# Introduction

The aim of my project was to improve the usability of syntax errors for novice programmers. Something about the original requirements and what I’ve actually achieved.

This project was inspired by my experience teaching programming over the summer. I noticed that, while students would not have problems understanding how code logically functioned, they struggled to understand when and why certain punctuation was needed, and its effect on the structure and meaning of the code. A lot of students expressed their frustration as “why can’t it just know what I mean?”

In recent years there has been a large number of graphical programming languages such as scratch (or sketch? I can never remember what it is called), which involves dragging and dropping blocks in order to build programs. This program structures the code for expressions on behalf of the user. Blocks are shaped like puzzle pieces, and when they do not fit together that is another syntax error. This use of metaphors helps the user understand that they do not fit together. These programming languages teach the logical meaning behind code but do not teach the user the concept behind the structure of a programming language.

I’m not really sure what is meant to go into the introduction.

I could do with a brief overview of related work here. Let’s assume that will take up about two hundred words and will mark the end of my introduction.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris placerat imperdiet augue ac dictum. Nulla pulvinar nec magna non posuere. Sed ac posuere arcu. Mauris vitae tortor nec sem sollicitudin convallis. Aliquam et accumsan neque, vitae ornare odio. Vestibulum volutpat tellus a nisi suscipit, sit amet aliquam nunc imperdiet. Nunc tincidunt quis ante quis aliquet.

Nulla suscipit sapien vitae sollicitudin venenatis. Aenean accumsan, tellus eu posuere dictum, lorem sapien placerat enim, sit amet varius nulla felis ac nunc. Nulla sit amet felis eget eros laoreet porttitor ut vel augue. Nunc ac rutrum odio. Pellentesque eget efficitur massa. Sed id rutrum ligula. Nulla malesuada augue eros, ut porta tellus varius non.

Sed mattis volutpat neque sed rhoncus. Praesent pharetra libero enim, a ullamcorper purus faucibus vel. Curabitur fringilla lectus tempus, iaculis tortor quis, iaculis lorem. Ut ac suscipit risus. Praesent at mi ipsum. Maecenas convallis magna non feugiat lobortis. Nunc rhoncus efficitur ligula mollis varius. Aliquam quis pulvinar justo, sit amet suscipit felis. Proin semper ligula vitae ligula vulputate auctor. Aliquam eleifend accumsan est, tincidunt ultrices lorem malesuada nec. Sed venenatis imperdiet ex pulvinar convallis. Aenean non tempor massa. Nunc et rhoncus metus. In quis hendrerit erat. Nullam non justo interdum.

# Preparation

“It should show how the project proposal was further refined and clarified, so that the Implementation stage could go smoothly rather than by trial and error.”

Well that that is me buggered.

It was decided that the Skulpt library would be used for this project very early on in the project. Skulpt is a library that compiles python to javascript, which can then be run natively the browser. The main advantage to this was allowing the project to be made using a webpage that could be run locally without a web server being used to execute the code then run the result.

The largest part of the preparation was understanding how the Skulpt parser worked. Do I really need to launch into huge amounts of detail here regarding how it works? Something about LRARRLRLLLRL lookahead parsers of some description. The parser is very interesting (haha) but I don’t know how much they would appreciate an in-depth analysis of how the parser functions.

Of course, there was also the rather fun task of getting a barebones Skulpt implementation up and running and seeing which errors popped out where in the parser. From here, I made a very, very detailed plan of which errors I wanted to fix and my strategies for doing so. I definitely did do this before I started implementing. Definitely.

Planning-wise, I just kind of attacked the project at whatever angle seemed to be the most important. It was very much trial and error.

I was working in javascript for this project. I probably should have learnt how javascript worked properly (hello prototypes) but if I’m honest I just kind of started typing stuff until it worked and googled a lot of stuff. But yes, I did definitely did learn javascript beforehand.

This section is just going to be a huge pack of lies and I don’t even care. Maybe I can twist the truth. Or at least the passage of time throughout my project.

# Implementation

As explained in the preparation section, I had tried to produce as many errors in python as I could and see how Skulpt reacted and which errors popped up where. From here, I had a plan of how to handle each error.

The main categories of error I fixed were:

* Unterminated string
* Unbalanced brackets
* Unexpected token

And now I might as well explain how I attempted to fix each one.

## Unterminated strings

Unterminated strings are conceptually very simple to fix; insert a closing quote somewhere. YES, I GOT A SEMI-COLON INTO MY DISSERTATION!! The main problem I faced was where to insert the missing quote. I decided to insert a quote such that the brackets after the quote was inserted are balanced. Since an expression must have matching brackets, this seemed like a sensible way of closing the quote. Maybe also I could have paid attention to commas if it was in a function argument list. Maybe it is too late.

This problem is found by Skulpt when it attempts to tokenize the string. The Tokenizer uses a regular expression to find the next token to be extracted from the string. As soon as it encounters the unterminated string it will not be able to find a match and will fail. I decided to fix this problem by checking whether all strings balance in the original string before attempting to parse. This was to ensure that the unterminated string fixes and unbalanced bracket fixes worked together nicely.

## Unbalanced brackets

Unbalanced brackets are also conceptually very simple to fix; insert or remove brackets in order to balance them. This one had a little more flexibility in how I could choose to insert or remove brackets leading to multiple fixes being suggested to the user allowing them to choose which one they wanted to use. Since it is not known what the user originally meant, this is inherently guesswork. For unmatched opening brackets, I have decided to either:

* Remove the offending bracket
* Insert a corresponding closing bracket immediately afterwards
* Insert a corresponding closing bracket immediately before the next bracket
* Insert a corresponding closing bracket immediately before the next closing bracket at the correct nesting depth

Any unmatched closing brackets are simply removed. I could have used similar techniques to insert matching brackets, but this would have led to a lot of options being produced, which may have swamped the user. These suggestions are then listed allowing the user to choose which one they want to try.

This problem was originally found by Skulpt in the Parser? I don’t know I need to figure out where it originally occurred. Basically, I fixed it before the string ever touches the parser to make sure it interacted nicely with the string fixing.

Oh gee I am hungry.

## Unexpected token

Now, this is the most exciting and interesting one. Basically, the way the parser is structured is it will be in a state and has a stack of incomplete constructs. For each construct, there is a DFA showing the transitions that can be made. When the parser is handed a token, it will look at the possible transitions that can be made in its current state. A transition can be made if it represents shifting a token, in which case the tokens match, or it represents pushing a non-terminal construct, which can be done if the token is in that construct’s first set. If the parser currently has a token which does not match any of the transitions that can be made then that is a parse error.

I feel a diagram would be a good idea.

Normally Skulpt just goes ‘no thank you I don’t know what to do with this <Exception>’. Instead, we can attempt to find fixes.

The most basic, generic fix that can be attempted is to retry the parse without the offending token. This may work if the user has accidentally inserted some punctuation.

The slightly more sophisticated generic fix that can be attempted is to insert a symbol. A naïve way to select symbols to insert is to just try every symbol, but this is obviously not a good idea and would try a lot of symbols that just wouldn’t work. I have generated the tokens to try by looking at the first set of the highest level unfinished construct. This will generate a set of tokens that are guaranteed to be accepted by the parser at the current moment. By inserting the previously offending token we can see if the fix worked or not; the fix is accepted if the parser accepts the previously offending token. If the parser still cannot figure out what to do with the token then the fix is rejected.

In order to narrow down the number of fixes attempted, the tokens are sorted into equivalent categories before each one is inserted. This is done by creating a new parser and initialising it to be in the state of the previous parser before the token was rejected. Each token is inserted and the final state and top-level construct of the parser are recorded. These tokens are then combined into a group, and only one token from the group is actually tried for the fix, but each possibility from the group is displayed to the user.

There are a number of more specific fixes that I have also implemented, such as concatenating adjacent names, which is only done if the concatenated name has already been used in the program or is a keyword, or inserting correct punctuation for a possible argument list. More specific fixes could be added fairly easily, such as detecting the use of a keyword (e.g. for, if) and showing the syntax specific to that type of statement.

After a successful fix has been found, the parser will then attempt to parse the rest of the line. If it can successfully parse the line then it will display all suggested fixes to the user. It will not attempt to fix errors in the rest of the line. If it encounters another error in this part of the line it will give up and show the partial fixes suggested. This means if there are two syntax errors in a single line, fixes will only be generated for the first error encountered.

## Turtle

The

## Syntax trees

Another part of the project was trying to help users to visualise and compare how the syntax of each fix differs. But I don’t think it really helps that much because there isn’t really a way for them to understand where each part comes from. I think maybe if I had more time to work on this then this part of the project could have been made much more usable but I doubt a novice programmer will understand what is going on with the trees.

## Interface

# Evaluation

# Conclusions