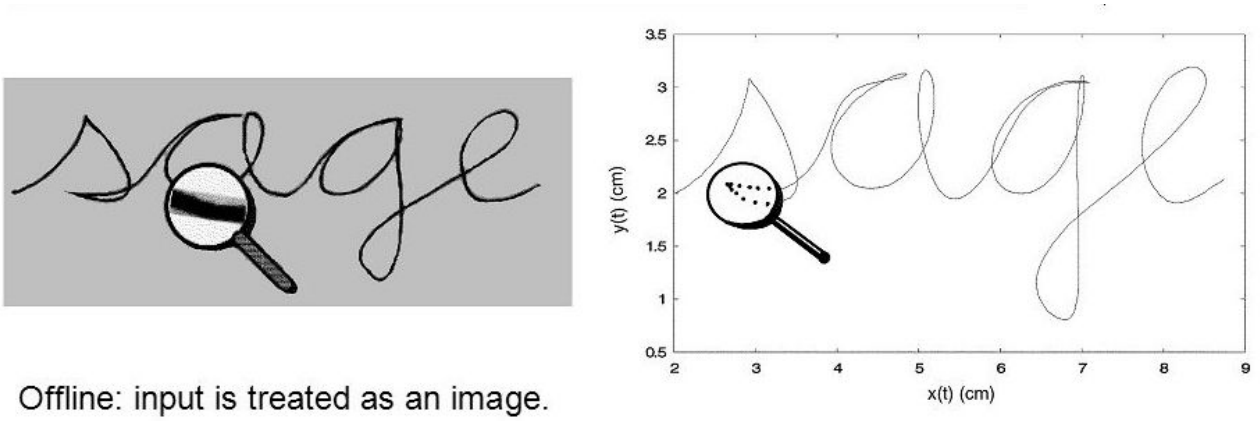


Figure 2.1: Offline vs Online Recognition

There are two distinct writing styles: printed and cursive. Some people also interchange between the writing styles in one piece of work. Past research projects that aim to build handwriting recognition systems from the ground up sometimes restrict themselves to printed handwriting, as this is an easier subproblem compared to that of cursive and mixed handwriting.

It is extremely difficult to create an algorithm that can recognise the handwriting of every single person. There is huge variability in stroke numbers, their order, shape and size, and tilting angle. Therefore, a decision would have to be made on creating a writer dependent or writer

independent recognition system, the latter being a significantly harder problem requiring more training data.

A number of preprocessing steps may need to be carried out because otherwise the input to a handwriting recognition system can have huge variability, for example in the size and angle of the writing. The most common preprocessing strategies implemented in a handwriting recognition system include: thresholding (separate foreground from background), noise removal (removal of ink splotches or lines of ruled paper) and character and word segmentation.

2.1 Building a handwriting recognition system from the ground up...

Hidden Markov Models (HMMs) and Neural Networks are popular algorithms used in the development of both offline and online handwriting recognition systems. Therefore, I considered whether I could modify these algorithms directly to work for source code recognition rather than for English language. My thoughts are discussed in the following subsections.

2.1.1 Hidden Markov Models

Hidden Markov Models are a statistical approach to solving the problem of handwriting recognition having been successful in the speech recognition domain. They can be viewed as extensions of discrete-state Markov processes. A Markov process is a stochastic process that satisfies the Markov condition: future behaviour depends only on the present state. A discrete state Markov process can be in one of a set of N distinct states, $S_1, S_2, ...S_N$. The Markov condition in probability notation is $P(Q_n = S_i|Q_{n-1} = S_j, Q_{n-2} = S_a, ..., Q_0 = S_b) = P(Q_n = S_i|Q_{n-1} = S_j) \forall i, j, a, b, n$

The Markov process is characterised by its initial state probabilities and the state transition probabilities. For hidden markov models, the observation at each state is probabilistic instead of deterministic (see figure 2.2). Three problems need to be solved to develop an effective Hidden Markov Model: evaluation problem, decoding problem and training problem.

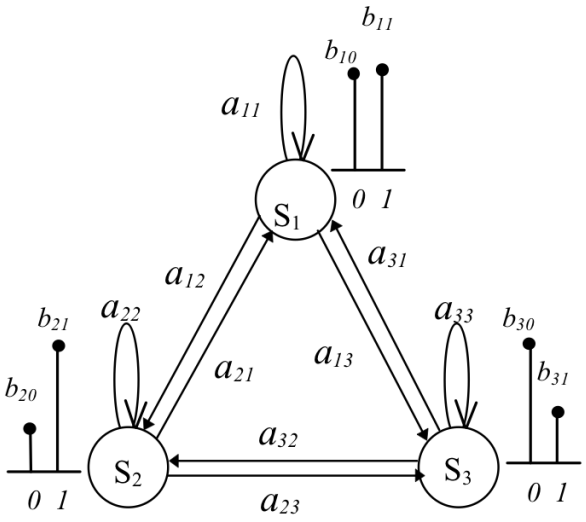


Figure 2.2: Hidden Markov Model with 3 states [3]

HMMs can be used to model letters, words and sentences [3]. In Han Shu's experiments he found that a 7 state left-to-right hidden markov model gave the maximum recognition accuracy for 89 individual letters and symbols. Words can be modelled by joining together the hidden markov models of the individual characters. However, for cursive writing you need to take into account letters such as "i", "j" and "t" where a stroke is made to complete the letter at the end. These dot and cross strokes can be added to the end of the hidden markov model made up of the composition of the hidden markov models representing each letter.

In the context of recognition of source code, the modelling of letters would be exactly the same as for natural English, but numbers and symbols need to have good models as well. However, it may be possible to change the way words and sentences are modelled for source code. Dot notation, camel casing and the general syntax of programming languages can be taken into account for the HMMs modelling words and sentences. However, the question arises whether this modification will change the recognition accuracy of standard english words, which are also plentiful in source code. For example, "Hash", "Map", and "HashMap" would each have their own hidden markov model. Distinguishing between "HashMap" and "Hash Map", i.e. the spacing, will be a problem that is solved outside of the hidden markov model.

2.1.2 Neural Networks

A neural network is an artificial system in computer science that is modelled on neurons in the human brain. It is made up of an input layer and an output layer with one or more hidden layers in-between. The neural network approximates a function that takes the input data and provides a reasonably correct output based on training on example inputs and outputs. The number of nodes in the input layer and output layer need to be decided by the user and is based on the classification task being done. For example, if we were building a neural network for recognising handwritten numbers then the output layer could have 10 nodes, one node to represent each digit. A weighted summation function is applied at each node in the hidden layers and the output layer, and the function uses the values output from the nodes in the previous layer. The

weights are the values determined during the training of the neural network. The result of the weighted summation function is then passed to an activation function, which determines if the neuron is activated or not, and to tell the nodes in the next layer this information.

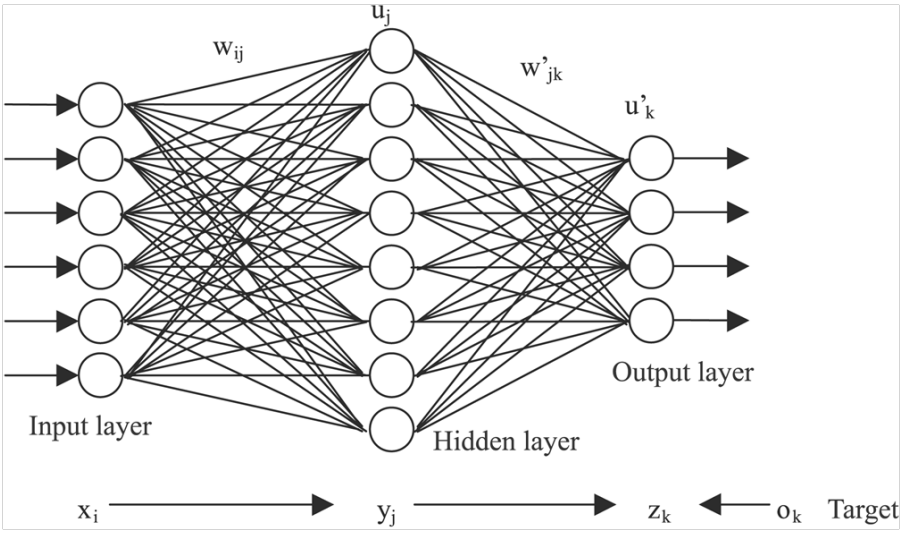


Figure 2.3: Neural Network [4]

For natural English, a handwriting recognition system can be modelled in the form of a Convolutional Neural Network (CNN). CNNs are feed forward neural networks (connections between nodes do not form a cycle) that are used to classify images and are inspired by the function of the visual cortex. Each layer of the CNN extracts features from the previous layer. Low level components such as edges and features are extracted and combined to give high level components. Filters then slide over the image to identify the translation invariant low level and high level features.

CNNs can be trained to recognise individual English words. This can hinder performance when the input is handwritten source code that has dot notation and camel casing, since the model would not have been trained on these instances. For example, dot notation could lead to some words having their first letter replaced with a capital letter (e.g. "m" to "M") since the model has been trained on many of these scenarios. It would be interesting to see if a CNN could be trained on several samples of handwritten source code in a given programming language. However, this would require a huge number of samples covering a large variety of handwriting styles. Gathering this data may not be viable given the time constraints of this project.

Alternatively, individual characters could be passed into a CNN for classification. This would require a method for segmenting the individual characters. One research project on handwriting recognition developed Long Short Term Memory (LSTM) networks to construct bounding boxes around individual characters for segmentation. However, by trying to recognise individual characters, this may hinder performance of recognising standard english words, which there are plenty of in source code. In addition, individual character recognition would not make use of the syntactic structure of programming languages.

An interesting paper on handwriting recognition systems titled Scan, Attend and Read: End-to-End Handwritten Paragraph Recognition with MDLSTM Attention [5] provides a different approach to handwriting recognition. The paper acknowledged a state of the art handwriting

recognition system at the time, which was created using Multi-Dimensional Long Short-Term Memory Recurrent Neural Networks (MDLSTM-RNNs), and aimed to build on this work by creating a multiline recognition system. Many other handwriting recognition systems break the input text into individual lines/words before the recognition phase. Furthermore, this model does not make use of a language model i.e. it is completely dependent on the visual classification of each character without consideration of context. This fact may benefit recognition of handwritten source code because other recognition systems that use a language model are most notably tripped up by the use of camel casing, dot notation and the mix of symbols and letters in source code. However, this may affect the recognition of other words that would otherwise normally be correctly recognised.

It is clear that there is some compromise between using existing algorithms that work for the English language and a new algorithm tailored towards source code. The latter being an original problem that could take some time and resources to solve.

2.2 Augmenting existing handwriting recognition systems to handle recognition of source code

2.2.1 Tesseract OCR (offline)

Research on using offline handwriting recognition systems for source code has been done in the past. Iris [6] is a handwriting recognition system developed specifically for Ruby code. The main idea was for Iris to post process the output of Tesseract, an optical character recognition (OCR) engine that would take as input an image of handwritten source code.

Tesseract is different to other OCR engines as it provides an API to further train the model against new languages and character sets. This gives way to the possibility of training Tesseract to recognise handwriting. However, as OCR engines are designed to recognise computer fonts (which are printed), there rises an issue with forcing it to recognise handwriting that is cursive. Tesseract also makes use of two word lists. The first list contains all the words of the language, and the second word list contains the most frequent words. If during the recognition phases Tesseract runs into a string of ambiguous characters that can be made into an actual word, then low confidence characters can be replaced to complete the word.

Since Iris is aimed at recognising Ruby code, the frequent words list can be composed of the keywords in the programming language. The developer of Iris also trained the Tesseract model for his handwriting using the API.

However, Iris employs another technique that would not suit the purposes of this project. A domain specific language (DSL) was created for Iris to recognise instead of the Ruby syntax as-is. The developer wanted the handwritten text to be as simple as possible in order to increase the accuracy of the recognition results. Therefore, instead of using symbols like '+' and '-' the DSL creates an alias, 'plus' and 'minus' respectively. This would not suit the purpose of this project, where the person is required to handwrite in the correct syntax of the programming language of their choice. The output of Tesseract is then post processed to fix common error strings.

Iris achieved varying levels of consistency and accuracy that for me indicates is not good enough for the purposes of my application. If the user is required to manually fix several changes in the recognised output (using a virtual/physical keyboard), then it still severely hinders the process of practicing. The developer of Iris suggested future work to apply a machine learning layer to improve the accuracy. However, considering that OCR engines are designed to achieve an accuracy between 80% and 95%, I feel that applying a post processing step to the results may not be enough to get the improvement in consistency that I desire.

Codeable [7] is an app developed to compile handwritten source code written in the programming language C. The project took inspiration from Iris, but targeted improving the preprocessing stage (so that it could work for images of whiteboards) and the postprocessing stage (to get better accuracy). The frequent word list used in the Tesseract engine included the keywords of the programming language as well as common variable names like 'str', 'total' and 'count'. Misspellings in the output of Tesseract was fixed by calculating the edit distance of the output with common strings. Missing delimiters in the output was fixed using regular expressions. If the code could not compile at this point, then the output was further processed to handle two types of errors: missing brackets and undeclared identifiers. However, the strategies for handling these errors were naive, for example always setting the type of an undeclared variable to int. The project restricted investigation to printed handwriting, which made the raw OCR output have a high accuracy, but the post processing made minimal improvements, and in some cases made the results worse.

However, considering that the interviewee will be handwriting in an environment that does not really give time for printed writing, I do not justify a technique that utilises an OCR engine. That being said, I think that some of the other techniques used in these projects could be investigated individually myself to see if they are worth including in my system.

2.2.2 MyScript (online)

Personally, I am very familiar with how handwriting on a tablet has evolved with almost all of my academic notes taken on an iPad since 2012. In preparation for my technical interviews, I continued to use my iPad with the GoodNotes app to handwrite my source code. The GoodNotes app uses the MyScript handwriting engine for handwriting recognition, and so I can use this preparation material to evaluate the MyScript handwriting engine as-is on source code. Although the results are not as accurate as recognition of natural English, they were quite close, and many of the errors felt to be common enough, and potentially fixable in post processing. Given this insight, I was more inclined to take advantage of an already existing state of the art online handwriting recognition system.

Zhi and Metoyer [8] investigated utilising the MyScript engine for source code recognition. They focused on the Python programming language, and aimed to post process the results from MyScript using the following pipeline.

- Statement classification: Look at first word of each line and then classify that statement.

- Statement Parsing: A top down LL parser was created to parse input and perform left-most derivation.

- Token Processing: Keep track of non-keywords, if later in the source code we come across similar (according to Levenshtein distance) non-keywords then make both instances the same word.

- Statement Concatenation: Remove extra spaces (e.g. extra space may come in recognising dot notation).

Evaluation of the project was based on word error rate (WER) and character error rate (CER) on 45 handwritten source code samples comparing original MyScript with the augmented MyScript pipeline. The original MyScript had 31.31% WER and 9.24% CER compared to augmented MyScript with 8.6% WER and 3.6% CER. The percentages are calculated using the Levenshtein distance between the recognised sequence and the true value:

$$\frac{D + I + S}{L} * 100\%$$

where D is the number of deleted units, I is the number of inserted units, S is the number of substituted units and L is the total number of units. A unit is a word for WER and a character for CER.

Despite the improvement in recognition, there were still errors that slipped through including incorrectly recognising nonkeywords (which affects the token processing step), incorrect recognition of symbols and mismatching brackets. There was also no user interface to fix recognition errors, and so the only way to do this was to copy and paste the text into a text editor and correct it using a physical/virtual keyboard.

Furthermore, despite the final results being worse in comparison to Codeable, the MyScript engine is able to handle a wide variety of handwriting styles, most importantly it can handle cursive writing as well.

2.3 My proposed solution

The final product should be usable by any developer, and so should work with a variety of handwriting styles. Therefore, the focus will be on utilising the currently existing state of the art online handwriting recognition system provided by MyScript. Given the time constraints of this project, it would not be feasible to collect a large amount of training data to build a recognition system from scratch using machine learning techniques. This project will instead aim to fix the recognition errors in the final output of MyScript through a pipeline of different processing strategies. As a result, this project also avoids solving the same sub problems associated with building a handwriting recognition system such as noise removal and character/word segmentation.

This project will focus on handwriting source code in the Java programming language. The core pipeline can then be developed around the syntactic structure and semantics of the programming language. For example, symbols such as brackets, square brackets and curly braces

are important in the language (and in the majority of cases are required). Java is also a strongly typed language, which means that you cannot assign something that is of the wrong type to a declared variable. This will allow us to fix recognition errors such as int i = o (which should be int i = 0).

Inspiration will be taken from the Compilers group project (WACC), where this pipeline will make use of the generated Lexer and Parser provided by ANTLR and the Java grammar. The tokens and parse tree provide a way to navigate the handwritten source code, and give some correspondence to location. In addition, semantic recognition fixes can be done through a class that uses the visitor pattern to walk the parse tree, and monitor functions, variables and types through a symbol table.

In order to avoid requiring access to a computer to complete the feedback loop, any final errors should be fixable by the user through the UI. Execution of the code will either be on a local server or via a third party website that provides such technical interview problems e.g. LeetCode and InterviewBit. The idea is to limit redirecting the user to a computer and physical keyboard.

Furthermore, this project will also avoid any configuration changes in the MyScript runtime engine that are dependent on the choice of programming language. For example, it is possible to provide the engine with a custom lexicon so that there is an increase in probability of recognising certain types of words. This would be useful for correcting the recognition of certain variable names and keywords as well as camel casing. However, if the app was to provide support for multiple programming languages, then the engine would need to be reconfigured, and at the moment that is not possible to do without recompiling the whole application.

2.4 Pipeline Decisions

An initial thought for this project was whether the *correct* source code could be created through just the text recognition candidates offered by the MyScript output. For example, could one just swap in an alternative candidate at each point a compiler error was thrown? After a few handwritten source code examples, I ruled out that this would not be possible as in some scenarios the candidates list did not contain the 'correct' alternative. This strategy also does not take advantage of the few certainties in this problem such as the syntactic structure of the programming language.

After reviewing a few more samples of handwritten source code along with the recognition output, I came up with a couple of heuristics that the pipeline could be built around on.

(1) Symbol Pairs Heuristic: '()', '[]' should come in complete pairs on a single line. Therefore, if there is no matching pair I can check the alternative candidates to see if the missing symbol exists. This is a fair heuristic to have as it only breaks down when either the user overflows a single statement onto multiple lines, or in an implementation of an anonymous inner class (see an example below). These are rare instances in the technical interview problems that will be solved. However, the same approach cannot be done for $\langle, \rangle$ since these are not only used for generics in Java, but also for boolean conditions e.g. "i $\langle$ 10" in for loops.

```
Arrays.sort(arr, new Comparator<String>() {
```

```
    @Override public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
```

(2) End of Line Heuristic: The last character on a line should be a semi colon or an opening or closing curly brace. If it is a closing curly brace, then this must be the only character on that line. This heuristic breaks down if the user adopts a style where both the opening and closing curly braces go on new lines, if one-statement 'if' blocks are used (curly braces optional) or if the user does not have enough space on the line, and overflows onto the next line. However, I feel that the user can avoid these scenarios, and still write source code that will not be judged differently in an interview environment.

(3) Post Lexer Heuristic: The lexer will output a series of tokens. There are a few tokens that have a compulsory previous/next symbol token e.g. "for (", "} else". My first thought was for recognising an issue and making a change within the ANTLR parser, but I found that since the parser does a look ahead, any errors may prevent the parser from reaching the stage of noticing and allowing me to make the change. The ANTLR parser is also time consuming. Therefore, instead I decided to do this step after getting the tokens from the lexer, and then looping through the tokens.

Formatting: Two common recognition errors in the MyScript output are spacing (sometimes spaces are inserted where the user did not intend for, and sometimes spaces are not inserted when the user did want a space) and camel casing (a word with camel casing will have spaces inserted between the individual words). For example, "if (p != 1)" is recognised as "if (p! = 1)". Also "HashMap" is recognised as "Hash Map". I decided that it would be best to 'redo' the spacing in the recognition, as opposed to fixing each spacing issue. This is because identifying a spacing issue is difficult since there are multiple scenarios. I figured that this would best be handled in the post lexer output as my method uses the tokens.

ANTLR: This stage of my pipeline will be handling identifier names and the semantics of the source code. This is because identifier names can be tracked through the symbol table data structure, and I can track when I run into an identifier using the visitor pattern. Although this stage requires building the parse tree, which is a time consuming process, it is easier to track when a variable/function declaration is being made (to add to the symbol table) and when future uses of a variable/function are done, in comparison to looping through the lexer tokens.

The final pipeline is illustrated in the figure below.

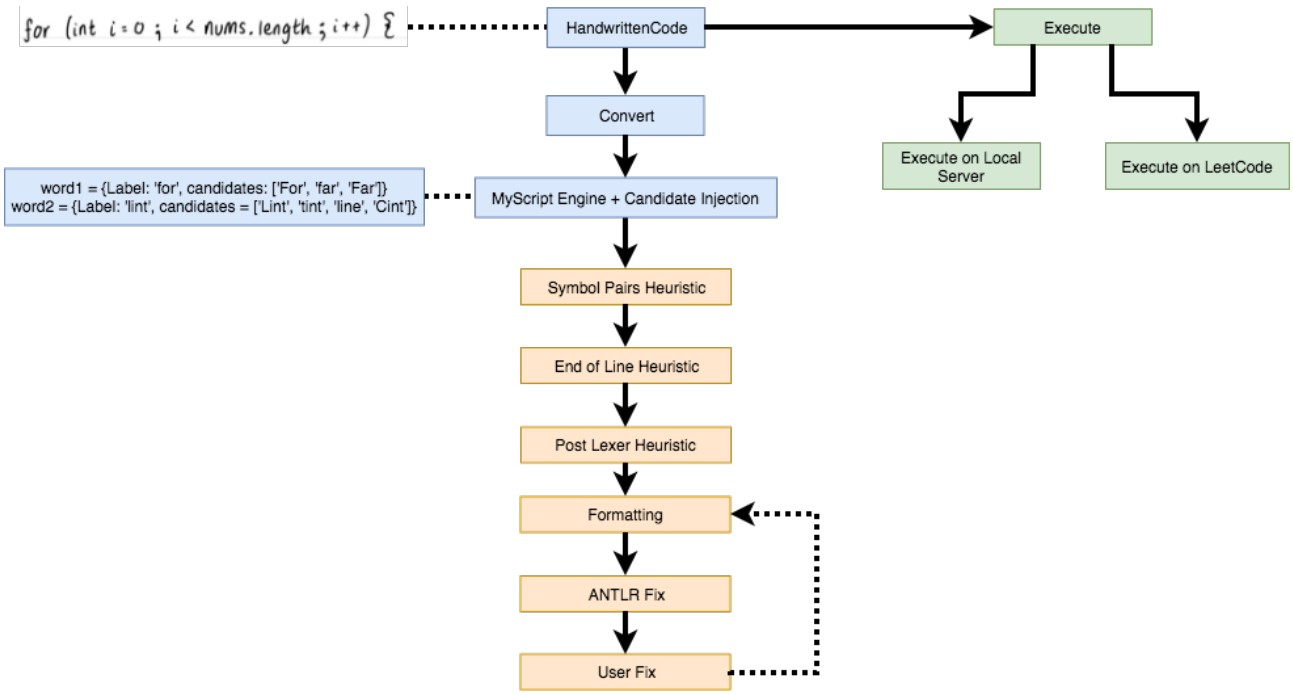


Figure 2.4: Pipeline

Every time a user fix is made, part of the pipeline is repeated starting from the formatting part. This is because I need to print the syntax highlighted output of the code in the UI. Note that the formatting stage also needs to be done prior to ANTLR, because formatting reduces errors caused by spacing issues that would otherwise cause more error nodes to appear in the parse tree. Any user fix should help reduce the number of errors in the source code, which will benefit the ANTLR part of the pipeline since it will encounter less error nodes. The whole pipeline could be repeated, but I decided against this because a single user fix is likely to not have a major effect to the previous parts of the pipeline.

As the ANTLR part of the pipeline is the most time consuming, the user can toggle the ANTLR on and off so that every user fix does not trigger the ANTLR stage.

One technical challenge of implementing this pipeline is tracking the changes that are being made in order to then display in the UI where a particular change was made in the pipeline. This problem will be explained in the implementation details.

The aim with this pipeline is to get it to be configurable for different programming languages. The language independent recognition fixes are done early on in the pipeline, and the language dependent recognition fixes are done at the end in the ANTLR stage.

Chapter 3

Final Application Flow Chart

- 1: On application launch the user is presented with a screen to select a question type. The user can select a question from a third party website or just create a blank file that will just execute locally.
- 1a: List of LeetCode questions along with their description, acceptance, difficulty level etc.
- 2: If custom is selected on the previous screen, then a new file is created that will execute handwritten code on a local server.
- 2a: Handwritten custom code.
- 2b: Handwritten custom code that is recognised using the MyScript engine and processed by my pipeline. The highlight toggle is enabled to display the bounding boxes.
- 2c: Execution of handwritten custom code on a local server. Works through SSH and SFTP. Compiler errors or execution result is displayed in-app.
- 2d: Renaming a file can be done by tapping the document title, which will bring up a popup.
- 2e: List of previously saved documents.
- 2f: Custom file settings. You can update the global injection candidate list, server settings and LeetCode settings.
- 3: Creation of a new file with the same title as the LeetCode question selected.
- 3a: Handwritten source code answering chosen LeetCode question.
- 3b: Handwritten source code that is recognised using the MyScript engine and processed by my pipeline. The highlight toggle is enabled to display the bounding boxes.
- 3c: Execution of the code on the LeetCode server and display of results.
- 3d: LeetCode file settings. You can update the global injection candidate list, server settings and LeetCode settings. Also displays the LeetCode question information here in case the user had forgotten the description.

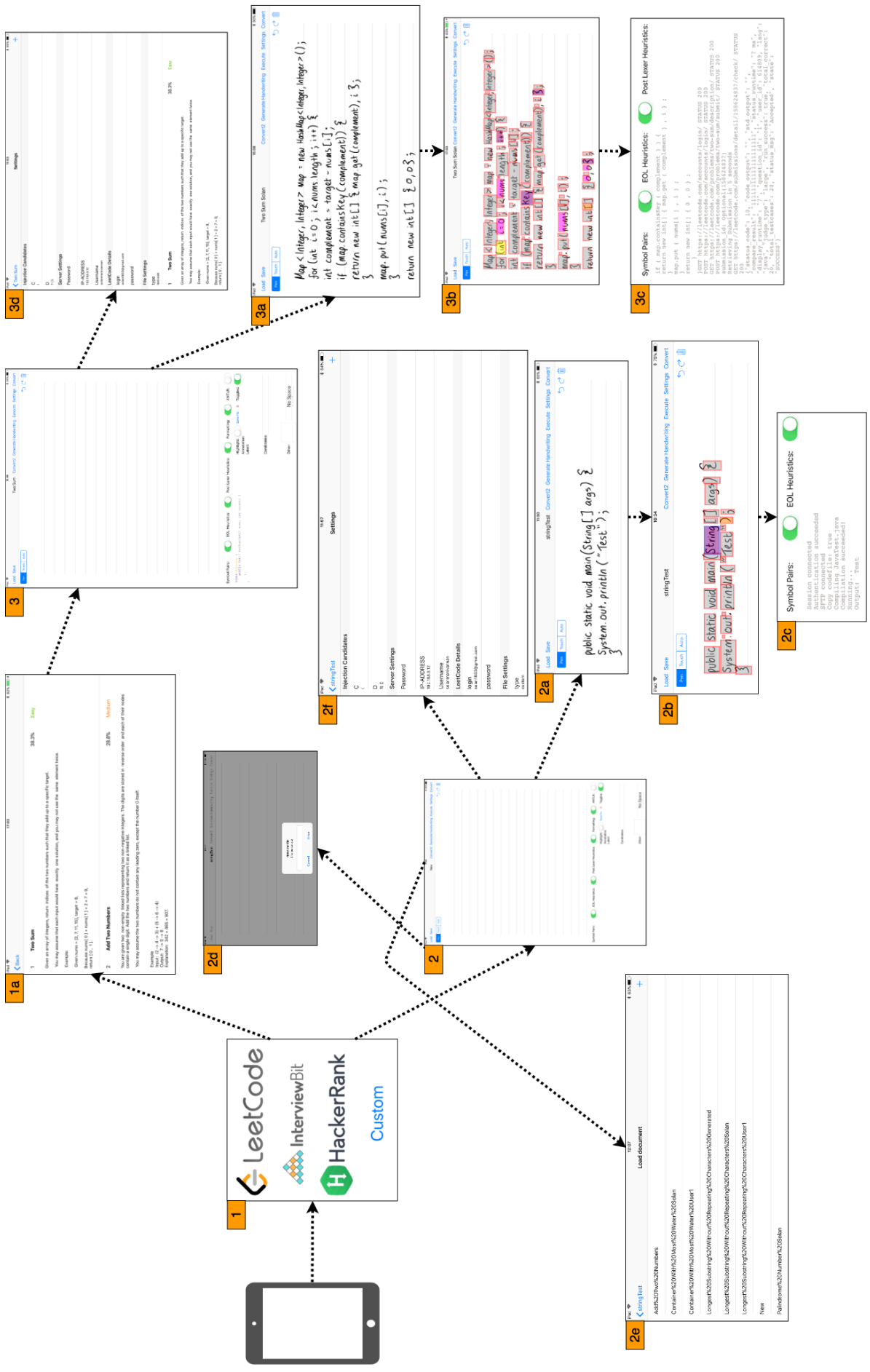


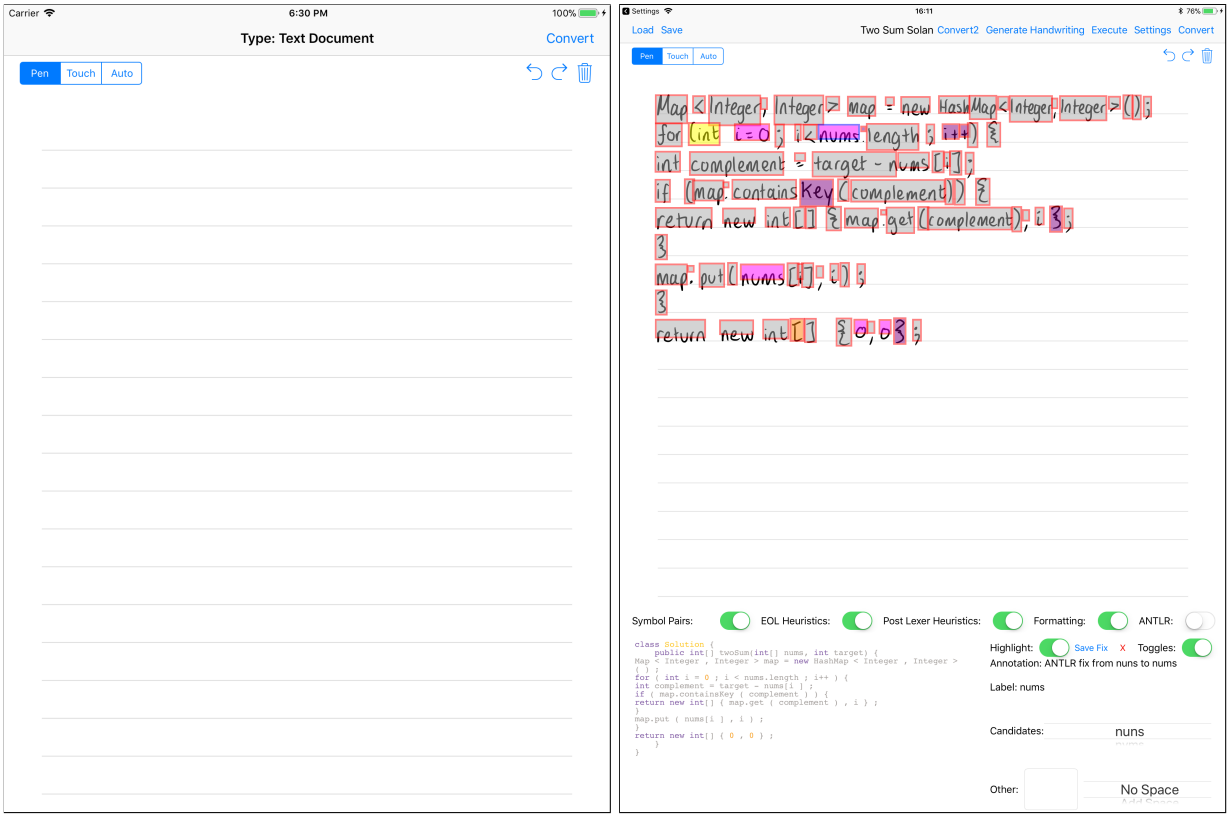
Figure 3.1: iOS Application Flow Chart

Chapter 4

Implementation Details

This chapter will follow a chronological order of the pipeline outlined above in the proposed solution. The remaining discussion will be on the other technical features of the product such as loading/saving content, execution on a local server, execution on third party websites etc

The starting point for this project is a fork of the 'Starter' project [9] provided by MyScript that provides a basic user interface for the user to draw on and then convert to text via the MyScript runtime engine. It also provides undo and redo functionality for the handwritten strokes. Throughout my project I continue to extend the implementation to offer other important functionality.



(a) Starter App - Forked Repo

(b) Final App Example

Figure 4.1: Before and After Implementation

The MyScript handwriting engine can be controlled through a configuration file. In this project the only change that has been made from the default configuration is increasing the number of text recognition candidates to 20 (the upper limit as opposed to 3)

4.1 Aggregating information on each handwritten object

The MyScript runtime engine is able to provide access to attribute information on each handwritten object (decided by the bounding boxes placed by MyScript). However, there is no API to easily access this information, we can instead export the handwritten data in a JIIX (short for JSON Interactive Ink eXchange format) string. Attributes such as bounding box coordinates, width, height, label (highest scoring text recognition candidate) and a list of alternative text recognition candidates can be extracted from this string. These attributes are stored in a newly created data structure called Word (detailed in figure 5.2), and will be used later on. The Word objects are collected together into a double array [[Word]. The first array contains the Word objects of the first line of handwritten source code, the second array contains the Word objects of the second line of handwritten source code etc etc.

Note: The text recognition candidate list is ordered in descending likelihood, but the score value for each candidate is unavailable.

4.2 Pipeline 1: Candidate Injection

At this stage of the project, I decided to evaluate the chosen label and the text recognition candidates for each handwritten object that was output by the MyScript engine. It did not take too many examples to notice that there were common recognition errors that did not have the correct fix in the list of candidates. As a result, I decided to have a candidate injection list, which the user can manually update. At the time of building the Word objects, I inject more options into the candidate list based on the label. e.g. '+D' $\rightarrow$ ['+1)', '+))'] However, it is important to try to keep this list as small as possible because it is computationally expensive (string matching and string building both need to be done, as well as covering all possible permutations of the injection). Future investigation into reinforcement learning, could allow for the app to dynamically update this list.

4.3 Pipeline 2: Mismatching symbol pairs

This now sets up the first passthrough at fixing recognition errors - specifically the pairwise symbols (curly brackets, square brackets). I have found that the number of false positives for these symbols is very small. Assuming that the user conforms to a code-style where each line of code has 'complete' pairs of these symbols then we can do a first pass of the code to count the number of each open and closing symbol. If there is a mismatch in the count, we then go through the candidates for each word on that line to see if that has the missing symbol. We count how many potential fixes there are, if there is only one potential fix, then that fix is chosen. This technique clearly breaks down when the recognition fails to find both the open

and closing symbols. Provided that the user follows 'good' code style, we should expect the count of open and closing symbols on each line to be small.

4.4 Pipeline 3: End of line heuristics

As previously mentioned, we are looking at the last character of each line to see if it is a semi colon or an opening or closing curly brace. A closing curly brace should be the only character on that line. The way this check is done is by looking at the last Word object on each line, and if we spot that it is the only character on that line, then we can replace it with a closing curly brace. If there are other characters on that line and it does not match an opening curly brace or a semi colon, then my implementation will check the alternative recognition candidates to see if it matches either of these symbols. If there is a match, then a replacement is made. This technique allows me to easily update the Word object without having to track indices in a string or token.

4.5 Pipeline 4: Post lexer heuristics

Example: "for lint" to "for (int"

The lexer will output a series of tokens, which we can quickly loop through and identify a token we want to check by comparing the token types (integers). At this stage we are only trying to identify a small number of tokens ("for", "if", "while", "else" etc) and either their preceding/succeeding token. If the preceding/succeeding token is not what is expected, then a change is made to the corresponding Word object. This technique requires us to also keep track of where we are in the [[Word]] and also at what index in the current Word (since multiple tokens maybe within the same Word). Therefore, once we spot an error, a change can be made to the correct Word at the correct location.

4.6 Pipeline 5: Formatting

Example: "if (p! = 1)" to "if (p!= 1)", "Hash Map" to "HashMap" , "MAX _ VALUE" to "MAX_VALUE"

This stage of the pipeline also works on the tokens output from the Lexer. It essentially identifies where spaces are supposed to be inserted rather than identifying spacing issues from MyScript and then fixing them.

The camel casing issue is fixed by joining a series of identifier tokens, provided that the starting identifier token is not the word "String" since that is a type. This method does break down when new object types are used such as "NodeList a = new NodeList();" being output as "NodeLista = new NodeList();". The user can fix this in the UI by tapping on the bounding box and then tapping on the add space button. In the reinforcement learning part of this app, the app can start to keep track of these types and ignore this case like for "String". Another

issue that came up was the method in the String class called "charAt" that would be recognised as "char At" and would *not* be corrected by this stage of the pipeline since char is not recognised as an identifier. In these cases the user would have to manually delete the space in the UI by tapping the delete space button.

The other spacing issues such as "! =", "+ +", "nums [0]" are sorted out by identifying when these two tokens appear next to each other. Many errors are sorted in this stage of the pipeline, but there is no indication to the user by a change in the colour of the bounding box (like what is done for other fixes). This is because spaces are not found within bounding boxes.

This stage of the pipeline also handles outputting syntax highlighted code to the bottom left of the screen, and this is only possible when having a code string with spacing all sorted out. Syntax highlighting is done using the Highlightr framework [10].

4.7 Pipeline 6: ANTLR Semantic Fixes

Examples: "int i = o" "int[] nums = {o, o}; int x = nuns.length"

The ANTLR4 runtime was added to the iOS project so that a lexer and parser could be created and used natively on the device without having to connect to a local server. The Java 9 grammar was also used to create the lexer and parser files for the Java programming language.

In order to keep track of the match between a token and a Word object, I first identify the line of the starting token of the code body (since if the user is answering a LeetCode question they do not handwrite the method header). A parse tree is then created using ANTLR, which is the most time consuming part of this whole pipeline. The parse tree is then visited using a custom tree walker and symbol table. When a terminal or error node is visited, this can increase the indices to keep track of the location with the [[Word]] objects. When a variable declaration is made, the variable is added to the symbol table. Any other identifier that is visited is then checked against the symbol table to see if it exists (or if a close match exists using Levenshtein distance). For single character identifiers like "i", and "o", they can be replaced with "1" and "0" if the type is an integer variable and "i"/"o" is not an existing variable, since this recognition error happens quite often.

I decided to set the Levenshtein threshold value to 65%, since a lot of variable names are three letters such as ans and res, and sometimes their recognition has an incorrect letter. However, this does affect other cases such as identifiers with the names "min" and "max". Both of these identifiers are usually encountered within the same snippet of code when answering a LeetCode question.

I decided to only calculate the Levenshtein distance between words that have the same length. Since we are using an online handwriting recognition system, all strokes should generate a character - ideally there should not be extra characters or fewer characters recognised. However, I have since found that mixing symbols, operators and numbers with characters does sometimes cause fewer characters to be recognised. Therefore, it may be useful to calculate Levenshtein distance between words of different lengths as well. This may be a setting that could be controlled by the user if they find that MyScript frequently recognises fewer characters.

Another decision that needs to be made with the Levenshtein algorithm is whether or not to take into account case sensitivity. In some examples I have found that words beginning with "i" or "l" are recognised with a capital letter instead of lower case. If the algorithm took into account case sensitivity, then it would not replace the incorrectly recognised variables "L" and "I" with the lowercase counterparts since the score would be below the threshold. Therefore, I have decided to ignore case sensitivity.

Any compiler errors in the input to the ANTLR parser can trigger one or more Error Nodes to appear in the parse tree. If this happens, then this prevents any fixes from being made for those error nodes. One way to counter the fact that the parse tree may contain Error Nodes that prevent potential fixes from being made, is to use ANTLR to classify each line of code. With that in mind, instead of using the visitor pattern, which breaks down when Error Nodes exist in the parse tree, we can instead loop through the tokens and fix additional recognition issues such as "int[] test = {0,0 3" since ANTLR can tell us that this is a local variable declaration with a type of int[].

4.8 Pipeline 7: User Fixes

The bounding boxes (UIViews) displayed on the user interface (created from the information stored in the Word object) each have a UITapGestureRecognizer attached to them. When the user taps on a bounding box, the border colour of the bounding box changes, and the bottom right part of the screen loads information about the bounding box. This opens up the opportunity for the user to make any manual recognition fixes.

The user can choose an alternative text recognition candidate via a scroll wheel (UIPickerView), they can add/delete a space (UIPickerView) or they can enter a completely new alternative (UITextField) via the virtual keyboard. Any recognition fixes made in the pipeline will also have an annotation displayed in this section of the user interface. If the pipeline triggered multiple recognition fixes for the same bounding box, then all annotations will be displayed in order.

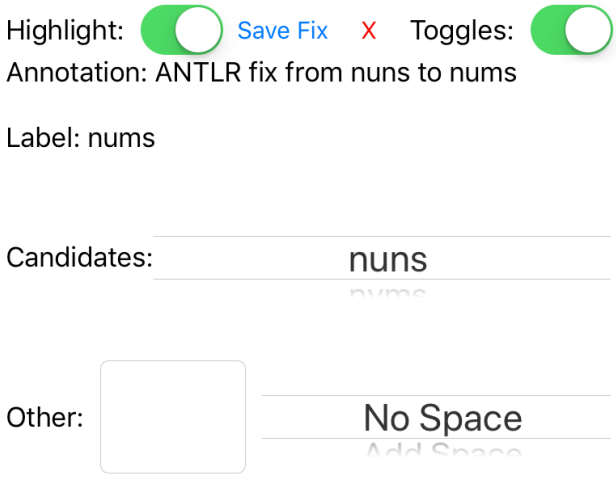


Figure 4.2: User Fixes UI

However, a single manual recognition fix made by the user here can actually improve the results of the pipeline - especially the ANTLR stage. This is because the ANTLR stage is less effective the more Error Nodes there are. So if you can remove an error through a manual user fix, you can increase the effectiveness of the ANTLR stage of the pipeline. Therefore, once the user presses save fix, and provided the ANTLR toggle is enabled, the ANTLR stage of the pipeline will be repeated.

4.9 Execution

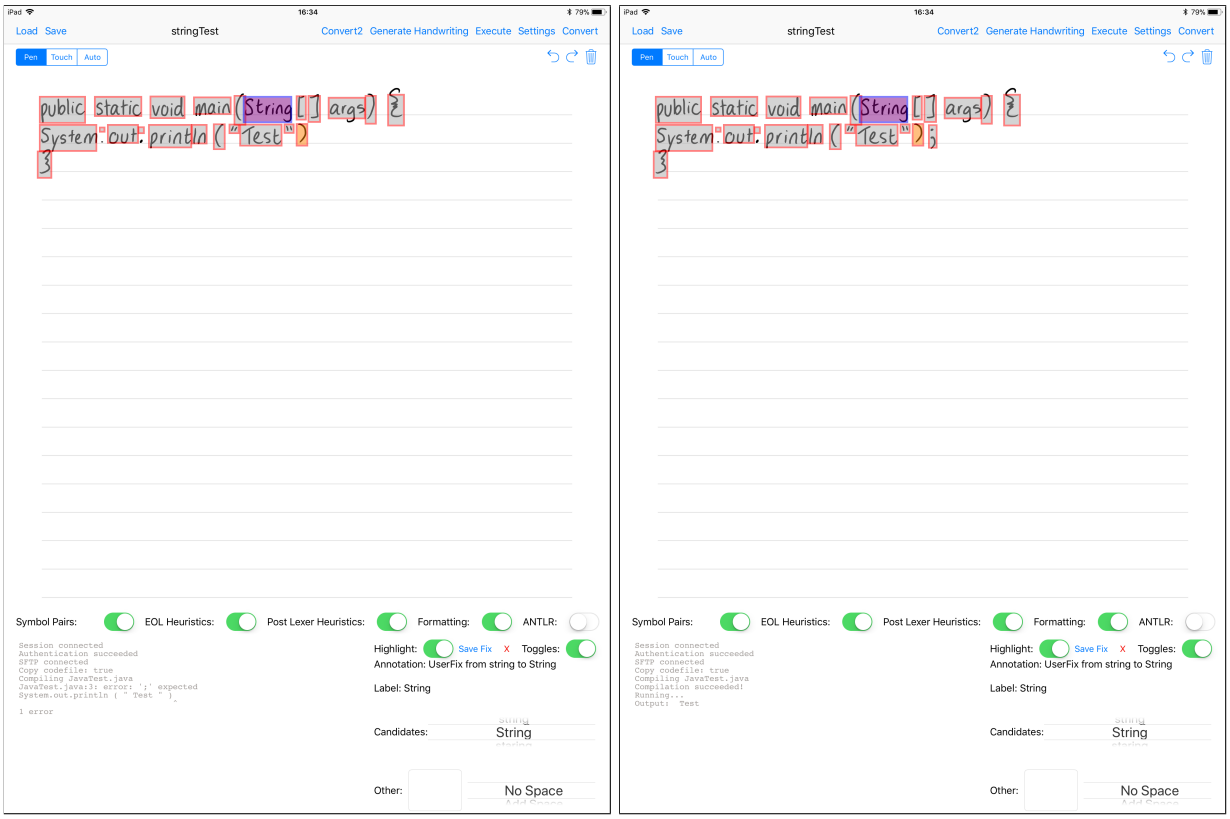
4.9.1 Custom - SSH local server

Custom files are instances where the user is, for example, answering a question from "Cracking the Coding Interview". As there is no online web application that provides a testing facility for these questions, I decided to still allow the user to run this code on a local server against test cases that they write. Custom files are also useful for just writing and running short snippets of code.

The user handwrites their source code as normal, but has the option to either place their code in an empty class and run whatever they have in their main method, or replace a {body} placeholder in a Java file located on their computer, and then run whatever their main method is.

If the user decides to put their source code in an empty class, then the application will then create the text file on the iOS device and transfer it to the local server using SFTP. Commands can be run to compile and execute the Java code in the same SSH session. All results (including stderr) are redirected to stdout, and then returned to the iOS application.

The implementation for this part of the project centred around using the NMSSH framework [11]. The local server settings can be configured within the app in the settings page (IP address and password).



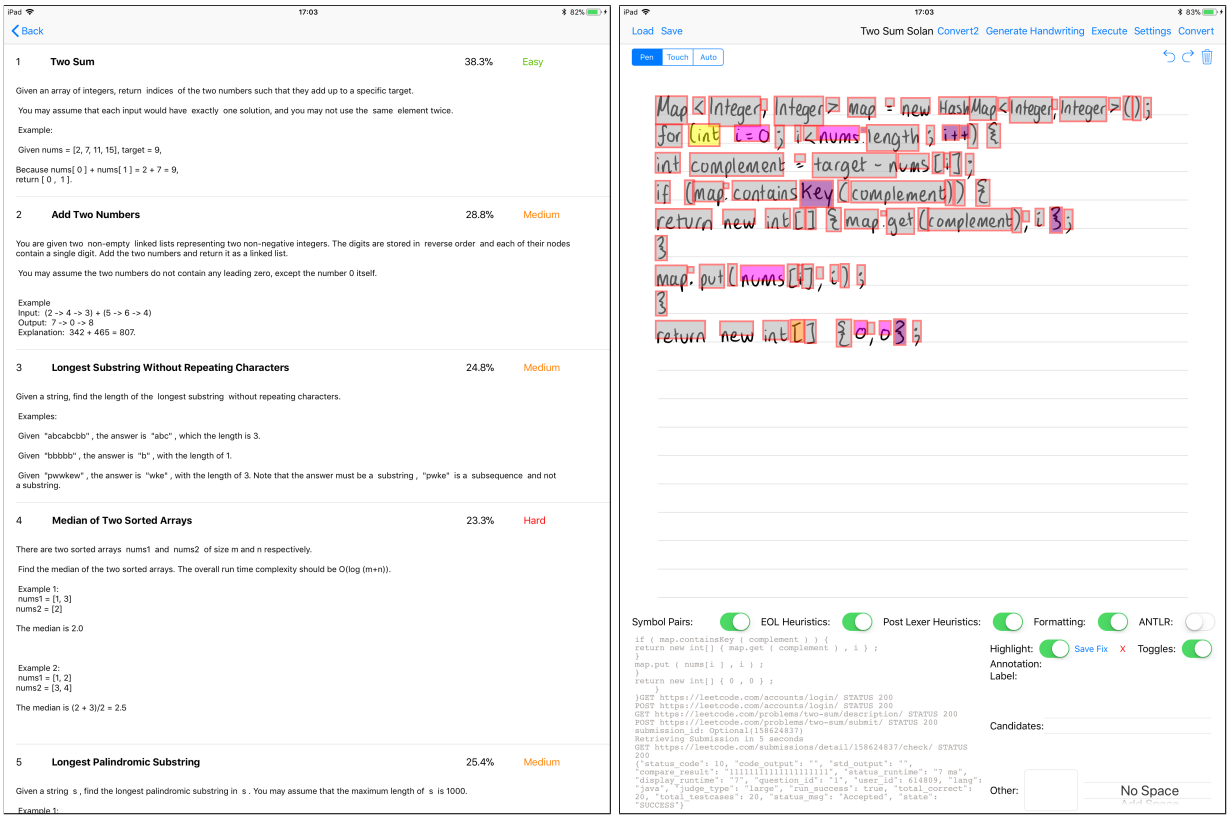
(a) Custom Code - Execution Compiler Error (b) Custom Code - Execution Success

Figure 4.3: Custom Code Execution Examples

4.9.2 Third Party Integration - LeetCode

The LeetCode website [12] offers a service for developers looking to practice their technical interview skills. Once you signup (free), you have access to a number of questions that can be answered using their online IDE. You can then submit your solution, which will be run against their test cases and also checked against their time estimates.

However, LeetCode currently does not have an API to get access to their questions database, and submit solutions to their testing environment. Instead, I decided to pre-scrape the questions along with their description, url and template code using Python. I decided to prescrape the questions on my laptop and store them in a Swift file as opposed to scraping them within the iOS application because the LeetCode website is full of Javascript code that is rendered by the browser. Instead of creating a UIWebKitView and loading the LeetCode website and waiting for all the JavaScript to render on an iPad, it was easier to do this scraping on a laptop using Selenium with Python [13].



(a) LeetCode Questions (b) LeetCode Submission

Figure 4.4: LeetCode UI

The user can then select a question from the main menu of the iOS application, which creates a new file with the same title as that of the question. A new file setting is created to state that this file is associated with a LeetCode question and the LeetCode question id is associated with it. The user can rename the file if they want to have multiple attempts for the same LeetCode question.

Upon execution of the handwritten code, the user is automatically signed in (the user provides their login details in the settings page) using HTTP requests. The Cross Site Request Forgery (CSRF) token is obtained from the cookie obtained when the user logs in, and is used when the post request is sent that submits the solution in the body of the request. This CSRF token is needed, along with setting the appropriate HTTP headers, in order for LeetCode to accept the request. A final GET request is sent 5 seconds after a successful submission to retrieve the results, and report the results in the iOS application. A time delay is required since if the request to retrieve results is sent too early, then the response will say so. The browser usually sends 2-3 of these retrieval requests instead of waiting after a certain number of seconds. These HTTP requests are sent on a separate thread, and the results are printed on the bottom right of the screen on the main thread. This multithreading prevents the user interface of the application freezing until all the requests have been sent and received a response. With this method, the user has the added advantage of logging into their LeetCode account on a computer and viewing (and changing) their submitted code in the online IDE. The LeetCode website also provides comments on time and space complexity for submissions, which the user also has access to when making a submission through this iOS application. This helps complete the feedback loop when answering a technical interview question.

4.10 Saving Files + Recognition Fixes

The original starter project provided a way to save and store one file that contains the information of the digital ink. The file has a IINK (Interactive Ink) file extension, which is proprietary to MyScript. I have extended the project to support loading and saving multiple IINK files. This feature along with renaming files was important for moving the project along as it allowed for testing the result of the pipeline processing on the *same* digital ink files.

However, the user may want to save the recognition fixes that have been made by both the pipeline and by the user. For example, if the user wants to submit the code to LeetCode once they regain internet connection, then they need to save the recognition fixes because otherwise, the user would have to repeat the same fixes again, which would be frustrating. The recognition fixes information is saved by storing the [[Word]] in a JSON file, and this is done by making the Word object implement the Codable protocol. The JSON file has the same name as the IINK file, and so when opening a file, the application can just load both files. By implementing the Codable protocol, encode and decode methods can be called when saving and loading the JSON file. Then when loading a file later on and enabling the Highlight toggle in the user interface, the bounding boxes along with the previous recognition fixes made by both the pipeline and the user are immediately displayed, i.e. there is no need to hit the convert button and run through the manual recognition fixes.

An important feature for this project is allowing the user to add/change their handwritten code between conversions without losing any manual user fixes they have made. The MyScript API does not provide a way of tracking what was changed between conversions, and instead does a completely new conversion every time. Therefore, my approach for tracking what was changed is to compare the bounding box width and height values and the final recognised label for all the Word objects of the old and new conversions. As a result, two separate conversion buttons are provided in the user interface, one that does a partial conversion and does not remove the old recognition fixes (but merges the new changes), and another that does a completely new conversion and replaces all of the existing recognition fixes. An alternative method that I considered was to keep track of the strokes by the user after the last conversion. I could then use the x and y positions of these strokes to track what was last changed, and correctly replace the minimal number of Word objects. However, this method would not work when the user uses the join lines or break lines gestures. These gestures are important, particularly in this type of application, as it allows the user to move their code up or down in case they want to add or delete code that they have already written. These gestures are represented by a straight vertical line on the line that wants a new line added (vertical line drawn from top to bottom) or the line to be joined with the preceding line (vertical line drawn from bottom to top).

Renaming files was also another important feature, especially when conducting user testing for multiple users. Renaming a file is done by tapping the title of the file in the user interface. Renaming a file involves renaming the .IINK file, renaming the .JSON file (if it exists), and also updating the name in the FileSettings dictionary stored in the UserDefaults (lightweight database).

4.11 Generated Handwriting

Each handwritten character, number, symbol or operator is made up of a set of strokes. Each stroke is made up of a series of x points, y points, force values. The first point of the stroke indicates pen down and the last point of the stroke indicates pen up and all other points indicate pen movement. I managed to create a dataset of every character, number, symbol and operator that stores the strokes information and also the width of the character. The stroke information can be extracted by exporting into a JIIX string (as mentioned previously). This dataset has been saved permanently in a Swift file (DataSet.swift). One point to make, however, is that while collecting this data there were some symbols such as "|", "/". "\" that were not correctly recognised, which goes to show that even without context, single character recognition with MyScript is not 100% correct. After normalising the data of each character (x position is left aligned, y position is on the first line of the lines guide in the app), I wrote a function that could handwrite a given string of source code using this dataset. One control parameter when artificially handwriting a given string is the spacing between each character. Having access to such a dataset and function allows another way of evaluating this application.

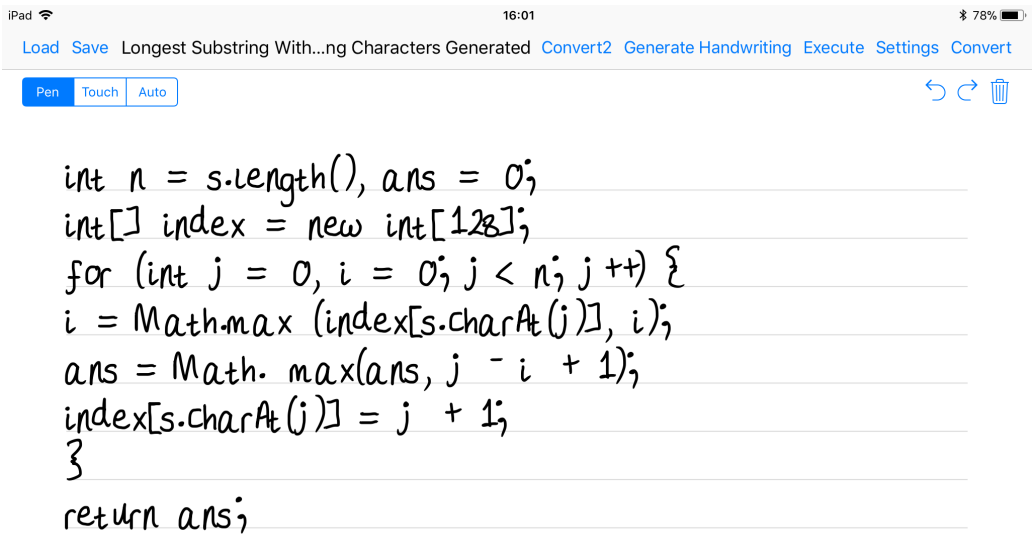


Figure 4.5: Generated Handwriting Example

4.12 iOS Application Structure

iOS applications are predominantly written using Objective-C and/or Swift. As this project was building on the 'starter' project by MyScript Interactive Ink, there is a mix of Objective-C (MyScript runtime engine) and Swift (my additional implementation). An alternative to this was to create a web application that uses MyScript's CloudKit framework, and although this would allow for working with multiple devices, it would not offer complete support for the Apple Pencil. Palm rejection would not be supported, and so this gives potential for more noise strokes to be drawn that would affect the recognition, and thus also the pipeline.

4.12.1 MyScript Interactive Ink SDK

The Interactive Ink runtime is setup by first creating and configuring the engine. For this project, the default configuration is setup, and the only change that is made is to the en_US.conf file located in the Recognition Assets folder where the value for SetWordListSize is changed from 3 to 20. This increases the number of recognition candidates to the maximum. The Interactive Ink model is made up of a *package* (a container for digital ink), which itself is made up of a collection of standalone content units called *parts*. These *parts* have an associated type: Text Document, Text, Math, Drawing or Diagram. In this project, each .iink file is a single package with a single part with type "Text Document". This allows for both math and text recognition, which is required since source code can have an arbitrary mix of English words and mathematical expressions in any location.

The *renderer target* is the UIView that will display the digital ink, and implements the IINKRenderTarget protocol. A *canvas* object is needed to provide an implementation of the drawing commands (pen down, pen move, pen up), and follows the IINKCanvas protocol. A *renderer* is an object responsible for knowing how to render content of each layer, and knowing which areas of the model need to be refreshed. For performance reasons, the renderer works on four different layers. The background layer in this project contains the line guides. The model layer contains everything that has been processed by the engine at that current point in time. A temporary layer is used to handle objects in a temporary state, for example those in a drag-and-drop operation - this project does not have any interaction with this layer. Finally, a capture layer exists to render the ink drawn on the screen, but has yet to be processed by the engine.

4.12.2 MVC Design Strategy

The main design strategy used in my implementation is MVC (Model View Controller). MVC splits the data model from the view that displays this data, making the controller the communicator between the two entities. For iOS applications, the view is what the user can see and interact with, and is created using Apple's UIKit framework. The model is the persistent data and can be stored in a variety of formats (CoreData, Property Lists, UserDefaults and Swift files). The controller is an object that contains the logic of the application and handles what happens when the user interacts with the views and when there is a change in the model.

The HomeViewController is the main controller of this application as it handles the screen where the user can draw on, convert and execute. As a result, the HomeViewController file contains methods that handle the logic for these interactions by the user. The model for this controller is the [[Word]] that are encoded into JSON and stored in a .json file.

The LeetCodeChooseQuestionTableViewController controls the table that lists all the LeetCode questions that have been prescraped along with important details in the table below. The question information is stored in a swift file called LeetCodeDataSet.swift (model).

ID	Title	URL	Submission Acceptance Rate	Difficulty	Description	Template Code

Figure 4.6: LeetCode Question Information

This information is not only used to provide the necessary details to the user, but also to put together the HTTP requests that are sent when making a submission of the final source code output to the LeetCode website.

Unlike the other UI components created for this application, the LeetCodeQuestionTableViewCell is created in a separated .xib file as this cell is used in two places: LeetCodeChooseQuestionTableViewController and SettingsTableViewController (so that the user can review the problem description again, if they had forgotten it). This prevents the need for recreating the same UI component in the storyboard file twice.

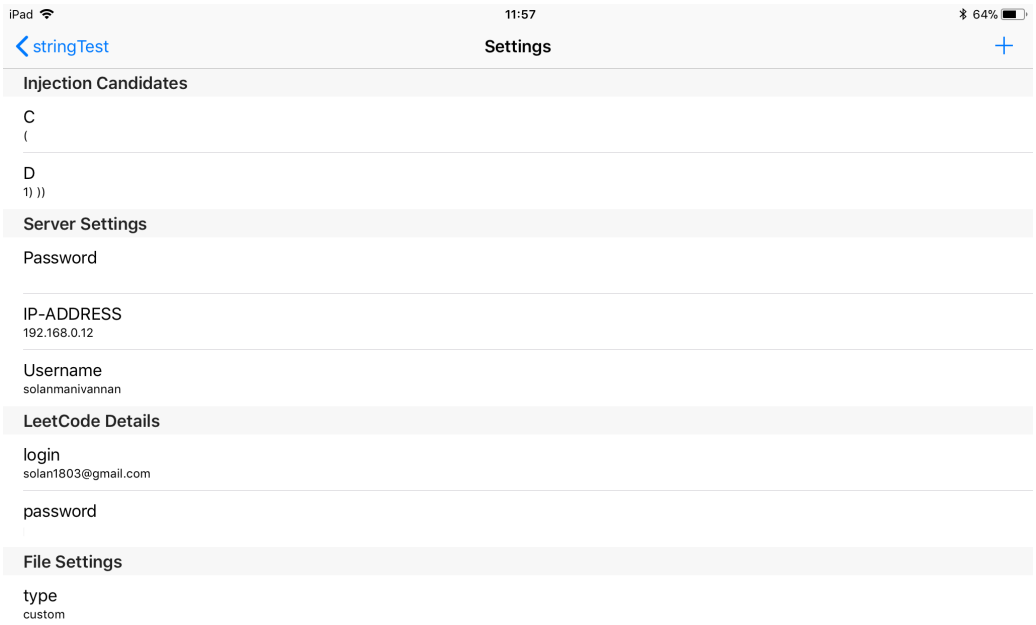


Figure 4.7: Settings Table View

The main settings (SettingsTableViewController) of the application contain information such as local server settings and LeetCode login details. The file settings contain information such as the type of the file: custom or LeetCode. For a LeetCode file, the LeetCode question id is also stored so that we can retrieve the question details from the LeetCode dataset. The file settings are stored in a lightweight database called UserDefaults (model), as this information will only be set once and read from.

The majority of the UI components are created in the Main.storyboard file using Apple's UIKit framework.

Below is a UML diagram of the main components that I have implemented for this project.

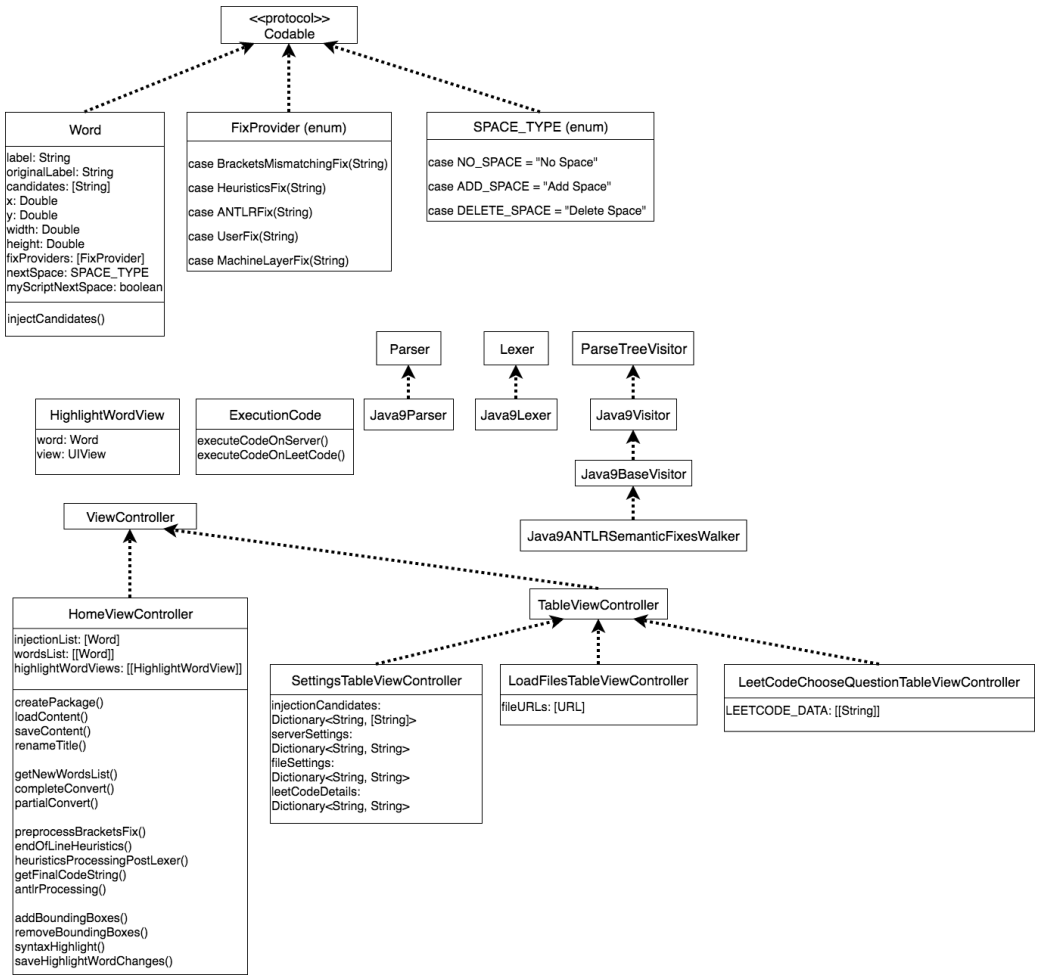


Figure 4.8: UML

4.13 Extending the application for multiple programming languages

If the user would like to handwrite source code in another programming language, then the following would have to be provided to configure the pipeline.

(1) For custom files, the user has to provide commands to compile and execute the source code file. Nothing needs to be changed for LeetCode questions, the choice of programming language needs to be specified in the HTTP request used for submission.

(2) The user needs to provide a grammar file in order for ANTLR to create a lexer and parser. A custom tree walker needs to be provided to walk the parse tree and create a symbol table data structure. For example, if Javascript was going to be added as another programming languages supported by this application, then since Javascript is a weakly typed programming language, a less rigorous custom tree walker fixer would be implemented; i.e. var i = o cannot check if the type of the variable i is an int before changing the o to a 0.

(3) As the pipeline is broken down into several stages that can be individually toggled, the user can disable parts of the pipeline that may not be relevant for that programming language. For

example, the Swift programming language has optional brackets for if, while, switch etc, and so the post lexer heuristics stage of the pipeline would not be relevant - unless the user would be willing to write source code with those brackets being compulsory.

However, this application suffers one issue with indentation. Once the digital ink is stored, it has no information about indentation and so every line of handwritten code is left aligned. So for programming languages such as Python, where spacing and indentation is important, handwritten source code will not be able to be used in this application until the MyScript API can handle storing indentation information.

Chapter 5

Evaluation

Evaluation of this project will require many different users to use the app. It may be appropriate to get computer science students with varying levels of experience, but also students looking to go into software development that study other degrees e.g. mathematics. It may also be important for the study that these people work through the problems themselves, rather than copying up a solution in their handwriting.

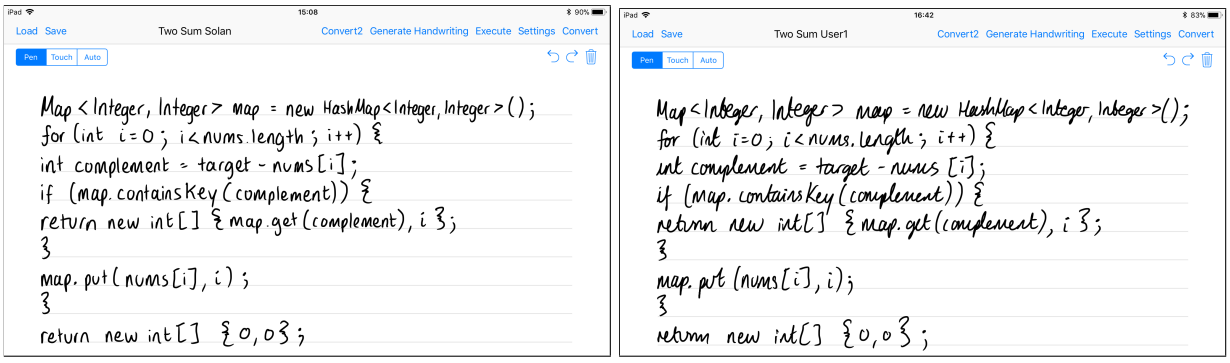
5.1 Quantitative Evaluation

Past projects in the area of handwritten source code recognition have done an evaluation on the word error rate and character error rate. However, in this project, as the focus is on compiling and executing the code, a more important measure of how successful the iOS application is can be based on the number of fixes required by the user to get a compilable and executable program that does what the user intended it to do. For example, a fix where the user selects an alternative text recognition candidate that may be made up of replacing 2 characters is considered as a single fix, whereas in past projects this would have been counted as two fixes.

The first experiment that I set out to do was assess the reduction in number of fixes that needed to be made for solutions to 4 LeetCode problems. An insertion, deletion or replacement of a character (including spaces) is counted as a fix, but for the application if the user found the correct word in the candidates list that has more than one character insertion, deletion or replacement, then this is counted as a single fix. I decided to pick 4 LeetCode problems that had solutions that would fit into the number of guidelines available on the screen. Furthermore, I got each user to write out the same solution so that there would be no variation in the number of fixes caused by a solution from the user having less/more words. For the 10 user average, the number of fixes required reduced by around 60-70% compared to copying and pasting the original MyScript output and correcting it using a physical/virtual keyboard.

LeetCode Question: (1) Two Sum			
	Number of Fixes Required	Plain MyScript Fixes	Percentage Reduction
Solan	6	19	68.42
Generated Handwriting	12	19	36.84
10 User Average	9	21	57.14
LeetCode Question: (3) Longest Substring Without Repeating Characters			
	Number of Fixes Required	Plain MyScript Fixes	Percentage Reduction
Solan	12	22	45.45
Generated Handwriting	14	26	46.15
10 User Average	12	29	58.62
LeetCode Question: (9) Palindrome Number			
	Number of Fixes Required	Plain MyScript Fixes	Percentage Reduction
Solan	4	21	80.95
Generated Handwriting	11	19	42.11
10 User Average	10	25	60
LeetCode Question: (11) Container With Most Water			
	Number of Fixes Required	Plain MyScript Fixes	Percentage Reduction
Solan	11	26	57.69
Generated Handwriting	13	23	43.48
10 User Average	10	33	69.69

Figure 5.1: Comparison of number of fixes required by the user between original MyScript output and post pipeline output



(a) Two Sum Solan (b) Two Sum Cursive Handwriting

Figure 5.2: TwoSum Examples

I find this to be a good size reduction given the pipeline in its current state, and there is potential for future work to fix some more issues. For example, a common error that required to be fixed was declaring new int array variables and initialising them "int[] arr = {0,0 3". A potential way of fixing this is to use ANTLR to recognise that this line of code is an int array declaration and initialisation, and then processing the tokens to replace the 3 with a curly brace. Another common issue (especially for cursive writing) that could be further investigated is allowing Levenshtein distance calculations to be made for words with different size strings, where the original MyScript recognition could recognise one less character than there actually was.

An important discussion point that came with more user testing was noticing pipeline fixes that changed something that was originally correct, and pipeline fixes that changed something that was originally incorrect, to something that is still incorrect. The latter problem is not so much of an issue for the user. The former problem happens on very few occasions and is mostly due to the mismatching symbols heuristic and the Levenshtein distance in ANTLR. For example, if the user has both Integer.MIN_VALUE and Integer.MAX_VALUE in their source code, then if Integer.MIN_VALUE appears first, MIN_VALUE is added to the symbol table, and when MAX_VALUE is visited, the Levenshtein distance would exceed the threshold value and MAX_VALUE would be replaced with MIN_VALUE. The user can fix this issue in the application in its current state by disabling ANTLR processing and making the change manually. For the mismatching symbols heuristic, one fix that caused an error was "for lint i = 0; i (5; i++)" instead of "for (int i = 0; i ⟨ 5; i++)". If the MyScript candidates do not have a "(int" candidate for "lint", but do have a "(" candidate for "⟨", then the left angle bracket is replaced with a opening bracket because of a mismatch count in the number of opening and closing brackets. This error did not happen so often, so I was not inclined to include a new entry in the candidate injection list for "l" to ["("]. Moving the post lexer heuristics stage of the pipeline above the symbol mismatch heuristic stage could help avoid this problem.

A followup technique that came about from user testing was keeping track of new types e.g. NodeList in order to help the formatting stage of the pipeline. Currently the user has to manually fix the spacing for "NodeLista = new NodeList" (insert a space before the 'a'). If another NodeList variable is declared, then the same error will need to be corrected by the user. Instead, once the first error is fixed, and ANTLR is run, it can add to a newly defined list, the set of new reference types defined by the user. Then the formatting stage of the pipeline could ignore joining identifiers that start with these words. As the user continues to use the app, this

	Average Typing Time (seconds)	Average Manual Fixing Time on App (seconds)
LeetCode Question: (1) Two Sum	132	66
LeetCode Question: (3) Longest Substring Without Repeating Characters	105	48
LeetCode Question: (9) Palindrome Number	186	55
LeetCode Question: (11) Container With Most Water	72	35

Figure 5.3: Comparison of time taken to get final source code

could also be added into the reinforcement part of the app so that the user does not even have to correct this error at all. In addition, once ANTLR has a list of function names as well, then when reprocessing the tokens, it is possible to check if any identifiers match function names. If there is a match, then the next token should be an opening curly bracket. Technically, this will just add to the "if", "while", "switch" etc cases in the Post Lexer Heuristic stage of the pipeline.

For generated handwriting I did notice that a huge factor that controlled how many errors appeared was to do with the spacing between characters. If there was not enough spacing, then consecutive symbols and letters would sometimes be recognised as a single letter. For example, "int[]" would be recognised as "intd" or some other letter instead of "[]". Therefore, in my experiment I decided to make the spacing as close to human handwriting.

In addition, another important comparison is how long it takes the user to recognise and make manual fixes v.s. how long it takes the user to type up the source code they handwrote from scratch. I decided to do this for the same LeetCode questions and 10 users. From the table, it is clear that the times spent for the user to get the final source code that they desire dropped by over 50%. The times in the table represent the 'wasted' time for the user in the process of answering a technical question. Although the time has dropped when using this iOS application, there is still a good amount of time wasted. Further development of the pipeline using the suggestions made above could help bring this down further.

5.2 Qualitative Evaluation

I found that typing out a handwritten solution became a more frustrating issue for the user, the more questions that were being answered. There is also potential for the user to make a mistake typing out a solution, and that mistake may or may not be caught by the compiler. One benefit of using my iOS application is that the user can see the typed output and the handwritten source code on the same screen, which makes making the manual recognition fixes

an easier process. Most users are also more easily distracted when typing out a long solution, which can lengthen the time spent on a single question unnecessarily.

The efficiency of using my iOS application became more apparent to the user after a couple of questions. The order in which I did the user testing was to get the user to handwrite their source code on the iOS application, run the pipeline and make manual recognition fixes, and then get the user to type out the solution on LeetCode for a comparison. After a couple of questions, the act of typing became more and more frustrating.

In terms of the handwriting experience, users found that writing source code on a tablet was beneficial with access to the quick gestures (scribbling out, adding new lines, joining new lines), and its integration with remembering manual recognition fixes. This prevented the user from repeating work that they had already done (manual recognition fixes) and allowing for a better experience at modifying their source code. If the user was handwriting their source code on paper, then corrections would eventually become quite messy, and it would be more difficult to type up the solution. In addition, since the user has no direct access to an IDE and physical keyboard, many of my users liked how this prevented them from avoiding handwriting a solution to a technical interview problem and straight away typing up a solution.

There are two main points suggested in user testing that should be implemented in further development. The first is allowing the canvas where users handwrite their source code to be scrollable. This allows for more than the restricted 17 line limit, and would be more appropriate for the harder technical interview questions. Once this is done, further evaluation can be done on how the number of lines of source code affects the number of fixes required by the user (i.e. how effective is the pipeline in reducing errors with more lines of source code). The second item of development is allowing users to write source code that overflows onto more than one line. This happened to be the case for if-statements that had more than one condition and the declaration of HashMaps. There needs to be some adjustment to the UI to help allow for this, since the end of line heuristic is definitely a useful stage of the pipeline.

There are also opportunities to increase the effectiveness of the app for preparing for technical interviews. Debugging is one of the key processes developers go through, and is important when going through a solution that does not work as expected. At the beginning of the project, I had debugging support as a potential technical goal to achieve. Some users suggested that this would be an important feature had they been solving problems from scratch rather than copying up a solution. Since the source code cannot be executed natively on iOS, this implementation would require communication with a local server. The LeetCode web application also offers debugging support, so there may be an opportunity to link up with that as well for LeetCode specific questions. In addition, execution of custom code files could also be improved to include automatic generation of test cases and comments on time/space complexity and code style. These were features that I also considered at the start of this project, but they were less important than improving the time taken to get an executable program.

Overall, the users of my app found it to help streamline the process of practicing for technical interviews by reducing the amount of typing that needs to be done by the user and the time taken to evaluate a solution. They also found that the other technical features such as saving recognition fixes gave more encouragement to use such an application.

Chapter 6

Conclusion

From this project, I have learnt that it is possible to build a post processing pipeline on top of an existing handwriting recognition system that can fix recognition errors on handwritten source code. Knowledge about the programming language, especially its syntactic structure, can help identify and fix recognition errors. There are also several ways of navigating the recognised source code that each provide their own advantages: a String, a series of tokens, a parse tree and the bounding boxes. This project managed to achieve around a 60-70% drop in the number of fixes that need to be made by the user in the recognised output. There was also further evidence in this project for more ways to fix common recognition errors. In addition, there is also a greater than 50% drop in the time taken for the user to get the final typed up source code that is identical to the handwritten source code. This is due to both the pipeline that fixes recognition errors and also the user interface that allows for quick corrections to words. With regards to the core aim of this project, which was to streamline the process of technical interview preparation, these statistics do indicate some success. However, given that we still identified room for improvement, continuous investigation and research into this area is still possible.

I have found that the hardest problem that would probably best be solved with machine learning is recognition of closing brackets ")" and "]". This is because opening brackets can be tracked through the use of functions and array variables, however, it is difficult to know when the closing bracket is going to appear. Machine learning used in either the recognition of these individual symbols or the complete word can help with the recognition of these specific symbols. Similarly, it is difficult to know when an angle bracket is going to be used in a condition, and this scenario also suffers with bad recognition. Machine learning would also probably be the best solution for this. However, angle brackets used in generics are recognised fairly well in the original MyScript output.

An interesting problem that came about in this project is fixing errors that were not due to the MyScript recognition, but due to the user. For example, the user could replace all opening curly braces with the letter "E", and the pipeline would very likely change this to an opening curly brace (since this is highly likely to be in the list of candidates). These fixes could create a compilable program that produces the correct result, yet the user has clearly made errors in their source code (intended or not). One way of potentially dealing with this problem is to keep track of all the recognition fixes made in the current file, and see if the same fix is consistently being made. This is indeed another area of exploration in this project.

Chapter 7

Future Work

7.1 Reinforcement Learning

One way to improve the effectiveness of this application given more data on a particular user over time, is reinforcement learning. I identify two areas where this can occur. Firstly, the injection candidates list can automatically be updated by the app once it recognises that the user has made a manual fix, which involves the "Other" textfield, several times for the same label. An algorithm can be created to notice the difference between the old label and the user fix to create the new injection candidate. The second area for reinforcement learning is in the ANTLR stage of the pipeline. Inspiration can be taken from the code completion feature in some IDEs where suggestions are made based on the frequency of the most common words (identifiers) used by the user. A separate constant symbol table data structure could be maintained by the application that adds identifiers (variable names, function names) that are frequently used by the user, and this could help fix errors that appear on the first instance, rather than waiting for the user to fix the first instance and then fixing all other instances through Levenshtein distance.

7.2 Machine Learning Layer

Symbols and operators have the highest recognition errors in comparison to numbers and letters. One way to fix this issue is through a machine learning layer that works as another stage of the pipeline. This machine learning layer could make use of the stroke data that we have. The idea would be to pass in the stroke data for each character one at a time for classification. This would mean that context would not be taken into account, but that would be alright as we only need to recognise individual brackets, for example.

Appendix A

User Guide

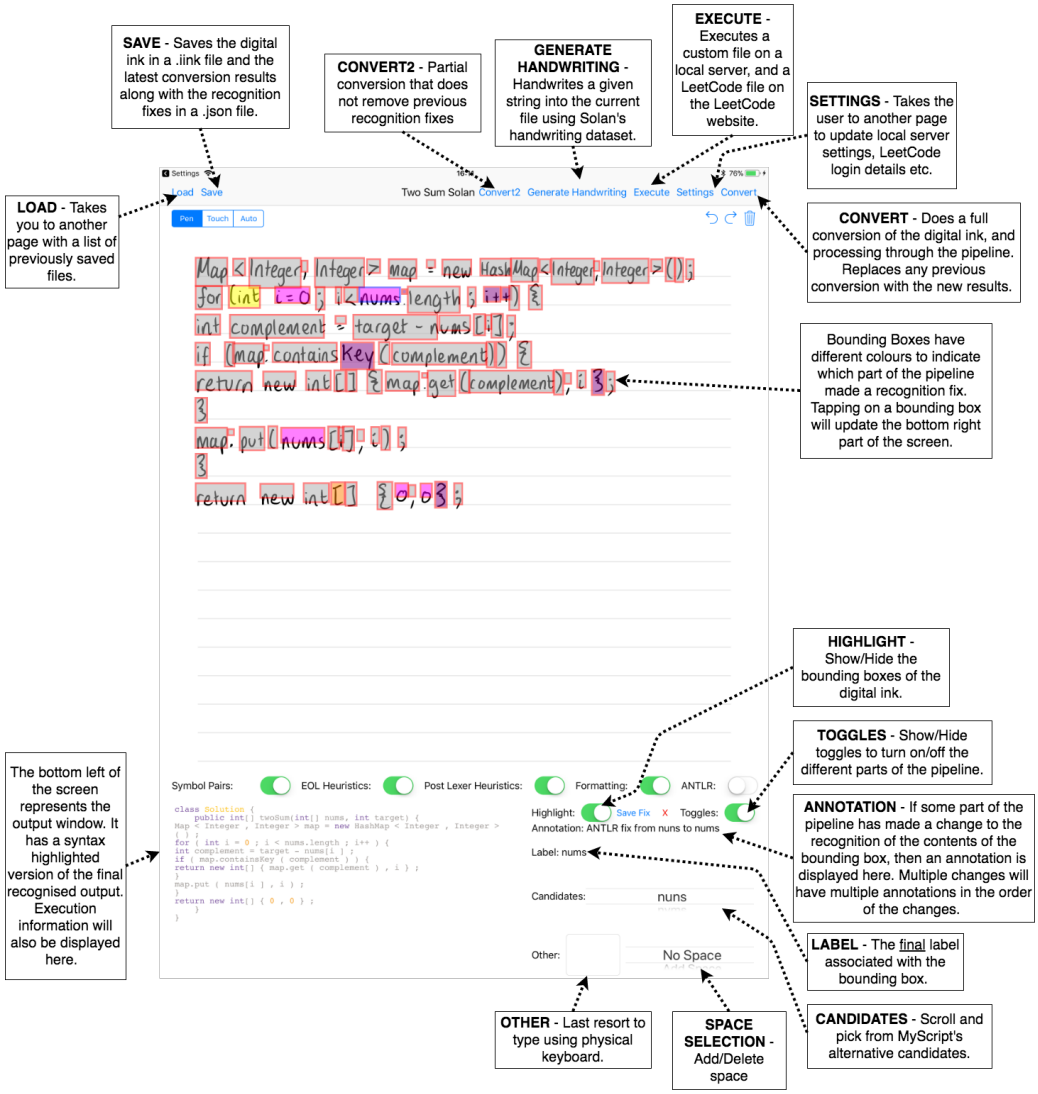


Figure A.1: User Guide

Appendix B

System Architecture

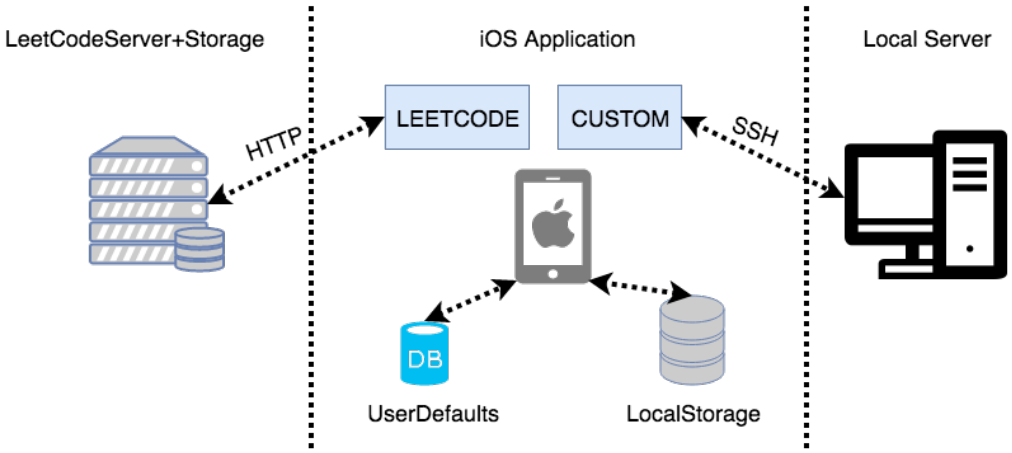


Figure B.1: System Architecture

Bibliography

[1] MyScript, "Myscript website." [Online]. Available: https://www.myscript.com

[2] T. B. Technology, "Goodnotes website." [Online]. Available: https://www.goodnotes.com

[3] H. Shu, "On-line handwriting recognition using hidden markov models." [Online]. Available: https://pdfs.semanticscholar.org/1d9f/1ca72190dd0b5e9f82b9a2a78315410e9373.pdf?_ga=2.24729453
1301992967.1528702736

[4] ExtremeTech, "artificial-neural-networks-are-changing-the-world-what-are-they." [Online]. Available: https://www.extremetech.com/extreme/215170-artificial-neural-networks-are-changing-the-world-what-are-they

[5] R. M. Theordore Bluche, Jerome Louradour, "Scan, attend and read: End-to-end handwritten paragraph recognition with mdlstm attention." [Online]. Available: https://arxiv.org/pdf/1604.03286.pdf

[6] B. Gonzalez, "Iris: A solution for executing handwritten code." [Online]. Available: https://brage.bibsys.no/xmlui/bitstream/handle/11250/137557/masteroppgave.pdf

[7] E. Thong, "Codeable: Generating compilable source code from handwritten source code." [Online]. Available: https://stacks.stanford.edu/file/druid:yt916dh6570/Thong_Recognition_of_Handwritten_Code.pdf

[8] R. M. Qiyu Zhi, "Recognizing handwritten source code." [Online]. Available: https://arxiv.org/pdf/1706.00069.pdf

[9] MyScript, "Myscript interactive ink examples." [Online]. Available: https://github.com/MyScript/interactive-ink-examples-ios

[10] R. GitHub, "Highlightr - ios & osx syntax highlighter." [Online]. Available: https://github.com/raspu/Highlightr

[11] N. GitHub, "Nmssh is an objective-c wrapper for libssh2." [Online]. Available: https://github.com/NMSSH/NMSSH

[12] "Leetcode website." [Online]. Available: https://www.leetcode.com/

[13] "Selenium with python." [Online]. Available: http://selenium-python.readthedocs.io