OCR A-Level NEA: Sevin Fernando

GCSE OCR Pseudo-code IDE

Table of Contents

Introduction

Currently, if a GCSE OCR student, teacher, or examiner wishes to test, but in certain specialised cases, mark, a GCSE OCR pseudocode problem (such as a past paper exam question response, an NEA pseudocode planning section, or personalised projects for the purposes of improving pseudocode skills), they will have to spend a large amount of time manually running through the program (via trace tables, mental notes, etc.). If mental notes are used, inaccuracies, mistakes, and oversights may easily be made, while also leading to half-hearted testing efforts (comprehensive white-box testing with multiple data inputs cannot be practically performed if testing one data input is significantly time-consuming), which further exacerbates inaccuracies due to a lack of thoroughness. As such, this IDE aims to provide all of the essential support for executing pseudocode problems quickly in a simple UI that reflects the relative simplicity of GCSE pseudocode problems, while consulting available interacting stakeholders for the provision of additional, useful features. To clarify, pseudocode can vary by definition, though OCR provides a fixed standard through which it provides model answers within mark schemes, etc., thus incentivising students to align themselves with these standards to maximise their chances of being marked correct. As such, undertaking this project is logically justified, aiming to also be aligned with current mark scheme standards. Finally, the interpreter should be complete (provides support for all of the GCSE OCR pseudocode syntax listed in the standard documentation) to ensure that it is applicable to the widest possible variety of pseudocode programs within the limits of OCR GCSE Computer Science.

Analysis

Computational Methods Relevant to the Solution

Problem Decomposition: To **describe**, when considering an OCR pseudocode text editor/interpreter solution, it is necessary to divide the problem into sub-problems (and then recombine them logically), **justified** as this facilitates modularity. This will aid in creating hierarchy diagrams for organised planning, robust unit-testing of small, identifiable components, future maintenance (largely because of unit testing, but also because adding components to the program - like a new code feature - will not largely destabilise the rest of the program's functionality if each component is reasonably independent), and proper iterative development (can work on the final solution piece-by-piece). For example, an initial idea of the subproblems involved include file management (opening and closing text files); managing the presentation of the text editor during development; executing the code (and returning error messages), which will be further divided into lexing, parsing, and interpreting; and finally outputting to a console.

Abstraction: To **describe**, when developing the interpreter, considering there can be a large amount of data to process, it is necessary to first remove unnecessary details such as comments and redundant whitespace, **justified** as this reduces clutter, enables more targeted (and therefore efficient) modular functions to be designed (thus reducing the bulkiness of the program consisting of over-generalised functions attempting to handle unnecessary edge cases), and alleviates the need for inefficient, avoidable long-term validation (due to the presence of cluttered information to work around). By reducing the required bulkiness of code, there will be less code repetition, greater clarity and readability, and it will therefore be easier to maintain going forwards.

Information Hiding: To **describe**, when creating a simple GUI for the user, layers of abstraction should exist to hide inner complexity, **justified** as this promotes user-friendliness. For example, users do not need to know that run-time variables are stored within data structures that are structured to manage scope, or when typing, they do not need to understand the loop or spacebar-triggered mechanism by which pretty printing is (likely to be) implemented. This will largely be carried out for the user as interpreting is a back-end process (all the user will see is the console output, the request for console input where applicable, and the 'Run' button). These heavy layers of abstraction are

necessary as the primary function of this application is to enable OCR GCSE pseudocode problems to be executed and extra details such as internal implementations of scope, abstract data types, etc. will distract and confuse the user, especially when a large portion of the intended demographic, OCR GCSE students, will have significant variance in skill level and programming experience.

Encapsulation: To **describe**, the use of Python classes (and encapsulation more generally) can hide information during the development of this project (e.g. the structure of the parser output can be made simple-to-use with classes in the interpreting stage), which will further aid in a modular approach (unit testing is easier when stages are represented independently). To **justify**, as interpreting is a large, multi-stage process, encapsulation to categorise components (binary operators, for loops, if statements, etc.) will be useful in creating succinct conditionals (to identify these components) and remaining organised amidst the large volume of data. For further **justification**, as some components will have overlapping features, encapsulation in classes allows for the inheritance of attributes/methods, which will reduce the repetition of code, thus making it clearer for readability, organisation, and future maintenance.

Amenability to a Computational Approach

Faster Execution Times: To **explain**, if a person were to go through a pseudo-code algorithm by hand, they would be limited by slowly written trace tables, time taken to logically understand the algorithm, time taken to re-check for mistakes, etc. However, a computing device, faster in focused processing (by design) with access to high-clock-speed CPUs, reliable instruction execution, parallel processing/multitasking, etc., will be able to execute larger OCR pseudocode programs much more quickly than a person. This can be further **justified** by a statistic - the average CPU has a clock speed (also the rate at which unit elementary instructions are completed) of between 1 and 3.8 GHz according to Lenovo. The computational **justification** for whether OCR pseudocode programs can be executed by an interpreter are it will exist on a Python backbone (existing protection from stack overflow and garbage collection, and we can limit loop counts explicitly) and the standard OCR pseudocode is consistent (and therefore conditionals can be programmed specifically - only finite possibilities for keywords exist and therefore it can be coded - and considering it contains all the basic programming constructs, it is Turing complete).

Ability to Store Files: To **explain**, computing devices have reliable, non-volatile secondary storage (e.g. HDD, SSD) in which OCR pseudocode files can be stored. This is advantageous over a real-life scenario, **justified** as when storing current work through traditional paper means, for example, programs can be misplaced, are more susceptible to physical damage, etc. - this is less true when in a digital format. Furthermore, it enables file handling to be done to a better extent by quickly opening files (using built-in mechanisms, such as that which transfers the file from secondary storage into RAM), allowing it to be saved, undone, closed, etc.

Added Functionality: To **explain**, an OCR pseudocode interpreter would benefit from being able to display error messages to aid in debugging, elements of pretty printing, etc. The aforementioned points are computationally achievable, **justified** as the screen can be used for display and using a separate programming language as a base enables the leveraging of existing error detection mechanisms (e.g. using Python's error detection to inform the interpreter's error detection) and pretty printing can be done automatically by the computer through the use of conditionals and running large checking mechanisms on the program (feasible due to high clock speeds and computing efficiency). These are specific examples, though generally the ability of a computer to multitask beyond normal human capabilities enables many useful features to exist simultaneously.

Greater Accuracy: To **explain**, as a person runs through an OCR pseudocode algorithm by hand, they are more likely to make mistakes (**justified** especially when handling edge cases and more finicky aspects of the code) than a computerised interpreter that works deterministically (by traditional computer design). If it is well-tested, this is especially true. Furthermore, if updates are to be delivered to the end-user, which may involve bug-fixing, for example, it being in a digital format

facilitates digital communication meaning that these updates can be delivered more quickly and efficiently.

Digital Interface: To **explain**, a computer can produce the development environment digitally by executing code on existing graphical platforms (e.g. Python Tkinter), which are pre-tested and optimised such that it is computationally feasible. In further **justification**, an OCR pseudocode editor/interpreter is limited in stored information (text files, interpreting code, few GUI elements, etc.), thereby requiring limited computational resources. The customisable aesthetic elements of a computer (with multi-coloured pixel technology, for example) can also make the development stage more user-friendly for the programming user.

Description of Stakeholders

To **describe**, this OCR pseudocode interpreter can be used by **GCSE students** who wish to freely create programs (as a means of practising their pseudocode and more generally coding skills) and execute their code responses to practise questions (thus being able to check for syntax and semantic errors, identifying misunderstandings to help prepare for their externals). It can also be used by **teachers** and **examiners** who wish to either practise in a similar sense (to maintain their skills and knowledge to later verify student work, especially in the case of free programs checked by an in-school teacher in the absence of a mark scheme) and as a means of marking student work (by ensuring the program works without errors).

To **explain**, conceivable **needs** for these demographics would be relatively fast execution times, easy-to-use file handling (to write programs, edit/undo, save them, etc.), a simple, aesthetically suitable GUI (to alleviate distractions during code-writing and for usability), a functional, feature-complete console (including the ability to display errors), a suitable range of errors (descriptive to aid in debugging), sufficient pre-built functions ('Print' to easily output to the console, 'Input' to accept user input, etc., such that users can focus on higher-level aspects of the code), and generally a development system that is responsive to their changing (or misunderstood) needs regarding the project.

To **explain**, The proposed solution can be **made use of** by satisfying the aforementioned needs (and further needs to be determined by primary and secondary research) - a user can open a text file within the text editor, have displayed an icon of the file on the left-hand-side (most likely) such that there is a file hierarchy for organisation, write their code in the text editor, perhaps have access to pretty printing within the text editor, execute the code by clicking a clearly visible 'Run' (or something similar) button, and have fairly quickly an output in the console (which may be the finalised, correct output, or an error message). This usage example assumes some undecided features (as in-depth analysis has not yet been carried out). These features are explained in more detail in the 'Design' section, but generally its sequential nature is a fairly accurate explanation.

This is **appropriate to the needs** of GCSE OCR CS students, teachers, and examiners as it is designed with them in mind (as discussed prior). To **explain**, with large, easy-to-read buttons, it fulfils usability requirements for the wide range of existing users. With minimalistic, clean features, it fulfils simplicity-of-use requirements. By using computation (and the relatively fast, focused processing of a computing device), it fulfils the requirement of fast execution times, with a clear, large console, it can represent the requirement of error indication and working output, it is easily accessible (can simply execute the code via an underlying interpreter and devices are often portable), etc.

To ensure that this project adheres closely to the needs of the relevant demographics, the interacting stakeholders **described** below will attempt to be representative of the desired variety:

- **Shao Shen Kuar:** He is a GCSE OCR CS student who, according to pre-interview conversations, has little past experience with programming and is currently studying the pseudocode section of the syllabus. Though he's using Python to gain transferable skills into standard pseudocode, he is currently struggling with remembering syntax and unsure of how to independently verify the correctness of his work when it differs explicitly from the mark scheme.

- **Toby Low:** He is a GCSE OCR CS student who is fairly experienced with programming, having completed numerous projects in Python and C. He finds little trouble in writing pseudocode programs, though he is still interested in a pseudocode interpreter to verify the correctness of larger, more complex programs for the purposes of both casual interest and skill-building for automaticity.

- **Tulin Kasimaga:** She is a GCSE OCR CS teacher who is primarily interested in having an interpreter to mark simple pseudocode responses to exam questions (that deviate slightly from the mark scheme, though are still potentially valid), project pseudocode planning sections (these are often large and complex, and if written in standard pseudocode, she can potentially verify their correctness, though this aspect is minimal), and large, complex programs (to aid ambitious students by helping them verify their implementations of difficult concepts).

Stakeholder Primary Research Interview

The goal of this section is to identify specific stakeholder success criteria through the use of targeted questions. Though the stakeholders are categorised into students/teachers, their actual experiences will likely be similar (generally just executing pseudocode), so I can ask them general-enough questions suitable for all. Furthermore, I will ask this question in a focus-group setting so that they can build off each other's answers and avoid repetition. With that said, the questions will be:

1. What past experiences do you have with text editors and IDEs?
2. Which features of these experiences did you find most useful?
3. Which features of these experiences do you think should be avoided?
4. What features of the pseudocode (e.g. for loops, if statements) do you want to be recognised by the interpreter, and which are unnecessary?
5. What would your preferred aesthetic be for a custom-built text editor/interpreter, including the relative window size?
6. Is there anything else you would like to mention?

To **justify**, questions 1, 2, and 3 establish a history with pre-existing text editors (and similar software) to determine whether future responses are informed by relevant experience, while trying to ascertain what key features to implement and potential missteps to avoid. Question 4 aims to determine which programming constructs to enable/disable, thus avoiding unnecessary work and highlighting areas to focus on. Question 5 aims to specify aesthetic preferences. Question 6 provides the opportunity to discuss anything missed.

Responses to Interview Questions

What past experiences do you have with text editors and IDEs?

Shao Shen Kuar: I've mainly used PyCharm for most of the Python programs I've completed recently, though I've also looked at Repl with its cloud-storage mechanism - opening files from anywhere is a great feature.

Toby Low: Though I used to use PyCharm, I prefer using a simple, clean text editor like Atom or Sublime Text for Python and some web-based applications I developed - I like the freedom of using the command-line interface to not only interpret obviously, but generally for navigation and establishing connections between files.

Tulin Kasimaga: When I teach, we all use Python as the primary language and I recommend, and use myself, Repl as a means of doing that. That way, students can work on school devices and continue the code-writing process at home if they wish or if it's assigned to them as homework. In my own time though, I use Atom mainly as a text editor, and interpret, along with other features, with Command Prompt, a command-line interface.

Which features of these experiences did you find the most useful?

Shao Shen Kuar: Overall, considering this is an OCR pseudocode text editor, I'll base my answers off that. I found the ability to easily run programs very useful. Though I know of using Command Prompt, it's nice to just have a simple, clear button like "Run" to just press and have it work. To add, clear error messages, including the type of error; ways to easily handle files - being able to open the file from within the IDE, edit it, and save it, etc.; some pretty printing features, like highlighting keywords with different colours; and a large console to display output.

Toby Low: I think that largely covers what I would like out of an OCR pseudocode IDE. Obviously, the main thing and where emphasis should most be placed is the interpreter - it should work for all of the syntax within OCR's standard pseudocode.

Tulin Kasimaga: I'd agree with both of them here. Obviously, OCR pseudocode doesn't make available standard error messages within the syllabus, but I think a fairly comprehensive, descriptive list of error types would be great. Ultimately, if it's about learning, clarity and help in understanding where someone may have gone wrong is important. Also, having multiple files open at the same time to add to what Shao Shen said about file handling is a nice extra feature that I think would make this project more user-friendly.

Which features of these experiences do you think should be avoided?

Shao Shen Kuar: I think for an OCR pseudocode interpreter, extra clutter could be avoided. As Toby said, what matters here mainly is the output after entering your code in the code-writing section, so perhaps avoid a box for current run-time variables like in PyCharm, implementing a whole debugger is perhaps a little much when you can just have clear, simple error messages,

Toby Low: True, you could also avoid links to GitHub and other connection facilities and even file importing, because you're unlikely to need multiple files to solve one pseudocode exam question, but if someone else wants it that's obviously fine.

Tulin Kasimaga: Also, I suppose a lot of IDEs offer quite a lot of support for multiple file types. Considering the OCR pseudocode syntax and the limited availability of different functions, I think it would be inadvisable to offer too many extra functions - I suppose stick to what the syllabus says and try not to make things much easier. In that vein, I would prefer no support for databases and SQL-related items, and provide support only for text files, or whichever format you use to enter the data into, while still having simple file handling like what Shao Shen described before.

What features of the pseudocode (e.g. for loops, if statements) do you want to be recognised by the interpreter, and which are unnecessary?

Shao Shen Kuar: I'd say just looking at OCR's standard GCSE pseudocode document and implementing everything there would be best, including those extra functions like substring() and length().

Toby Low: I would agree with him here. Trying to implement everything within that document, and preferably nothing more, is the best way forward as far as creating a useful practice and exam-marking application.

Tulin Kasimaga: I suppose there is that extra bit of file-handling in the syllabus. Though it doesn't come up too often, I think implementing it, especially with a Python foundation, should be achievable and would generally make it complete.

What would your preferred aesthetic be for a custom-built text editor/interpreter, including the relative window size?

Shao Shen Kuar: I think in terms of accessibility for all, a full-screen window is the best way to go. In terms of the actual layout, and what I think is true for all of us considering what we've said so far, is that simplicity is key. On the left-hand-side, like in traditional IDEs, you could have the file-handling part with clear buttons for direction. In the middle, you could have the actual text editor with the text file. On the right, you could have the input/output console.

Toby Low: Yeah, that traditional layout, obviously a clear run button at the top, and a relatively large font size would be great. That said, what Shao Shen said about the file-handling buttons on the left, I think it would be better to have them at the top in the same row as the run button, but more top-left as opposed to centre. I would also think that full-screen mode limits you from interacting with other windows simultaneously, such as looking at the exam response. That said, I do agree that the window size should be large, so something along the lines of 1400x700 on a standard 1920x1080 is something I've enjoyed before.

Shao Shen Kuar: Actually, that sounds much better. Sure, I would be happy with that. I suppose if there are limited file-handling features, as there obviously would be with GCSE pseudocode, it would be bad to dedicate a whole column to it. With regards to the window size, what Toby said about interacting with other windows simultaneously makes sense, and imagining 1400x700, I would be entirely happy with that.

Tulin Kasimaga: I think what they described makes complete sense. Also, if you're implementing the ability to have multiple files open at the same time, having the tabs near the top that you could navigate with would be great. But overall, a sort of half-text-editor-half-console with the buttons at the top and access to pretty printing with preferably a choice of overarching themes - though dark mode is the usual, I've always gotten use to light mode and being able to choose between the two would be nice - that sounds great.

Is there anything else you would like to mention?

Shao Shen Kuar: I don't necessarily have too much to say. Just a simple, clean look with two main windows - the text editor and the main console, with the 'Run' and file-handling buttons at the top, and all of the nice editor and thematic features we described before. Beyond that, just something that adheres as tightly as possible to OCR's document would be ideal. I'm looking forward to it.

Toby and Ms. Tulin agreed with Shao Shen, essentially reiterating the importance of what was discussed earlier - a simple GUI that has usability features lending itself to the large variety of GCSE

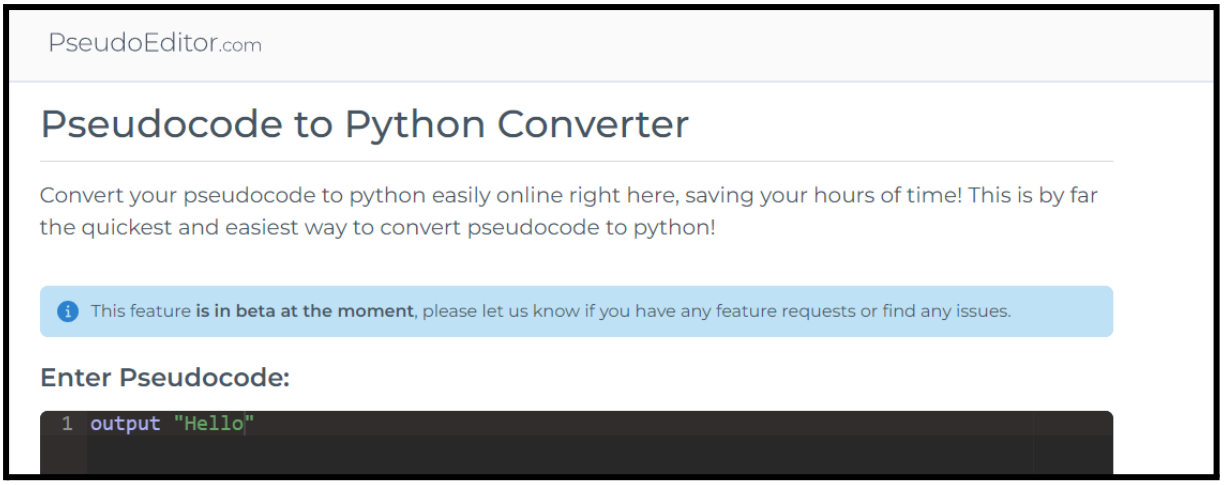
CS students and an interpreter that supports all of the programming constructs within OCR's documentation.

Findings of Stakeholder Interview

All stakeholders have had some experience in programming with text editors and IDEs, **justifying** that their preferences are based on observed ideas from existing, similar solutions. From here, we can generally divide the project into two overarching, relatively independent sections - the GUI and the interpreter. The GUI will clearly need to be simple and buttons and font large/visible, **justified** as this aids in usability, with the stakeholders even drafting a simple layout - the full screen divided with the text editor on the left-hand-side and the console on the right-hand-side, with file-handling buttons and 'Run' on top. Though this is a solid idea, they also mentioned that having multiple files open at the same time (by switching tabs) would be preferable. As such, we shall look more deeply into existing solutions to decide whether their initial draft is better than another idea such as dividing the screen into three - a file-handling window, a text editor, and a console, for example. Furthermore, multiple colour themes should be available. The interpreter itself, likely the most computationally intensive aspect, will require everything mentioned in OCR's documentation.

Researching Similar, Existing Solutions

PseudoEditor.com Pseudocode to Python Converter



Limitations of This Solution: In this solution, free-form pseudocode is understood by an AI-driven engine and converted to its corresponding Python code, after which the code can be executed by insertion into a Python interpreter. Firstly, the usage of free-form pseudocode will not be enabled in the OCR Interpreter. This is **justified** as according to OCR's specification, there is a standard pseudocode (which has syntax rules), and though there may be leeway, the extent of this is unspecified and maximising practise for pseudocode involves adhering strictly to the syntax rules, thus maximising user benefit. Also, there is an intermediate code here in the form of Python transferred to the user, from which it is transferred to a Python interpreter, which will not be implemented. This is **justified** as it is possible to implement a single OCR Interpreter which completes execution in a single session (on a Python background), which would be simpler to a user (particularly with little IDE experience) than a multi-stage interpretation process, and simplicity is a user preference. Also, the use of AI to understand what the user intends to write (and thus execute it by converting it to an executable intermediary code) will not be implemented. This is **justified** as due to the predefined syntax rules, execution can be exactly implemented without ambiguity (that an AI has to see through), and the use of AI would likely lead to longer execution times and larger file sizes (due to execution being more intensive, with overheads), which would be an inefficient solution.

What I Can Apply: Here, Python is the intermediate language. Though we will avoid multi-stage execution, I will use Python as the foundation of the OCR Interpreter. This is **justified** as Python has useful libraries including Tkinter, which can be used to implement a functional GUI. It is also interpreted, meaning simultaneous interpretation and execution without the creation of an intermediary executable file, avoiding a multi-stage process, facilitating simplicity, a user preference. It is also Turing complete containing all programming constructs contained logically within the OCR Interpreter, so it will be computationally achievable. The entirety of the interpretation is also contained within a single window, which I will implement. This is **justified** as a single window means that the user will not have to navigate excessively to use program features, facilitating simplicity, a user preference. It also has logical benefits as implementing multi-window functionality in Tkinter in code may require communication between the windows, which would require further classes to handle each window, increasing the code/file size, and this is inefficient.

Pseudocode.DeepJain.com Pseudocode Editor & Compiler

```
// This tool intends to "run" pseudocode for teaching purposes.
// It is consistent with the IB Computer Science syllabus. It may not work for other syntaxes.
//
// Examples:

// For Loop
number = 5
factorial = 1
loop for i from 2 to number
    factorial *= i
end loop
output number, "!", "=", factorial  // 120

// Input, Random, DIV, MOD
input number1
number2 = random(2, 9)
output number1, "div", number2, "=", number1 div number2
output number1, "mod", number2, "=", number1 mod number2

// Collection
c = collection([-1, 0, 1])
```

Limitations of This Solution: Here, the pseudocode syntax is that of the IB CS syllabus, which I will not implement. This is **justified** as though there are similarities, it will be inefficient to practise OCR pseudocode via a different medium, and is against the stakeholder requirement of consistent syntax with OCR pseudocode. Also, there is the pre-defined function random() and addItem(), in addition to a Collection ADT, which I will not implement. This is **justified** as these functions are not inherent to the GCSE OCR Pseudocode, and their addition may lead to false assumptions of the contents by students, which would disadvantage their exam experience. Furthermore, a Collection does not exist within the OCR Pseudocode and instead alternative data types (the elementary data types) should be implemented to maximise consistency.

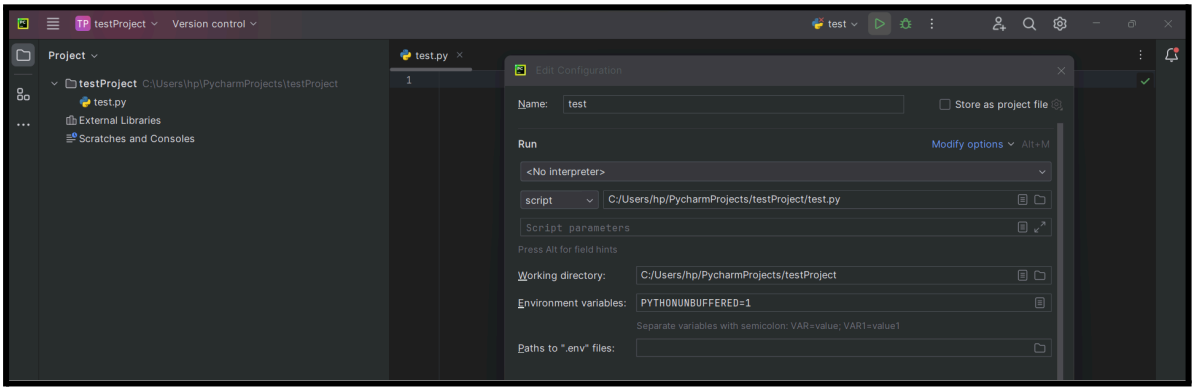
What I Can Apply: I will implement the programming constructs of predefined input/output functions and for loops as contained above. This is **justified** as these are included within the GCSE OCR Pseudocode, and will enable users to interact with common exam questions, a user preference. There is also a simple execution mechanism which I will implement, involving a single-stage button press as opposed to a multi-stage, multi-interpreter requirement. This is **justified** as a single stage is straightforward for the user to follow, facilitating simplicity, a user preference. It is also logically beneficial as if there were multiple steps, the vulnerability to errors would increase (as debugging would have to be performed intensively over multiple stages, including transfer between stages, in a limited development window). Multiple stages would therefore reduce the security in functionality of the final product, which is undesirable.

College Board Pseudocode Interpreter

Limitations of This Solution: Evidently, the program does not provide post-interpreter final output clearly, which I will not implement. This is **justified** as the real, formatted output should be displayed in order for it to be checked against a mark scheme (or expected output), while avoiding the redundancy of output function, which is an integral aspect of the OCR Pseudocode syntax. Also, the output of the Lexer and Parser stage is displayed, which I will not implement. This is **justified** as avoiding this acts as a layer of abstraction, enabling the user to avoid the technical complexities of intermediate stages to execution, which allows them to focus on development, thus maximising user productivity. There is also ambiguity when arguments to DISPLAY are not bracket-enclosed, which I will not implement. This is **justified** as opting instead for brackets to enclose arguments enables multi-argument function/procedure calls, which is consistent with OCR Pseudocode as subroutines may be defined with multiple parameters. It also improves readability, which will aid inexperienced users, thus improving user satisfaction.

What I Can Apply: Though I will not display the Syntax Tree and Tokens, I will implement them internally. This is **justified** as Lexing-Parsing-Interpreting is a standard compiler structure (as in the syllabus), and creating a token list collects the useful text editor data while removing unnecessary comments and whitespace, while the abstract syntax tree provides an easily traversed data structure for the interpreter to process for execution. The Token structure above contains an ID and NUMBER field, which I will implement by alternative names. This is **justified** as the ID will be useful for identifying how a particular token should be processed when checking syntax against predefined rules at Parsing, while the NUMBER, or more generally VALUE, field, will be needed for the final processing at Interpreting (e.g. adding the exact values of two numbers).

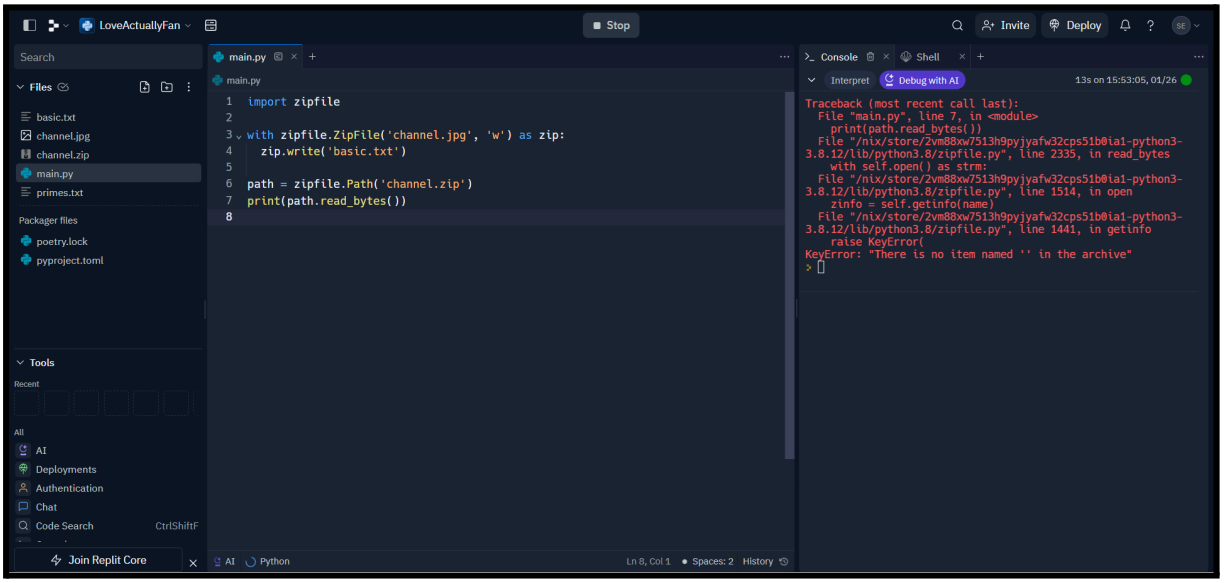
PyCharm Main Window



Limitations of This Solution: When looking at PyCharm's main window, we are primarily trying to find a well-structured, reputed UI to model after. That said, the division of the text editor and terminal in a top-down structure is undesirable, especially as it leads to the console being fairly cramped vertically and possibly forcing smaller font sizes within it. This conclusion is **justified** as stakeholder preferences specify the text editor and terminal being divided vertically as opposed to horizontally and a simple GUI with large features (including font size) with sufficient space. As such, we shall continue with the vertical division (screen-split into three main sections, including the file hierarchy). Furthermore, the 'Run' button at the top is quite small, represented purely by an icon and not a text description. To **justify**, avoiding a text description would limit ease-of-use usability for those unfamiliar with icon paradigms, such as those perhaps with limited exposure to technology or IDEs. To add, stakeholder preferences also specify large, visible buttons for readability and simplicity in understanding. As such, we shall use a large 'Run' button with both the word 'Run' and the 'Play'-iconography beside it to maximise usability aid. Finally, the 'Edit Configuration' window attempts to connect a local, on-device Python interpreter to PyCharm. For further **justification**, to expect every GCSE OCR CS student to perform a connection to a separate interpreting file would be unreasonable, requiring multiple extra steps, multiple installations, etc. adding to usage complexity and infringing on the stakeholder preference of simplicity. As such, we shall integrate the interpreter into our program (user will only need to press 'Run', not worrying about file connection), thus adding a layer of abstraction for ease of use.

What I Can Apply: The file hierarchy on the left-hand-side makes sense, **justified** as it is in line with stakeholder preferences of an easy-to-use file management system (thus having a standalone, independent place to perform these operations will help distinguish it, allowing for a simpler UI). Right-clicking enables various features such as opening, saving, deleting, etc., though we shall probably use large buttons, **justified** this is a stakeholder preference and maximising visibility is an important usability feature (as the intended audience - GCSE OCR CS students - is relatively large and variance in eyesight will exist). Furthermore, the inclusion of line numbers in the sizable text editor is important, **justified** as it provides for ease-of-use navigation and for referencing line numbers during error messages which shall be discussed later.

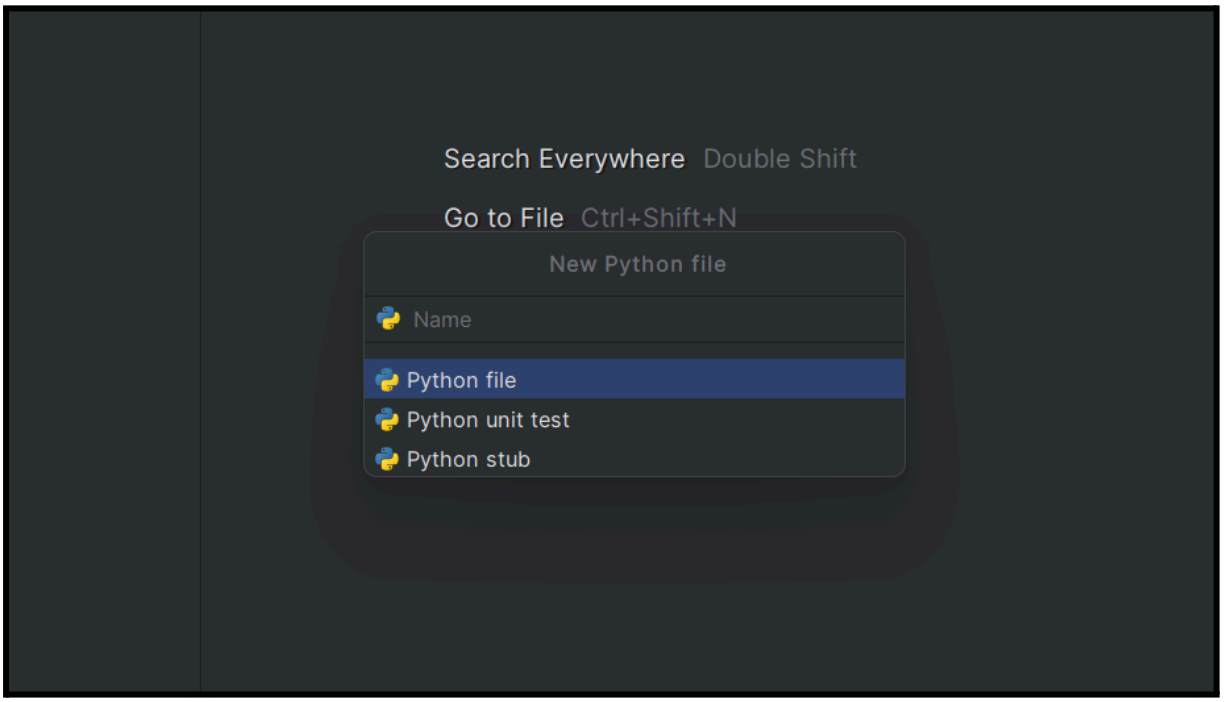
Replit Main Window

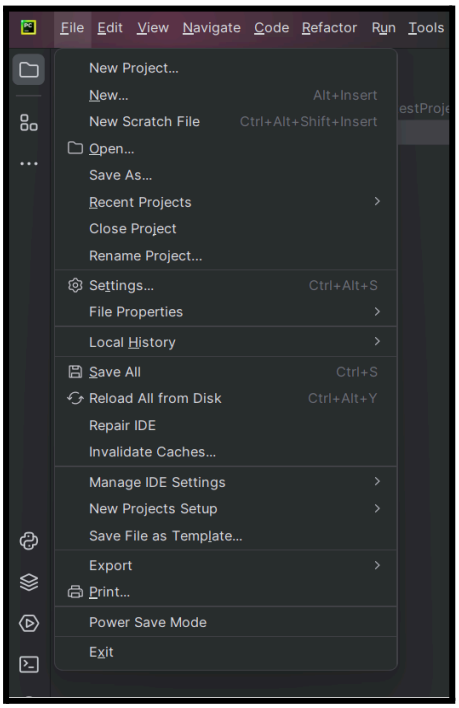


Limitations of This Solution: Overall, Replit provides an ideal base from which we can format the GUI. That said, there are still additional, extraneous features that will be unnecessary for relatively simple OCR GCSE pseudocode such as the 'Tools' section on the bottom-left, with features such as 'AI' and 'Deployments', **justified** as they are not likely to be used by a student responding to exam questions; the division of the 'Console' and the 'Shell', which are quite similar functionally; and the top bar from which we only really need to retain 'Run' (currently says 'Stop' - implementing this is extraneous, especially as it may require communication between the interpreting file and the GUI interface, which can add to programming complexity without significant gain, thus **justifying** its exclusion). Steps forward from these observations include the non-inclusion of these extra sections (**justified** by the stakeholder preference of two to three simple, vertical divisions with the primary purposes of file handling, text editing, and inputting/outputting and that of usage simplicity and self-explainability) and the inclusion of only the console (**justified** as stakeholders require a place for program input/output to occur, and these features are clearly necessary for some program functionality/usability).

What I Can Apply: The program will be split into three main vertical divisions (file-handling window, text editor, and console), **justified** by the aforementioned reasons of usage simplicity and stakeholder preference. The use of the word 'Files' above the file-handling window is a useful idea and we can incorporate the same feature above the text editor and the console. This is **justified** as because there will be variance within the relatively large population of OCR GCSE CS students, it is important to maximise usability aid - explaining each component to the greatest extent possible supplemented by succinct text descriptions can be a part of that. The clear 'Run' button, though not pictured, is green - the use of colour associations (e.g. green is often associated with starting) is useful, **justified** as this will further aid in usability (by leading to faster recognition times of button functions). Neutral buttons, such as the theme picker, can be yellow, for example. Finally, the error message pictured should be included, **justified** as it satisfies the stakeholder preference of having descriptive errors and maximising user support in debugging. That said, this one is quite drawn-out and unclear (due to the inclusion of multiple errors). Instead, we shall structure the error message with error type, line number, and do this for the first instant an error is found (i.e. a single error).

PyCharm File Management System





Limitations of This Solution: Most apparently, the 'New File' function enables multiple file types to be opened. This is unnecessary for this GCSE OCR pseudocode interpreter, **justified** as there is no support within the documentation as far as file-handling is concerned beyond text files. As such, we can simplify usage (further **justified** as it meets the stakeholder preference of a simple, easy-to-use GUI) by having a single, clear 'New File' button (further **justified** as it meets the stakeholder preference of clear, visible buttons) that enables the user to only open text files, perhaps with an error pop-up if the file is not a text file to avoid difficult-to-foresee errors. Another limitation is that file-handling requires navigating to the small-text 'File' tab, then navigating the drop-down to again a myriad of small-text options, etc. We can simplify this process by using large, clear buttons (justified previously) of 'New File', 'Import File', 'Save File', and 'Delete File', **justified** as these are the main file-handling operations that would be required by a user and that cannot be executed locally on the text editor (like undoing, for example). Furthermore, the use of differentiating colour and large, visible font is ideal, **justified** as it can make clear the function of each button, making it more accessible/readable to those struggling with eyesight issues. We can also avoid the use of 'Projects' and in-folder file management, **justified** as despite the preference of stakeholders of having the ability to have multiple files open, for a given pseudocode project (whether it be the NEA or a simple exam question), it will most likely require a single file, especially as the GCSE OCR pseudocode documentation does not support imports. As such, multiple files may be open (in tabs, not exactly open simultaneously), but they may be from disparate folders.

What I Can Apply: When clicking the 'New File' button, there will have to be a mechanism for naming the file, **justified** as that is its purpose. Here, the floating window solution provided above

will work well, **justified** as it exists a layer above the rest of the IDE and therefore will not infringe on space or require restructuring. However, here, naming the file stores it within the current project - which as **justified** before is not a mechanism we are supporting. As such, our floating window will be the standard File Explorer file-creation mechanism. When clicking the 'Import File' button, the user is automatically brought to File Explorer - this can be incorporated into our IDE, **justified** largely by the inclusion of that feature within Python Tkinter, which is the graphical package I have opted for, and again by the fact that the File Explorer window exists a layer above the IDE and that satisfies the user preference of a simple, vertically divided, constant UI.

Essential Features of the Proposed Computational Solution

Fundamentally, the proposed computational solution can be divided into an aesthetically simple GUI, three primary, independent interface features - a file-management system, a text editor, and a console, and a back-end interpreter. To **explain**, though the GUI is intertwined with the aforementioned interface features, they are sufficiently functionally complex to be considered as standalone. The specific details of these features, and other user-preference, ease-of-use, non-essential features, will be explained and further justified in the 'Stakeholder Requirements' part of the 'Analysis' section. The explanations for these essential features, including a description of how they will be used and justification, are:

- **Aesthetically Simple GUI:** Despite functionally being non-essential, aesthetic simplicity and its associated intuitiveness will be needed to cater to the potential variance in skill levels of a possibly wide user base (GCSE OCR students, teachers, and examiners). This will be implemented, **explained** briefly, through the use of three primary vertically divided sections (intuitively, this will coincide with the aforementioned interface features, thus having logical, ease-of-use aspects), large, visible, icon-supplemented buttons (**explained** as this accommodates for eyesight/language variance and contributes generally to descriptive clarity), large font sizes (**explained** as this accommodates for eyesight variance and contributes to readability), pretty printing (**explained** as this supplements the user's learning of keywords, which is a primary purpose of this application, while helping identify syntax errors promoting user support), and a theme picker (**explained** as this satisfies a stakeholder preference, being aware of the notion that different users prefer different colour settings while providing support for certain colour-blindness conditions). The GUI will be implemented with Python Tkinter as it is a fairly comprehensive graphical platform still providing the freedom and complexity required for this project, and Python exists as a useful backbone with garbage collection, etc.

- **File Management System:** For a user to write programs, computer-stored text files will be the main medium (**explained** as other files types are not necessary as the OCR pseudocode documentation does not specify functionality beyond reading/writing to text files, and text files provide ample support for the entry of textual code), and validation will be done to ensure that this is the only openable file type. This window will reside on the left-hand-side (**explained** as this is standard in existing solutions) and will have four buttons for the primary file operations - 'New File', Import File', Save File' and 'Delete File' (these are intuitively placed above the file hierarchy section, which is just a list of file names that can be clicked to be opened, being logical and understandable to users, while being a fairly exhaustive list of the necessary file operations that cannot occur within the text editor).

- **Text Editor:** This is fairly self-explanatory, being a space in which the GCSE OCR pseudocode is written for eventual interpretation. Considering some pseudocode problems may be long (e.g. planning sections of the NEA), this will have to be scrollable (especially as the font size is ideally large as explained previously) with line numbers (for the location-identification of errors and for keeping organised, providing user support), thus **explaining** it. Furthermore, pretty printing will be an additional feature incorporated (as **explained** in the 'Aesthetically Simple GUI' section).

- **Console:** To **explain**, this will be the input/output mechanism for pseudocode programs. As such, it is of limited interaction, not having functional write capabilities when a program is not asking for input, and this will be enforced in code. There is also no need for traditional console functions such as directory-changing and file-listing, **explained** as OCR pseudocode, in its relatively elementary applications, will not benefit significantly from this added functionality, especially when considering the extra complexity a user must then digest.

- **Interpreter:** To **explain**, existing in the back-end (layer of abstraction for simplicity as users do not need to see how their code is specifically executed), this will have to support everything listed and syntaxed in the OCR GCSE pseudocode documentation (as per stakeholder preference, but intuitively completeness is a useful feature). We shall make use of the current standard (**explained** as this is the most up-to-date, exam-applicable - which is of primary importance to our stakeholders - guide) - the OCR GCSE 9-1 Pseudocode Guide as of February 2015. Specifically, there will be variable (local and global) declaration, casting functions, inputting/outputting, loop structures (for, while, do-until), various operators, selection structures (if, switch/case), string-handling functions, subroutines (functions, procedures), file-handling (reading, writing), and comments (in addition to descriptive error messages for user support, **explained** as it facilitates faster debugging as per stakeholder preference).

Limitations of the Proposed Solution

Overall, the proposed solution provides the effective base features necessary for a useful GCSE OCR pseudocode IDE (with additional ease-of-use features such as pretty printing for learning), though some useful, additional features are not included either due to subjectively unnecessary complexity or the indifference of stakeholders.

- **Lack of a Built-in Debugger:** To **explain**, an expansive, built-in debugger (with breakpoints, etc.) is not included. This is **justified** as there are already extra debugging capabilities within descriptive error messages in the console. Furthermore, providing too much support to students could limit how active they are in debugging themselves which dampens their learning efforts, contradicting a primary goal of this application as pointed out by Ms. Tulin. To **justify** further, it also adds to computational complexity and the usage of computational resources, which unreasonably increases hardware requirements (decreasing access) disproportionately to the potential gain of easier debugging.

- **Lack of a 'Project' System:** To **explain**, there is also not a 'Project' system, in which IDE windows are organised by folders. This is **justified** as the relative simplicity of pseudocode algorithms does not require communication between multiple files (and the GCSE OCR pseudocode documentation does not provide support for imports for this to occur in the first place), so files can be considered independently and without the traditional 'Project' system in mind. To **justify** further, while this may subvert standards, it is not significant enough to be too noticeable, and it will not affect the main functionality of the solution (executing pseudocode programs).

- **Slow Python Backbone:** To **explain**, Python is a relatively slow high-level programming language, thus likely leading to higher execution times. This is **justified** by the need for a comprehensive graphical platform (Python has Tkinter), necessary for various GUI functions we plan to implement (button-clicking, changing windows, an independent window, etc.); the fact that highly optimised execution times are unnecessary for relatively simple GCSE pseudocode programs (already quick enough); and that Python is sufficiently comprehensive anyway (Turing complete, having access to conditionals, loop structures, classes, etc., useful for conditional interpreting, modularity, etc.)

Requirements of the Proposed Solution

Software Requirements

Requirement	Justification
Pre-Installed Python Interpreter	The code will be written in Python (primarily due to the existence of Tkinter, a sufficiently comprehensive graphical platform, but also due to features such as possessing various programming constructs, garbage collection, etc.), so a Python interpreter will be required. Furthermore, as an executable file cannot be produced, an external application interpreter will be required
Python Tkinter	This is a sufficiently comprehensive graphical platform which provides useful features that cover our GUI needs, such as an independent separate window (which can be full-screened), vertical-scroll text boxes, buttons, drop-downs, etc. Furthermore, the module is reputed and well-tested, thereby reducing extraneous, unnecessary further validation and allow us to focus on implementing the main interpreting functionality
Windows (7 or Later), macOS, or Linux Operating System	According to Python's official documentation, these are the minimum OS requirements for Python to run on a computing device (Python is hardware-agnostic)

Hardware Requirements

Requirement	Justification
Computer with Standard Peripherals (Pointing Device and Keyboard) or Laptop with Built-in Keyboard and Pointing Device	A computer/laptop capable of installing a Python interpreter and executing the eventual machine code is necessary to run the program as it is built on a Python base. A keyboard is necessary as there are ASCII character entry requirements when coding/inputting. A pointing device, with cursor control and clicking features is necessary to interact with the buttons in the program. A screen is necessary to view console output and view UI, etc.
At least a 1 GHz Processor (e.g. Intel Atom® Processor or Intel® Core™ i3 Processor)	According to Python's official documentation, these are the minimum processor requirements for Python to run on a computing device (Python is hardware-agnostic)
At least 1 GB RAM	According to Python's official documentation, this is the minimum RAM required for Python to run on a computing device. Also, necessary due to heavy use of call stack (nested statements handled by the interpreter)

At least 250 MB Disk Space	According to Python's official documentation, this is the minimum secondary storage space required for Python to run on a computing device (to avoid partial installations or cancelled installations preventing IDE use)

Stakeholder Requirements / Success Criteria

Requirement/Criterion	**Justification**	**How To Evidence/Measure (Measurable)**
1400px by 700px GUI	Preferred by interacting stakeholders during interview; Provides a large space for UI elements, such as large font sizes, large buttons, a wide text editor, etc., thus improving readability and usability, especially for those with weaker eyesight, while not being entirely full-screen such that other windows can be interacted with simultaneously	Screenshot of 1400px by 700px Tkinter window GUI after opening IDE + Screenshot of section of code in which 1400px by 700px GUI is implemented
GUI divided into three primary, different-coloured, vertically divided sections (excluding top bar)	Minimal windows preferred by interacting stakeholders during interview; Horizontal divisions have been shown to typically squash console output with PyCharm, which is undesirable; Removing unnecessary windows allows the main features to be highlighted/provides maximum space for text-editing, console output, etc., which have been identified as the main features by the interacting stakeholders; Different colours divide each section clearly	Screenshots of three primary vertically divided sections of different colours for each available theme
Each primary section has headings 'Files', 'Text Editor', 'Console' (from left to right)	Makes clear the purpose of each section, especially for those unacquainted with traditional IDE layouts	Screenshot of three section headings being 'Files', 'Text Editor', 'Console' (from left to right)
Large font size in text editor and console	Preferred by interacting stakeholders during interview; Larger font improves readability, being an important usability feature for those with weaker eyesight	Screenshot of text in Text Editor + Console + Statements of acceptance from each interacting stakeholder saying font size in text editor and console is sufficiently large
Coloured buttons consisting of a text description and a suitable icon	Preferred by interacting stakeholders during interview; Colour helps to differentiate buttons from the background and improve visibility; A	Screenshots of each button, showing the colour, text description, and icon

	combination of a text description and an icon provide greater accessibility to a greater variation in user language and experience	
‘Change Theme’ button opens a drop-down of themes for ‘Dark Mode’ and ‘Light Mode’	‘Dark’ and ‘Light’ preferred by interacting stakeholders during interview; Provides usability support for both preferences and reduced harshness on eyes; A drop-down creates minimal disturbance to the rest of the application, which makes sense as theme-picking is a minor aspect of the solution	Screenshot of drop-down menu with themes ‘Dark Mode’ and ‘Light Mode’ after clicking ‘Change Theme’ + Screenshot of ‘Dark Mode’ IDE after clicking ‘Dark Mode’ + Screenshot of ‘Light Mode’ IDE after clicking ‘Light Mode’
File hierarchy consisting of file names (support multiple) that can be clicked to be opened	Necessary for the navigation and improved organisation of multiple files, a feature preferred by the interacting stakeholders during interview	Screenshot of file hierarchy consisting of file names + Screenshot of file being opened in the text editor after clicking it in the hierarchy
‘New File’ button allows an empty (excluding line number), new text file to be created	It is empty by default, apart from the initial line number, as the latter is necessary to avoid unwanted syntax errors and the former is standard practice	Screenshot of empty text editor after ‘New File’ is clicked + Screenshot of empty file added to the file hierarchy
‘Import File’ button opens File Explorer, allowing an existing text file to be selected	File Explorer is a standard file-handling mechanism, and it provides a well-tested, reliable mechanism for file opening	Screenshot of File Explorer being opened after clicking ‘Import File’ + Screenshot of the file appended to the file hierarchy
‘Save File’ button saves file currently opened in the text editor	File storage was preferred by the interacting stakeholders during the interview; Enables work to be saved for future reference and transfer (e.g. to teacher for marking)	Screenshot of saved file with changes before and after clicking the ‘Save File’ button respectively
‘Delete File’ button deletes file currently opened in the text editor	The file-deletion mechanism is preferred by the interacting stakeholders during interview; Useful for freeing storage space easily when a program file is no longer necessary	Screenshot of file present and absent in its intended location before and after clicking the ‘Delete File’ button respectively
‘Import File’ only enables the opening of text files	OCR GCSE pseudocode only uses ASCII characters completely supported by text files; The only file-reading/writing capabilities supported by the OCR GCSE pseudocode documentation are for text files; Extra file support would be inaccurate and extraneous complexity-wise	Screenshot of section of code where this is implemented + Screenshot of error messages produced when non-text files are opened

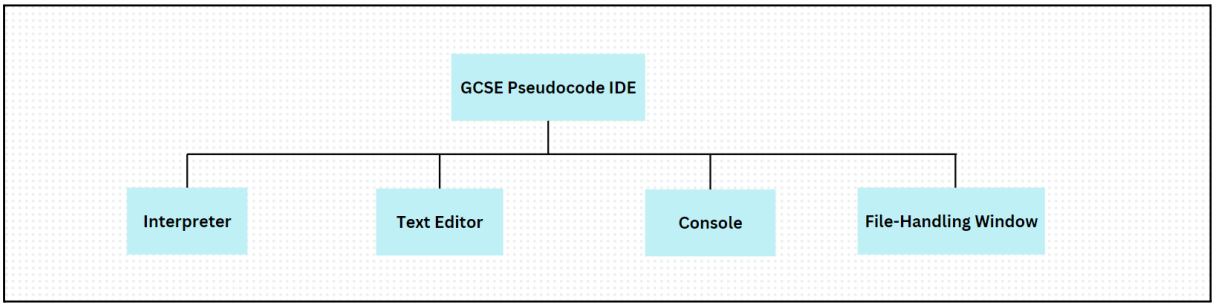
‘Run’ button executes program, updating console where applicable	Some way to execute the program is necessary (the primary purpose of this solution); A button is clear, visible, and a recognisable standard	Screenshots of console being updated (where applicable) after clicking the ‘Run’ button
Pretty printing in the text editor	Preferred by the interacting stakeholders during interview; Highlights keywords aiding students in memorising and understanding them, and education is an important auxiliary purpose of this solution	Screenshots of each keyword category being highlighted a particular colour
Vertically scrollable text editor with line numbers	The text editor may contain large programs that cannot be observed at once, especially due to large font sizes and the large nature of programs such as the planning sections of the NEA. As such, there should be a way to scroll the text editor to navigate to previous/later sections of pseudocode; Line numbers are necessary to remain organised amidst high-line-count programs and so that error messages can reference them for debugging aid	Screenshot of text editor with line numbers (incremented by 1 each line) + Screenshot of working vertical scroll bar
Console has input/output capabilities	There needs to be a section for the program output to be seen (and therefore checked or used usefully); There needs to be a section for input as some programs use the input() function, thereby being necessary for functionality and completeness	Screenshot of console output after a program is run by pressing the ‘Run’ button + Screenshot of console input after a program requiring input is run by pressing the ‘Run’ button
The interpreter must produce the correct output for a program consisting of any of the statement types mentioned in the OCR GCSE Pseudocode documentation	Intended output is obviously necessary, otherwise making the solution misleading and inaccurate; Must work for all syntax for the purposes of completeness, wide applicability (e.g. marking any student response, planning an extensive NEA project), and accuracy	Thorough white-box testing for each foreseeable usage of syntax + Faultless beta black-box testing results from the interacting stakeholders
The interpreter produces and the console consequently outputs error messages consisting of the error type and line number	Error messages provide debugging aid such that users are sufficiently supported and have a reasonable understanding of where the error is, thus speeding up debugging so they can focus on code-writing; The	Screenshots of various error message types, providing evidence of the error type and line number

	debugging aid is limited such that not too much support is provided to students (as in the case of an exhaustive debugger), which may be useful in simulating exam conditions	

Design

Systematic Breakdown of Problem

The goal of this section is to build a general hierarchy diagram that logically divides the overarching solution - building a useful, stakeholder-satisfying IDE for GCSE OCR pseudocode - into smaller, computationally suitable problems that can be targeted independently, **justified** as this is done with the purpose of maintaining modularity for organisational and robust, unit-testing purposes. The detailed implementation of the leaves of this hierarchy diagram and the nature of the logical recombination will be described in the 'Definition of Solution Structure' and 'Algorithms of the Solution' sections. As discussed previously and as is evident within the success criteria, the large, independent sections of the solution are the interpreter, text editor, console, and file-handling window, which will be **explained/justified** and further broken down now. Firstly though, we can build a surface-level depiction of the hierarchy diagram.



Further Breakdown of Interpreter

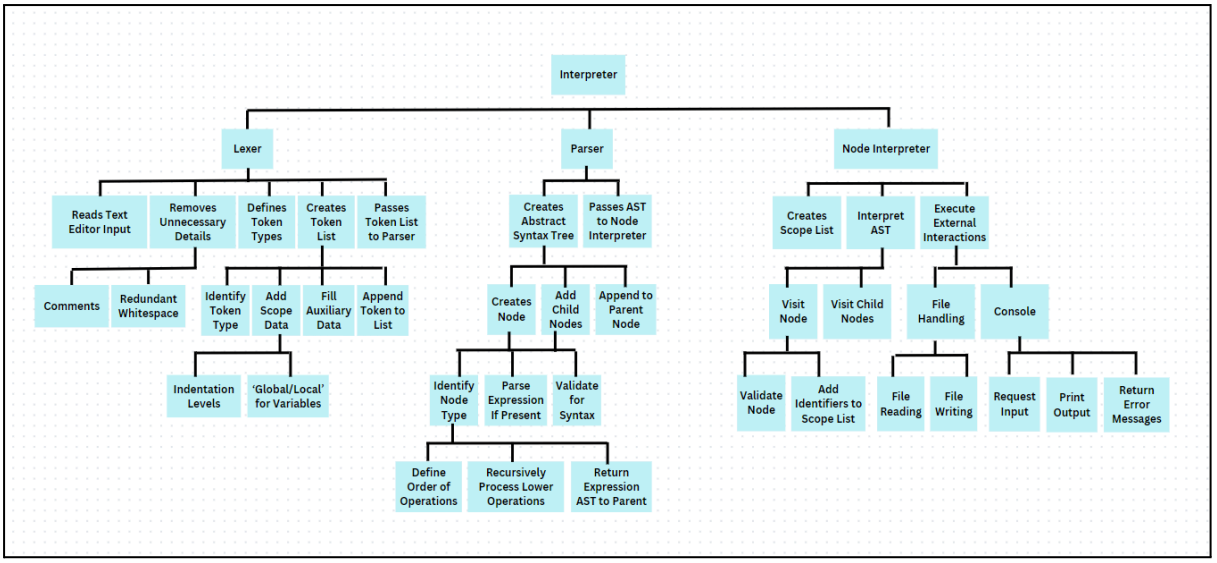
The **justification** of the interpreter being an independent feature is primarily that it exists in the back-end, performing the more computationally intensive/integral aspect of the solution - running pseudocode - (to **explain**) connected to the rest of the application in a limited fashion only by the 'Run' button (primary connection to GUI), the returning of error messages, input requests, and program outputs (primary connection to the console), and the file-reading of the text editor input (primary connection to the text editor) (explanation). Beyond that, it has no direct connection to the file-handling window, with most complex functionality occurring self-contained, translating high-level pseudocode into interpretable Python analogues via sufficiently complex algorithms. The **justification** of its importance is intuitive - the primary function of the solution beyond extra stakeholder-preferred IDE features is pseudocode interpretation - and the interpreter is the component that carries this process out in a complete sense (as specified in the Success Criteria).

Traditionally, the first stage of interpretation is the 'Lexer'. To **explain**, this reads the text editor input, removes initial unnecessary details (for interpretation) of comments and redundant whitespace (**justified** abstraction as otherwise, constant validation would be required within multiple functions to handle these extraneous features, which unnecessarily repeats code and lengthens the program, making maintenance and unit-testing more difficult), defines token types (**justifiably** a leaf node as this can be done after combing through the OCR GCSE pseudocode documentation), creates a list of tokens, which represent the fundamental function of a keyword, etc. within a program (**justified**

abstraction as once again, unnecessary details such as line number, character sequence number, etc. can be removed, making it easier for the next stage of processing), and passes the list of tokens to the 'Parser', the next stage of interpretation. To **explain**, these stages of lexing can be considered sufficiently simple to form leaf nodes of the hierarchy diagram as they are sequential (pass from one stage to another directly) and are sufficiently logically simple to implement (**justified** as reading the text editor via Python's file-reading mechanism, the removal of unnecessary details can be done through selective conditionals while incrementing through the text editor input, tokens can be generated through searching for the presence/absence of keywords, passing the token list to the 'Parser' is simply a matter of creating the list and returning it to a 'Parser' object, for example), though this will be expanded upon during the 'Definition of Solution Structure' section. To **explain** the token list, this consists of identifying the token, which consists of identifying scope information such as determining indentation levels, and filling the appropriate auxiliary data for the token (**justified** as a leaf node as though breakdown is possible into what this means for each token type, that will be intuitive when we reach the 'Algorithms of the Solution' section) and the presence/absence of the 'global' keyword for variable declaration statements and appending the token to the token list (all **justifiably** leaf nodes as indentation levels depend upon whitespace count and the 'global' keyword can be determined while character reading - identifiable criteria).

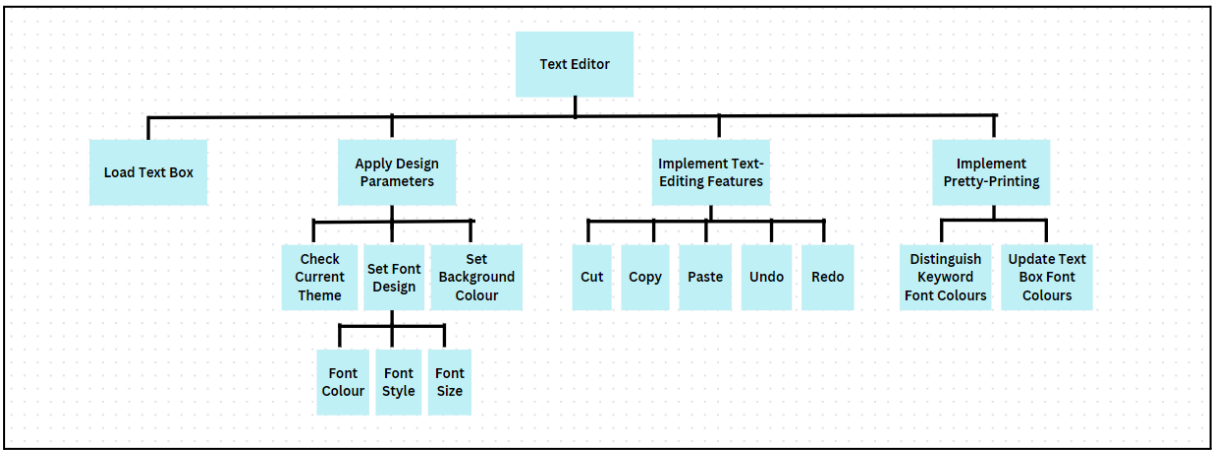
The second stage of interpretation is the 'Parser'. To **explain**, this broadly creates an abstract syntax tree (a tree structure that orders the components of a program hierarchically, thus determining the order of operations and processing the interpreter must finally carry out) and passes the abstract syntax tree to the 'Interpreter', the final stage of interpreting (**justifiably**, this is a leaf node as it is sufficiently simple to implement by returning the statement-specific node classes connected with pointers - or by enclosed attributes - to the 'Interpreter'). To **explain** the creation of the abstract syntax tree, the first stage would be to identify the node type (which can be done through the identification of unique elements within that statement, such as 'if' or 'for', thus **justifiably** being a simple leaf node); validate for syntax errors (**justifiably** a leaf node as these will be obvious due to the strict syntax of OCR's documentation); append internal statements to this node (for example, 'if' statements contain variable declaration statements, nested 'if' statements, etc. within them. To **justify**, this sufficiently simple to be a leaf node as the process by which the internal nodes are defined is exactly the same as the current node, and node definition will soon be completely fleshed out in a leaf-node fashion); and finally append the current node to the parent node in a similar recursive manner (**justifiably**, sufficiently simple to be a leaf node as it can be implemented intuitively through returning from a recursive call by the parent node). To **explain** the identification of node type, though this is sufficiently simple to implement through the identification of unique keywords, some components are still fairly complex, and they will be broken down now. Namely, the existence of expressions (arithmetic, Boolean, etc.) is fairly complex and can be broken down into defining an order of operations (**justifiably** a leaf node by using PEMDAS and the standard Boolean analogue), recursively process increasingly lower-level operations with the order of operations in mind (**justifiably** a leaf node as this is purely based off of the previous part), and return the expression abstract syntax tree to the parent node of the 'Parser' (**justifiably** a leaf node as a simple Python 'return' function can be used with the local abstract syntax tree of the expression).

The final stage of interpretation is the 'Node Interpreter'. To **explain**, this creates a scope list to manage the scope of identifiers (**justifiably** a leaf node as multi-dimensional lists can simply be used to emulate scopes); visits a node (beginning at the overarching program node), which entails validating it (**justifiably** a leaf node as though errors can be further broken down, this is situational and can be discussed in the 'Algorithms of the Solution' section), visiting child nodes (justifiably a leaf node as visits are simply a function call. To **explain**, children node-visiting is necessary to recursively process lower levels), suitably adding identifiers to the correct location in the scope list (**justifiably** a leaf node as this can be simply implemented through a stack mechanism); and executing on the console where appropriate, which entails returning error messages when necessary, requesting for input, or printing output (**justifiably** leaf nodes as these are each simple, single function calls).



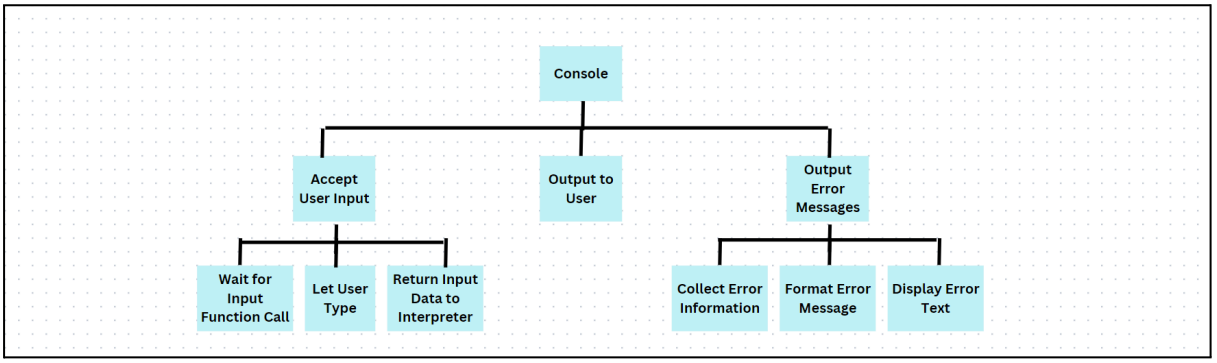
Further Breakdown of Text Editor, Console, and File-Handling Window

The **justification** for the text editor being an independent feature is that it has limited connection to the other program component, connected only to the interpreter during the text-reading stage of the 'Lexer' and the GUI Features by the 'Change Theme' button. To **explain**, it is self-contained with a primary purpose of enabling users to conveniently enter pseudocode for later interpretation, with the bonus of pseudocode-specific aesthetic features. **Explaining** the breakdown, the stages are loading the text box (**justifiably** a leaf node as Tkinter has a sufficiently simple Text class); applying the design parameters, which can be further broken down into checking the current theme (**justifiably** a leaf node as this can be stored in an accessible variable), setting the background colour, the font colour, the font style, and the font size (**justifiably** leaf nodes by altering the attributes of the Text object); implementing the text-editing features, which can be further broken down into 'Cut', 'Copy', 'Paste', and 'Undo' fairly exhaustively (**justifiably** a leaf node as keyboard shortcuts and auxiliary variables can aid simply in the data storage required for all of these features); and implementing pretty-printing, which can be further broken down into distinguishing keyword font colours (**justifiably** a leaf node as this information can be stored simply in auxiliary ADTs) and updating text box font colours (**justifiably** a leaf node as this can be simply implemented through a loop mechanism searching for keywords or a spacebar-triggered action).

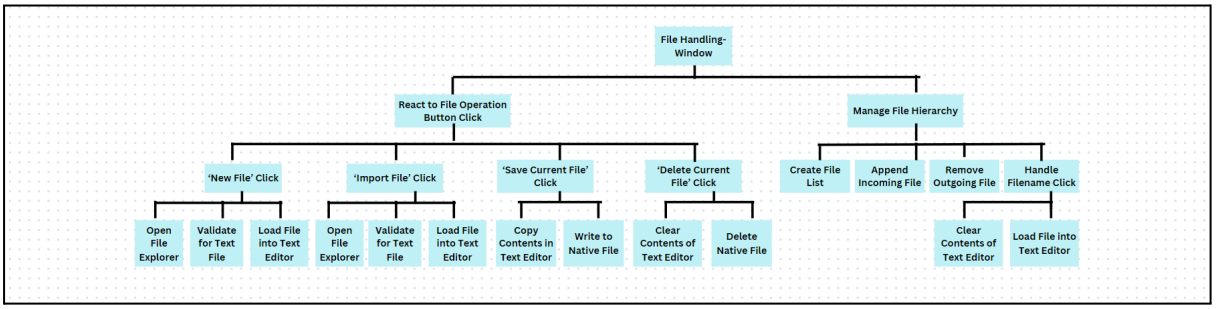


The **justification** of the console being an independent feature is that it is largely self-contained with a primary purpose of being an intermediary between the interpreter and the user, only connected to the interpreter during the 'Node Interpreter' stage and the GUI Features due to a visible design.

Breakdown/**explaining**: the stages are accepting user input, which can be broken down into waiting for the input function call (**justifiably** a leaf node as this connection can be made within the input function call processing), letting the user type (**justifiably** a leaf node as this can be implemented simply through a Text object), and returning the input data to the interpreter (can be simply done with a 'return' statement); outputting to the user (**justifiably** a leaf node as can simply display required text from interpreter input); and outputting error messages, which can be further broken down into collecting error information (**justifiably** a leaf node as the interpreter can return this data), formatting the error message (**justifiably** a leaf node as this is a simple case of string manipulation), and displaying error text (**justifiably** a leaf node as the console will have a Text element to display to).



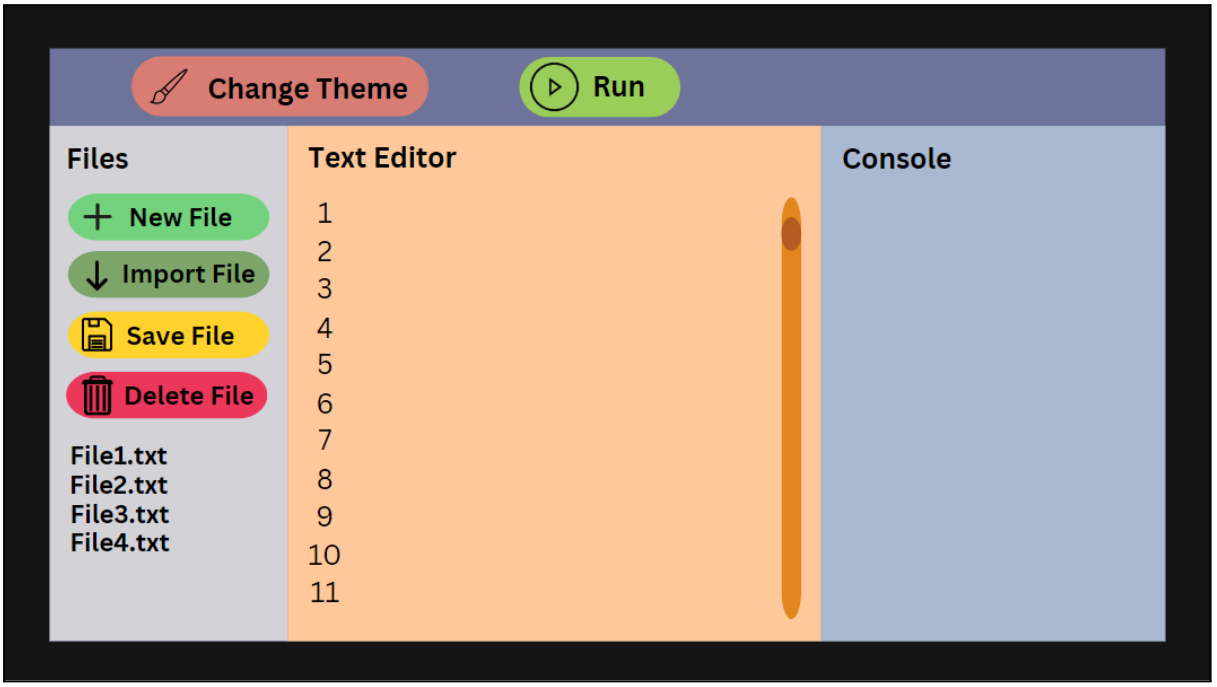
The **justification** of the file-handling window being an independent feature is that it is only linked to the text editor by the file operations acting upon the current text file and the GUI Features by the file operation buttons, having a primary purpose of transferring/managing files between the text editor and the computing device. Breaking it down/**explaining**, the stages are reacting to the file operation button click, which can be further broken down into 'New File', which opens File Explorer (sufficiently simple to implement through built-in Tkinter file creation features), validates to ensure it is a text file (**justifiably** a leaf node as a conditional on the file extension can be executed), and loads the file into the text editor (**justifiably** a leaf node as this can be implemented by mirroring the file text); 'Import File', which similarly opens File Explorer, validates to ensure it is a text file, and loads the file into the text editor (all leaf nodes for the above reasons); 'Save File, which copies the text in the text editor and writes the text into the file currently opened (**justifiably** leaf nodes as file-reading/writing mechanisms are built into Python); and 'Delete File', which clears the contents of the text editor (sufficiently simple via resetting the text editor UI section) and deletes the current file (**justifiably** a leaf node through Python's file-deletion features); and manages the file hierarchy, which involves creating a list of files to display (sufficiently simple via a list of file names, initially empty), appending an incoming file (sufficiently simple via appending to the file list and displaying), removing an outgoing file (**justifiably** a leaf node by removing from the file list and displaying), and functionality for clicking a file name, which clears the contents of the text editor (**justified** previously) and loads the clicked file into the text editor (**justified** previously).



Detailed Definition of Solution Structure

GUI Structure

Firstly, an image of the planned GUI will be displayed, which will be **justified** afterwards. The colours used within this image do not reflect the final colours of the program and it should be taken as a visualiser for content and layout alone.



As we can see, it fulfils the default GUI success criteria listed prior. Specifically, a [1400 x 700 pixel GUI] (implied), [GUI divided into three primary, different-coloured, vertically divided sections], [Each primary section has headings 'Files', 'Text Editor', 'Console' (from left to right)], [Large font size in text editor and console] (constant font size across both, so implied), [Coloured buttons consisting of a text description and a suitable icon], ['Change Theme' button opens a drop-down of themes for 'Dark Mode' and 'Light Mode'] (though the drop-down will be a default one from Tkinter with modifications to match font styles. The specifics of 'Dark Mode' and 'Light Mode' will be expanded upon in the 'Colour Palette Structure' section), [File hierarchy consisting of file names (support multiple) that can be clicked to be opened], and [Vertically scrollable text editor with line numbers]. Overall, the **justification** of this GUI design is primarily that, as evidenced, it contains all of the default GUI features required by stakeholders while also maintaining a general simplicity in its minimalistic style (lack of harsh borders, large, clear text in a readable font, etc.) that reflects the intended simplicity of an OCR Pseudocode IDE, potentially used by inexperienced, novice students. As all GUI Success Criteria were considered when designing the GUI and were described above in this paragraph, this definition of GUI structure is **detailed**.

Colour Palette Structure

Overall, the colour palette is dependent upon the current theme selected, which will vary, as per stakeholder preference between 'Light Mode' and 'Dark Mode'. To clarify, the former mostly uses light colours (often close to white) in its background whereas the latter uses darker colours. In pre-justification of the colour palette to be described, the UI colours existing within the foreground will need to contrast these schemes such that the interactable features (buttons, text, etc.) are maximally visible, and this will be visualisable when the colour palette table is completed. This has the benefits of usability for those with weaker eyesight in addition to general aesthetic simplicity, a

stakeholder preference that cannot be directly measured but is desirable nonetheless. Considering each element's colour will be defined here, the Colour Palette Structure is **detailed**.

GUI Feature	'Light Mode' Colour	'Dark Mode' Colour
Top Bar Background	**#E0E0E0**	**#111145**
File-Handling Window Background	**#EDF0F1**	**#2E2E6C**
Text Editor Background	**#FFFFFF**	**#34348E**
Console Background	**#CCCCCC**	**#202057**
'Files' Header	**#474747**	**#F8F8FC**
'Text Editor' Header	**#474747**	**#F8F8FC**
'Console' Header	**#474747**	**#F8F8FC**
Text Editor Text (Not Pretty Printing) + Line Numbers	**#282828**	**#FDFDFD**
Console Text	**#282828**	**#FDFDFD**
Console Error Text	**#F33A3A**	**#F33A3A**
Filename Hierarchy Text	**#282828**	**#FDFDFD**
Button Text	**#282828**	**#FDFDFD**
'New File' Background	**#82E769**	**#3C9805**
'Import File' Background	**#5CDE5D**	**#07870F**
'Save File' Background	**#FFE240**	**#FF8E00**
'Delete File' Background	**#FF5C62**	**#F64214**
'Change Theme' Background	**#D98CFF**	**#B203E7**
'Run' Background	**#C0FA33**	**#07B742**

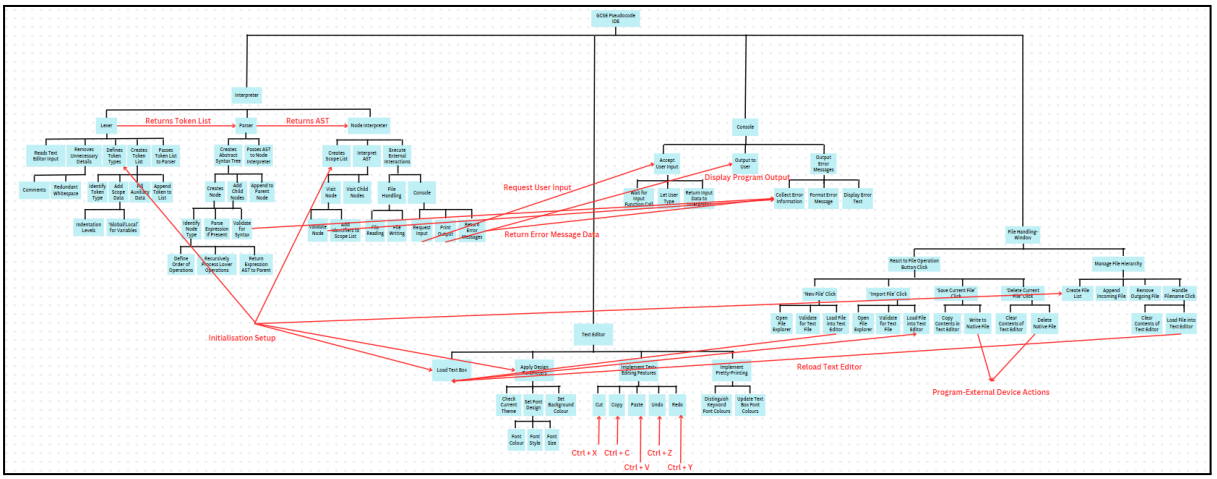
Logical Structure

This section aims to qualify how components are connected together, which will be useful in the **justification** of how the algorithms to be designed in the 'Algorithms of the Solution' section form a complete solution to the problem. This will be done by exploring exhaustively how the program could be used, ending with a visualisation. Firstly, each feature is static until interacted with when the program is opened and the GUI loaded. These start-up initialisations include the setting of the default theme to 'Light Mode' (arbitrary, but 'Light Mode' being an older standard makes this conventional); the loading of the text box, the application of the default design parameters (as per 'Light Mode', standard font style, etc.), and the creation of a default unsaved, empty file with one line for typing available by default within the text editor; nothing beyond basic GUI definition for the console as this purely interacts with the interpreter; and the declaration of the file list (initially empty) within the file hierarchy.

Most other component-to-component transitions occur via button clicks. Specifically, the 'Run' button initiates the 'Lexer' stage of the interpreter; the 'Change Theme' button is self-contained and

leads to the theme dropdown, where selecting one switches the design parameters which must be applied to the entire GUI; the ‘New File’ button initiates the ‘New File’ click functionality within the file-handling window, loads the file into the text editor (changing the text box), and appends the file name to the file hierarchy; the ‘Open File’ button is identical though following the ‘Open File’ click functionality route; the ‘Save File’ connects externally, being an intermediary between the text editor and the device (there is also the necessity of checking whether this file exists within the computer or not. If not, a ‘Save As’ function must be provided connected to File Explorer); and the ‘Delete File’ button clears the text editor (thus re-loading the text box) also connecting externally to remove the file from the device. Furthermore, when a file name is clicked, that file is opened in the text editor (in addition to the file-click functionality described in the file hierarchy diagram above), thus reloading the text box. Other interactions include the ‘Cut’, ‘Copy’, ‘Paste’, and ‘Undo’ functionality specified in the ‘Text Editor’ hierarchy diagram, which will be activated using the keyboard shortcuts of ‘Ctrl + X’, ‘Ctrl + C’, ‘Ctrl + V’, ‘Ctrl + Z’, and ‘Ctrl + Y’ respectively as is the industry convention (users will likely have prior experience with this functionality, thus **justifying** it for ease of use). The implementation of the pretty printing function call is currently planned to be a character-triggered one, such that whenever a word is completed (as ended by a ‘space’, ‘tab’, ‘newline’ etc.), the pretty printing function is called and performs iterations on the text editor text, thus rewriting it in the appropriate colour. As such, it is fairly self-contained and this will be made apparent on the diagram beneath this section.

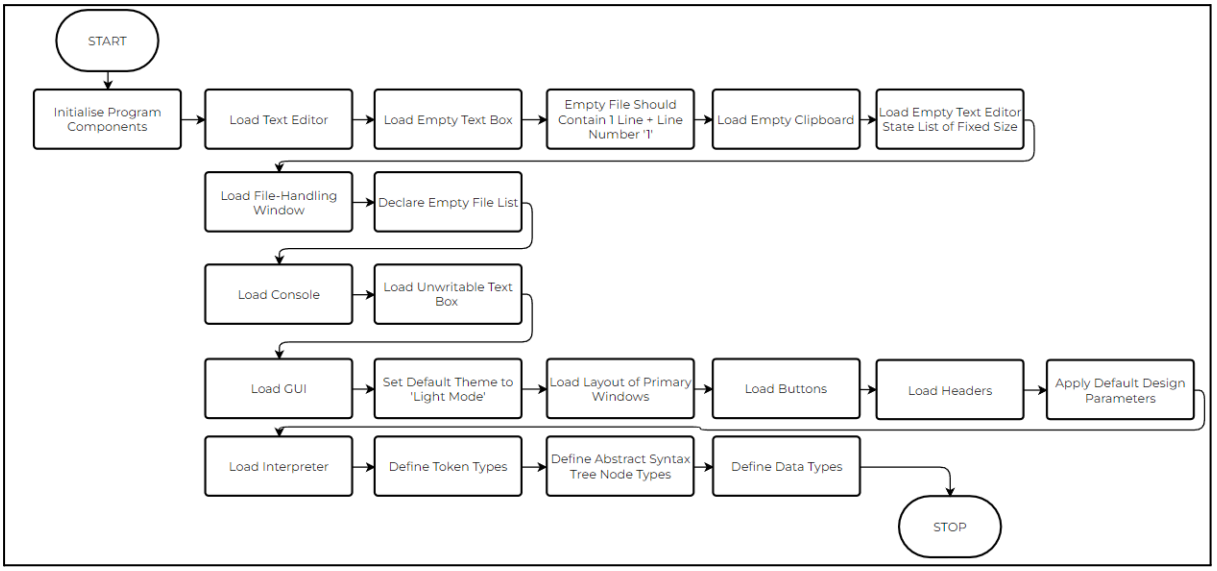
Finally, the interpreter has a large quantity of self-contained functionality, only connected to the rest of the application via the ‘Run’ button and to the console with input, output, and error information. **Justifiably** then, direct connections should be emphasised linearly between the ‘Lexer’, ‘Parser’, and ‘Node Interpreter’ via return functions - the code first passes through the ‘Lexer’ where a stream of tokens is produced, the ‘Parser’ where those tokens are turned into an abstract syntax tree, and the ‘Node Interpreter’ where that tree is traversed logically to execute line-by-line instructions. The ‘Node Interpreter’ also connects directly to the console either requesting for input, returning output information to the console for display purposes, or performing a similar action with error data. To add complete detail for the proposed logical structure described above, the visualisation below connects the hierarchy diagrams completely with annotations for each inter-component transition and its trigger action. The layers are uneven for space reasons, but logically each component in the same layer is decomposed equivalently. As the entire program is decomposed logically, visualised in the image below, with arrows even describing the interactions between components useful for the design of algorithms, the Logical Structure is **detailed**:



Algorithms of the Solution

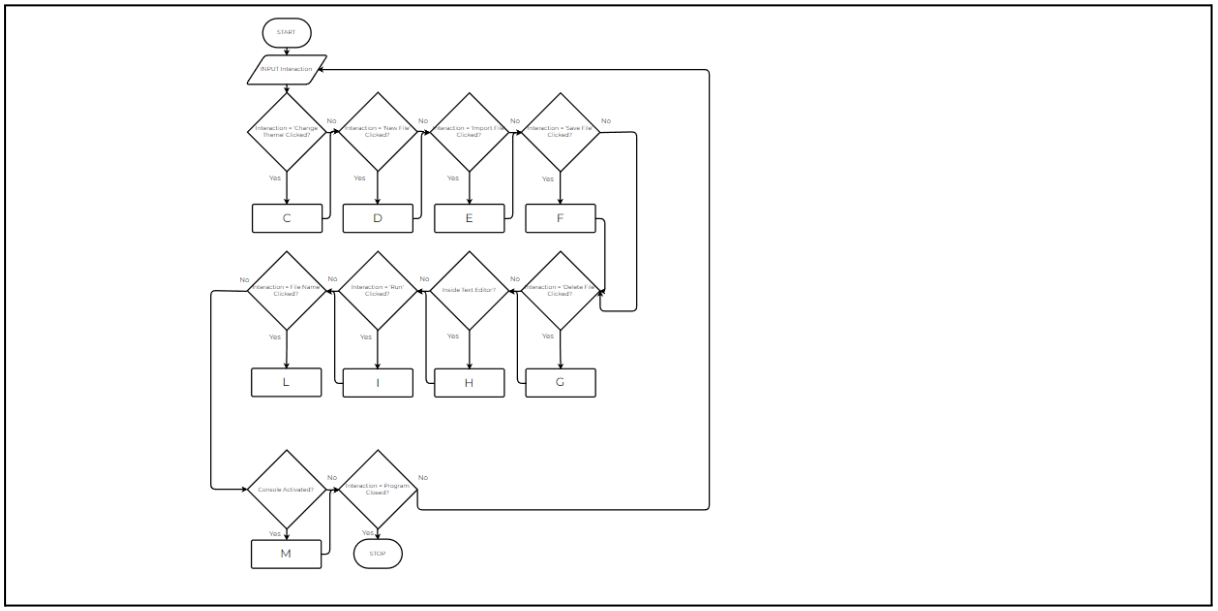
In this section, the primary algorithms that are sufficiently complex to break down further will be detailed explicitly. Helper functions, where used, will exist within these primary algorithms and their functions will be made obvious. Further built-in Tkinter functions, which are relatively standalone such as button clicks triggering further functionality within the program will also be detailed for completeness though will not be emphasised as significantly due to standard implementations for these functions already existing on Tkinter documentation.

Initialisation Algorithm (A)



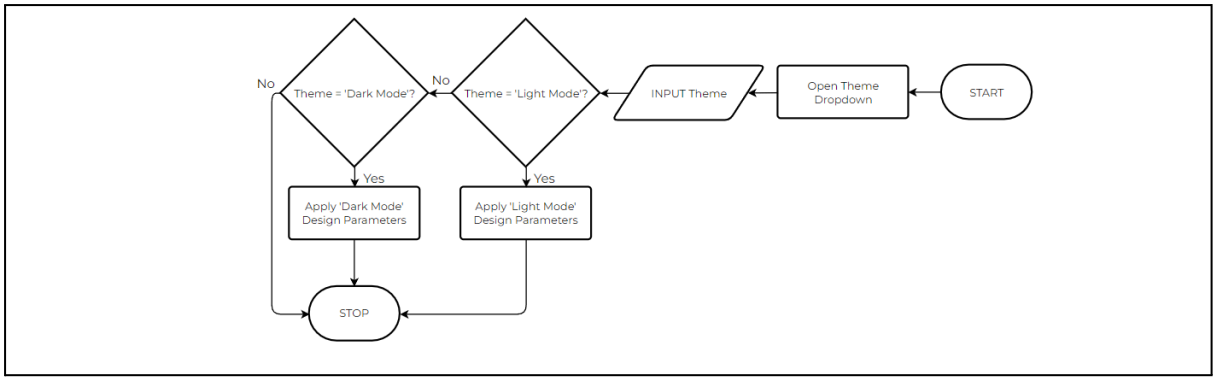
To **describe**, the primary function of this algorithm is to load the required elements of each component before processing begins. Pre-loading these sections is computationally **justifiable** as otherwise, this would need to be done each time a component is interacted with, which leads to wastage in processing time, RAM usage, etc. It is also sufficiently simple to implement as a lowest-level algorithm, **justified** largely by the simplicity of each process box in the flowchart being of a one-line, Python-supported nature or being multi-lined in a very repeatable way, such as the 'Define Token Types' process, which would involve declaring multiple strings (a very simple process). Obviously, one could break down processes such as 'Load Layout of Primary Windows' by defining relative heights and widths, etc., though this has already been performed qualitatively in the 'GUI Structure' section previously and can be managed further during development with stakeholder feedback as it is very much a stakeholder preference that is not directly quantifiable. As all primary components identified in Problem Breakdown are accounted for, it is **complete**.

Interaction-Selection Algorithm (B)



This algorithm forms the backbone of the program, **justified** as the IDE beyond back-end processing is highly interaction-based as discussed previously. To **describe**, the flowchart passes through each potential interaction, being 'Change Theme', 'New File', Import File', 'Save File', 'Delete File', 'Text Editor' internal interactions, 'Run', 'File Hierarchy' filename clicks, 'Console' activation, and the closing of the program for completeness. This is a **complete** list of all of the interactions, **justified** essentially in the 'Logical Structure' section previously, because it encapsulates all of the button clicks and the additional interactions within the text editor, console, and file-handling window (not the interpreter necessarily as this follows from the 'Run' button click and is entirely in the back-end). It is looped in a 'Do Until' style as the program by nature does not have a fixed end point (for example, one can continue to open files, edit them, run them, change the theme multiple times, etc.), and this flowchart only ending when the program is manually closed accounts for this. From here, the other algorithms beyond initialisation can exist for a **complete** program discussed as per the 'Success Criteria' and the 'Definition of Solution Structure'.

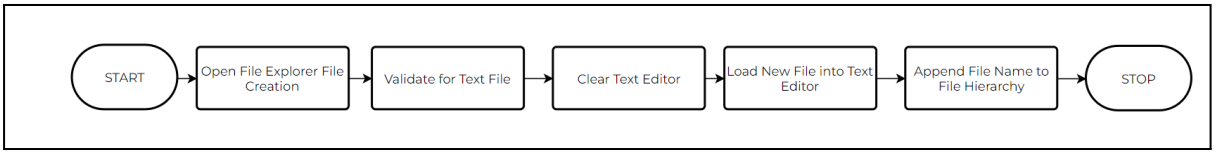
Change Theme Algorithm (C)



To **describe**, this is a fairly self-contained algorithm enabling users to change the colour theme of the overall IDE as per the colours mentioned in the 'Colour Palette Structure' section. This is **justified** as it is a stakeholder requirement to have different colour themes, particularly 'Light Mode' and 'Dark Mode' as this was divided between the stakeholders. To add, there is a usability aspect as different users may prefer different backgrounds, with 'Dark Mode' potentially being less harsh on

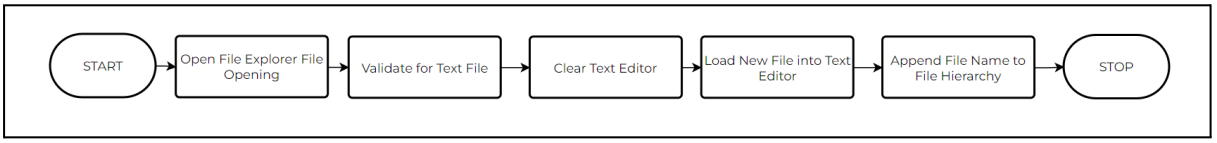
the eyes for those with strained eyesight. The dropdown menu is used as it is a common paradigm and will be easily understood by users, working through intuitive button clicks. As it accounts for both modes, it is **complete**.

New File Algorithm (D)



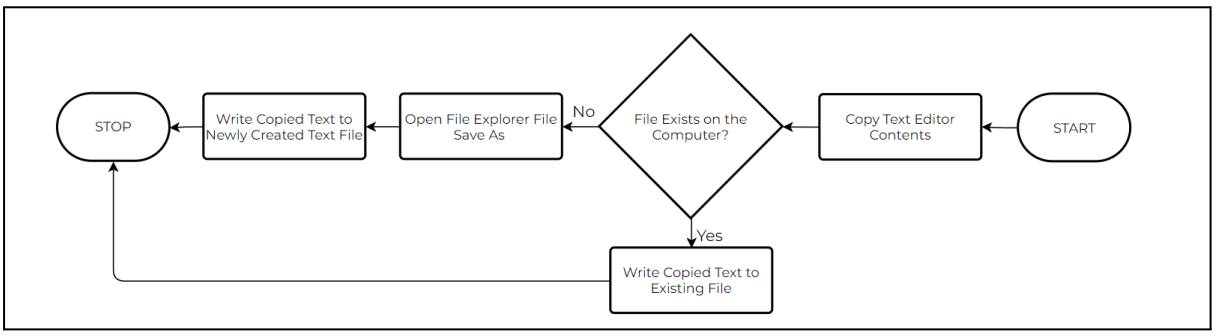
The **justification** of this algorithm is justified as it satisfied both the stakeholder requirements of 'New File' opening text files only (mentioned validation) and that 'New File' uses the simple, supported method of file creation of File Explorer. To **describe**, there is interaction between the file-handling window and text editor here as planned in the 'Definition of Solution Structure' section, where the text editor is cleared and reloaded with the newly created file, with additional processing for appending the file name to the file hierarchy. As discussed, this is a **complete** solution to the file creation problem and is intuitive for end users as after they create the file, a large quantity of processing (loading the file into the text editor, etc.) is automated and abstracted by the back-end.

Import File Algorithm (E)



To **describe**, as 'Open File' and 'New File' are essentially the same with the only difference being the mode in which 'File Explorer' is used (though this is independent of the rest of the flowchart as other processing begins after the finalised selection of the file), the **justification** for this algorithm is identical to that discussed above. As File Explorer is self-contained and prevents files that do not exist on the computing device from being opened, there is no extra validation in checking for existence required, thus it is **complete** in its current form.

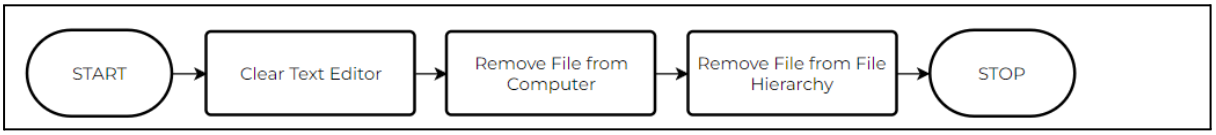
Save File Algorithm (F)



The **justification** of this algorithm is justified as file-saving is a stakeholder requirement but more generally, the ability to easily save work and store it digitally is a USP for this project. It is **complete** as it accounts for both the cases of the file already existing and not existing on the device (in the case of a file being deleted mid-edit). To **describe**, it will also use File Explorer for the 'Save As' functionality as it is well-reputed and reliable, while considering the device interactions discussed in the 'Definition of Solution Structure' section. Considering data structures, in storing the file

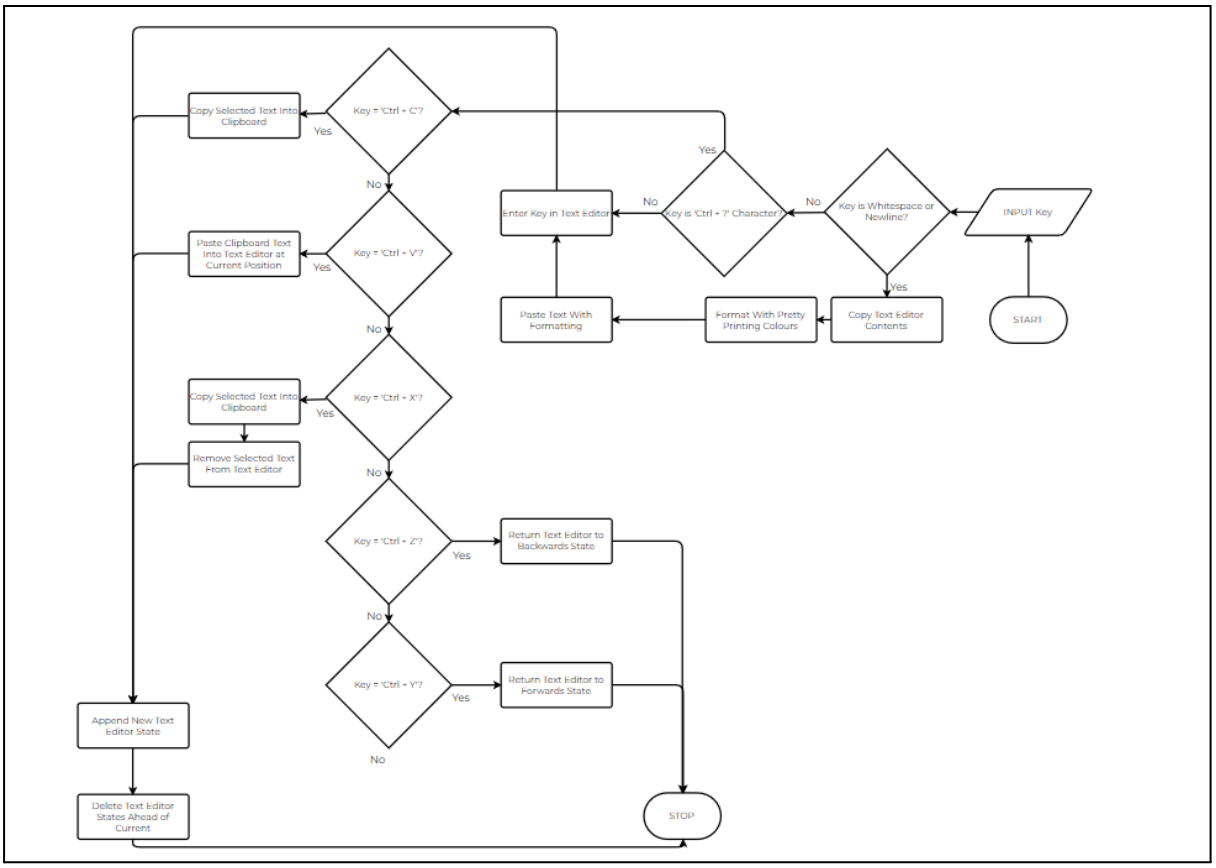
information for this algorithm and for much of the other file-handling algorithms, we may create a file class storing name, location, etc., though this will be detailed later on.

Delete File Algorithm (G)



This is a very simple algorithm yet is included for completeness. To **justify**, is necessary as file-deletion is a stakeholder requirement, being part of the 'Success Criteria'. To **describe**, it is **complete** as it completes all connections established in the 'Logical Structure' section, being the clearing of the text editor, the external connection to the device, and the deletion of the filename from the file hierarchy. Naturally, it is self-contained, only being connected to the flowchart by the file-deletion interaction as with a majority of the algorithms discussed prior, which is **justifiably** with purpose for modularity and decomposition.

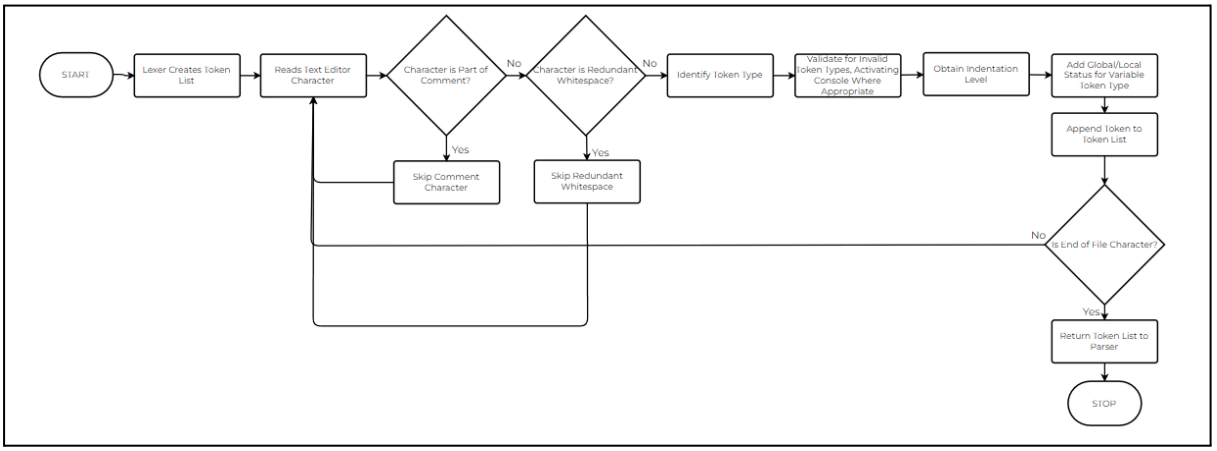
Text Editing Algorithm (H)



To **describe**, this algorithm works on a character-by-character basis, recording one keypress and depending on which key it is, determining what to do. No matter what character, a text editor state list, as created during initialisation, will be updated (this allows undoing to work by indexing the final element in this list in the manner above). If the character is of a 'Ctrl + _' variety, the functionality is also described above, in which the clipboard is also mentioned. The clipboard will hold text copied and cut, and will be necessary in implementing the paste functionality (by writing to the selected position with clipboard contents). The pretty-printing functionality will also be implemented here (not entirely separate as it is inexplicably linked to the text-editing process naturally) by using a

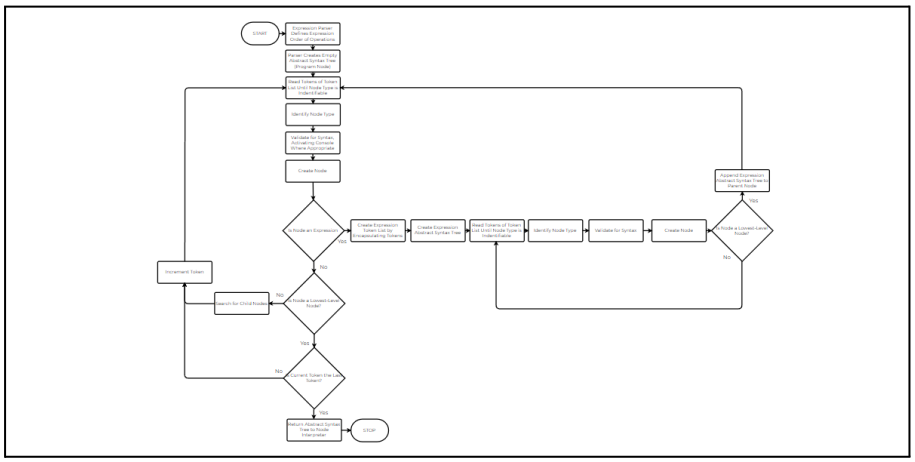
whitespace-triggered system (by limiting this relatively time-consuming process to only when it is necessary, which is a when a new word is formed, we optimise for time, thus improving usability). Obviously, multiple keys are likely to be pressed in sequence (e.g. typing a word), and yet a loop structure is not found within the algorithm. This is because there is an overarching loop structure for the whole program which can be seen when looking at algorithm B, and this should be interpreted as being in effect here. Overall, the necessity of the text-editing algorithm and pretty-printing is **justified** by their being in the 'Success Criteria', with the text-editing in particular being integral to an IDE. The use of the 'Ctrl + _' key presses is not explicitly stated in the 'Success Criteria' due to the stakeholders not mentioning it, though these are very common features in most text-editing tools and is likely to be expected by users. As such, their implementation is important for **completeness**.

Lexing Algorithm (I)



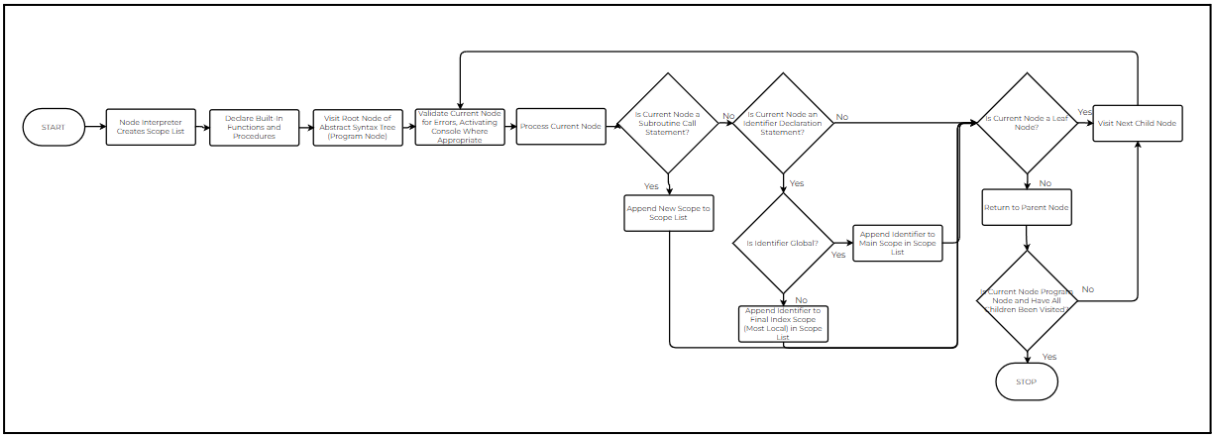
To **describe**, this algorithm reads the text editor contents character-by-character with the aim of creating a token, accumulating necessary information such as type, indentation level, and global/local status, until a complete token list is ready for parsing. This is **justified** as lexing is a useful stage before the creation of the abstract syntax tree, essentially removing extraneous information that would make later processing more tedious and overdone. It will connect to the parser directly, and is complete in its own right as it produces a simple list of all essential tokens (and has some validation with invalid token types, thus allowing for some preprocessing before parsing and node interpreting). As all token types will be accounted for and all extraneous information removed, this is **complete**.

Parsing Algorithm (J)



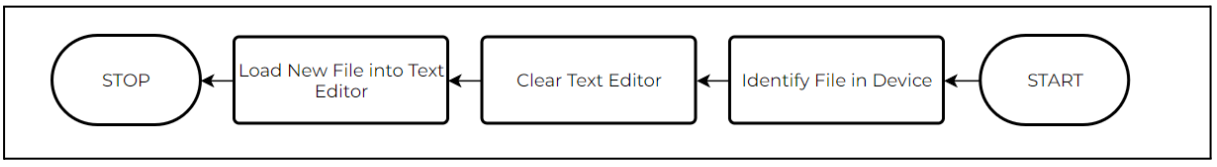
To **describe**, this flowchart encapsulates both the functionality of the main parser, which essentially attempts to create a list of statement nodes that contain further nodes existing within each other to produce a tree-like structure from which processing can be done (the nature of these nodes, including type, will be detailed later and will be based on the pseudocode documentation), and the expression parser, which works in a similar fashion, though its nodes are of a different type, consisting of arithmetic and boolean data, operator nodes, etc. This works in a loop structure, **justified** as a tree creation algorithm requires constant reference to parent and children nodes during traversal. This continues until the entire token list is exhausted. Overall, this is **complete** as all of the details for the abstract syntax tree will be produced for node interpretation.

Node Interpreting Algorithm (K)



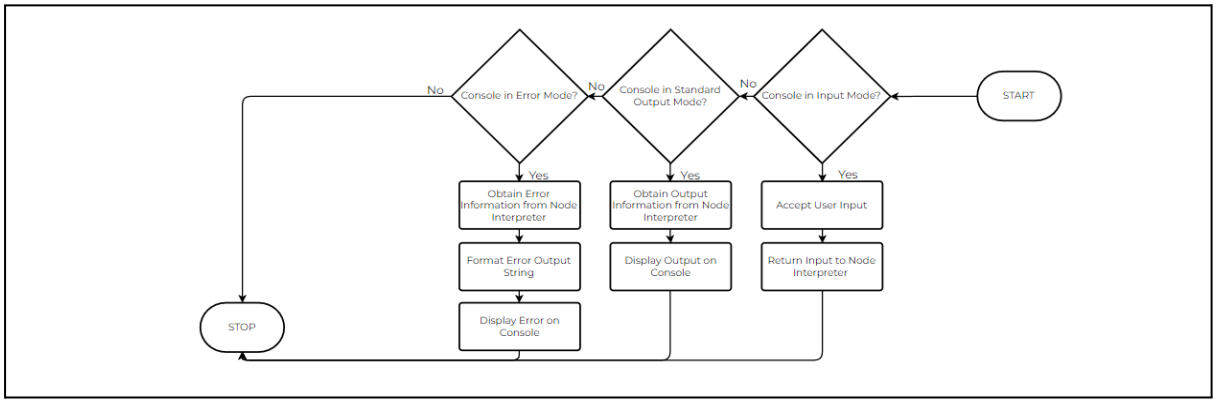
To **describe**, this algorithm takes the abstract syntax tree, uses a general visit method to initially visit the overarching program node, and completes this for each child node in a recursive fashion. As such, each node of the abstract syntax tree will be explored ensuring the entire program is executed **completely**. As such, this algorithm together with I and J complete the interpretation aspect of the solution. There is validation support for syntax and semantic errors as well as function calls, permitting console-interpreter interaction as required, while making explicit the notion of scope and accounting for local/global keywords. By carefully defining syntax, we ensure that this algorithm is accurate for purpose, and so we shall reflect OCR's documentation as closely as possible. There is also an overarching loop structure, **justified** as there is a need to visit multiple nodes with support for parent/child traversals as it is ultimately a tree.

Filename Click Algorithm (L)



To **describe**, this algorithm identifies the file using information stored within a self-made File object, clears the text editor and reloads it with the contents of the new file after reading. It is **justified** as it is a part of the success criteria, largely because it supports the complexity of having multiple files handled simultaneously, a stakeholder preference. It is also **complete** as it communicates with the text editor as established in the 'Logical Structure' section.

Console Algorithm (M)



The algorithm is **complete** as it accounts for all three console modes (input, output, and error messaging), (to **describe**) accepting either interpreter or user input depending on the mode and returning to the display or the interpreter. It will be used for both mid- and post-interpretation due to its input/output nature. The necessity of this algorithm is **justified** as a console is a stakeholder requirement and is generally an integral part of an IDE, especially as programs require input/output capabilities in order to be run.

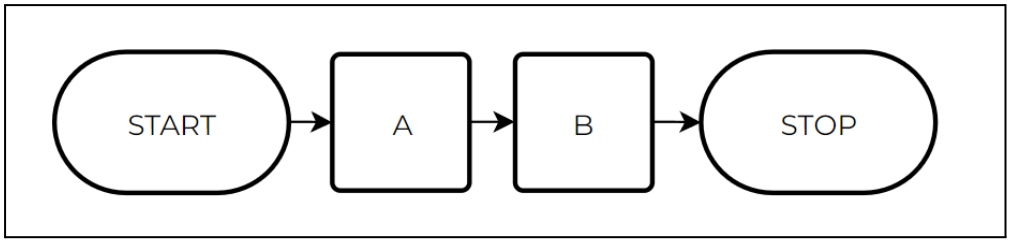
Justification of Algorithmic Completeness

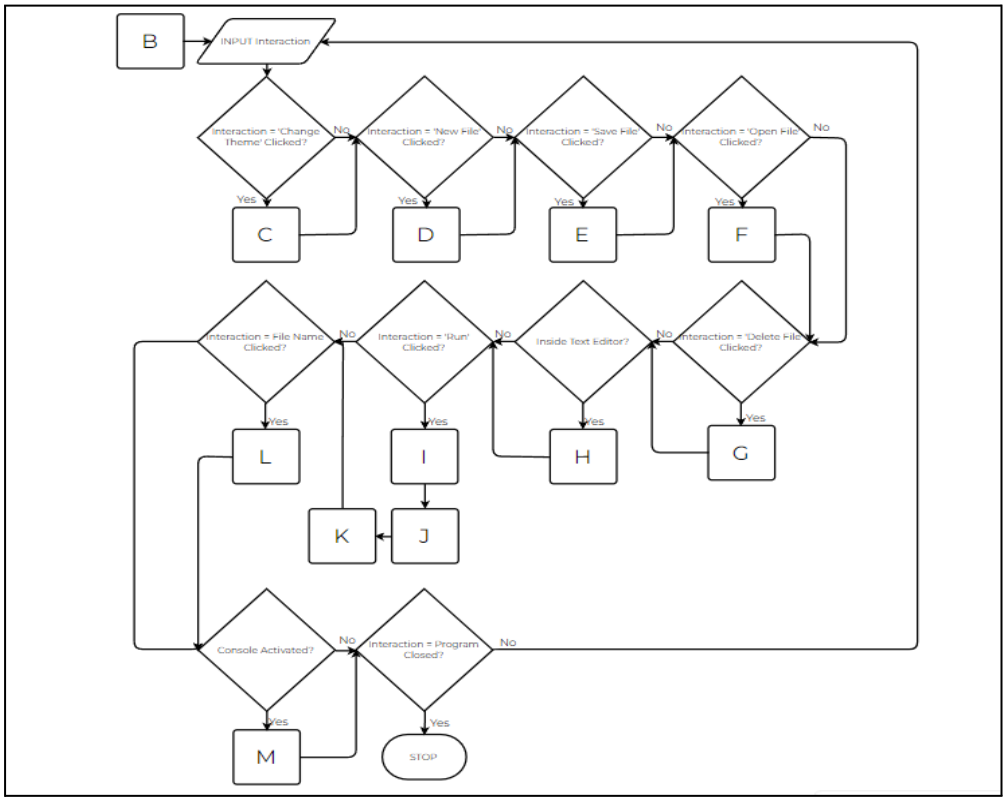
Though there are 13 algorithms in total, all forming a part of a relatively large, disparate initial problem, the breakdown mentioned above attacks each problem separately, permitting modularity, with further benefits of unit testing and less code repetition, while ensuring all aspects are satisfied. This is **justified** as the main program components, the file-handling window, console, and interpreter, have full functionality with algorithmic support, while their trigger actions are accounted for (e.g. button press, 'Ctrl + _' press, key press). This can be **justified** by looking at each logical success criteria - excluding GUI features as this has been addressed - and explaining where each criterion is evidenced.

Logical Success Criterion	**Evidence in Algorithms**
'Change Theme' button opens a drop-down of themes for 'Dark Mode' and 'Light Mode'	Drop-down with both modes in Algorithm C
File hierarchy consisting of file names (support multiple) that can be clicked to be opened	File hierarchy initialised in Algorithm A and filename clicks are supported in Algorithm L
'New File' button opens File Explorer, allowing a new text file to be created	File-creation with File Explorer supported in Algorithm D
'Import File' button opens File Explorer, allowing an existing text file to be selected	File-opening with File Explorer supported in Algorithm E
'Save File' button saves file currently opened in the text editor	File-saving for both unsaved files and saved files supported in Algorithm F
'Delete File' button deletes file currently opened in the text editor	File-deletion supported in Algorithm G
'New File' and 'Import File' only enable the creation/opening of text files	Validation for text files in both cases supported in Algorithms D and E
'Run' button executes program, updating	Interpretation supported in Algorithms I, K, and

console where applicable	K. Algorithm M supports console modes.
Pretty printing in the text editor	Pretty printing including the trigger action found in Algorithm H
Console has input/output capabilities	Algorithm M supports all console input/output features, with input/output function interpretations supported in Algorithm K
The interpreter must produce the correct output for a program consisting of any of the statement types mentioned in the OCR GCSE Pseudocode documentation	Algorithms I, J, and K showcase a complete interpretation cycle, the syntax of which will be modelled after the documentation. Therefore, it has the capacity to be correct
The interpreter produces and the console consequently outputs error messages consisting of the error type and line number	Error validation supported in algorithms I, J, and K. Algorithm M supports the console error mode, including the required error formatting

As such, we can see the algorithms fully encapsulate the aforementioned design requirements and satisfy all logical success criteria. To generalise, it is **complete** as it accounts for all interactions and works in a loop structure to ensure that interactions are continuously supported until the program is closed. For further **justification**, here are the final flowcharts showing the entire program which are complete when considered together:





Usability Features

To **describe**, one example of a usability feature is the use of large buttons with large-font text and an associated symbol. This is true for all buttons, namely 'Run', 'Change Theme', 'New File', 'Import File', 'Save File', and 'Delete File', and is **justified** as using multiple descriptors makes the button's function more recognisable, especially when there are multiple, disparate functions within the program (Run, Change Theme, etc.). The large font makes the text more readable, which can help when users have weaker eyesight, and the font chosen in the 'GUI Structure' section is standard and simple to avoid textual confusion. The buttons themselves being large facilitate the large font, while buttons are also a common medium for user interaction being intuitive to use and suitably closed in function (i.e. unlike textual inputs, there is no unnecessary space in extra validation and user ambiguity).

To **describe**, another usability feature is the window size itself. 1400 x 700 is suitably large such that a majority of the screen is taken up. This is **justified** as the interacting stakeholders preferred this approach due to it providing space for large internal features (buttons, font, etc.) for high readability. At the same time, it avoids the disadvantages of (borderless) full-screen as it still maintains space for other windows to be opened simultaneously. This is **justified** for an IDE as one may want to have the OCR documentation window open for syntax, their NEA specification for guidance, etc. There are also few primary UI elements unlike some other IDEs, limited here only to the essentials of the file-handling window, console, and text editor. This is **justified** as it provides maximum space for internal elements like the text in the text editor (large font improves readability), and minimises the presence of awkward multi-line statements (justified as the program's maintainability may suffer if lines are difficult to read). These generous proportions and the contrasting colours found within the window (as evidenced in the 'Colour Palette Structure' section) are also aesthetically pleasing, which is a usability feature in the sense that it may improve the user's coding experience, thus enabling better programming sessions. The contrast can also aid in cases of weaker eyesight, **justifying** this decision further.

To **describe**, to reduce ambiguity and thus improve usability, there will also be further descriptors. These include the primary window headers ('Files', 'Console', 'Text Editor'), which is a **justifiable** decision as those inexperienced with IDEs (possible as a large portion of the intended demographic is GCSE students) may not recognise traditional layout paradigms and be unaware of the function of some program elements (headers provide a good starting point for learning about them). To **describe** further, within the error messages themselves, there is a logical formatting for each message including the error type (and a general description) and the line number. Debugging is a valuable usability feature, **justified** as it enables students to understand where they went wrong (useful for learning, while allowing them to focus more on actual coding and problem-solving, particularly if they are in a rush to debug). The consistent formatting also improves the professionalism of the solution, **justified** if it is to be used in a semi-formal setting with teachers and examiners.

To **describe**, another usability feature is that file interactions are limited to text files. This can avoid ambiguity for users who may otherwise open other file types and find them unusable within the IDE due to a lack of pseudocode support in the documentation. To **justify** further, it enables users to write to text files outside of the IDE (if they prefer a different aesthetic environment, for example) and import the file into the IDE for the purposes of execution. As such, users have much more flexibility in where they can write pseudocode programs as opposed to being limited purely to the IDE. To add, the 'Import File' function enables older programs stored in the device (written in text files) to be executed. This compatibility with old programs is **justified** as students may have written programs before they want to be checked, examiners may want to check past consistency, etc.

To **describe**, the final main usability feature is the division of the colour themes into 'Light Mode' and 'Dark Mode'. This is **justified** as different users prefer different background colours as evidenced by the stakeholders, and this provides variety. Furthermore, 'Dark Mode' in particular can be less

harsh on the eyes, which facilitates fewer prolonged breaks (thus improving productivity in coding sessions) and supporting those with already-weaker eyesights.

Description of Usability Feature	**Justification**
Large buttons (with a large font size and an appropriate symbol attached). True for all buttons 'Run', 'Change Theme', 'New File', 'Import File', 'Save File', 'Delete File'	Multiple descriptors makes the button's function more recognisable, especially when there are multiple, disparate functions within the program. The large font makes the text more readable, which can help when users have weaker eyesight, and the font chosen in the 'GUI Structure' section is standard and simple to avoid textual confusion. The button themselves being large facilitates the large font, while buttons are also a common medium for user interaction being intuitive to use and suitably closed in function.
1400 x 700 Window Size	Suitably large such that a majority of the screen is taken up, justified as the interacting stakeholders preferred large space for larger internal features (e.g. buttons, font) for high readability, while avoiding the disadvantages of borderless full-screen as space is maintained for other windows to be opened simultaneously, useful as an OCR student may wish to open the official syntax guidelines online, for example.
Few Primary UI Elements (only the essentials of the file-handling window, console, and text editor)	Provides maximum space for internal elements like the text in the text editor (large font improves readability), and minimises the presence of awkward multi-line statements (justified as maintainability may suffer if lines are difficult to read). Generous proportions are also aesthetically pleasing, ideal in the sense that it may improve the user's coding session.
Primary Window Headers ('Files', 'Console', 'Text Editor')	Those inexperienced with IDEs (possible as GCSE Computer Science may be the first interactions with programs for a student) may not recognise traditional layout paradigms and be unaware of the function of some program elements (headers provide a good starting point for learning about them).
Logical Formatting for each error message including the error type, an error message, and the line number	Enables students to understand where they went wrong (useful for learning, while allowing them to focus more on actual coding and problem-solving, particularly if they are in a rush to debug). The consistent formatting also improves the professionalism of the solution, which may be useful if it is to be used in a semi-formal setting with teachers and examiners.
File interactions are limited to text files	Avoids ambiguity for users who may otherwise open other file types and find them unusable within the IDE due to a lack of pseudocode support in the documentation. Furthermore, it enables users to write to text files outside of the

	IDE (if they prefer a different aesthetic environment, for example) and import the file into the IDE for the purposes of execution. As such, users have much more flexibility in where they can write pseudocode programs as opposed to being limited purely to the IDE. To add, the 'Import File' function enables older programs stored in the device (written in text files) to be executed. This compatibility with old programs is justified as students may have written programs before they want to be checked, examiners may want to check past consistency, etc.
Division of colour themes into 'Light Mode' and 'Dark Mode'	Justified as different users prefer different background colours as evidenced by the stakeholders, and this provides variety. Furthermore, 'Dark Mode' in particular can be less harsh on the eyes, which facilitates fewer prolonged breaks (thus improving productivity in coding sessions) and supports those with already-weaker eyesight. Furthermore, the contrast is maintained between backgrounds and foregrounds (even when theme switching), thus maximising readability, thus improving the coding experience and productivity.

Key Variables

Identification of Name	Identification of Data Type	Justification of Use
TextEditorStateList	1D, fixed-size linked list of strings containing text editor content states behaving like a queue. Default size is 20	Storage of backwards and forwards states must be maintained for 'Undo' to work. Fixed size as creating too many high-word-count copies will take a lot of space, most of which will be wasted as users are not likely to use 'Undo' indefinitely. Custom linked list as functionality, such as maintaining a fixed size, is also personalised
Clipboard	String containing recently cut/copied text from text editor	Storage of this text is necessary for 'Paste' to work, which is justified as basic text-editing operations are a stakeholder requirement
FileList	1D, fixed-size linked list of File objects representing files in the file hierarchy. Default size is 5	Storage of files is necessary for representation in the file hierarchy, especially as multiple files should be maintained simultaneously. Fixed size as only a few files will generally be needed for pseudocode projects; this reduces wasted space
ThemeSetter	Class with attributes for the colour of each item which varies with 'Light Mode' and 'Dark Mode', toggled by a custom setTheme function. Attributes are logically represented in the 'Colour Palette Structure' section	This improves modularity and reduces code repetition as all required theme features can be set here a single time and the ThemeSetter only needs to be referenced once during initialisation. Also improves readability as each colour will be associated with an equivalently named attribute, bettering maintainability
File	Class representing a text file that will be handled using the file operation buttons and found within the file hierarchy. Attributes are Name, Location	Necessary as these files will need to be stored for the aforementioned handling. Name and Location are justified as these are the attributes required for display and Save/Delete functionality respectively
Error	Class representing errors that will occur during interpretation. Attributes are Type, Message, LineNumber	Necessary as this keeps errors organised (as there may be multiple times they are required) and Type, Message, and LineNumber are required for more readable formatting

KeyPress	1-Character String representing the last key that was pressed by the user in the text editor	Necessary for the conditional functionality of 'Ctrl + _' shortcuts and white-space triggered pretty printing
TextPalette	Dictionary that contains word : colour pairs for each keyword that has a distinct colour in the text editor for pretty printing	Makes implementing pretty printing faster and the dictionary produces value with a single-line function in O(1). Also, keeping the text palette in one location reduces code repetition, thus improving maintainability
Lexer	Class responsible for reading the text editor input and creating a token list character-by-character. Attributes are Text, CurrentCharacter, and CurrentPosition	Necessary for creating the token list for parsing. Text is justified as working on a local string of the text editor input avoids extra inter-file communication. CurrentCharacter necessary for conditional token type determination and CurrentPosition necessary to determine EOF
Parser	Class responsible for reading the token list token-by-token and creating an abstract syntax tree. Attributes are TokenList, CurrentToken, and CurrentPosition	Necessary for creating the abstract syntax tree for node interpretation. Token-by-token reading provides greater local precision when creating nodes. CurrentToken necessary for validating and completing nodes. CurrentPosition necessary for determining end of list
ExpressionParser	Class responsible for reading a section of the token list in which an expression resides and breaking it up into an expression abstract syntax tree. Attributes are TokenList, CurrentToken, CurrentPosition, BracketCount, TermOrder	Standalone ExpressionParser preferred as this reduces the bulkiness of the Parser while allowing for extra, expression-specific attributes like BracketCount, which is necessary for unopened/unclosed bracket validation and TermOrder, which is used to define the order of operations
NodeInterpreter	Class responsible for executing the final logic of the code while traversing the abstract syntax tree. Attributes are ScopeList	Necessary for executing the code finally. ScopeList is justified as this enables scope functionality (including global/local) to be implemented
Token	Class representing a basic program unit (token), which includes identifiers, literals, operators, keywords, newline, brackets, periods, and commas. Attributes are Type, Value	The Parser benefits from tokens as opposed to raw text as extraneous details are removed, avoiding bulky validation processing and improving efficiency. Type is

		necessary to differentiate between different token functions. Value is necessary for further processing during node interpretation
Node	Class representing a unit of the abstract syntax tree, which includes the overarching program node, nodes for each statement type, a return node, a branch node, literal nodes, identifier nodes, and operator nodes. Attributes are dependent upon which node type is used	Necessary to create each unit of the abstract syntax tree and allows for separation based on function (e.g. if statements and for loops can be differentiated). This allows targeted processing during node interpretation, ensuring greater readability.
ScopeList	2D list of identifiers (variables, functions, and procedures), where each internal list represents a scope in the program	2D list necessary as each internal list may contain multiple identifiers. Furthermore, it may be organised such that the last index is the most local scope, which can be processed for the identification of undeclared identifier errors
Number	Class representing a final number data type for node interpretation. Will facilitate arithmetic operations, comparison operations, and type conversion as a number would. Attributes are Value	Improves modularity if all the number operations can be defined in a single place, reducing code repetition and facilitating modularity. This also makes the Node Interpreter less bulky. Value is necessary to store the actual number
Integer	Class that inherits from Number, which stores non-fractional numbers. Will facilitate the same operations as Number as well as modulo and integer division	This being a child class of the similar Number class reduces code repetition while gaining the same benefits justified above. Creating a separate class is necessary as modulo and integer division cannot be performed on floats
Float	Class that inherits from Number, which stores fractional numbers. Will facilitate the same operations as Number	This has the same justified benefits of inheritance and code repetition as the Integer justification, and is separate for the reasons of modulo and integer division
String	Class representing a final string data type for node interpretation. Will facilitate concatenation, comparison, and type conversion. Attributes are Value	Improves modularity if all the string operations can be defined in a single place, reducing code repetition and facilitating modularity. This also makes the Node Interpreter less bulky. Value is necessary to store the actual string

Boolean	Class representing a final Boolean data type for node interpretation. Will facilitate Boolean operations, comparison operations, and type conversions. Attributes are Value	Improves modularity if all the Boolean operations can be defined in a single place, reducing code repetition and facilitating modularity. This also makes the Node Interpreter less bulky. Value is necessary to store the actual Boolean
Procedure	Class representing a final procedure data type for addition to ScopeList and node interpretation. Will facilitate comparison. Attributes are Value, which is a 3-element list. The elements are Name, ParameterNameList, and StatementList	A separate Procedure identifier class is needed as there is more information than just value in the case of variables. In this case, these are the parameters and statements for later potential execution
Function	Class representing a final function data type for addition to ScopeList and node interpretation. Will facilitate comparison. Attributes are Value, which is a 3-element list. The elements are Name, ParameterNameList, and StatementList	A separate Function identifier class is needed as there is more information than just value in the case of variables. In this case, these are the parameters and statements for later potential execution

Explanation and Justification of Necessary Validation

To **explain**, validation will be required to differentiate between the 'Save' and 'Save As' file-saving routes through the program. To **justify**, this is necessary as they have different logics, as taking the 'Save As' route erroneously would lead to the opening of the File Explorer, which would require the users to take extra steps in order to save a file beyond a single button click, which would lead to inefficiency and wastage in time. Potentially, it could also open up further errors to account for such as file clashes when attempting to store an identically named file in the same location, perhaps requiring extra, unnecessary user interactions, leading to dissatisfaction. To **explain**, we shall compare for equality the Name attribute of the File object to each file in the list of files existing currently in its Location attribute. If there is a match, we shall opt for the 'Save' route, and if there isn't, we shall opt for the 'Save As' route.

To **explain**, another error may occur when the 'Delete File' button is clicked either when there is an unsaved file in the text editor (as in the case of program startup, beginning with an empty, unsaved text file) or when a file currently being edited was deleted manually (via the system, not the IDE solution) mid-edit. To **justify**, validation is required as in the first case, a clear system error would be thrown if we attempt to access a text file with no existing location, with potential further errors if the default temporary address actually contains a file with a coincidentally identical name, leading to unwanted corruption. In the second case, validation is **justified** as a system error would also be thrown if we attempt to delete a non-existent file, which could lead to unnecessary system windows being opened that the user would unwantedly deal with, leading to dissatisfaction. Furthermore, as these errors are out of the control of the IDE, the vulnerability to unexpected system errors grows. To **explain**, to avoid the first case, we shall assign the default value None to an unsaved, on-startup text file, and use a conditional to check for this case when the 'Delete File' button is clicked for non-action . To avoid the second case, we shall perform the same FileExists method described in the above paragraph (comparing the Name attributes with the names in Location). If the file does not exist, we shall not take further action in deletion.

To **explain** further, we have imposed length limits on FileList and TextEditorStateList. To **justify**, due in part to the use of large, readable font for readability purposes, enabling too many files would lead overflow in the screen space available, leading to the inaccessibility of files incurring an error only fixed by a hard-reset (which backtracks in progress and productivity for the user). **Justification** with TextEditorStateList is that this would build up largely over time (because the user is likely to type many characters and edit repeatedly), leading to high storage space taken up, which is inefficient memory-wise as simultaneously they are unlikely to undo too many times. To **explain**, we shall create a custom-linked list on which we can perform length checks (by using an internal counter). Once this exceeds the stated maximum size, we can in the first case display a warning box saying too many files are in the hierarchy (as we would not want to replace an existing file as this could overwrite the user's work) and in the second case, delete the oldest element and add this into the slot of the newest element. In the latter case, this works like a queue data structure.

To **explain**, we also need validation to ensure only text files reach the text editor. To **justify**, this is because as discussed, text files themselves are a suitable medium to write pseudocode (contains alphanumeric and symbol characters), and if other files were opened, as the IDE is custom-written, there may be errors if one of a long list of unsupported file types is opened, particularly if pseudocode functions such as file-reading on a PNG occurs (because this is unsupported in OCR's documentation). As such, to avoid potential, unexpected errors and display errors (**justified** as the text editor is only designed to support Unicode textual characters, validation is required. To **explain**, files only enter the system via the 'New File' and 'Import File' buttons. As such, whenever a file is either created or opened, before it is loaded into the text editor, the file extension will be checked. If it is a .txt, the file should be loaded successfully. If not, loading will be halted and an informational message displayed to the user to avoid confusion.

Finally, to **explain**, there are internal errors within the pseudocode itself (i.e. syntax errors, DivisionByZero, etc.). Though these are technically termed errors, their display is a part of the successful working of the program and is thus not exactly validation, but they are included for completeness. To **justify** their inclusion, the console differs in logic between normal output and error output, particularly with error formatting. As such, if an error occurs, this should be identified before communication with the console. Furthermore, there are potential issues if these errors are not identified. For example, if a DivisionByZero error is not caught by the Node Interpreter, it will be caught by the Python interpreter. We do not want errors to travel this far as otherwise, Python will halt the program entirely, which is undesirable as this may mean that work is unsaved, progress is backtracked (**justification**), etc. As such, this Type Check validation to identify errors is necessary. To **explain**, they will be handled as they arise. For example, syntax errors will be accounted for by defining syntax rules within the parser and strictly searching for these patterns (if they are not met, a syntax error will be caught). DivisionByZero obviously requires a Value Check to see if a BinaryOperatorNode has a RightNode of 0 and an Operator of '/'. UndeclaredIdentifier will be caught when searching the ScopeList each time an identifier is used. Unopened/Unclosed Bracket will be caught by performing a pre-program check (by obtaining a running total which increments and decrements, checking for negativity to signify an error) and within the ExpressionParser itself, etc.

Test Data For Iterative Development

As discussed, the program is split into primary components. Namely, these are the file-handling window, text editor, console, and interpreter. Before we detail the test data, we must determine the specific stages of the iterative development process. As the interpreter is completely standalone in terms of functionality (connected outwardly to the console, but input/output can be temporarily performed by using a standard Python console while testing), Stage 1 will be the Interpreter, with overarching prototypes of the Lexer, Parser, and Node Interpreter in that order. As the text editor is directly connected to the interpreter (providing direct input in the form of the text file contents), Stage 2 can be the Text Editor + Interpreter, barring the multi-file handling as this is not required for pretty printing implementations, etc. Next, as the file-handling window is directly connected to the

text editor (and not to the console), Stage 3 will be the File-Handling Window + Text Editor + Interpreter, where the handling of multiple files can occur (including file operations connected to the text editor). As the console is the last logical primary component and at this stage possesses all of its direct connections, Stage 4 will be the Console + File-Handling Window + Text Editor + Interpreter, where all errors identified during node interpretation will be passed on for display to the console and input features accounted for. Finally, as all of this exists within an overarching GUI, the extra buttons for activation (file-handling will have already been implemented) and general GUI features (including the colour themes) will be implemented in Stage 5. To summarise, here are the stages for iterative development, preceding the test data required for each stage:

- **Stage 1:** Interpreter
- **Stage 2:** Text Editor + Interpreter
- **Stage 3:** File-Handling Window + Text Editor + Interpreter
- **Stage 4:** Console + File-Handling Window + Text Editor + Interpreter
- **Stage 5:** Complete GUI

Stage 1 Test Data

Lexer

Test Data	**Expected Output**	**Justification**
'X=3+*/-"Hello"+True+False>3.0<2>=0<=10=="There"!=False^+(100*100)'	[IDENTIFIER, ASSIGNMENT, INT, PLUS, MULTIPLY, DIVIDE, MINUS, STRING, PLUS, BOOLEAN, PLUS, BOOLEAN, GREATER, FLOAT, LESSER, INT, GREATER EQUALITY, INT, LESSER EQUALITY, INT, EQUALITY, STRING, NON EQUALITY, BOOLEAN, POWER, PLUS, LPAREN, INT, MULTIPLY, INT, RPAREN]	Normal/Valid test data, specifically all tokens that do not require spaces to be differentiated are tested here. This is necessary to see if non-space-separated values can be detected by the Lexer correctly (as opposed to incorrectly combining characters logically)
'MOD DIV AND OR NOT if then elseif else endif switch : case default endswitch for to next while do endwhile until procedure endprocedure function endfunction return ifthen'	[MOD, DIV, AND, OR, NOT, IF, THEN, ELSEIF, ELSE, ENDIF, SWITCH, COLON, CASE, DEFAULT, ENDSWITCH, FOR, TO, NEXT, WHILE, DO, ENDWHILE, UNTIL, PROCEDURE, ENDPROCEDURE, FUNCTION, ENDFUNCTION, RETURN, IDENTIFIER]	Normal/Valid test data, specifically all tokens that do require spaces are tested here. As these are letter-containing, this is justified to ensure these keywords are not counted as identifiers. The ifthen hopes to ensure the Lexer is space-sensitive with these keywords, thus classifying it as an identifier
'\n\n\t X = 3 + 5* / - True\n + "Hello" MOD 3 DIV\n'	[NEWLINE, NEWLINE, IDENTIFIER, ASSIGNMENT, INT, PLUS, INT, MULTIPLY, DIVIDE, MINUS, BOOLEAN, NEWLINE, PLUS, STRING, MOD, INT, DIV, NEWLINE]	Normal/Valid test data, serving two purposes. One is that each newline is identified as these are important for identifying indentation errors at the parsing stage. Secondly, it ensures that space-separated tokens and non-space separated tokens are identified correctly despite coexisting in the input
'% 5 + Var'	InvalidTokenError at [LINE	Erroneous test data, necessary

	NUMBER]	to ensure invalid characters (not defined as tokens) are identified as such, throwing a custom InvalidTokenError, and that further processing stops afterwards (to avoid time wastage)
‘Variable1’	[IDENTIFIER]	Normal/Valid test data, necessary to ensure that variables with numeric characters (at the end) are correctly identified as identifiers. Also correctly creates a token list despite only having one token.
‘1Variable’	SyntaxError at [LINE NUMBER]	Erroneous test data, necessary to ensure that numbers followed by characters (as they are not keywords nor valid variable names) are considered invalid tokens, invoking an InvalidTokenError

Lexer + Parser

Test Data	**Expected Output**	**Justification**
‘Var1 = 3*3^5*2+(6/6-1)’	(VariableDeclaration: Var1, (BinOp: (BinOp: (BinOp: 3, MULT, 3), POWER, (BinOp: 5, MULT, 2)), PLUS, (BinOp: (BinOp: 6, DIVIDE, 6), MINUS, 1))	Normal/Valid test data, necessary to ensure that all arithmetic operators follow the correct order of operations. Also ensures variable declaration statements work correctly.
‘Var2 = 9 * 9 MOD 3 - 2 DIV 1’	(VariableDeclaration: Var2, (BinOp: (BinOp: (BinOp: 9, MULT, 9), MOD, 3)), MINUS, BinOp: (2, DIV, 1))	Normal/Valid test data, necessary to ensure MOD and DIV are equal in order to multiplication (and thus higher than addition and subtraction)
‘Var3 = 3 > 4+5’	(VariableDeclaration: Var3, (BinOp: 3, GREATER, BinOp: (4, PLUS, 5)))	Normal/Valid test data, necessary to ensure that inequalities/equalities have the lowest priority (hence the use of PLUS)
‘Var4 = True AND False OR 3’	(VariableDeclaration: Var4, BinOp: (True, AND, BinOp: (False, OR, 3)))	Normal/Valid test data, necessary to ensure all Boolean operators follow the correct order of operations

‘if Var10 == -1 then
  print(“Hello”)
  customProc(3)
endif’
	(IfStatement: (BinOp: Var10, EQUALITY, (UnaryOp: MINUS, 1)), [ProcedureCall: ‘print’, [‘Hello], ProcedureCall: ‘customProc’, [3]])	Normal/Valid test data, necessary to ensure if statements are supported syntactically, unary operators are identified correctly (-1), and

		that if statements support multiples lines of statements

'for count = 3 to 5+1
  Var1 = 1
next count
\n\n\n
while True
  Var2 = 2
endwhile
do
  Var3 = 3
until False'
	((ForStatement: 1, (BinOp: 5, PLUS, 1), count, [VariableDeclaration: Var1, 1]), (WhileStatement: True, [VariableDeclaration: Var2, 2]), (DoUntilStatement: False, [VariableDeclaration, Var3, 3]))	Normal/Valid test data, necessary to ensure that all loop structures work as required, in addition to ensuring that complex expressions work correctly beyond variable declaration statements. Also, the three newlines in the middle are to ensure that extra empty lines are skipped without disruption

'function Func1(par1) print(3) endfunction'	(FunctionDeclaration: Func1, [par1, par2], [ProcedureCall: 'print', 3])	Normal/Valid test data, necessary to ensure function declaration and procedure call statements are identified correctly, while ensuring brackets work for parameter enclosing (and not just expressions)

if Var1 == 1 then
  Var1 = 1
    Var2 = 2
endif
	IndentationError	Erroneous test data, necessary to ensure that inconsistent indentation is not allowed as per OCR documentation, in which case an IndentationError is thrown

'3 = Var2'	SyntaxError	Erroneous test data, necessary to ensure that variable declaration statements only follow the Variable, Assignment, Expression syntax, otherwise throwing a SyntaxError
'\tVar1 = 10'	IdentationError	Erroneous test data, necessary to ensure extra indentation is picked up on in the first line (as it is unique without previous lines)

Lexer + Parser + Node Interpreter

Test Data	**Expected Output**	**Justification**
Name = input() print("Hello + " Name)	Input will be "John Doe" 'Hello John Doe'	Normal/Valid test data, necessary to ensure that both console-interaction subroutines (and subroutines generally) work as expected

Var1 = int(input())
if Var1 == 1 then
  print("One")
elseif Var1 == 2
  print("Two")
endif
	Input will be "1"
	'One'
		Normal/Valid test data, necessary to ensure that the correct conditional branch in a selection structure is executed. Furthermore, this checks that type conversion (specifically

		string to int) works correctly
Var2 = 10 for count = 0 to 3 Var2 = Var2 + 1 next count print(Var2)	'14'	Normal/Valid test data, necessary to ensure that loop structures (specifically a for loop) work correctly, and that the indentation within a for loop is not misidentified as not being within the global scope
Var1 = 2 / 0	DivisionByZeroError	Erroneous test data, necessary to ensure that division by zero is not enabled, thus identifying a semantic error
function GetValue(par1) Var1 = 1 endfunction print(Var1)	UndeclaredVariableError	Erroneous test data, necessary to ensure that variables in the local and global scopes are allocated correctly, and that local variables are not recognised globally (scopes avoid variable clashes, improve modularity, etc.)
Var3 = 10 + "Hello"	TypeError	Erroneous test data, necessary to ensure that TypeErrors can be identified correctly, with this example being an invalid INT, PLUS, STRING expression

Stage 2 Test Data

Text Editor

Test Data	**Expected Output**	**Justification**
'Ctrl + C' on selected text in the text editor, followed by 'Ctrl + V' at a different point	Text should be duplicated at the new point, and the copied text should remain at the old point	Normal/Valid test data, necessary to ensure copy works in conjunction with paste, checking the clipboard
'Ctrl + X' on selected text in the text editor, followed by 'Ctrl + V' at a different point	Text should be duplicated at the new point, and the cut text should be gone from the old point	Normal/Valid test data, necessary to ensure cut works in conjunction with paste, checking the clipboard
'Ctrl + Z' in the text editor	Text editor should return to a previous state (based on last key press)	Normal/Valid test data, necessary to ensure that undo works, thus testing the backwards traversal of the text editor state list
'if then else name 123 for print() input() = + - for case "Hello"'	'if then else name 123 for print() input() True = + - == for case "Hello"'	Normal/Valid test data, necessary to ensure pretty printing works correctly, such that there is consistent highlighting with keywords

Stage 3 Test Data

File-Handling Window

Test Data	Expected Output	Justification
‘New File’ button clicked	Current contents of text editor cleared, except for first line number, and ***.txt appended to file hierarchy	Normal/Valid test data, necessary to ensure that the new file functionality works completely as the discussed algorithm defines
‘Import File’ button clicked with text file attempted to be opened	File explorer opened, with the name of the selected file appended to the file hierarchy and the file contents appearing in the text editor	Normal/Valid test data, necessary to ensure that the import file functionality works completely as the discussed algorithm defines
‘Import File’ button clicked, with PNG file attempted to be opened	PNG is not imported, and a warning box displayed stating that only text files can be opened	Erroneous test data, necessary to fulfil the Success Criteria of non-text files being unopenable due to their inefficiency and vulnerability
‘Save File’ button clicked with pre-saved file in the text editor	When the on-device file is opened, the contents are updated to the state when the button was clicked	Normal/Valid test data, necessary to ensure that saving a pre-saved file follows the ‘Save’ and not ‘Save As’ route
‘Save File’ button clicked with unsaved file in the text editor	File explorer opened, with a new file with identical contents to that in the text editor is stored in the input location, with ***.txt updated	Normal/Valid test data, necessary to ensure that saving an unsaved file follows the ‘Save As’ and not ‘Save’ route
‘Delete File’ button clicked with pre-saved file in the text editor	On-device version of the file in the text editor is removed from its location and file hierarchy cleared, and the text editor is cleared	Normal/Valid test data, necessary to ensure that the ‘Delete File’ function works as expected with an existing file
‘Delete File button clicked with unsaved file in the text editor	No file is deleted. A warning box is displayed stating that an unsaved file cannot be deleted, but the text editor is cleared	Erroneous test data, necessary to ensure no unwanted deletions or Python errors are thrown when attempting to delete an unsaved file
Filename in file hierarchy clicked	Contents of the aforementioned file are loaded in the text editor	Normal/Valid test data, necessary to ensure that the file hierarchy names are clickable and act as expected

Stage 4 Test Data

Console

Test Data	Expected Output	Justification
Attempt to type when console is not in Input mode	Nothing should be written to the console	Erroneous test data, necessary to ensure that the console is in read-only mode until input() activates read-write
'Name = input("Name? ") print("Hello " + Name)'	Output should be "Hello" + input to 'Name? '	Normal/Valid test data, necessary to ensure that both input() and print() work as expected
'3 = Var1'	Syntax Error: Invalid Assignment Syntax, Line Number [LINE NUMBER]	Erroneous test data, necessary to ensure that errors are formatted by Type, Message, Line Number and are outputted to the console

Stage 5 Test Data

Other GUI Features

Test Data	Expected Output	Justification
'Change Theme' button clicked, and 'Dark Mode' clicked from the dropdown when current theme is 'Light Mode'	Colour palette changes from that of 'Light Mode' to 'Dark Mode'	Normal/Valid test data, necessary to ensure that that 'Change Theme' works when the alternative theme is selected
'Run' button clicked, with text editor contents being: 'print("Hello World")'	Console outputs 'Hello World'	Normal/Valid test data, necessary to ensure that after restructuring the UI, the console-interpreter interaction still works. Also ensures that 'Run' works as a trigger action

These tests are comprehensive, **justified** as they test all of the intended functionality described in the Success Criteria and the algorithms themselves. As such, when they are fulfilled, the program from a white-box perspective will be developed entirely and can then be passed on to stakeholders for final review and general beta-testing.

Further Data For Post-Development

Post-development testing will first be a final white-box runthrough of the expected functionality of the IDE in its entirety, all done in one continuous session. This continuity is **justified** as in order to simulate a real work session, actions should be performed in series as they are interdependent. This will also provide a qualitative sense of how the program would feel for a user, which will help inform final review as user experience is vital to the overall success of a development environment. Though there may be repetitions, this is **justified** as we should ensure functionality is maintained after developing the other components. The test data, chronologically after first executing the IDE program, is shown here:

Test Data	Expected Output	Justification
Press ‘Delete File’ button	Pop-up window warning of deleting an unsaved file	Ensures that unwanted deletion of an unknown file does not occur, and that the error has not reached Python
Press ‘New File’ button, and attempt to create an empty text file	File Explorer opens in file-creation mode. Empty text file created in location	Ensures that the ‘New File’ button works as expected, creating an empty text file

Enter into text editor:
Name = “Lorem”
if Name == “Lorem” then
  print(“Ipsum”)
endif
	Name = “Lorem”
	if Name == “Lorem” then
	  print(“Ipsum”)
	endif
		Ensures pretty-printing works consistently across a valid text file with keywords, strings, etc.

Enter ‘Ctrl + C’ on “endif”, ‘Ctrl + V’ after “endif”, ‘Ctrl + Z’, ‘Ctrl + Y’, ‘Ctrl + X’ on the second “endif’, ‘Ctrl + V’ after endif, and then delete the second “endif”	After 1st ‘Ctrl + V’, extra “endif” should be produced. ‘Ctrl + Z’ removes this “endif”. ‘Ctrl + Y’ brings it back. The ‘Ctrl + X’ removes the 2nd “endif”, with ‘Ctrl + V’ bringing it back.	This ensures that all ‘Ctrl + _’ functionality works within the text editor as expected
Press ‘Run’ button	“Ipsum”	Ensures the “Run” button is connected to the interpreter and that the interpretation occurs successfully, finally connecting to the console for output
Press ‘Save File’ button	On-device file has contents updated with that of the text editor	Ensures the ‘Save File’ button correctly follows the ‘Save’ functionality route with an already-existing file, rewriting its contents
Close Program	Program window closes	Necessary to ensure that attempting to close the window does not throw an unexpected error. Also, sets up for the next few tests
Press ‘Import File’ button, opening the file worked on just prior	Opens File Explorer in file-opening, then loading the text editor into the state it was in prior	Necessary to ensure the ‘Import File’ functionality works correctly and that the previously saved file was saved in an error-free format
Press ‘Delete File’ button	Text editor content clears and on-device file is non-existent	Necessary to ensure file-deletion works correctly for a pre-saved file

This white-box testing is sufficiently thorough as it completes all the interactions, including file-handling operations (creation, importing, saving, and deleting), text editor functionality (pretty printing, ‘Ctrl + _’ character presses), and interpreter-console interactions (print, in this case) as one may expect in a coding session, so no further post-development testing can be done with interactions. Previous interpreter/console functionality was tested during Stage 1 and Stage 4 in the iterative cycle. As such, the IDE from an empirical point of view works as per Success Criteria and is therefore **complete,** providing complete **justification**. Finally, for robustness, we shall also input larger text editor inputs to test the interpreter’s ability to handle multiple variables, constructs, etc.,

Test Data	Expected Output	Justification

Age = int(input("Age: "))
if Age >= 90 then
  print("Age is 90+")
elseif Age >= 80 then
  print("Age is 80-90")
else
  print("Age is less than 80")
endif
	Input will be 85
	'Age is 80-90'
		Useful to determine the successfulness of type conversions, the input function, variable declaration, multi-branch if statements (including comparison operators), and the print function

Number = 0
for Count = 1 to 4
  Number = Number + Count
next
print(Number)
while Number > 7
  Number = Number - 1
  print(Number)
endwhile
do
  Number = Number + 10
  print(Number)
until Number > 30
	10
	9
	8
	7
	17
	27
	37
		Useful to determine the successfulness of all loop constructs (for loops, while loops, do until loops), and multi-line indented blocks (in the while loop)

function Func1(par1)
  return par1
endfunction
print(Func1(1))
procedure Proc1(par1)
  print(par1)
endprocedure
Proc1(9)
	3
	7
		Useful to determine the successfulness of function declaration and functional calls, and procedure declaration and procedure calls

Data = "A"
switch Data:
  case "A":
  print("Data is A")
  case "B":
  print("Data is B"
  default:
  print("Data is not A or B")
endswitch
	"Data is A"	Useful to determine the successfulness of switch statements and comparison operators for strings

Str1 = "Hello" print(Str1.length) Num1 = 10 print(Num1 MOD 3) print(Num1 DIV 3)	5 "ell" 1 3	Useful to determine the successfulness of comments (not being processed), all built-in functions (length), and MOD and DIV as unique arithmetic operators (due to their being letters)

This is a comprehensive white-box test of the interpreter, **justified** because, when considered together with Stage 1 testing, all program features have been tested. Specifically, all loop structures, the use of scopes, all conditional branching structures, functions and procedures in terms of declaration and calling, variable declaration, pre-written functions and procedures, operators, and type conversions. If all of this is met, then we have **completely** fulfilled the success criteria with sufficient evidence for final evaluation, providing **complete justification**.

Iterative Development and Testing

Stage 1: Interpreter

Prototype 1: Lexer

Firstly, we must define the token types. This is **related to 'Analysis'** by the Success Criterion of requiring the correct output for any given combination of program statements as per OCR's documentation (tokens are a necessary part of the interpreter as they define essential program units, thus abstracting the problem for parsing and future processing, thus required to produce the correct output), exemplified in **Problem Breakdown** by the 'Define Token Types' process in Algorithm A. To **explain**, I shall create a class called TokenTypeContainer, which consists of attributes storing each token type as a string established during the __init__ function capitalised to reinforce their function as constants, while potentially enabling flexibility in valid tokens for different use cases. This is **justified** as it organises tokens in a single location promoting modularity while preventing accidental overwriting due to IDE autocompletion, thus reducing the likelihood of errors. It also avoids global variables, thus promoting modularity.

```
# Class Containing Definition of Token Types With Consistent Variable Names
# Ideal to Organise Tokens for Future Token Addition While Preventing Accidental Overwriting
# Attributes: Token Types Listed Separately for Organisation, Grouped By Function, Commented By Use Cases
class TokenTypeContainer:
    def __init__(self):
        self.IDENTIFIER = 'IDENTIFIER' # For Any Non-String-Enclosed, Non-Keyword, Letter-Starting Word
        self.INTEGERLITERAL = 'INT' # For Non-String Enclosed Numerics Without Periods
        self.STRINGLITERAL = 'STRING' # For Double-Quote Enclosed Groups of Characters
        self.FLOATLITERAL = 'FLOAT' # For Non-String Enclosed Numerics With Periods
        self.BOOLEANLITERAL = 'BOOL' # For Non-String Enclosed Values in [True, False]
        self.PLUS, self.MINUS = 'PLUS', 'MINUS' # For Non-String Enclosed Values in [+, -] respectively
        self.MULTIPLY, self.DIVIDE = 'MULT', 'DIVIDE' # For Non-String Enclosed Values in [*, /] respectively
        self.MOD, self.DIV = 'MOD', 'DIV' # For Non-String Enclosed Values in [MOD, DIV] respectively
        self.LPAREN, self.RPAREN = 'LPAREN', 'RPAREN' # For Non-String Enclosed Values in [(, )] respectively
        self.POWER = 'POW' # For Non String Enclosed Value <^>
        self.AND, self.OR, self.NOT = 'AND', 'OR', 'NOT' # For Non-String Values in [AND, OR, NOT] respectively
        self.ASSIGNMENT = 'ASSIGNMENT' # For Non-String Enclosed Value <=>
        self.EQUALITY, self.NONEQUALITY = 'EQUALITY', 'NONEQUALITY' # For Non-String Values in [==, !=]
        self.LESSER, self.GREATER = 'LESSER', 'GREATER' # For Non-String Values in [<, >] respectively
        self.LESSEREQUALITY = 'LESSEREQUALITY' # For Non-String Value [<=]
        self.GREATEREQUALITY = 'GREATEREQUALITY' # For Non-String Value [>=]
        self.IF, self.ELSE, self.THEN = 'IF', 'ELSE', 'THEN' # For Non-String Values in [if, then] respectively
        self.ELSEIF, self.ENDIF = 'ELSEIF', 'ENDIF' # For Non-String Values [elseif, endif] respectively
        self.SWITCH = 'SWITCH' # For Non-String Value <switch>
        self.COLON = 'COLON' # For Non-String Value <:>
        self.CASE = 'CASE' # For Non-String Value <case>
        self.DEFAULT = 'DEFAULT' # For Non-String Value <default>
        self.ENDSWITCH = 'ENDSWITCH' # For Non-String Value <endswitch>
        self.FOR, self.TO, self.NEXT = 'FOR', 'TO', 'NEXT' # For Non-String Values in [for, to, next]
        self.WHILE, self.ENDWHILE = 'WHILE', 'ENDWHILE' # For Non-String Values [while, endwhile]
        self.DO, self.UNTIL = 'DO', 'UNTIL' # For Non-String Values [do, until]
        self.PROCEDURE = 'PROCEDURE' # For Non-String Value <procedure>
        self.ENDPROCEDURE = 'ENDPROCEDURE' # For Non-String Value <endprocedure>
        self.FUNCTION = 'FUNCTION' # For Non-String Value <function>
        self.ENDFUNCTION = 'ENDFUNCTION' # For Non-String Value <endfunction>
        self.RETURN = 'RETURN' # For Non-String Value <return>
        self.NEWLINE = 'NEWLINE' # For Actual Newline Character With Value <\n>
```

Next, we shall implement Errors, which signify primarily a mistake in the program (text editor input). This is **related to 'Analysis'** by the Success Criterion of the interpreter producing correct error messages for invalid text editor inputs, and Errors will encapsulate these outputs. This is **related to Problem Breakdown** by each validation box in Algorithms I, J, and K (e.g. validate for token types), as these will be required when alternative invalid inputs are found. To **explain**, I shall create an Error class, with attributes type (the form of error mentioning the case of its arising, defaulted to 'UnknownError'), message (a more specific description of what the exact error is, defaulted to None - remedial action, for example), and lineNumber (line of text editor where error arose). To **justify**, the class enables encapsulation of an Error, making the code more readable, promoting modularity. Type is justifiably defaulted to 'UnknownError' as this avoids untyped errors during development,

thus avoiding Python exceptions. Message is justifiably defaulted to None as some errors do not have messages (e.g. self-explanatory or unknown). LineNumber is justifiable as debugging is aided if the location is given, thus enabling more focused development time.

```
# Class Enclosing an Error Object, Neceesary for the Eventual Throwing of Syntax/Semantic Errors When Found
# Will Be Called By returning Error(type, message, lineNumber) at any stage of interpretation
# Attributes: Type (The Form of Error Describing the Cause of its Arising), Message (A More Specific Description
# of What the Exact Error is, Perhaps With Remedial Actions), and LineNumber (Line of Text Editor Where Error Arose)
class Error:
    # Initialises the Token with Parameters Identical to the Class Attributes Described Above
    def __init__(self, lineNumber, type = 'UnknownError', message = None):
        self.statementList = []
        self.type = type # Default Value is Unknown Error, Used for Catch-Alls Where an Exact Error is Not Identifiable
        self.message = message # Default Value is None, for When the Error is Self-Explanatory or Unknown
        self.lineNumber = lineNumber # Not Assigned a Default Value as all Errors Should Have Line Numbers, So it Must be Assigned
    # String Representation of Error when Printed
    def __repr__(self):
        if self.message is None: # Validation: Avoids Meaninglessly Printing None to the Console When a Message is Lacking
            return f'{self.type} at Line Number {self.lineNumber}' # in an Error
        return f'{self.type} at Line Number {self.lineNumber}: {self.message}' # Returns Only if a Message is Present in the Error
```

Now that we have token types, we should define a token structure for the purposes of addition to the token list for later parsing. This is **related to 'Analysis'** also by the Success Criterion of requiring the correct output for any given combination of valid program statements, as tokens are required to abstract the problem for parsing, which is ideal, optimised processing to eventually produce the correct output. It is **related to Problem Breakdown** by the 'Lexer Creates Token List' process in Algorithm I, as the token list is eventually populated with these elements. To **explain**, I shall create a Token class, with attributes type (descriptor of function within the program) and value (the scalar for later processing, defaulted to None). To **justify**, the class enables encapsulation of the Token, such that the code becomes more readable, also promoting modularity. Type is necessary for syntax analysis (as only certain token types can follow another) and value is justifiably defaulted to None as it is not a required aspect of some tokens (e.g. the COLON token is value-less)

```
# Class Enclosing a Complete Token to be Added to the Token List by the Lexer
# Each Token Represents a Fundamental Functional Unit of the Pseudocode Program
# Attributes: type (Descriptor of Function Within the Program, Necessary for Syntax Analysis),
# value (If Applicable, the Scalar of the Token for Later Processing - Node Interpretation)
# lineNumber (Line Number in Which Token Was Found in Text Editor)
class Token:
    # Initialises the Token with Parameters Identical to the Class Attributes Described Above
    def __init__(self, type, lineNumber, value = None):
        self.type = type # Not Assigned a Default Value as all Tokens Should Have a Type, So it Must be Assigned
        self.lineNumber = lineNumber # Not Assigned a Default Value as it is a Required Attribute (All Tokens Must Have It)
        self.value = value # Default Value is None, Representing Cases Where a Token Lacks a Scalar
    # String Representation of Token (type: value) when Printed
    def __repr__(self):
        if self.value is None: # Validation: Avoids Meaninglessly Printing None to the Console When
            return f'{self.type}' # a Value is Lacking in a Token
        return f'{self.type}: {self.value}' # Returns Type: Value only if a Value Exists in the Token
```

Next, as the Lexer requires the contents of the file for character-by-character file-reading, we shall need to read the file text. This is **related to Analysis** by the requirement of the interpreter producing the correct output given valid program statements, justified by the notion of file-reading required to determine the program statements input. This is **related to Problem Breakdown** by the 'Lexer Reads Text Editor Character' process of Algorithm I, justified as the Lexer will need the entire input string before character-by-character reading. To **explain**, I created a helper function which reads the file contents, returning it in a single, continuous string, throwing a FileError if the file does not exist, taking the input of filePath, which is a string with the file's location. To **justify**, the algorithm is necessary for file-reading, and the purpose of catching the error is to prevent Python's FileNotFound error from halting the program, instead opting for a warning box approach which shall be implemented with the GUI.

```
# Helper Function Returning the Text Contents of a File in a Single String
# Will Be Called By the Lexer in Declaring its Text Attribute
# May Be Used in the Future for Further File-Handling in the File-Handling Window (Maintenance)
# Parameters: filePath (String of Location of File Including Entire Path to be Opened For Reading)
def getFileText(filePath):
    try: # Validation: Only Returns the Contents of the File if the File Exists at the Location Specified in the File Path
        with open(filePath, 'r') as file:
            return file.read() # Returns the File Contents as a Single, Continuous String (Including Special Characters)
    except FileNotFoundError: # Throws a FileError Instead of a Python FileNotFound To Prevent Auto-Closing of the Program
        return Error('N/A', 'FileError', 'File Not Found In Given Location') # Line Number Unspecified as It is a System Error
```

Next, we shall define the initialisation and traversal methods for the Lexer. This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, justified as lexer–traversing is required for character-reading and therefore token production (necessary for parsing). This is **related to Problem Breakdown** by the 'Skip Character' and 'Reads Text Editor Character' processes of Algorithm I, justified as for character identification, the ability to move between characters is necessary. To **explain**, I created a Lexer class, with attributes text (file contents), currentPosition (index of text), currentCharacter (character at index currentPosition), and currentLineNumber (line of text editor) **justified** as these are stores of the characters for conditional token selection. The methods enable both committed traversals (where currentPosition and currentCharacter change) and uncommitted traversals (or peeks, where they remain static), **justified** as there is a need to both move through the file contents to ensure all program statements are processed and immediate peeking to identify multi-character tokens and white-space enclosed tokens. To **justify** overall, the Lexer class enables modularity with the lexing stage of interpretation, facilitating unit-testing of token-identification, and character-by-character reading is necessary as tokens are character-separated (so precision is needed to differentiate between them).

```
# Class Which Reads the Token List Character-by-Character to Produce a Valid Token List for Parsing
# Attributes: text (File Contents in a Single, Continuous String, Via Helper Function), currentPosition(Index of text That is Currently
# Being Pointed at for the Purposes of Reading and Peaking), currentCharacter (Actual Character Being Pointed At, Necessary for Conditional
# Execution so That the Corresponding Token Can Be Determined Thereafter)
class Lexer:
    # Initialises the Lexer with Paramters Identical to the Class Attributes Described Above
    def __init__(self, filePath):
        self.text = '\n' + getFileText(filePath) # Future Maintenance: Can Declare text with Smaller String Inputs for Testing Purposes
        self.currentPosition = -1 # Set to -1 as advanceByCharacter Will Be Called To Begin Loop, so This Increments to (Valid) 0, The Start
        self.currentCharacter = None # Set to None to Avoid Processing Before a Character is Accessed (Future-Proofing)
        self.currentLineNumber = 1 # Set to 1 as First Character Will Be Read at Line Number 1
    # Decrements currentPosition and currentCharacter by 1 (Goes Backwards by 1)
    def decrementByCharacter(self):
        self.currentPosition -= 1
        if self.currentPosition >= 0: # Validation: Avoids Accidental Indexing of Final Characters (e.g. -1) if Decremented Too Much
            self.currentCharacter = self.text[self.currentPosition] # Sets currentCharacter to Previous Position Only if currentPosition is Non-Negative
        return Error('N/A', 'OutOfBoundsError', 'Decremented Character in Lexer Too Far Backwards') # Line Number Unspecified as It is a System Error
    # Increments currentPosition and currentCharacter by 1 (Goes Forwards by 1)
    def advanceByCharacter(self):
        self.currentPosition += 1
        if self.currentPosition >= len(self.text): # Validation: Avoids Python's IndexOutOfBounds to Prevent Program Halting
            self.currentCharacter = None # Set to None Instead of Throwing an Error as Over-Advancing May Occur Temporarily While Reading Last Character
        else:
            self.currentCharacter = self.text[self.currentPosition] # Sets currentCharacter to Next Position Only if currentPosition is Within text Length
    # Returns Next Character Without Changing the Value of currentPosition or currentCharacter, Used For Variable Multi-Character Token Selection
    def peek(self):
        if self.currentPosition >= len(self.text) - 1: # Validation: Avoids Python's IndexOutOfBounds to Prevent Program Halting
            return '' # Set to '' Instead of Throwing an Error as Over-Advancing May Occur Temporarily While Reading Last Character
        return self.text[self.currentPosition + 1] # Returns Next Character Only if currentPosition is Within text Length
    # Returns Previous Character Without Changing the Value of currentPosition or currentCharacter, Used For Whitespace-Enclosed Token Selection
    def peekBack(self):
        if self.currentPosition <= 0: # Validation: Avoids Accidental Indexing of Final Characters (e.g. -1) if Decremented Too Much
            return '' # Returns '' Instead of Throwing an Error as Under-Advancing May Occur Temporarily While Reading Last Character
        return self.text[self.currentPosition - 1] # Returns Previous Character Only if currentPosition is Within text Length
```

Next, we shall create the complete token list, the primary purpose of the Lexer. This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, **justified** as the token list is necessary for parsing, which is the pre-structuring stage to final execution. This is **related to Problem Breakdown** by Algorithm I beginning 'Read Text Editor Character' to 'Return Token List to Parser'. To **explain**, we shall read character-by-character, using the current character to identify the associated token type (and associated value, if applicable), and append it to the token list, halting successfully when the end of the text file contents is reached, and unsuccessfully is a syntax error is identified midway. To **justify**, the algorithm is necessary as tokens should be organised in an ordered fashion (as syntax analysis at parsing relies on order to check syntax rules), hence the need for character-by-character (from start to end) file-reading. The **justification** for adding currentLineNumber to each Token is for error expression during the parsing and node interpretation stages (as the text file will not be accessed then). The justification for using

helper functions for the later Tokens as seen below is to utilise modularity, enabling unit-testing of the multi-line, more complex token identification processes (thereby aiding maintenance) and to reduce the number of lines within the backbone of the token identification algorithm (to aid readability). The **justification** for the returning of errors upwards from the helper functions and beyond the Lexer is that warning box checks (for the GUI at Stage 5) will occur at high levels (e.g. at the roots of the Lexer, Parser, etc.), thus minimising repeated code.

```
# Returns a 1D List of Token Objects In Order of Their Appearance in the Program Input
def getTokenList(self):
    tokenList = [] # Begins With Empty List as No Tokens Are Read Yet
    while True: # Indefinite Iteration as Programs are of Variable Length
        self.advanceByCharacter() # Main Means by Which Next Character is Read for Next Token Determination
        if self.currentPosition >= len(self.text): # Returns the Final tokenList If End of Text is Reached
            return tokenList # Maintenance: Will Be Passed to Parser
        if self.currentCharacter in ' \t':
            pass # Ignores Unnecessary Whitespace (Linked to "Removes Unnecessary Whitespace" in Design)
        elif self.currentCharacter == '(': # Maintenance: Validation for LPAREN/RPAREN Closing Will Be Done During Parsing, Not Here
            tokenList.append(Token(TokenTypeContainer().LPAREN, self.currentLineNumber)) # No Value as LPAREN are Standalone
        elif self.currentCharacter == ')': # Maintenance: Validation for RPAREN Closing Done During Parsing, Specifically Expression Parsing
            tokenList.append(Token(TokenTypeContainer().RPAREN, self.currentLineNumber)) # No Value as RPAREN are Standalone
        elif self.currentCharacter == '^':
            tokenList.append(Token(TokenTypeContainer().POWER, self.currentLineNumber)) # No Value as Operands are Considered Separately (Maintenance: BinaryOpNode)
        elif self.currentCharacter == '*': # Future Maintenance: MULTIPLY Will Be Used for Both String Multiplication (Duplication) and Numeric Multiplication
            tokenList.append(Token(TokenTypeContainer().MULTIPLY, self.currentLineNumber)) # No Value as Operands are Considered Separately (Maintenance: BinaryOpNod
        elif self.currentCharacter == '/':
            tokenList.append(Token(TokenTypeContainer().DIVIDE, self.currentLineNumber)) # No Value as Operands are Considered Separately (Maintenance: BinaryOpNode)
        elif self.currentCharacter == '+': # Future Maintenance: PLUS Will Be Used for Both String Concatenation and Numeric Addition
            tokenList.append(Token(TokenTypeContainer().PLUS, self.currentLineNumber)) # No Value as Operands are Considered Separately (Maintenance: BinaryOpNode)
        elif self.currentCharacter == '-':
            tokenList.append(Token(TokenTypeContainer().MINUS, self.currentLineNumber)) # No Value as Operands are Considered Separately (Maintenance: BinaryOpNode)
        elif self.currentCharacter == ':':
            tokenList.append(Token(TokenTypeContainer().COLON, self.currentLineNumber)) # No Value as Colon is Just Standalone, Purely for Syntax Analysis at Parsing
        elif self.currentCharacter == '\n':
            tokenList.append(self.handleNewline()) # Call to Helper Function to Handle Newline Separately for Future Maintenance
            self.currentLineNumber += 1 # Line Number Increases by 1 Everytime a Newline is Encountered (Specifically Just After)
```

```
elif self.currentCharacter == '=':
    tokenList.append(self.handleEquals()) # Call to Helper Function as Equals May be ASSIGNMENT or Multi-Character (EQUALITY)
elif self.currentCharacter == '!':
    currentToken = self.handleExclamation() # Call to Helper Function as Exclamation May Be INVALIDTOKEN or NONEQUALITY (Multi-Character)
    if type(currentToken).__name__ == 'Error': # Validation: Returns Error Upwards if Invalid Token is Determined
        return currentToken # Maintenance: Purpose of Upwards Return is Warning Box Checks Will Be Done at Root and Lexing Should be Halted
    tokenList.append(currentToken)
elif self.currentCharacter == '<':
    tokenList.append(self.handleLesser()) # Call to Helper Function as Lesser May Be LESSER or LESSEREQUALITY (Multi-Character)
elif self.currentCharacter == '>':
    tokenList.append(self.handleGreater()) # Call to Helper Function as Greater May Be GREATER or GREATEREQUALITY (Multi-Character)
elif self.currentCharacter == '"':
    currentToken = self.handleQuote() # Call to Helper Function Due to Complexity of Variable-Length Strings + Unclosed Quote Errors
    if type(currentToken).__name__ == 'Error': # Validation: Returns Error Upwards if Unclosed Quote is Determined
        return currentToken # Maintenance: Purpose of Upwards Return is Warning Box Checks Will Be Done at Root and Lexing Should be Halted
    tokenList.append(currentToken)
elif self.currentCharacter.isnumeric() or self.currentCharacter == '.':
    currentToken = self.handleNumeric() # Call to Helper Function Due to Complexity of Variable-Length Numbers + Multiple Dot Errors
    if type(currentToken).__name__ == 'Error': # Validation: Returns Error Upwards if Multiple Dots is Determined
        return currentToken # Maintenance: Purpose of Upwards Return is Warning Box Checks Will Be Done at Root and Lexing Should be Halted
    tokenList.append(currentToken)
elif self.currentCharacter.isalpha() or self.currentCharacter == '_':
    currentToken = self.handleAlpha() # Call to Helper Function Due to Complexity of Variable-Length Identifiers + Multiple Keywords
    if type(currentToken).__name__ == 'Error': # Validation: Returns Error Upwards if Syntax Error is Determined
        return currentToken # Maintenance: Purpose of Upwards Return is Warning Box Checks Will Be Done at Root and Lexing Should be Halted
    tokenList.append(currentToken)
else:
    return Error(self.currentLineNumber, 'InvalidTokenError', 'Unrecognised Token') # Validation: Catch-All if Unrecognised Token is Found (Halts Lexing)
```

Next, we shall define the helper functions mentioned above, which create relatively complex tokens. This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, **justified** as the these complex tokens are a part of OCR's syntax in documentation, so they must be included. This is **related to Problem Breakdown** by the 'Identify Token Type' and following processes in Algorithm I, justified as these handle particular tokens post-identification. To **explain**, I created separate helper functions for each character that may produce a valid token, though the specific token/validity is dependent upon the following character. As such, the peek() method defined previously is often used here. There is limited nesting with conditionals, **justified** as these tokens (apart from numerics which use while loops justified as they are of indefinite length) are maximally two characters in length. To **justify**, these helper functions facilitate modularity within token identification in the Lexer, enabling unit-testing of different tokens for robustness, while less nesting makes the code more readable and logically easier to follow.

```
# Helper Function Which Returns a Newline Token and Updates the Position of the Position Pointer
def handleNewline(self):
    spaceCount = 0 # This is the value of a Newline Token, Equal to the Leading Whitespace of the Next Line for Indentation
    tempPosition = self.currentPosition # We Do Not Want to Update Current Position as This Will Occur Automatically in the Main Loop
    if tempPosition < len(self.text) - 1: # Validation: Shall Only Attempt to Get the Next Position if We Are Within Text Length Limits
        tempPosition += 1 # Increments the Position to Read the Next Character
    while self.text[tempPosition] == ' ': # Updates Whitespace Value While Whitespace is Found (Includes Tabbing By Python's Implementation
        spaceCount += 1 # Leading Whitespace Value Increments Each Time Leading Whitespace is Found
        tempPosition += 1 # Goes to the Index of the Next Character
        if tempPosition >= len(self.text): # Just in Case The Final Few Characters are Whitespace (In Some Text Implementations)
            break # Exits the Loop as All Leading Whitespace Has Been Found
    return Token(TokenTypeContainer().NEWLINE, self.currentLineNumber, spaceCount) # Returns the Newline Token With Value Leading Whitespac
# Helper Function Which Return an Assignment or Equality Token Depending on Following Character
def handleEquals(self):
    nextCharacter = self.peek() # Looks at Next Character, Used for Testing Token Determination
    if len(nextCharacter) == 0: # Validation: In the Case of EOF (In Some Editors)
        return Token(TokenTypeContainer().ASSIGNMENT, self.currentLineNumber) # Return Assignment as Single-Character Equals
    if nextCharacter == '=': # Maintenance: This is for Equality, But Alternative Characters Done Via Repeated If Blocks
        self.advanceByCharacter() # To Reach End of Equality, Main Loop Then Increments Past This Token
        return Token(TokenTypeContainer().EQUALITY, self.currentLineNumber) # Returns Equality As No Other Option Beyond This
    return Token(TokenTypeContainer().ASSIGNMENT, self.currentLineNumber) # Returns Assignment As Assignment is Non-Space-Enclosed
# Helper Function Which Returns a Non-Equality Token or InvalidTokenError, Depending on Following Character
def handleExclamation(self):
    nextCharacter = self.peek() # Looks at Next Character, Used for Testing Token Determination
    if len(nextCharacter) == 0: # Validation: In the Case of EOF (In Some Editors)
        return Error(self.currentLineNumber, 'InvalidTokenError', '! Unrecognised Token') # ! Alone is Unrecognised Token
    if nextCharacter == '=': # Maintenance: This is for Non-Equality, But Alternative Characters Done Via Repeated If Blocks
        self.advanceByCharacter() # To Reach End of Non-Equality, Main Loop Then Increments Past This Token
        return Token(TokenTypeContainer().NONEQUALITY, self.currentLineNumber) # Returns Non-Equality As No Other Option Beyond This
    return Error(self.currentLineNumber, 'InvalidTokenError', '! Unrecognised Token') # ! Alone is an Unrecognised Token
# Helper Function Which Returns a Lesser or LesserEquality Token, Depending on Following Character
def handleLesser(self):
    nextCharacter = self.peek() # Looks at Next Character, Used for Testing Token Determination
    if len(nextCharacter) == 0: # Validation: In the Case of EOF (In Some Editors)
        return Token(TokenTypeContainer().LESSER, self.currentLineNumber) # Return Lesser as Single-Character Lesser Than
    if nextCharacter == '=': # Maintenance: This is for LesserEquality, But Alternative Characters Done Via Repeated If Blocks
        self.advanceByCharacter() # To Reach End of LesserEquality, Main Loop Then Increments Past This Token
        return Token(TokenTypeContainer().LESSEREQUALITY, self.currentLineNumber) # Returns LesserEquality As No Other Option Beyond This
    return Token(TokenTypeContainer().LESSER, self.currentLineNumber) # Returns Lesser As Lesser is Non-Space-Enclosed
```

```
# Helper Function Which Returns a Greater or GreaterEquality Token, Depending on Following Character
def handleGreater(self):
    nextCharacter = self.peek() # Looks at Next Character, Used for Testing Token Determination
    if len(nextCharacter) == 0: # Validation: In the Case of EOF (In Some Editors)
        return Token(TokenTypeContainer().GREATER, self.currentLineNumber) # Return Greater as Single-Character Greater Than
    if nextCharacter == '=': # Maintenance: This is for GreaterEquality, But Alternative Characters Done Via Repeate
        self.advanceByCharacter() # To Reach End of GreaterEquality, Main Loop Then Increments Past This Token
        return Token(TokenTypeContainer().GREATEREQUALITY, self.currentLineNumber) # Returns GreaterEquality As No Other Option Beyond This
    return Token(TokenTypeContainer().GREATER, self.currentLineNumber) # Returns Greater As Greater is Non-Space-Enclosed
# Helper Function Which Returns a StringLiteral Token, With Value of String Contents
def handleQuote(self):
    stringValue = '' # Declares Empty String as String Concatenation Will Take Place While Traversing
    self.advanceByCharacter() # Goes to Next Character, to Check Whether the String Continues or Ends
    while self.currentCharacter != '"": # Loop Will Only Exit if a Double Quote (Signifying End of String) is Found
        if self.currentPosition >= len(self.text) or self.currentCharacter == '\n': # Validation: Checks Whether String is Unclosed By End of Line
            return Error(self.currentLineNumber, 'SyntaxError', 'Double Quotes Unclosed') # If So, A Syntax Error is Thrown as Strings are Single-Lined
        stringValue += self.currentCharacter # While Valid, Keep Adding to the Contents of the String
        self.advanceByCharacter() # Repeats Character Advancing, In Effect a Do While Loop
    return Token(TokenTypeContainer().STRINGLITERAL, self.currentLineNumber, stringValue) # When the String is Closed, Return the StringLiteral Token to Lexer
# Helper Function Which Returns an IntegerLiteral or FloatLiteral Token, Depending on the Presence of a Decimal Point
def handleNumeric(self):
    dotCount = 0 # Validation: Establishes Decimal Point Count for Token Identification and Too Many Dots Error Identification
    numberValue = '' # Holds as String as Otherwise, Leading Zeroes May Lead to Unwanted Token Creation of Mulitple Empty Numeric Tokens
    while self.currentCharacter.isnumeric() or self.currentCharacter == '.': # Only Grows numberValue if Numeric Characters (+ .) are Found
        numberValue += self.currentCharacter # Grows numberValue by Concatenating Immediate Numeric Character
        if self.currentCharacter == '.': # Checks to See Whether Decimal Point is Found
            dotCount += 1 # If So, After Prior Concatenation, Increase Dot Count to Reflect
        self.advanceByCharacter() # Goes to the Next Character to Further Check for Numeric Values (or Halting)
    self.decrementByCharacter() # Goes Backwards as While Advancing, We Went 1 Character Too Far (Would Lead to Token Skipping Due to Main Loop Advancing)
    if dotCount > 1: # Validation: Checks Whether Multiple Dots in Numeric (This is an Invalid Number Format)
        return Error(self.currentLineNumber, 'SyntaxError', 'More Than 1 Dot in Number') # Throws a Syntax Error of Too Many Dots if So
    if dotCount == 0: # No Decimal Points Signify an Integer
        return Token(TokenTypeContainer().INTEGERLITERAL, self.currentLineNumber, int(numberValue)) # Converts numberValue to INT for Arithmetic in Future
    return Token(TokenTypeContainer().FLOATLITERAL, self.currentLineNumber, float(numberValue)) # Case of 1 Decimal Point: Float, Converted for Arithmetic
```

Next, we shall define the final lexing element, handling keywords and identifiers. This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, justified as these tokens are a part of OCR's syntax in documentation, so they must be included. This is **related to Problem Breakdown** by the 'Identify Token Type' and following processes in Algorithm I, justified as these handle the post-identification processes of keywords and identifiers. To **explain**, words are collected in the beginning through a while loop (justified as words could be of variable length, thereby requiring indefinite iteration), including the preceding and following characters. This is **justified** as some words are necessarily white-space enclosed (e.g. OR, AND) or enclosed by a limited set of characters, and so this information is needed. For further **justification**, this enables partial syntax validation, which does not take place in the Parser (where most syntax analysis occurs) as these are errors that can only be identified in the Lexer (because of its direct access to the text editor, where immediate characters are searchable).

```
# Handles All Non-String Alphanumeric Words (Identifiers + Keywords)
def handleAlpha(self):
    previousCharacter = self.peekBack() # Necessary as Some Words are Syntantically Valid Only When Enclosed By Specific Characters (Back, Here)
    currentWord = '' # Empty String as Word Will Be Built by Concatenation Throughout Traversal
    while self.currentCharacter.isalpha() or self.currentCharacter == '_' or self.currentCharacter.isnumeric(): # Collects Contiguous Alphanumerics
        currentWord += self.currentCharacter # Grows currentWord by Concatenating Immediate Alphanumeric Character
        self.advanceByCharacter() # Goes to the Next Character to Further Check for Alphanumeric Values (or Halting)
    self.decrementByCharacter() # Goes Backwards as While Advancing, We Went 1 Character Too Far (Would Lead to Token Skipping Due to Main Loop Advancing)
    nextCharacter = self.peek() # Necessary For Syntax Validation of Enclosing Characters (Here, the Forward Character)
    elif currentWord == 'AND': # Case Sensitive as per OCR Documentation, Logical AND
        if previousCharacter not in ' \t' or nextCharacter not in ' \t': # Validation: Checks for Valid Enclosings for This Token
            return Error(self.currentLineNumber, 'SyntaxError', 'AND Not Whitespace-Enclosed')
        return Token(TokenTypeContainer().AND, self.currentLineNumber)
    elif currentWord == 'OR': # Case Sensitive as per OCR Documentation, Logical OR
        if previousCharacter not in ' \t' or nextCharacter not in ' \t': # Validation: Checks for Valid Enclosings for This Token
            return Error(self.currentLineNumber, 'SyntaxError', 'OR Not Whitespace-Enclosed')
        return Token(TokenTypeContainer().OR, self.currentLineNumber)
    elif currentWord == 'NOT': # Case Sensitive as per OCR Documentation, Logical NOT
        if previousCharacter not in ' (\t' or nextCharacter not in ' \t': # Validation: Checks for Valid Enclosings for This Token
            return Error(self.currentLineNumber, 'SyntaxError', 'NOT Not Whitespace-(-Enclosed')
        return Token(TokenTypeContainer().NOT, self.currentLineNumber)
    elif currentWord == 'if': # Case Sensitive as per OCR Documentation, Begins an if Statement
        if previousCharacter not in ' \t\n' or nextCharacter not in ' \t': # Validation: Checks for Valid Enclosings for This Token
            return Error(self.currentLineNumber, 'SyntaxError', 'if Not Whitespace-\n-Enclosed')
        return Token(TokenTypeContainer().IF, self.currentLineNumber)
    elif currentWord == 'then': # Case Sensitive as per OCR Documentation, Ends if and elseif Lines of if Statement
        if previousCharacter not in ' \t' or nextCharacter not in ' \t\n': # Validation: Checks for Valid Enclosings for This Token
            return Error(self.currentLineNumber, 'SyntaxError', 'then Not Whitespace-\n-Enclosed')
        return Token(TokenTypeContainer().THEN, self.currentLineNumber)
    elif currentWord == 'elseif': # Case Sensitive as per OCR Documentation, Begins Alternative Branches (Not Else) to if Statement
        if previousCharacter not in ' \t\n' or nextCharacter not in ' \t': # Validation: Checks for Valid Enclosings for This Token
            return Error(self.currentLineNumber, 'SyntaxError', 'elseif Not Whitespace-\n-Enclosed')
        return Token(TokenTypeContainer().ELSEIF, self.currentLineNumber)
```

Prototype 1 Testing

Test Data	Expected Output	Actual Output
'X=3+*/-"Hello"+True+False>3.0<2>=0<=10=="There"!=False^+(100*100)'	[IDENTIFIER, ASSIGNMENT, INT, PLUS, MULTIPLY, DIVIDE, MINUS, STRING, PLUS, BOOLEAN, PLUS, BOOLEAN, GREATER, FLOAT, LESSER, INT, GREATER EQUALITY, INT, LESSER EQUALITY, INT, EQUALITY, STRING, NON EQUALITY, BOOLEAN, POWER, PLUS, LPAREN, INT, MULTIPLY, INT, RPAREN]	[IDENTIFIER, ASSIGNMENT, INT, PLUS, MULT, DIVIDE, MINUS, STRING, PLUS, BOOLEAN, PLUS, BOOLEAN, GREATER, FLOAT, LESSER, INT, GREATER EQUALITY, INT, LESSER EQUALITY, INT, EQUALITY, STRING, NON EQUALITY, BOOLEAN, POWER, PLUS, LPAREN, INT, MULTIPLY, INT, RPAREN] ✅ `[IDENTIFIER: X, ASSIGNMENT, INT: 3, PLUS, MULT, DIVIDE, MINUS, STRING: Hello, PLUS, BOOL: True, PLUS, BOOL: False, GREATER, FLOAT: 3.0, LESSER, INT: 2, GREATEREQUALITY, INT: 0, LESSEREQUALITY, INT: 10, EQUALITY, STRING: There, NONEQUALITY, BOOL: False, POW, PLUS, LPAREN, INT: 100, MULT, INT: 100, RPAREN, NEWLINE: 0]`
'MOD DIV AND OR NOT if then elseif else endif switch : case default endswitch for to next while do endwhile until procedure endprocedure function endfunction return ifthen'	[MOD, DIV, AND, OR, NOT, IF, THEN, ELSEIF, ELSE, ENDIF, SWITCH, COLON, CASE, DEFAULT, ENDSWITCH, FOR, TO, NEXT, WHILE, DO, ENDWHILE, UNTIL, PROCEDURE, ENDPROCEDURE, FUNCTION, ENDFUNCTION, RETURN, IDENTIFIER]	[IDENTIFIER, IDENTIFIER, AND, OR, NOT, IF, THEN, ELSEIF, ELSE, ENDIF, SWITCH, COLON, CASE, DEFAULT, ENDSWITCH, FOR, TO, NEXT, WHILE, DO, ENDWHILE, UNTIL, PROCEDURE, ENDPROCEDURE, FUNCTION, ENDFUNCTION, RETURN, IDENTIFIER] ❌ `[IDENTIFIER: MOD, IDENTIFIER: DIV, AND, OR, NOT, IF, THEN, ELSEIF, ELSE, ENDIF, SWITCH, COLON, CASE, DEFAULT, ENDSWITCH, FOR, TO, NEXT, WHILE, DO, ENDWHILE, UNTIL, PROCEDURE, ENDPROCEDURE, FUNCTION, ENDFUNCTION, RETURN, IDENTIFIER: ifthen, NEWLINE: 0]`
'\n\n\t X = 3 + 5* / - True\n + "Hello" MOD 3 DIV\n'	[NEWLINE, NEWLINE, IDENTIFIER, ASSIGNMENT, INT, PLUS, INT, MULTIPLY, DIVIDE, MINUS, BOOLEAN, NEWLINE, PLUS, STRING, MOD, INT, DIV,	[NEWLINE, NEWLINE, IDENTIFIER, ASSIGNMENT, INT, PLUS, INT, MULTIPLY, DIVIDE, MINUS, BOOLEAN, NEWLINE, PLUS, STRING, MOD, INT, DIV,

	NEWLINE]	NEWLINE] ❌ `[NEWLINE: 0, NEWLINE: 2, IDENTIFIER: X, ASSIGNMENT, INT: 3, PLUS, INT: 5, MULT, DIVIDE, MINUS, BOOL: True, NEWLINE: 2, PLUS, STRING: Hello, IDENTIFIER: MOD, INT: 3, IDENTIFIER: DIV, NEWLINE: 0]`
'% 5 + Var'	InvalidTokenError at [LINE NUMBER]	InvalidTokenError at Line Number 1 ✅ `InvalidTokenError at Line Number 1: Unrecognised Token`
'Variable1'	[IDENTIFIER]	[IDENTIFIER] ✅ `[IDENTIFIER: Variable1, NEWLINE: 0]`
'1Variable'	SyntaxError at [LINE NUMBER]	SyntaxError at LineNumber 1 ✅ `SyntaxError at Line Number 1: Variable Name Cannot Begin With Number`

Prototype 1 Remedial Action

Tests 2 and 3 failed, and as observed, every instance of both MOD and DIV were replaced by IDENTIFIERs that had values MOD and DIV, which is incorrect. This occurred as during the handleAlpha() helper function of the Lexer class, MOD and DIV were not included in the items to check. As such, the action I shall take is to create two separate conditional if statements that include MOD and DIV. This is **justified** as the notion of creating separate conditionals for each separate token type is consistent with the rest of the code, improving readability, and it enables further unit-testing of singular token types.

```
if currentWord == 'MOD': # Case Sensitive as per OCR Documentation, Numeric Remainder
    if previousCharacter not in ' \t\n' or nextCharacter not in ' \t\n':
        return Error(self.currentLineNumber, 'SyntaxError', 'MOD Not Whitespace-Enclosed') # Validation: Checks for Valid Enclosings for This Token
    return Token(TokenTypeContainer().MOD, self.currentLineNumber)
elif currentWord == 'DIV': # Case Sensitive as per OCR Documentation, Numeric Integer Division
    if previousCharacter not in ' \t\n' or nextCharacter not in ' \t\n': # Validation: Checks for Valid Enclosings for This Token
        return Error(self.currentLineNumber, 'SyntaxError', 'DIV Not Whitespace-Enclosed')
    return Token(TokenTypeContainer().DIV, self.currentLineNumber)
```

For **full justification**, the previously failed tests have now been completed successfully:

'MOD DIV AND OR NOT if then elseif else endif switch : case default endswitch for to next while do endwhile until procedure endprocedure function endfunction return ifthen'	[MOD, DIV, AND, OR, NOT, IF, THEN, ELSEIF, ELSE, ENDIF, SWITCH, COLON, CASE, DEFAULT, ENDSWITCH, FOR, TO, NEXT, WHILE, DO, ENDWHILE, UNTIL, PROCEDURE, ENDPROCEDURE, FUNCTION, ENDFUNCTION, RETURN, IDENTIFIER]	[MOD, DIV, AND, OR, NOT, IF, THEN, ELSEIF, ELSE, ENDIF, SWITCH, COLON, CASE, DEFAULT, ENDSWITCH, FOR, TO, NEXT, WHILE, DO, ENDWHILE, UNTIL, PROCEDURE, ENDPROCEDURE, FUNCTION, ENDFUNCTION, RETURN, IDENTIFIER] ✅ `[MOD, DIV, AND, OR, NOT, IF, THEN, ELSEIF, ELSE, ENDIF, SWITCH, COLON, CASE, DEFAULT, ENDSWITCH, FOR, TO, NEXT, WHILE, DO, ENDWHILE, UNTIL, PROCEDURE, ENDPROCEDURE, FUNCTION, ENDFUNCTION, RETURN, IDENTIFIER: ifthen, NEWLINE: 0]`
'\n\n\t X = 3 + 5* / - True\n + "Hello" MOD 3 DIV\n'	[NEWLINE, NEWLINE, IDENTIFIER, ASSIGNMENT, INT, PLUS, INT, MULTIPLY, DIVIDE,	[NEWLINE, NEWLINE, IDENTIFIER, ASSIGNMENT, INT, PLUS, INT, MULTIPLY, DIVIDE,

	MINUS, BOOLEAN, NEWLINE, PLUS, STRING, MOD, INT, DIV, NEWLINE]	MINUS, BOOLEAN, NEWLINE, PLUS, STRING, MOD, INT, DIV, NEWLINE] ✅ [NEWLINE: 0, NEWLINE: 0, IDENTIFIER: X, ASSIGNMENT, INT: 3, PLUS, INT: 5, MULT, DIVIDE, MINUS, BOOL: True, NEWLINE: 0, PLUS, STRING: Hello, MOD, INT: 3, DIV, NEWLINE: 0]

Prototype 1 Evidence

```
C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOCR.py
[NEWLINE: 0, IDENTIFIER: Var2, ASSIGNMENT, INT: 100, PLUS, INT: 40, PLUS, INT: 30, NEWLINE: 0, IDENTIFIER: Var3, ASSIGNMENT, INT: 199, MULT, INT: 78, POW, INT: 2, NEWLINE: 0, IDENTIFIER: Var4, ASSIGNMENT, BOOL: True, NEWLINE: 0, FUNCTION, IDENTIFIER: Func1, LPAREN, IDENTIFIER: par1, RPAREN, NEWLINE: 2, RETURN, IDENTIFIER: par1, NEWLINE: 0, ENDFUNCTION, NEWLINE: 0]

C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOCR.py
SyntaxError at Line Number 2: function Not Whitespace-
-Enclosed

C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOCR.py
[NEWLINE: 0, IDENTIFIER: Var1, ASSIGNMENT, IDENTIFIER: Var2, ASSIGNMENT, IDENTIFIER: Var3, ASSIGNMENT, FUNCTION, THEN, NEWLINE: 0, INT: 90000, PLUS, INT: 90000, ASSIGNMENT, INT: 100000, NEWLINE: 0]

C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOCR.py
[NEWLINE: 0, IDENTIFIER: VariablesAreCool, ASSIGNMENT, INT: 109809, PLUS, IDENTIFIER: VariablesAreCool, NEWLINE: 0, IDENTIFIER: Milop, ASSIGNMENT, IDENTIFIER: Milop, NEWLINE: 0, FUNCTION, IDENTIFIER: Func1, LPAREN, IDENTIFIER: par1, RPAREN, NEWLINE: 2, IDENTIFIER: Var2, ASSIGNMENT, IDENTIFIER: Milop, PLUS, IDENTIFIER: VariablesAreCool, NEWLINE: 0, ENDFUNCTION, NEWLINE: 0]
```

Prototype 1 Review

At Stage 1 Prototype 1, we fully completed the Lexer, which was responsible for creating a 1D token list passed to the Parser in Prototype 2. We successfully used encapsulation in the representation of a token using the Token class with attributes currentLineNumber (for future processing as parsers will not have direct access to the text editor), value, and type (which has further been encapsulated under the TokenTypeContainer class). Therefore, in completing this stage, while having all tokens necessary for the complete syntax defined in OCR's documentation, we also maintained flexibility in a modular approach (considering classes, separate helper functions, etc.), thus making future unit-testing in post-development (and in the next few prototypes easier). We also implemented Errors, which were used successfully for validation in all necessary token identification conditionals in the Lexing stage and used further in the Interpreter and eventually in the other primary windows. To reinforce the success of this prototype, we completed robust testing (justified in the Design stage) of the Lexing stage, where all tests defined previously passed without fail.

Prototype 2: Lexer + Parser

First, we shall define the primary, overarching node to be established in the abstract syntax tree created by the Parser (essentially the root node). This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, **justified** as all parsing occurs after the root node of the AST, and AST is necessary for final execution (due to structuring). This is **related to Problem Breakdown** by the first 'Create Node' process of Algorithm J, **justified** as this will be used to declare the root node. To **explain**, I created a class ProgramNode with an attribute statementList (holds program statement nodes, declared as an empty list) and function addStatement (enables the appending of statement nodes during parsing to the statementList). To **justify**, defining ProgramNode separately in a class facilitates modularity, enabling unit-testing of this node relative to others. This is also justified as other nodes may have different functionality.

```
# Class Representing Overaching Node in AST of Main Parser
# Attributes: statementList: (1D List of Nodes, Each of Which are Complete Statement Blocks
# Such as If Statements, For Loops, Variable Declaration, etc.)
class ProgramNode:
    # Initialises a ProgramNode Object with Class Attributes Described Above
    def __init__(self):
        self.statementList = [] # Empty List as It Is Populated with Nodes Using the addStatement Helper
    # String representation of ProgramNode When Called by print()
    def __repr__(self):
        if len(self.statementList) == 0: # Validation: Checks to See Whether Program is Empty
            return Error('N/A', 'SystemError', 'Cannot Have Empty Program') # Maintenance: No Line Number as SystemError
        return f'(PROGRAM: {self.statementList})' # Only Returns Valid String if Program Contains Valid Elements
    # Helper Which Adds a Statement to the Program Upon Discovery
    def addStatement(self, statementNode):
        self.statementList.append(statementNode) # Adds to End of List as Statements are Traversed in Start->End Order
```

Next, we shall define each enclosed statement node to be potentially added to the AST (within the root program node). This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, justified as these statements are defined within the OCR documentation, and as such must be implemented. This is **related to Problem Breakdown** by the 'Identify Node Type' post-root-node, justified as separate statements are identified separately. To **explain**, I created classes for each statement node, with attributes dependent upon the particular statement type (if statements contain a branch list composed of the conditionals that satisfy the branch and its nested statements (1D list justified as elements are traversed in order, so incremental traversal of list represents if statement functionality), variable declaration statements consist of the variable name and assigned expression, with the exclusion of the assignment operator **justified** as because this is found in all assignment operations, it can be abstracted away, etc.). To **justify**, the classes facilitate modularity, enabling unit-testing of each statement node (faster debugging), while their differences (as evidenced above) can be accounted for by using classes of varying attributes.

```
# Class Representing an If Statement in the AST
# Attributes: branchList: Consists of BranchNodes in the Order They are Mentioned in the If Statement
# Will Be Used Whenever an If Statement is Found in the Token List of the Main Parser
class IfStatementNode:
    # Initialises an IfStatementNode Object with Paramters Identical to the Class Attributes Described Above
    def __init__(self):
        self.branchList = [] # Empty List as It is Populated With Nodes using the addStatement Helper Function
    # String representation of IfStatementNode When Called by print()
    def __repr__(self):
        if len(self.branchList) == 0: # Validation: Checks to See Whether If Statement is Empty
            return Error('N/A', 'SystemError', 'Cannot Have Empty If Statement') # Maintenance: No Line Number as SystemError
        return f'<IF: {self.branchList}>'
    # Helper Function Which Adds a Branch to the If Statement Upon Discovery
    def addBranch(self, branchNode):
        self.branchList.append(branchNode) # Adds to End of List as Branches are Traversed in Start->End Order

# Class Representing a Swtich Statement in the AST
# Attributes: branchList: Consists of BranchNodes in the Order They are Mentioned in the Switch Statement
# Will be Used Whenever a Switch Statement is Found in the Token List of the Main Parser
class SwitchStatementNode:
    # Initialises a SwitchStatementNode Object With Parameters Identical to the Class Attributes Descriebd Above
    def __init__(self):
        self.branchList = [] # Empty List as it is Populated with Nodes using the addStatement Helper Function
    # String representation of SwitchStatementNode When Called by print()
    def __repr__(self):
        if len(self.branchList) == 0: # Validation: Checks to See Whether Switch Statement is Empty
            return Error('N/A', 'SystemError', 'Cannot Have Empty Switch Statement') # Maintenance: No Line Number as SystemError
        return f'<SWITCH: {self.branchList}>'
    # Helper Function Which Adds a Branch to the Switch Statement Upon Discovery
    def addBranch(self, branchNode):
        self.branchList.append(branchNode) # Adds to End of List as Branches are Traversed in Start->End Order
```

Next, we shall define the elements (operators, literals, variables, and function calls) potentially used in the Expression Parser. This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, **justified** as expression elements are necessary for using complex conditionals, variable declaration, etc. (in OCR's documentation). This is **related to Problem Breakdown** by the 'Identify Node Type' process in the Expression Parser route of Algorithm J, **justified** as node identification is aided by defining expression node types prior. To **explain**, I shall create classes for each expression element, with attributes being dependent upon type (literal nodes consist of a single token attribute, justified as the tokens possess all essential properties for interpretation - line number and value specifically - function call nodes consist of a name, **justified** as this is necessary for scope validation during node interpretation, and a parameter node list, **justified** as parameters may themselves be complex expressions, requiring complex node

representation, etc.) To **justify**, the classes facilitate modularity, enabling unit-testing of each expression node separately, while being a necessary precursor to the syntax analysis of the Expression Parser itself.

```
# Class Representing a Number Literal in the AST of the Expression Parser
# Attributes: numToken (Consists of the Number Literal Token from the Lexer)
# Encapsulated When a Valid Number Literal Token is Encountered During Expression Parsing
class NumberLiteralNode:
    # Initialises a NumberLiteralNode Object with Parameters Identical to the Class Attributes Described Above
    def __init__(self, numToken):
        self.numToken = numToken # Token Used as All Important Info (Line Number, Value) is Already Encapsulated
    # String Representation of NumberLiteralNode When Called by print()
    def __repr__(self):
        return f'{self.numToken}' # Validation for Token Presence Not Needed as Only Invoked Upon Token Discovery (So It Exists)

# Class Representing a Boolean Literal in the AST of the Expression Parser
# Attributes: numToken (Consists of the Boolean Literal Token from the Lexer)
# Encapsulated When a Valid Boolean Literal Token is Encountered During Expression Parsing
class BooleanLiteralNode:
    # Initialises a BooleanLiteralNode Object with Parameters Identical to the Class Attributes Described Above
    def __init__(self, boolToken):
        self.boolToken = boolToken # Token Used as All Important Info (Line Number, Value) is Already Encapsulated
    # String representation of BooleanLiteralNode When Called by print()
    def __repr__(self):
        return f'{self.boolToken}' # Validation for Token Presence Not Needed as Only Invoked Upon Token Discovery (So It Exists)
```

Next, we shall define the initialisation and traversal methods for the Expression Parser. This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, **justified** as in order to evaluate expressions (necessary for valid processing of conditionals, assignments, etc.), we must be able to move between tokens. This is related to **Problem Breakdown** by the 'Read Tokens Until Node Type is Identifiable' process in the Expression Parser section of **Algorithm** J, **justified** as reading tokens is dependent on being able to traverse towards them. To explain, the attributes tokenList, currPos, and currToken and methods decrementToken and advanceToken are similar to those in the Lexer (with Characters), justified similarly. There is also bracketCount, which keeps track of the brackets unopened or unclosed simultaneously, and termOrder, which is a 2D tuple of token types, where higher indexes represent a higher priority in collating when processed by the Expression Parser. To **justify**, bracketCount is necessary for the validation of brackets unopened/unclosed errors (and an integer is used as the value is discrete and sign represents the binary nature of unopened/unclosed), and termOrder is necessary to enforce the order of operations as per the OCR documentation (and 2D tuples enable difference between layers, but multiple items within each priority later).

```
# Class Which Reads the Expression Token List Token-By-Token and Creates and Expression AST for the Main AST
# Attributes: tokenList (List of Tokens Encapsulating Intended Expression), currPos (Index of Current Pointer to tokenList),
# currToken (Element at currPos Index of tokenList), bracketCount (Integer, Sign Represents Left/Right Brackets, and Magnitude'
# Represents Amount Unopened/Unclosed), termOrder (2D Tuple of Operator Token Types, Where Index Represents Priority)
class ExpressionParser:
    # Initialises an ExpressionParser Object with Attributes Identical to the Class Attributes Described Above
    def __init__(self, tokenList):
        self.tokenList = tokenList # Populated List Validated in the Parser Stage
        self.currPos = -1 # Negative 1 as advanceToken() at the End Will Increase it to 0, to Begin
        self.currToken = None # Is None, Though advanceToken() Will Update it to First Element (Necessary to Validate advanceToken())
        self.bracketCount = 0 # Validation: Sign Represents Brackets Opened/Closed Currently, Magnitude -> Number Unmatched
        self.termOrder = ( # Lower Indexes are Lower Priority, Determined Via Standard Expressions and OCR Documentation
            [TokenTypeContainer().LESSER, TokenTypeContainer().LESSEREQUALITY, TokenTypeContainer().GREATER, TokenTypeContainer().GREATEREQUALITY],
            [TokenTypeContainer().OR, TokenTypeContainer().PLUS, TokenTypeContainer().MINUS],
            [TokenTypeContainer().MULTIPLY, TokenTypeContainer().DIVIDE, TokenTypeContainer().MOD, TokenTypeContainer().DIV, TokenTypeContainer().AND],
            [TokenTypeContainer().POWER]
        )
        self.advanceToken() # Necessary to Update Both currToken and currPos
    # Goes 1 Step Backwards in Both currPos and currToken
    def decrementToken(self):
        self.currPos -= 1 # Decrements currPos Regardless of Position in List, Validation Occurs After
        if self.currPos >= 0: # Validation: Avoids Accidental Indexing of Final Tokens as Opposed to the First
            self.currToken = self.tokenList[self.currPos] # Only Updates if Within the Size Limits of tokenList
    # Goes 1 Step Forwards in Both currPos and currToken
    def advanceToken(self):
        self.currPos += 1 # Increments currPos Regardless of Position in List, Validation Occurs After
        if self.currPos < len(self.tokenList): # Validation: Avoids Python IndexOutOfBounds Error (and Unwanted Halting)
            self.currToken = self.tokenList[self.currPos] # Only Updates if Within the Size Limits of tokenList
        else:
            self.currToken = None # Set to None as Opposed to Throwing an Error as Temporary Over-Advancement May Occur
```

Next, we shall create the recursive expression processing functionality of the Expression Parser. This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, **justified** as this creates the AST for the interpretation of complex expressions (necessary for conditionals, assignments, etc. in the OCR documentation). This is

related to Problem Breakdown by the Expression Parsing branch of Algorithm J, **justified** as the ExpressionParser class is the standalone implementation of breaking down complex expressions. To **explain**, parseGeneralTerm is a template function which executes recursively, traversing to higher indices of the termOrder 2D tuple, collating Operator Nodes (while differentiating between Unary and Binary Operators), with the base case of calling parseFactor, which represents the highest priority of the defined order of operations (last index). To **justify**, recursion is used as the same LHS - Operator - RHS pattern is found in BinaryOperatorNodes, a symmetry that allows near-identical processing of expression parts.

```
# Template Function to Be Used to Return Final Expression Node by Recursively Checking for Each Layer of termOrder Throughout Expression
def parseGeneralTerm(self, termIndex):
    factorFlag = False # Determines Whether Token is Fundamental (Low-Level Operand), False Initially as Processing is Top-Down
    if termIndex == len(self.termOrder) - 1: # These Operands are Highest-Priority, Thus Occupying the Final Index of termOrder
        factorFlag = True # Sets factorFlag to Execute Standalone Factor Processing (Does Not Follow General Template)
    leftNode = self.parseGeneralTerm(termIndex + 1) if not factorFlag else self.parseFactor() # Recursively Checks Next Index or Follows Non-Standard Route
    self.advanceToken() # Traverses to Next Token, Which Should Be the Operand (Validation Already Occurs Internally)
    while self.currPos < len(self.tokenList): # Only Continues Processing if tokenList is Not Fully Traversed
        if self.currToken.type not in self.termOrder[termIndex]: # If Binary Operator Does Not Exist (Unary), Processing is Complete
            self.decrementToken() # Decrements by One so Recursive Processing Includse First Token Performing Initial Advancement
            return leftNode # Returns UnaryOperatorNode if Above Conditions Satisfied
        opToken = self.currToken # Operators are Always One Token Long in OCR Documentation, So No Loops Required
        self.advanceToken() # Moves on the Next Token to Process RHS Operand
        if self.currPos >= len(self.tokenList): # Validation: If RHS Operand Does Not Exist, This is Invalid Syntax for a Binary Operation
            return Error(self.currToken.lineNumber, 'SyntaxError', 'Invalid Expression Syntax') # SyntaxError Thrown if RHS Operand Does Not Exist
        rightNode = self.parseGeneralTerm(termIndex + 1) if not factorFlag else self.parseFactor() # Performs Identical Recursive Processing (Symmetrical)
        leftNode = BinaryOpNode(leftNode, opToken, rightNode) # When LHS, Operator, RHS Structure Complete, All Transferred to leftNode (Combining)
        self.advanceToken() # Moves on to Next Token to Check for Right Parenthesis
        if self.currPos >= len(self.tokenList): # If No Right Parenthesis, Moves Back to Continue Recursive Processing
            self.decrementToken() # Moves Backwards to Begin Recursive Processing at Correct Index
            return leftNode # Returns Complete BinaryOpNode Upwards as LHS of Above Recursive Layer
        if self.currToken == TokenTypeContainer().RPAREN: # Decrements as bracketCount Functionality Elsewhere
            self.decrementToken() # Moves Backwards to Begin Recursive Processing at Correct Index
            return leftNode # Returns Complete BinaryOpNode Upwards as LHS of Above Recursive Layer
    self.decrementToken() # Moves Backwards to Begin Recursive Processing at Correct Index
    return leftNode # Returns Complete BinaryOpNode Upwards as LHS of Above Recursive Layer
```

Next, we shall implement the main Parser, which creates the Abstract Syntax Tree for input to the Node Interpreter. This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, **justified** as the AST organises program statements logically for straightforward execution. This is **related to Problem Breakdown** by the Main Parser branch of Algorithm J, justified as the Parser class is the standalone implementation for creating the AST (and collating nodes). To **explain**, I created a Parser class with initialisation and traversal attributes and methods identical to the Expression Parser (except the expression-specific attributes bracketCount and termOrder), **justified** as they both traverse identically structured token lists. The parsing algorithm for each statement works by first identifying the statement type (via unique keywords, such as 'if' triggering the parsing of an 'if statement', etc.) and then following the syntax rules for that statement (e.g. 'if' -> 'expression' -> 'then'...) until it is complete, identifying SyntaxErrors if there are deviations from these predefined rules. To **justify**, this is achievable only because these rules are mentioned explicitly in the OCR documentation, and the breakdown of parsing statements to different functions promotes modularity, enabling unit-testing of different statement types.

```python
# Class That Creates AST, Organising Nodes Logically for Straight-Forward Execution by Node Interpreter
# Attributes: tokenList (List of Tokens Encapsulating Intended Expression), currPos (Index of Current Pointer to tokenList),
# currToken (Element at currPos Index of tokenList)
class Parser:
    # Initialises Parser Object with Attributes Identical to the Class Attributes Described Above
    def __init__(self, tokenList):
        self.tokenList = tokenList # Complete tokenList Generated from Text Editor Input as Returned by Parser
        self.currPos = -1 # advanceToken() at End Will Increase it to 0 to Begin
        self.currToken = None # advanceToken() at End Will Update it to the First Element, Necessary to Validate advanceToken()
        self.advanceToken() # Sets currPos and currToken to Correct Value (Within Limits of tokenList)
    # Decrements currPos and currToken by a Single Position
    def decrementToken(self):
        self.currPos -= 1 # Decrements currPos by a Single Position
        if self.currPos < 0: # Validation: Avoids Accidental Indexing of Final Elements When Accessing First Element
            return Error('n/a', 'SystemError', 'tokenList Indexed Too Far Backwards at Parsing') # No Line Number as SystemError
        self.currToken = self.tokenList[self.currPos] # Decrements currToken by a Single Position
    # Advances currPos and currToken by a Single Position
    def advanceToken(self):
        self.currPos += 1 # Increments currPos by a Single Position
        if self.currPos < len(self.tokenList): # Validation: Avoids Python IndexOutOfBounds Error and Halting
            self.currToken = self.tokenList[self.currPos] # Only Updates currToken if Within the Limits of the Length of tokenList
        else:
            self.currToken = None # Set to None as Temporary Over-Advancement May Occur (So No Error Thrown)
    # Builds the Overarching ProgramNode Consisting of a Collection of All Statements Within Program Ordered Logically
    def parseProgram(self):
        programNode = ProgramNode() # Creates Empty ProgramNode (Filling Occurs Post-Creation)
        while self.currPos < len(self.tokenList): # Continues Updating programNode With Statements Only Until End of tokenList
            while self.currToken.type == TokenTypeContainer().NEWLINE: # Skips Extra Newline Characters Where Statements Are Not Found
                self.advanceToken()
                if self.currPos >= len(self.tokenList): # If End of tokenList is Reached, programNode is Complete
                    return programNode # Returns programNode, Containing Complete AST, for Node Interpretation
            self.decrementToken() # Goes to Newline Token of Previous Line as This Has Following Whitespace (Indentation Testing)
            if self.currToken.value != 0: # Validation: All Beginnings of Statements (Only Checks First Line, Not Including Nesting)
                return Error(self.currToken.lineNumber + 1, 'IndentationError', 'Indentation for Statement Start != 0') # Line Number + 1 as Previous Line
            self.advanceToken() # Advances to Beginning of Next Line for Parsing of Next Statement
            statementNode = self.parseStatement() # Obtains AST Branch of Statement for Appending to programNode
            if type(statementNode).__name__ == 'Error': # Validation: Checks to See Whether an Error is Found While Parsing Internal Statement
                return statementNode # Returns Error Upwards to Root For Error Output to Console
            programNode.addStatement(statementNode) # Adds Statement to programNode
        return programNode
```

Prototype 2 Testing

Test Data	Expected Output	Actual Output
'Var1 = 3*3^5*2+(6/6-1)'	(VariableDeclaration: Var1, (BinOp: (BinOp: (BinOp: 3, MULT, 3), POWER, (BinOp: 5, MULT, 2)), PLUS, (BinOp: (BinOp: 6, DIVIDE, 6), MINUS, 1))	(VariableDeclaration: Var1, (BinOp: (BinOp: (BinOp: 3, MULT, 3), POWER, (BinOp: 5, MULT, 2)), PLUS, (BinOp: (BinOp: 6, DIVIDE, 6), MINUS, 1)) ✅ `(PROGRAM: [<ASSIGNMENT: IDENTIFIER: Var1 = ((INT: 3, MULT, (INT: 3, POW, INT: 5)), MULT, INT: 2), PLUS, ((INT: 6, DIVIDE, INT: 6), MINUS, INT: 1))>])`
'Var2 = 9 * 9 MOD 3 - 2 DIV 1'	(VariableDeclaration: Var2, (BinOp: (BinOp: (BinOp: 9, MULT, 9), MOD, 3)), MINUS, BinOp: (2, DIV, 1))	(VariableDeclaration: Var2, (BinOp: (BinOp: (BinOp: 9, MULT, 9), MOD, 3)), MINUS, BinOp: (2, DIV, 1)) ✅ `(PROGRAM: [<ASSIGNMENT: IDENTIFIER: Var2 = ((INT: 9, MULT, INT: 9), MOD, INT: 3), MINUS, (INT: 2, DIV, INT: 1))>])`
'Var3 = 3 > 4+5'	(VariableDeclaration: Var3, (BinOp: 3, GREATER, BinOp: (4, PLUS, 5)))	(VariableDeclaration: Var3, (BinOp: 3, GREATER, BinOp: (4, PLUS, 5))) ✅ `(PROGRAM: [<ASSIGNMENT: IDENTIFIER: Var3 = (INT: 3, GREATER, (INT: 4, PLUS, INT: 5))>])`
'Var4 = True AND False OR 3'	(VariableDeclaration: Var4, BinOp: (True, AND, BinOp: (False, OR, 3)))	(VariableDeclaration: Var4, BinOp: (True, AND, BinOp: (False, OR, 3))) ✅ `(PROGRAM: [<ASSIGNMENT: IDENTIFIER: Var4 = (BOOL: True, AND, BOOL: False), OR, INT: 3)>])`
'if Var10 == -1 then	(IfStatement: (BinOp: Var10,	(IfStatement: (BinOp: Var10,

print("Hello") customProc(3) endif'	EQUALITY, (UnaryOp: MINUS, 1)), [ProcedureCall: 'print', ['Hello], ProcedureCall: 'customProc', [3]])	EQUALITY, (UnaryOp: MINUS, 1)), [ProcedureCall: 'print', ['Hello], ProcedureCall: 'customProc', [3]]) ✅ `(PROGRAM: [<IF: [(BRANCH: (IDENTIFIER: Var10 , EQUALITY, (MINUS, INT: 1)) -> [<PROCEDURE: print; PARAMS: [STRING: Hello]>, <PROCEDURE : customProc; PARAMS: [INT: 3]>]>])]>])`
'for count = 3 to 5+1 Var1 = 1 next count \n\n\n while True Var2 = 2 endwhile do Var3 = 3 until False'	((ForStatement: 1, (BinOp: 5, PLUS, 1), count, [VariableDeclaration: Var1, 1]), (WhileStatement: True, [VariableDeclaration: Var2, 2]), (DoUntilStatement: False, [VariableDeclaration, Var3, 3]))	((ForStatement: 1, (BinOp: 5, PLUS, 1), count, [VariableDeclaration: Var1, 1]), (WhileStatement: True, [VariableDeclaration: Var2, 2]), (DoUntilStatement: False, [VariableDeclaration, Var3, 3])) ✅ `(PROGRAM: [<FOR: ITER: INT: 0 - INT: 1; STAT EMENTS: [<ASSIGNMENT: IDENTIFIER: Var2 = BOO L: False>]>])`
'function Func1(par1) print(3) endfunction'	(FunctionDeclaration: Func1, [par1, par2], [ProcedureCall: 'print', 3])	(FunctionDeclaration: Func1, [par1, par2], [ProcedureCall: 'print', 3]) ✅ `(PROGRAM: [<FUNC DECLARE: Func1; PARAMS: [ID ENTIFIER: par1]; STATEMENTS: [<PROCEDURE: pr int; PARAMS: [INT: 3]>]>])`
'if Var1 == 1 then Var1 = 1 Var2 = 2 endif'	IndentationError at [LINE NUMBER]	IndentationError at Line Number 2 ✅ `IndentationError at Line Number 2: Invalid I ndentation`
'\tVar1 = 10'	IdentationError at [LINE NUMBER]	(VariableDeclaration: Var1, 1) ❌ `(PROGRAM: [<ASSIGNMENT: IDENTIFIER: Var1 = I NT: 10>])`

Prototype 2 Remedial Action

The final test failed as it ignored the Indentation Error and considered the statement valid, creating a complete Abstract Syntax Tree for a normal assignment statement. To fix this error, I shall add a dummy newline character to the text editor input to the Lexer (at the beginning). To **justify**, clearly the error is limited to not noticing Indentation Errors for the first line (as evidenced by the second-last test being successful), and this may be attributed to the standard newline check not occurring for this line (as it does not have a previous line with a newline token containing a following whitespace value, thus not triggering the check). As such, by adding a dummy newline to the beginning, this token will be considered and the check triggered, while not changing the functionality of the program (as extra newlines are skipped during parsing).

```
# Initialises the Lexer with Paramters Identical to the Class Attributes Described Above
def __init__(self, filePath):
    self.text = '\n' + getFileText(filePath) # Future Maintenance: Can Declare text with Smaller String Inputs for Testing Purposes
```

For **full justification**, the previously failed test has now been completed successfully:

'\tVar1 = 10'	IdentationError at [LINE NUMBER]	IndentationError at Line Number 1 ✅

		IndentationError at Line Number 1: Invalid I ndentation

Prototype 2 Evidence

```
C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOCR.py
(PROGRAM: [<PROCEDURE: print; PARAMS: [INT: 1]>, <PROCEDURE: print; PARAMS: [INT: 2]>, <PROCEDURE: print; PARAMS: [STRING: Hello]>])

C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOCR.py
(PROGRAM: [<IF: [(BRANCH: (INT: 3, EQUALITY, INT: 3) -> [<PROCEDURE: print; PARAMS: [INT: 100]>])]>])

C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOCR.py
(PROGRAM: [<ASSIGNMENT: IDENTIFIER: Var2 = ((INT: 100, PLUS, INT: 40), PLUS, INT: 30)>, <ASSIGNMENT: IDENTIFIER: Var3 = (INT: 199, MULT, (INT: 78, POW,
 INT: 2))>, <ASSIGNMENT: IDENTIFIER: Var4 = BOOL: True>])

C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOCR.py
(PROGRAM: [<ASSIGNMENT: IDENTIFIER: Var2 = ((INT: 100, PLUS, INT: 40), PLUS, INT: 30)>, <ASSIGNMENT: IDENTIFIER: Var3 = (INT: 199, MULT, (INT: 78, POW,
 INT: 2))>, <ASSIGNMENT: IDENTIFIER: Var4 = BOOL: True>, <FUNC DECLARE: Func1; PARAMS: [IDENTIFIER: par1]; STATEMENTS: [<RETURN: IDENTIFIER: par1>]>])
```

Prototype 2 Review

At Stage 1 Prototype 2, the Parser was created, which converted the token list from the Lexer into a logically organised Abstract Syntax Tree for node interpretation. Modularity was maintained in having a standalone main Parser class, with separate methods for parsing different statement types, thus enabling the unit-testing of statements (as evidenced in testing). Furthermore, statement types were also stored in standalone nodes with particular attributes, which helped during node interpretation as the expected functionality was named accordingly. A separate class for the Expression Parser also was made, in which the complexity of handling an order of operations between different operation types - Boolean, arithmetic, String - was abstracted by defining order in terms of a 2D tuple and using recursion (considering the symmetry of binary operations). Overall, this prototype was successful, as the Lexer and Parser combined successfully, as evidenced in testing where all test cases (including tests for robustness in the case of returning Syntax and Indentation errors) passed successfully.

Prototype 3: Lexer + Parser + Node Interpreter

First, we shall define all data types within classes. This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, **justified** as data types help determine which operations are allowed, etc., thus allowing the validation of syntax. This is **related to Problem Breakdown** by the 'Define Data Types' process of Algorithm A, justified as there will be a class per data type as defined in OCR's documentation. To **explain**, each class (data type) will have methods defining unary/binary operations that can occur with them as operands (e.g. addTo, multiplyBy). To **justify**, classes facilitate modularity, enabling the unit-testing of operator syntax, while the breakdown into methods enables validation to occur in a readable location.

```
# Parent Class of Integer and Float Data Types (To Avoid Code Repetition)
# Attributes: value (Represents the Value of the Number, Obtained Originally at Lexing)
# Will Be Used When Returning a Number From a FunctionNode or When a (Number)LiteralNode is Found
class Number:
    # Initialises a Number Object With the Class Attributes Described Above
    def __init__(self, value):
        self.value = value # Refers to Numerical Value Specifically (Used During Operations)
    # String Representation of Number Object When print() is Called
    def __repr__(self):
        return f'{self.value}' # Only Returns the Value as This Simplifies Console Output
    # Adds Two Number Together (In BinaryOpNode With + Operator and Number on LHS)
    def addTo(self, data):
        if not isinstance(data, Integer) and not isinstance(data, Float): # Validation: Addition With a Number Can Only Happen With Another Number
            return Error('n/a', 'TypeError', f'{type(self).__name__} & {type(data).__name__} Invalid Operands For +') # Throws a TypeError for Incompatible Types
        if self.value + data.value != round(self.value + data.value): # Checks to See Whether Current Number is a Float
            return Float(self.value + data.value) # In This Case, It Returns a Float Data Type (To Include Possible Decimal Part of Addition)
        else: # Else Means That the Result is an Integer
            return Integer(self.value + data.value) # In This Case, There is No Decimal Part, so Returning an Integer is Safe and Will Not Throw a Python Error
    # Subtracts One Number From Another Number (In BinaryOpNode With - Operator and Number on LHS)
    def subtractBy(self, data):
        if not isinstance(data, Integer) and not isinstance(data, Float): # Validation: Subtraction With a Number Can Only Happen With Another Number
            return Error('n/a', 'TypeError', f'{type(self).__name__} & {type(data).__name__} Invalid Operands For -') # Throws a TypeError for Incompatible Types
        if self.value - data.value != round(self.value - data.value): # Checks to See Whether Current Number is a Float
            return Float(self.value - data.value) # In This Case, It Returns a Float Data Type (To Include Possible Decimal Part of Subtraction)
        else: # Else Means That the Result is an Integer
            return Integer(self.value - data.value) # In This Case, There is No Decimal Part, so Returning an Integer is Safe and Will Not Throw a Python Error
    # Subtracts One Number By Another Number (In BinaryOpNode With * Operator and Number on LHS)
    def multiplyBy(self, data):
        if not isinstance(data, Integer) and not isinstance(data, Float): # Validation: Multiplication With a Number Can Only Happen With Another Number
            return Error('n/a', 'TypeError', f'{type(self).__name__} & {type(data).__name__} Invalid Operands For *') # Throws a TypeError for Incompatible Types
        if self.value * data.value != round(self.value * data.value): # Checks to See Whether Current Number is a Float
            return Float(self.value * data.value) # In This Case, It Returns a Float Data Type (To Include Possible Decimal Part of Multiplication)
        else: # Else Means That the Result is an Integer
            return Integer(self.value * data.value) # In This Case, There is No Decimal Part, so Returning an Integer is Safe and Will Not Throw a Python Error
```

Next, we shall finally execute each instruction (and perform semantic analysis and the remaining syntax analysis). This is **related to 'Analysis'** by the requirement of the interpreter producing the correct output given valid program statements, **justified** as by now, statements will be completely validated and extra functionality (built-in subroutines, including the necessary-for-interaction print() and input() will be declared within the visitFunctionCallNode() method). This is **related to Problem Breakdown** by Algorithm K, hinged largely on the 'Visit Node' process. To **explain**, a general visit method is called, the node type used to determine the node-specific visit function, and the node is processed, with input-output interaction happening through validated procedure calls, for example. To **justify**, it is done on a node-by-node basis as the abstract syntax tree is created purely using nodes and as they are already organised (e.g. the statement list in the program node is in logical order), visiting in this way is a simple and readable traversal method.

```
# Class that Represents the Node Interpreter, Which Interprets the Main AST From the Parser By Visiting Lower Levels of the Tree in Logical Order
# Attributes: symbolTable (2D List of Identifiers, Highest Index Being the Most Recently Opened Scope (Within the Latest Subroutine) and First Index Being the Global Scope)
# symbolTable Prevents Identically Named Identifiers in Different Scopes Clashing
# Used Directly After the Parser Creates the Main AST
class Interpreter:
    # Initialises an Interpreter Object with the Same Class Attributes Described Above
    def __init__(self, symbolTable = []):
        self.symbolTable = symbolTable # Defaulted to an Empty List As This is Generally What it Will Be, But Can Alter for Testing Purposes (Maintenance)
    # Checks if a Given Identifier (By Name) Exists and Returns the Most Local Copy (If There Are Multiple With the Same Name), Returns None if Identifier is Not Found
    def getIdentifierValue(self, name):
        for scope in self.symbolTable[::-1]: # Reverses the symbolTable to Begin Searching From the Highest Index (Most Local Scope)
            if name in scope: # Checks if in the Current Scope (1D List), The String of the Name of the Desired Identifier Exists
                return scope[name] # If So, Returns that Node (Internal List is Actually a Dictionary as This Is O(1) Searching)
        return None # Returns None if Identifier Not Found in symbolTable
    # General Visit Method, Identifies The Type of Node Being Pointed to, and Transfers the Visiting to a Specific Visit Method for That Node
    def visit(self, node):
        if Console.cget('state') == tk.NORMAL:
            return
        methodName = f'visit{type(node).__name__}' # This Identifies Which Node Is Being Pointed at, and Therefore Determines the Name of the Node-Specific Function
        visitMethod = getattr(self, methodName, self.visitNotFound) # Allows for the Possibility of an Unknown Node Being Visited (Mainly If a Non-Node is Accidentally Visited
        return visitMethod(node) # Visits the Node Specifically (Or the Non-Node)
    # If A Non-Node is Visited, the AST Has Been Built Up Unsuccessfully
    def visitNotFound(self, node):
        return Error('n/a', 'SystemError', 'Visit Method to Non-Node Not Defined') # No Line Number as System Error
    # Visits Overarching Program Node of AST
    def visitProgramNode(self, node):
        if Console.cget('state') == tk.NORMAL:
            return
        self.symbolTable.append({}) # Appends Initially the Global Scope to symbolTable (This Visit Method is Called First, so This Will Be at Index 0)
        for statementNode in node.statementList: # Goes Through Each Program Statement in Order (Logical Processing Direction)
            retVal = self.visit(statementNode) # Checks Whether a ReturnNode is Found Within That Overall Statemen
            if retVal is not None: # In This Case, It Should Not Be as a Function is Never a Standalone Statement (In The Way Defined in the Parser)
                return Error('n/a', 'SystemError', 'Return Statement Cannot Exist Outside Of A Function') # No Line Number as SystemError
        self.symbolTable.pop() # Clears the Global Scope After All Statements Are Visited (Processing Complete) (Garbage Collection) (System Maintenance)
```

Prototype 3 Testing

Test Data	Expected Output	Justification

Name = input()
print("Hello " + Name)
	Input will be "John Doe"
	'Hello John Doe'
		'Hello John Doe' ✅
		`John Doe`
		`Hello John Doe`

Var1 = int(input())
if Var1 == 1 then
  print("One")
elseif Var1 == 2 then
  print("Two")
endif
	Input will be "1"
	'One'
		'One' ✅
		`1`
		`One`

Var2 = 10 for count = 0 to 3 Var2 = Var2 + 1 next count print(Var2)	'14'	'14' ✅ `14`

Var1 = 2 / 0	DivisionByZeroError	Python ZeroDivisionError ❌
		`File "C:\Users\hp\Desktop\Old Files\Deskt`
		`p\Programming\Projects\Language\NewOCR.py",`
		`line 1458, in divideBy`
		`return Float(self.value / data.value)`
		`~~~~~~~~~~~~~^~~~~~~~~~~~~~`
		`ZeroDivisionError: division by zero`

function GetValue(par1) Var1 = 1 endfunction print(Var1)	NameError	NameError ✅ `NameError: No Identifier Named Var1`
Var3 = 10 + “Hello”	TypeError	TypeError ✅ `TypeError: Invalid Operands for +`

Prototype 3 Remedial Action

Test 4 failed as instead of the Interpreter class catching and returning the DivisionByZeroError as an object, an actual Python ZeroDivisionError was thrown, thus halting the program. This occurred as the case when the RHS operand of division is 0 was not considered during development, and so it must be included now in an if statement before the actual division for value occurs. This is **justified** as if this action was not taken and the Python error was allowed to remain, the program with the complete GUI may halt entirely, thus requiring reloading each time this error is made, which would lead to a poor user experience and much slower development times.

```
if data.value == 0: # Checks Whether RHS Operand of / is 0
    return Error('n/a', 'DivisionByZeroError', 'Cannot Divide By 0') # Throws a ZeroDivisionError to Prevent Complete Halting
```

For **full justification**, the previously failed test has now been completed successfully:

Var1 = 2 / 0	DivisionByZeroError	DivisionByZeroError ✅ `DivisionByZeroError: Cannot Divide by 0`

Prototype 3 Evidence

```
C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOC
R.py
DivisionByZeroError: Cannot Divide by 0

C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOC
R.py
6
DivisionByZeroError: Cannot Divide by 0

C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOC
R.py
1800

C:\Users\hp\Desktop\Old Files\Desktop\Programming\Projects\Language>python NewOC
R.py
1
2
Hello
```

Prototype 3 Review

At Stage 1 Prototype 3, we fully completed the Interpreter, which traversed the Abstract Syntax Tree from the Parser stage in the order pre-set during Parsing, thus completing all required operations, console interactions, etc. Furthermore, by creating a standalone Interpreter class (to represent the Node Interpreter singly) and by having distinct methods for each visit function (for each node type

as defined during the Parser stage), modularity was maintained here, which will be useful during post-development testing (as in the case of unit-testing). Overall, with all test cases passed in a fairly robust set of test cases (including invalid test data), there is sufficient evidence that Stage 1 has been passed successfully and we can move on to Stage 2.

Stage 1 Evidence of Validation for All Key Elements

Lexer Validation

Range Check: Ensures that self.text is not accessed at a negative index (avoids the accidental accessing of the final elements when the first is intended, thus avoiding unwanted logical errors). Validation occurs inside the getTokenList() function.

```
if self.currentPosition >= 0: # Validation: Avoids Accidental Indexing of Final Characters (e.g. -1) if Decremented Too Much
    self.currentCharacter = self.text[self.currentPosition] # Sets currentCharacter to Previous Position Only if currentPosition is Non-Negative
return Error('N/A', 'OutOfBoundsError', 'Decremented Character in Lexer Too Far Backwards') # Line Number Unspecified as It is a System Error
```

Type Check: Ensures that self.currentCharacter is of a valid type that matches a recognised token. If a token is not able to be identified (where tokens are based completely off OCR's documentation), a custom error is thrown. Validation occurs inside the getTokenList() function.

```
else:
    return Error(self.currentLineNumber, 'InvalidTokenError', 'Unrecognised Token') # Validation: Catch-All if Unrecognised Token is Found (Halts Lexing)
```

Type Check: Throws an error if an exclamation mark is found alone as this is an unrecognised token, thus being useful for maintenance (as unrecognised tokens are handled strictly). Validation occurs inside the getTokenList() function.

```
if Len(nextCharacter) == 0: # Validation: In the Case of EOF (In Some Editors)
    return Error(self.currentLineNumber, 'InvalidTokenError', '! Unrecognised Token') # ! Alone is Unrecognised Token
```

Parser Validation

Presence Check: Checks to see whether binary operation has an RHS operand. If not, it throws an error as this is invalid syntax and would lead to Python errors during node interpretation. Validation occurs inside the parseGeneralTerm() function.

```
if self.currPos >= len(self.tokenList): # Validation: If RHS Operand Does Not Exist, This is Invalid Syntax for a Binary Operation
    return Error(self.currToken.lineNumber, 'SyntaxError', 'Invalid Expression Syntax') # SyntaxError Thrown if RHS Operand Does Not Exist
```

Format Check: Checks to see whether a right parenthesis is found without an 'Open Bracket' counterpart. If so, an error is thrown as this is invalid syntax (brackets must be balanced). Validation occurs inside the parseExpression() function.

```
elif self.currToken.type == TokenTypeContainer().RPAREN and self.bracketCount == 0: # Validation: Checks Whether Closed Bracket Exists Without an Open Counterpart
    return Error(self.currToken.lineNumber, 'SyntaxError', 'Bracket Unopened') # Throws an Error as Brackets Closed Must Be Opened
```

Type Check: Checks to see whether unary operators '+' or '-' precedes a Boolean value. If a Boolean type is found in this situation, an error is thrown as there are no defined OCR operations in which a unary '+' or '-' precedes a Boolean. Validation occurs inside the getOperatorSign() function.

```
if self.currToken.type in (TokenTypeContainer().PLUS, TokenTypeContainer().MINUS) and expressionType == 'Boolean': # Validation: Checks Whether +/- Precedes Boolean
    return Error(self.currToken.lineNumber, 'SyntaxError', 'Invalid Operands') # If So, Throws an Error as Booleans Don't Accept Unary +/-
```

Node Interpreter Validation

Type Check: Checks to see whether the RHS operand of a '/' binary operation is 0. If so, it throws a custom DivisionByZero error as otherwise, a Python ZeroDivisionError would lead to unwanted halting. Validation occurs inside the divideBy() function of the Number class.

```
if data.value == 0: # Checks Whether RHS Operand of / is 0
    return Error('n/a', 'DivisionByZeroError', 'Cannot Divide By 0') # Throws a ZeroDivisionError to Prevent Complete Halting
```

Type Check: Checks to see whether an RHS operand of a '+' operator with an LHS Number operand is a non-Number. If so, an error is thrown as that operation is invalid and would lead to Python SyntaxErrors (unwanted halting). Validation occurs inside the addTo() function of the Number class.

```
if not isinstance(data, Integer) and not isinstance(data, Float): # Validation: Addition With a Number Can Only Happen With Another Number
    return Error('n/a', 'TypeError', f'{type(self).__name__} & {type(data).__name__} Invalid Operands For +') # Throws a TypeError for Incompatible Types
```

Stage 1 Summary Review

At Stage 1 Prototype 1, we fully completed the Lexer, which was responsible for creating a 1D token list passed to the Parser in Prototype 2. We successfully used encapsulation in the representation of a token using the Token class with attributes currentLineNumber (for future processing as parsers will not have direct access to the text editor), value, and type (which has further been encapsulated under the TokenTypeContainer class). Therefore, in completing this stage, while having all tokens necessary for the complete syntax defined in OCR's documentation, we also maintained flexibility in a modular approach (considering classes, separate helper functions, etc.), thus making future unit-testing in post-development (and in the next few prototypes easier). We also implemented Errors, which were used successfully for validation in all necessary token identification conditionals in the Lexing stage and used further in the Interpreter and eventually in the other primary windows. To reinforce the success of this prototype, we completed robust testing (justified in the Design stage) of the Lexing stage, where all tests defined previously passed without fail.

At Stage 1 Prototype 2, the Parser was created, which converted the token list from the Lexer into a logically organised Abstract Syntax Tree for node interpretation. Modularity was maintained in having a standalone main Parser class, with separate methods for parsing different statement types, thus enabling the unit-testing of statements (as evidenced in testing). Furthermore, statement types were also stored in standalone nodes with particular attributes, which helped during node interpretation as the expected functionality was named accordingly. A separate class for the Expression Parser also was made, in which the complexity of handling an order of operations between different operation types - Boolean, arithmetic, String - was abstracted by defining order in terms of a 2D tuple and using recursion (considering the symmetry of binary operations). Overall, this prototype was successful, as the Lexer and Parser combined successfully, as evidenced in testing where all test cases (including tests for robustness in the case of returning Syntax and Indentation errors) passed successfully.

At Stage 1 Prototype 3, we fully completed the Interpreter, which traversed the Abstract Syntax Tree from the Parser stage in the order pre-set during Parsing, thus completing all required operations, console interactions, etc. Furthermore, by creating a standalone Interpreter class (to represent the Node Interpreter singly) and by having distinct methods for each visit function (for each node type as defined during the Parser stage), modularity was maintained here, which will be useful during post-development testing (as in the case of unit-testing). Overall, with all test cases passed in a fairly robust set of test cases (including invalid test data), there is sufficient evidence that Stage 1 has been passed successfully and we can move on to Stage 2.

Stage 2: Text Editor + Interpreter

Prototype 1: Pretty Printing

Firstly, the text editor must be displayed for the pretty printing results to be seen. As such, we shall also format the screen with all components, positioning the text editor relative to the other primary components to make Stage 5 easier. This is **related to 'Analysis'** by the requirement of the 1400 x 700 pixel GUI, the GUI being divided into three primary sections (ignoring the top bar), the vertically scrollable text editor, and the large font size in the text editor (this is a constant feature, so it is defined here). This is **related to Problem Breakdown** by the 'Load GUI' and 'Load Layout of Primary Windows' processes of Algorithm A, **justified** as this is essentially just formatting sections. To **explain**, a Tkinter window was created, with a ScrolledText object being used to represent the text editor, with packing used to position each object depending upon its height and width defined to fill the screen based upon the 'GUI Structure Design'. To **justify**, Tkinter was used as it is sufficiently customisable to allow for size variations (e.g. height, width, pack sides). The ScrolledText object is justified as this ensures that the scrollbar is directly next to the text window, which is the desired behaviour especially as other scrollbars might exist in other windows. Packing is justified as they are essentially rectangular windows that fill the space, and the simplicity of packing facilitates this.

```
# Initial Window Formatting
window = tk.Tk() # Creates a Tkinter Window
window.title('OCR Interpreter') # Sets the Tabbed Title of the Tkinter Window
window.geometry('1400x700') # As Per the Success Criteria, 1400x700 is Preferred
window.configure(background = 'lightgray')

# Sets up Frames Inside Windows Sectioning Off Primary Components
TopBar = tk.Frame(window, background = 'red', borderwidth = 0, height = 90, width = 1400) # Sets up the Top Bar To Later House Run and Change Theme
TextEditor = tk.scrolledtext.ScrolledText(window, borderwidth = 0, height = 610, width = 51, background = 'white') # ScrolledText as the ScrollBar for Success Criteria
TextEditorFontStyle = ('Roboto Mono', 20, 'normal') # Sets Font Style For Text Editor
TextEditor.configure(font = TextEditorFontStyle) # Adds the Font Style Set Prior to Text Editor
TextEditor.insert('1.0', '1') # Maintenance: Necessary as the First Line Number Must Exist Upon Startup (Cannot Be Triggered By Return as The Line is Automatic)
Console = tk.Text(window, background = '#A0ABBD', borderwidth = 0, height = 610, width = 20, wrap = "word") # Sets Up Position for Console, Though May Change to Text Later
ConsoleFontStyle = ('Roboto Mono', 20, 'normal')
FileOperationBar = tk.Frame(window, background = 'orange', borderwidth = 0, height = 350, width = 350) # Frame Necessary as Objects (Buttons) Must Exist Within It
FileHierarchyBar = tk.Frame(window, background = 'yellow', borderwidth = 0, height = 260, width = 350) # Frame Necessary as a File List Must Exist Within It

# Packing Necessary to Place Each Component Defined Above Graphically in the Window
TopBar.pack(side = 'top', fill = 'both', expand = True) # Prioritises Top as the Top Must Completely Take the Top Portion of the Window
Console.pack(side = 'right', expand = False, pady = 10) # Right as We Need Left Space for the File Windows
TextEditor.pack(side = 'right', expand = False, padx = 10, pady = 10) # Padding As Text Neighbouring the Edge is Aesthetically Unsuitable
FileOperationBar.pack(side = 'top', fill = 'both', expand = True) # Top as Right Space is Already Reserved By the Previous Components
FileHierarchyBar.pack(side = 'bottom', fill = 'both', expand = True) # Bottom as Right Space is Already Reserved By the Previous Components
```

Next, we shall need the helper functions and structures for pretty printing. This is **related to 'Analysis'** by the requirement of 'Pretty Printing in the Text Editor', **justified** as these functions will modularise the process of pretty printing and reduce the bulk of the main function, thus improving readability and maintenance. This is **related to Problem Breakdown** by the 'Format With Pretty Printing Colours' process in Algorithm H, **justified** as these functions are a useful precursor to the actual application of the pretty printing on the text. To **explain**, createTextEditorTags() defines each word type (and connects it to its associated font colour), prettyPrinter is a dictionary of lists, which specifically enumerates those words/characters that belong to the Keyword and Operator tags, and getTagType() takes a word as input by the main pretty printing function and returns its associated colour tag for formatting in the main function. To **justify**, createTextEditorTags() provides maintenance support as all tags are located in an easily accessible, shared location; prettyPrinter reduces the bulk of the main function by removing the need for excess if statements, which may otherwise worsen readability; and getTagType() also reduces the bulk of the main function by performing the required conditionals to check tag type within itself, while providing an alternative root with 'Other', in which case Text Editor tags do not need to be applied (for optimisation).

```
# Creates Each Separate Tag For Highlighting, Naming Them Appropriately, Including Font Color
def createTextEditorTags(TextEditor):
    TextEditor.tag_config('Keyword', foreground = '#f92672') # Sets a Consistent Colour for Keywords
    TextEditor.tag_config('String', foreground = '#a6e22e') # This is the Colour for Words Enclosed In Double Quotes (Including Double Quotes)
    TextEditor.tag_config('Number', foreground = '#fd971f') # This is for Numeric Characters Without Letters (and Outside of Strings)
    TextEditor.tag_config('Boolean', foreground = '#fd5ff0') # This is for True and False Specifically Outside of Strings
    TextEditor.tag_config('Operator', foreground = '#ae81ff') # Sets a Consistent Colour for Operators
createTextEditorTags(TextEditor) # Calls the Function to Create These Tags in One Place Before First Entering the Pretty Printing Function
```

```
# Dictionary for What Constitutes a Keyword and Operator to Reduce the Bulk on the Pretty Printing Function (Easy Access)
prettyPrinter = {
    'Keyword': [ # Lists All Keywords Defined During the Lexing Stage as Per OCR Documentation
        'if', 'then', 'elseif', 'else', 'endif', # If Keywords
        'switch', 'case', 'default', 'endswitch', # Switch Keywords
        'for', 'to', 'next', # For Keywords
        'while', 'endwhile', # While Keywords
        'do', 'until', # Do Until Keywords
        'procedure', 'endprocedure', # Procedure Keywords
        'function', 'endfunction', 'return' # Function Keywords
    ],
    'Operator': [ # Lists All Operators Defined During the Lexing Stage as Per OCR Documentation
        '+', '-', '*', '/', '^', # Arithmetic Operators Grouped for Maintenance
        'AND', 'OR', 'NOT', # Logical Boolean Operators
        '=', '>', '>=', '<=', '<', '!=', '==' # Comparison Operators and Assignment
    ]
}

# Helper Function Which Takes the Word Enclosed by the prettyPrintingFunction and Returns the Associated Colour Tag
def getTagType(currWord):
    if currWord in prettyPrinter['Keyword']: # Searches the Previously Defined prettyPrinter to Determine if Keyword
        return 'Keyword'
    elif currWord in prettyPrinter['Operator']: # Searches the Previously Defined prettyPrinter to Determine if Operator
        return 'Operator'
    elif '"' in currWord: # All Strings are Enclosed in Only Double Quotes, So This Will Work
        return 'String'
    elif currWord.isnumeric(): # If Only Numbers are Found Unenclosed by Double Quotes, It Must Be a Number
        return 'Number'
    elif currWord in ['True', 'False']: # True/False Unenclosed in Double Quotes (String Checked Prior)
        return 'Boolean'
    return 'Other' # Validation: Catch-All If Word Type is Not Specialised for Highlighting/Pretty Printing
```

Next, we shall need to implement the main pretty printing function. This is **related to 'Analysis'** by the requirement of 'Pretty Printing in the Text Editor', **justified** as this is the function that ultimately applies the predefined colours tags to display the pretty printed text. This is **related to Problem Breakdown** by the 'Format With Pretty Printing Colours' process in Algorithm H, **justified** as application (formatting) occurs here. To **explain**, a word is enclosed by iterating character by character through a while loop until a whitespace is found (newlines are ignored, justified as these are already removed during the split('\n') function, separating the data into lines due to Tkinter indexing in a line : character index format). When a word is identified, the associated tag is obtained through the getTagType() helper function, and that is applied using the tag_add() Tkinter function after formatting the start and final indexes to Tkinter's specification. To **justify**, the use of helper functions reduces the bulk of this function, which makes the code more readable, thereby promoting maintainability. For further justification, it is done on a word-by-word basis as pretty printed words are independent of each other (i.e. an 'if' will always be the same colour regardless of the words that precede or follow it). This also justifies the loop structure as each word is handled in the same loop in the same manner because they are ultimately independent.

```
# Main Pretty Printing Function Which Scans the Entirety of the Text Editor and Colours Words Where Appropriate
def prettyPrintTextEditor(event):
    print('-------------------------')
    TextEditor = event.widget # Gets Text Editor, Which is How the Pretty Printing Function Was Activated (Upon Key Release)
    TextEditor.tag_remove("Keyword", "1.0", "end") # Removal is Necessary as Otherwise, Overlapping Tags Leads to Unoptimised Performance,
    TextEditor.tag_remove("String", "1.0", "end") # Wasted Storage Space, and Potentially Confusion Within Tkinter Which Could Lead to
    TextEditor.tag_remove("Operator", "1.0", "end") # Accidental Highlighting or Even a Lack of Highlighting
    TextEditor.tag_remove("Number", "1.0", "end") # All Tags Defined Previously are Removed If They Exist
    TextEditor.tag_remove("Boolean", "1.0", "end") # Within the Words Currently in the Text Editor
    textEditorStateList.insert(TextEditor.get('1.0', tk.END)) # This Updates State List With the Current Text Editor State (For Undoing Later On)
    textLines = TextEditor.get('1.0', tk.END).split('\n') # Splits the Input from Text Editor Into Lines as Indexing Functionality Occurs With Line.Index Floats
    for lineNumber in range(len(textLines)): # Iterates Through all Lines One By One
        if lineNumber == len(textLines) - 1: # Validation: Avoids Python IndexOutOfBounds Error (To Prevent Unwanted Halting)
            break # If So, It Breaks to Stop Future Processing
        if len(textLines) == 1: # If Only 1 Line, This Processing Must Begin at Index 1 as Index 0 Already Contains the Automatic 1 (Line Number)
            charNumber = 1 # Sets Initial Processing Index to 1
        else:
            charNumber = 0 # Otherwise, Sets Initial Processing Index to 0
        while True: # Breaking Out of Loop Occurs With Internal Break Statements
            finalIndexFlag = False # Assumes Initial Character is Not the Last (Alternative Case Handled on Line Number 2139)
            startIndex = finalIndex = charNumber # Both Start and Final Indexes Required for tag_add Method of Tkinter
            currWord = '' # Word To Be Accumulated Over While Loop
            if charNumber >= len(textLines[lineNumber]): # Validation: Avoids Python IndexOutOfBounds Error (To Prevent Unwanted Halting)
                break # Validation: Avoids Python IndexOutOfBounds Error (To Prevent Unwanted Halting)
            currCharacter = textLines[lineNumber][charNumber] # If Within Bounds, Gets Exact Value of Current Character
            while currCharacter not in ' \t': # Loop Breaks (Word Found) When A Whitespace is Encountered
                currCharacter = textLines[lineNumber][charNumber] # Resets Current Word (Like a Do While Loop)
                if charNumber == len(textLines[lineNumber]) - 1: # Validation: Special Case Handling When Character at End of Line
                    if currCharacter != ' \t': # Does Not Include Extra Whitespace in Word
                        currWord += currCharacter # Because It Interferes With Conditionally Identifying Tag Type
                    finalIndex = str(lineNumber + 2) + '.' + '0' # lineNumber + 2 as Tkinter Indexing Begins at 1, and It Is Non-Inclusive, So Next Line Should Be
                    finalIndexFlag = True # Special Case Handling When Character at End of Line
                    break # Processing Stops as End of Line is Reached
                currWord += currCharacter # If Not End of Line, Characters Continue Accumulating
                charNumber += 1 # Goes to Next Indexing
```

```
                if charNumber >= Len(textLines[lineNumber]): # Validation: Avoids Python IndexOutOfBounds Error (To Prevent Unwanted Halting)
                    break # Validation: Avoids Python IndexOutOfBounds Error (To Prevent Unwanted Halting)
            tagType = 'Other' # General Value Before tagType Actually Determined
            currWord = currWord.replace('\t', '') # Removes Whitespace as This Interferes With Tag Type Identification (Because Exact String Values Compared)
            currWord = currWord. replace(' ', '')
            tagType = getTagType(currWord) # Calls Helper Function to Determine tagType
            if finalIndexFlag: # Special Case Handling if Last Character of Line
                startIndex = str(lineNumber + 1) + '.' + str(startIndex) # lineNumber + 1 as Tkinter Indexing Begins at 1 (So Must Be Increased by 1)
                if tagType != 'Other': # Adds Tag if it is Especially Defined
                    TextEditor.tag_add(tagType, startIndex, finalIndex) # Adds Tag if it is Especially Defined
                print('|' + currWord + '|', tagType, startIndex, finalIndex)
                tagType = 'Other' # Resets tagType to Ensure Unwanted Highlighting Does Not Occur for Alternative Words
                break # Avoids Future Processing As This Would Lead to Overlapping Tags
            startIndex = str(lineNumber + 1) + '.' + str(startIndex) # lineNumber + 1 as Tkinter Indexing Begins at 1 (So Must Be Increased by 1)
            finalIndex = str(lineNumber + 1) + '.' + str(charNumber - 1) # lineNumber + 1 as Tkinter Indexing Begins at 1 (So Must Be Increased by 1)
            print('|' + currWord + '|', tagType, startIndex, finalIndex)
            if tagType != 'Other': # Adds Tag if it is Especially Defined
                TextEditor.tag_add(tagType, startIndex, finalIndex) # Adds Tag if it is Especially Defined
            tagType = 'Other' # Resets tagType to Ensure Unwanted Highlighting Does Not Occur for Alternative Words
TextEditor.bind('<KeyRelease>', prettyPrintTextEditor) # Pretty Printing Function Called Everytime a Key is Released (Not Pressed So There is Time to Update)
```

Prototype 1 Testing

Test Data	Expected Output	Actual Output
if then else endif	1 if then else endif	1 if then else endif ✅ `1 if then else endif`

for count = 0 to 1
  print(“Hello”)
next
	1 for count = 0 to 1
	2 print(“Hello”)
	3 next
		1 for count = 0 to 1
		2 print(“Hello”)
		3 next ✅
		`1 for count = 0 to 1`
		`2  print( "Hello" )`
		`3 next`

Var1 = 2 + 2 AND True	1 Var1 = 2 + 2 AND True	1 Var1 = 2 + 2 AND True✅ `1 Var1 = 2 + 2 AND True`

switch case while endwhile
elseif do until MOD DIV
	1 switch case while endwhile
	2 elseif do until MOD DIV
		1 switch case while endwhile
		2 elseif do until MOD DIV ❌
		`1 switch case while endwhile`
		`2 elseif do until MOD DIV`

Prototype 1 Remedial Action

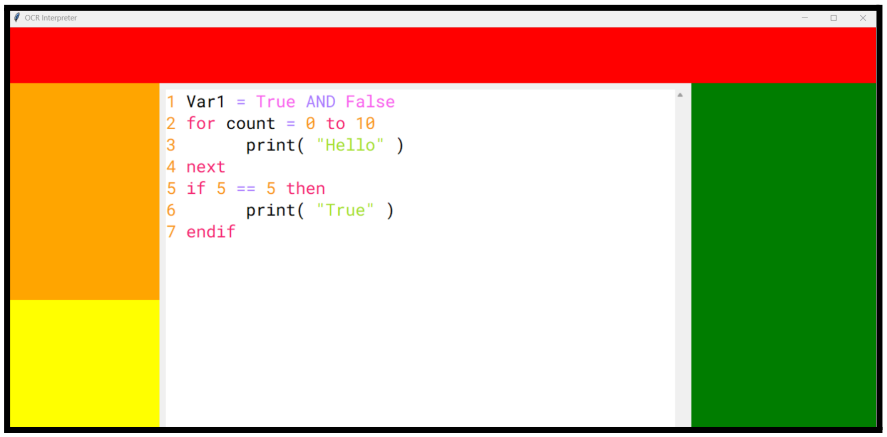
The final test failed as DIV was highlighted black (as an unspecialised word) instead of the purple associated with operators. This occurred as DIV was missed when defining the ‘Operator’ key of the prettyPrinter dictionary. The action that will be taken is to thus include DIV in the ‘Operator’ key, next to MOD for maintenance purposes as this organises similar operators for readability. This is **justified** as an alternative would be to check for DIV in the main function, but this would worsen readability as some operators would be in the prettyPrinter and some not, thus also worsening maintainability. Furthermore, unnecessary conditionals can be avoided, optimising performance.

```
'+', '-', '*', '/', 'MOD', 'DIV', '^', # Arithmetic Operators Grouped for Maintenance
```

For **full justification**, the previously failed test has now been completed successfully:

switch case while endwhile
elseif do until MOD DIV
	1 switch case while endwhile
	2 elseif do until MOD DIV
		1 switch case while endwhile
		2 elseif do until MOD DIV ✅
		`1 switch case while endwhile`
		`2 elseif do until MOD DIV`

Prototype 1 Evidence



Prototype 1 Review

At Stage 2 Prototype 1, we created a mechanism by which each key release triggered a main pretty printing function, which broke the text editor input into words and applied colour tags based on the nature of those words. Furthermore, by breaking the implementation into various functions, including helper functions that collectively created each tag type in a shared location and the prettyPrinter dictionary which avoided long nested conditionals in identifying Operators and Keywords, we reduced the bulk of a given code section and promoted modularity, thus improving readability and maintainability. There was also robust validation particularly in the form of length and range checks to avoid OutOfBounds errors during iteration to obtain words, which is useful in avoiding accidental halting, thus contributing overall to the smoothness of the IDE experience. Overall, pretty printing occurred successfully for all keywords, operators, literals, etc. as evidenced by all test cases being passed successfully, the test cases themselves being relatively robust in checking a variety of data input types. As such, there is sufficient evidence that Stage 2 Prototype 1 has been completed successfully, meaning we can move on from its development (until post-development testing).

Prototype 2: Pretty Printing + Undo

Copy, Paste, and Cut are already a feature of the ScrolledText object in Tkinter, so only the Undo feature will need to be implemented from scratch. First, we shall create a custom fixed-length Stack class. This is **related to 'Analysis'** by the necessity of a functional text editor, **justified** as the Stack is the data structure that the Text Editor State List (traversed to undo to previous states) will use, and this is optimal for a useful text editor. This is **related to Problem Breakdown** by the 'Load Empty Text Editor State List of Fixed Size' process in Algorithm A, **justified** as the Text Editor State List is a Stack object. To **explain**, the class will consist of custom nodes (with attributes data to hold strings of text editor states and next to point to the next StackNode, working like a linked list), with insertions and deletions occurring at the start index (due to the Stack being a LIFO structure) with length limitations as determined by the predetermined constant MAX_LENGTH. To **justify**, a custom class is useful as there is custom functionality, such as the fixed length of the state list, while the class promotes modularity useful for unit-testing. Further **justification** of the Stack itself is when a

new state is added, that is the one popped from the stack when ‘Undo’ is activated, thereby being LIFO.

```
# A Class of a Node Representing an Element in a Stack
# Shall Be Used in the textEditorStateList State for Undo Functionality
class StackNode:
    # Initialises a StackNode Object With Class Attributes Identical to Those Described Above
    def __init__(self, data):
        self.data = data # data Will be a String of the Text Editor State as Per a Tkinter get() Function
        self.next = None # next Will Point to the Next StackNode of the Stack (Connection Mechanism)

# A Class of a Stack (LIFO) To Be Used in the Implementation of the textEditorStateList
# Extra Functionality is a Fixed Size List, Set to 20 as This is a Fair Default Undo Amount
class Stack:
    # Initialises a Stack Object With Class Attributes Identical to Those Described Above
    def __init__(self):
        self.head = None # Represents the First Element in the Stack, Where Both Addition and Removal Occur
        self.MAX_LENGTH = 20 # Represents the Maximum Text Editor States Storable, Prevents Storing Extraneous Large Strings for Optimisation
        self.length = 0 # Sets the Initial Length (Amount of Attributes) to 0, Because a Stack is Created Empty

    # Adds a Node to the Stack If There is Space Available, Otherwise Does Nothing (Necessary as Extra Undoing May Be Attempted, So Halting Errors Should Not Be Thrown)
    def insert(self, data):
        newNode = StackNode(data) # Creates the Node Which Will Be Added, Including the Text Editor State
        if self.head is None: # If the Stack is Empty, self.head Must Be Explicitly Changed (With New Node)
            self.head = newNode # Explicit Changing of self.head
            self.length += 1 # Increases the Length by 1 to Reflect the Growing List (The New Node)
            return # Avoids Further Processing (As This May Lead to Unwanted Node Deletion)
        if self.length < self.MAX_LENGTH: # Validation: Only Increases the Length by 1 if the List is Not Maximum Size, Otherwise Extra Storage Issues May Occur
            self.length += 1 # Where It Is Not Maximum Size, Increases the Length by 1 to Reflect the Growing List (The New Node)
        newNode.next = self.head # Shifts the Current Head To Head + 1, as It is No Longer the Most Recently Added Node
        self.head = newNode # Updates Head to Newest Node To Reflect Stack Behaviour (LIFO)
        if self.length <= self.MAX_LENGTH: # Validation: Prevents Surplus Deletion If Max_Length is Not Reached As This Would Lead to a Prematurely Ungrowing List
            return # Stops Further Processing To Prevent Surplus Deletion
        finalNode = self.head # Temporary Node Which Will Iterate Until End of Stack
        while finalNode.next.next is not None: # Validation: Prevents Indexing Unknown Attribute of None (As It Stops Before Reaching the None Value)
            finalNode = finalNode.next # Continue Iterating Until Last Node of Stack
        finalNode.next = None # Deletes the Extra Final Node (Should Maximally Be 20)
```

Next, we shall implement the main undo function, which returns the text editor to the immediately preceding state. This is **related to ‘Analysis’** by the necessity of a functional text editor, as undoing is a common feature of text editors, fulfilling ease-of-use user needs. This is **related to Problem Breakdown** by the ‘Return Text Editor to Backwards State’ process of Algorithm H, **justified** as the undo function’s purpose is to return to the immediately preceding state. To **explain**, the textEditorStateList was defined as a Stack as discussed, and undoTextEditor() (binded to Control-Z) pops the current head of the Stack (current state), such that the new head is the preceding state, storing it. The text editor is then completely cleared and reinstated with the string contents of the variable storing the immediately preceding state. To **justify**, Control-Z is the binding as this is a standard commonly associated with undoing. The **justification** for the clear-reinstate sequence is that it is intuitive and readable. An alternative would be only storing the change that was made, which would require less storage space, though the benefits seem to be outweighed by the disadvantages of the further conditionals of identifying how that change compares to the previous text editor states and the complexity of finding a suitable storage format that works out these dependencies.

```
# Function Activated Upon the Key Press of Control-Z (As Per Convention)
# Returns the Text Editor State to the Immediate Previous Version Using Stack Functionality
# Event Attached Only to Text Editor So No Interference With Other Program Components (Maintenance)
def undoTextEditor(event):
    TextEditor = event.widget # Obtains Text Editor Object To Edit It Directly
    textEditorStateList.remove() # LIFO, So This Removes The Current Text Editor State, Making Front /Head the Exact Previous
    if textEditorStateList.isEmpty(): # Validation: Prevents Further Deletions if textEditorStateList is Empty, Avoiding Unwanted Deletions
        return # Halts Further Processing By Exiting the Function
    if textEditorStateList.isEmpty(): # In the Case of an Empty Text Editor State List, The Text Editor Itself Is Blank
        textEditorData = '' # As Such, We Can Prepare to Insert a Blank String (Representing Empty Text Editor)
    else: # If It is Not Empty, Text Editor is Not Blank
        textEditorData = textEditorStateList.getFront() # Previous State is Head of Datd
    TextEditor.delete('1.0', tk.END) # Clears Text Editor Completely To Smoothly Add Gathered Data
    TextEditor.insert('1.0', textEditorData) # Reloads Text Editor With Immediately Previous State, As Expected Of Undo
TextEditor.bind('<Control-z>', undoTextEditor) # Binds Ctrl+Z to Text Editor, For Undo Activation
```

Prototype 2 Testing

Test Data	Expected Output	Actual Output
‘Ctrl + C’ on selected text in the text editor, followed by ‘Ctrl + V’ at a different point	Text should be duplicated at the new point, and the copied text should remain at the old point	Text was duplicated at the new point, and the copied text remained at the old point ✅

		7 endif endif
‘Ctrl + X’ on selected text in the text editor, followed by ‘Ctrl + V’ at a different point	Text should be duplicated at the new point, and the cut text should be gone from the old point	Text was duplicated at the new point, and the cut text left the old point ✅ 1 Var1 = 2 100 10
‘Ctrl + Z’ in the text editor pressed once	Text editor should return to immediate previous state	Text editor returned to immediate previous state ✅ 1 Var1 = 2 100 10 10 10 10 10 10 10 10 10 1 Var1 = 2 100 10 10 10 10 10 10 10 10 1
‘Ctrl + Z’ in the text editor pressed twice	Text editor should return to immediate previous state twice, ending at second previous	Text editor returned to immediate previous state, but it did not do the same for the second Ctrl + Z ❌ 1 Var1 = 10 10 10 10 10 10 True False False True 2 Var2 = "Hello" + True 10 10 10 10 10 10 1 Var1 = 10 10 10 10 10 10 True False False True 2 Var2 = "Hello" + True 10 10 10 10 10 1 1 Var1 = 10 10 10 10 10 10 True False False True 2 Var2 = "Hello" + True 10 10 10 10 10 1

Prototype 2 Remedial Action

The final test failed as the second ‘Ctrl + Z’ key press did not lead to a reversion of the text editor into a previous state. This occurred as ultimately there was one call of textEditorStateList.remove() in the main undo function. Though this initially makes sense, it is problematic as the function appending text editor states is triggered upon key release, and technically pressing ‘Ctrl and Z’ leads to an extra key release that was not caught before. As such, when attempting to undo, the text editor remains in the same state as these effects cancel each other out. The action that will be taken is to add an extra textEditorStateList.remove() in the main undo function, thus deleting this “extra copy” of the current text editor state. This is **justified** as the alternative would be to avoid the extra insertion, and this would be relatively inefficient as extra conditionals would need to be added to check for the key that is pressed in multiple locations in the code. The extra deletion requires a single line, thereby being readable and maintainable.

```
# Function Activated Upon the Key Press of Control-Z (As Per Convention)
# Returns the Text Editor State to the Immediate Previous Version Using Stack Functionality
# Event Attached Only to Text Editor So No Interference With Other Program Components (Maintenance)
def undoTextEditor(event):
    TextEditor = event.widget # Obtains Text Editor Object To Edit It Directly
    textEditorStateList.remove() # LIFO, So This Removes The Current Text Editor State, Making Front /Head the Exact Previous
    if textEditorStateList.isEmpty(): # Validation: Prevents Further Deletions if textEditorStateList is Empty, Avoiding Unwanted Deletions
        return # Halts Further Processing By Exiting the Function
    textEditorStateList.remove() # LIFO, So This Removes The Current Text Editor State, Removes Extra Copy of Current State
    if textEditorStateList.isEmpty(): # In the Case of an Empty Text Editor State List, The Text Editor Itself Is Blank
        textEditorData = '' # As Such, We Can Prepare to Insert a Blank String (Representing Empty Text Editor)
    else: # If It is Not Empty, Text Editor is Not Blank
        textEditorData = textEditorStateList.getFront() # Previous State is Head of Datd
    TextEditor.delete('1.0', tk.END) # Clears Text Editor Completely To Smoothly Add Gathered Data
    TextEditor.insert('1.0', textEditorData) # Reloads Text Editor With Immediately Previous State, As Expected Of Undo
TextEditor.bind('<Control-z>', undoTextEditor) # Binds Ctrl+Z to Text Editor, For Undo Activation
```

For **full justification**, the previously failed test has now been completed successfully:

‘Ctrl + Z’ in the text editor pressed twice	Text editor should return to immediate previous state twice, ending at second previous	Text editor returned to immediate previous state, and did the same for the second key press ✅
		`1 Var1 = 10 10 10 10 10 10 True False False True`
		`2 Var2 = "Hello" + True 10 10 10 10 10 10`
		`1 Var1 = 10 10 10 10 10 10 True False False True`
		`2 Var2 = "Hello" + True 10 10 10 10 10 1`
		`1 Var1 = 10 10 10 10 10 10 True False False True`
		`2 Var2 = "Hello" + True 10 10 10 10 10`

Prototype 2 Evidence

Prototype 2 Review

At Stage 2 Prototype 2, we created an undo mechanism activated upon the key press of Control + Z, which traversed to the immediately preceding node in the text editor state list (representing the previous state), returning it for the overwriting of the text editor. Evidently, the process in detail was multi-stepped involving the creation of a fixed-length (for optimisation of memory) Stack class, which worked as a linked list with a shared insertion and deletion point, to represent the text editor state list, backwards traversal using the custom-built remove() and getFront() Stack methods in sequence, while accounting for the technicalities of key release. As such, the use of overarching methods such as insert() and undoTextEditor() being called minimally successfully employed abstraction which will be useful in the future maintenance of the system. The use of validation in the form of length/range checks to avoid OutOfBounds errors to prevent the accidental halting of the system is also useful once again to the overall smoothness of the IDE experience. Furthermore, all of the test cases passed, covering the built-in Control + _ functionality in addition to the custom implementation of Control + Z. As such, there is sufficient evidence that Stage 2 Prototype 2 has been completed successfully, meaning that Stage 2 in its entirety is sufficiently successful to be moved on from in the iterative development stage, and we can revisit robust testing at the post-development stage.

Stage 2 Evidence of Validation for All Key Elements

Pretty Printing Validation

Range Check: Ensures that textLines[lineNumber] is not accessed at an index value greater than its length, thus avoiding a Python OutOfBoundsError, which may otherwise lead to accidental system-based halting. Validation occurs inside the prettyPrintTextEditor() function.

```
if lineNumber == len(textLines) - 1: # Validation: Avoids Python IndexOutOfBounds Error (To Prevent Unwanted Halting)
    break # If So, It Breaks to Stop Future Processing
```

Type Check: Ensures that if currWord does not belong to a specialised, pretty printed word type, 'Other' is returned as a catch-all as opposed to a default None. This avoids a Python TypeError / AttributeError, thus avoiding accidental halting. Validation occurs inside the getTagType() function.

```
elif currWord in ['True', 'False']: # True/False Unenclosed in Double Quotes (String Checked Prior)
    return 'Boolean'
return 'Other' # Validation: Catch-All If Word Type is Not Specialised for Highlighting/Pretty Printing
```

Range Check: Another instance of ensuring that textLines[lineNumber] is not accessed at an index value greater than its length (again upon a charNumber incrementation), thus avoiding a Python OutOfBoundsError and halting. Validation occurs inside the prettyPrintTextEditor() function.

```
charNumber += 1 # Goes to Next Indexing
if charNumber >= len(textLines[lineNumber]): # Validation: Avoids Python IndexOutOfBounds Error (To Prevent Unwanted Halting)
    break # Validation: Avoids Python IndexOutOfBounds Error (To Prevent Unwanted Halting)
```

Undo Validation

Range Check: Ensures that processing is stopped before the deletion process begins as otherwise, deletions would occur for a list below maximum size, leading to a prematurely un-growing list. Validation occurs inside the insert() function of Stack.

```
if self.length <= self.MAX_LENGTH: # Validation: Prevents Surplus Deletion If Max_Length is Not Reached As This Would Lead to a Prematurely Ungrowing List
    return # Stops Further Processing To Prevent Surplus Deletion
```

Range Check: Ensures that length incrementation only occurs when the length is still at a growing stage, as otherwise, text editor states may accumulate beyond MAX_LENGTH if later checks use == as opposed to >= (worsens maintainability). Validation occurs inside the insert() function of Stack.

```
if self.length < self.MAX_LENGTH: # Validation: Only Increases the Length by 1 if the List is Not Maximum Size, Otherwise Extra Storage Issues May Occur
    self.length += 1 # Where It Is Not Maximum Size, Increases the Length by 1 to Reflect the Growing List (The New Node)
```

Length Check: Ensures that further deletions of textEditorStateList only occur when elements exist for deletion, thus avoiding potential Python Type Errors/Attribute Errors when handling None types, avoiding accidental halting. Validation occurs inside the undoTextEditor() function.

```
if textEditorStateList.isEmpty(): # Validation: Prevents Further Deletions if textEditorStateList is Empty, Avoiding Unwanted Deletions
    return # Halts Further Processing By Exiting the Function
```

Stage 2 Summary Review

At Stage 2 Prototype 1, we created a mechanism by which each key release triggered a main pretty printing function, which broke the text editor input into words and applied colour tags based on the nature of those words. Furthermore, by breaking the implementation into various functions,

including helper functions that collectively created each tag type in a shared location and the prettyPrinter dictionary which avoided long nested conditionals in identifying Operators and Keywords, we reduced the bulk of a given code section and promoted modularity, thus improving readability and maintainability. There was also robust validation particularly in the form of length and range checks to avoid OutOfBounds errors during iteration to obtain words, which is useful in avoiding accidental halting, thus contributing overall to the smoothness of the IDE experience. Overall, pretty printing occurred successfully for all keywords, operators, literals, etc. as evidenced by all test cases being passed successfully, the test cases themselves being relatively robust in checking a variety of data input types. As such, there is sufficient evidence that Stage 2 Prototype 1 has been completed successfully, meaning we can move on from its development (until post-development testing).

At Stage 2 Prototype 2, we created an undo mechanism activated upon the key press of Control + Z, which traversed to the immediately preceding node in the text editor state list (representing the previous state), returning it for the overwriting of the text editor. Evidently, the process in detail was multi-stepped involving the creation of a fixed-length (for optimisation of memory) Stack class, which worked as a linked list with a shared insertion and deletion point, to represent the text editor state list, backwards traversal using the custom-built remove() and getFront() Stack methods in sequence, while accounting for the technicalities of key release. As such, the use of overarching methods such as insert() and undoTextEditor() being called minimally successfully employed abstraction which will be useful in the future maintenance of the system. The use of validation in the form of length/range checks to avoid OutOfBounds errors to prevent the accidental halting of the system is also useful once again to the overall smoothness of the IDE experience. Furthermore, all of the test cases passed, covering the built-in Control + _ functionality in addition to the custom implementation of Control + Z. As such, there is sufficient evidence that Stage 2 Prototype 2 has been completed successfully, meaning that Stage 2 in its entirety is sufficiently successful to be moved on from in the iterative development stage, and we can revisit robust testing at the post-development stage.

Stage 3: File-Handling Window + Text Editor + Interpreter

Prototype 1: New File + Import File

First, we shall display the file operation buttons upon start-up. This is **related to 'Analysis'** by the necessity of coloured buttons with image and text descriptions, **justified** as the display of these buttons entails their formatting, in which colour and content will be defined. This is **related to Problem Breakdown** by the 'Load Buttons' process of Algorithm A, **justified** as post-display, the buttons will be loaded. To **explain**, the buttons are first created with attributes including background colour (set to the light theme colours defined in the GUI Structure) and text descriptions, with images also selected and placed within the buttons. Before the images are placed, they are formatted using a helper function which maximises their size within the button. To **justify**, the use of functions to carry out each operation promotes modularity, which aids unit-testing and facilitates maintenance (as implementations of similar features are grouped under a common section). The light theme colours are chosen because light theme is the default colour in which the IDE will be opened, though this is mainly a maintenance fail-safe in case the colour theme is not applied upon start-up. The **justification** for maximising the size of the icons within the button is to satisfy the usability feature of having large images to maximise readability and aid those with weaker eyesight.

```
# Transforms an Image in a Tkinter-Compatible Format After Applying Appropriate Formatting For Placement Into a Button
def getFileImage(fileName):
    fileImage = Image.open(fileName) # Opens the File Given Its Location
    fileImage = fileImage.resize((25, 25)) # (25, 25) Pixels Is Ideal Given the Standard Height of Tkinter Buttons Aesthetically
    fileImage = ImageTk.PhotoImage(fileImage) # Converts the Image into a Tkinter-Compatible Format For Placement Into Button
    return fileImage # Returns the Fully Formatted Image

NewFileImage = getFileImage('NewFileImage.png') # Maintenance: Images are Currently Locally Stored
ImportFileImage = getFileImage('ImportFileImage.png') # Maintenance: For Future Maintenance/Deployment, May Want to Convert to URL Format
SaveFileImage = getFileImage('SaveFileImage.png') # As the Internet Is Globally/Publicly Accessible
DeleteFileImage = getFileImage('DeleteFileImage.png') # Names Are Consistent for Future Maintenance and Readability While Stored Locally
ButtonFontStyle =  ('Roboto Mono', 15, 'normal') # Font Style Consistent With Text Editor (Aesthetics) and 15 is Ideal for Standard Butto

NewFileButton = tk.Button(FileOperationBar, text = ' New File', image = NewFileImage, font = ButtonFontStyle, compound = tk.LEFT, width =
ImportFileButton = tk.Button(FileOperationBar, text = ' Import File', image = ImportFileImage, font = ButtonFontStyle, compound = tk.LEFT
SaveFileButton = tk.Button(FileOperationBar, text = ' Save File', image = SaveFileImage, font = ButtonFontStyle, compound = tk.LEFT, width
DeleteFileButton = tk.Button(FileOperationBar, text = ' Delete File', image = DeleteFileImage, font = ButtonFontStyle, compound = tk.LEFT

# Creates Each File Operation Button Object and Positions Them Within the FileOperationBar In a Horizontally and Roughly Vertically Centre
def packFileOperationButtons():
    # Buttons are All Placed Within Frame FileOperationBar, Where Frames Were Chosen For Their Capacity for Organisation
    # compound Places the Image Relative to the Text, Where Left is Chosen as It Is Consistent With GUI Structure Design
    # Fixed Width 200 Is Ideal for 1400x700 Window as it Fills Out the FileOperationBar While Leaving Space for Aesthetics
    # Default Backgrounds are Set to Those of Light Mode in GUI Structure as This is the Default Colour Theme

    NewFileButton.pack(side = 'top', pady = 20) # 20 is Ideal for Filling Out the Vertical Space While Leaving Empty Regions for Aesthetic
    ImportFileButton.pack(side = 'top', pady = 20) # 20 is Used Consistently as Symmetry Improves the Professionalism of the IDE
    SaveFileButton.pack(side = 'top', pady = 20) # Top is Chosen as Bottom Space is Less Noticeable
    DeleteFileButton.pack(side = 'top', pady = 20) # Top is Used Consistely as This Again Promotes Symmetry
packFileOperationButtons() # Calls the Above Function so the Buttons are Actually Formatted and Displayed Upon Startups (They Are Static)
```

Next, we shall implement a mechanism for updating the file hierarchy when the status of a file is updated using a file operation button. This is **related to 'Analysis'** by the criterion of the file hierarchy consisting of multiple, clickable file names, **justified** as pre-display, the group of files must be ordered and formatted, in which clickability can be implemented. This is **related to Problem Breakdown** by the 'Declare Empty File List' and 'Load Layout of Primary Windows' process in Algorithm A, **justified** as initially the file list begins empty of known files and the file hierarchy is a primary window that needs to be displayed. To **explain**, fileHierarchyObjects contains the file locations of multiple files in a 1D list, and this will be updated during the file operations to be implemented. displayFileHierarchy() then iterates through the list in order, formatting both the file name and the button which will contain it, then displaying it in the frame of the file hierarchy. To **justify**, a list is used to contain the files as this enables multiple files to be inserted, which satisfies the criterion, while indexing using textEditorFileLocation enables files to be traversed between, which allows for flexibility in moving between newly created and deleted files. To **justify** the use of a frame for the file hierarchy, this enables the collection and formatting of elements relative to it, which is useful in this case as all file names will exist purely within the file hierarchy.

```
textEditorFileLocation = 0 # Sets Current File Index to 0 to Point to First New File Upon Start-Up
fileHierarchyObjects = ['***.txt'] # List of All File Names in IDE For Display in File Hierarchy In Order
# Function That Resets File Hierarchy and Updates Its Display With Every Item in fileHierarchyObjects
def displayFileHierarchy():
    global fileHierarchyObjects # Maintenance: In Future, if Assignment is Done, This Will Prevent Scope Errors
    for widget in FileHierarchyBar.winfo_children(): # Gets All Displayed File Objects in the File Hierarchy
      widget.destroy() # Clears Them, Necessary as Otherwise, Redundant Repetitions Will Occur
    for fileHierarchyObject in fileHierarchyObjects: # Goes Through File Names One By One, In Order of Update
        fileName = fileHierarchyObject.split('/')[-1] # Gets Only File Name From Complete File Path (Aesthetics)
        # Button Chosen As Formatting is Reliable and In Future, May Potentially Use For Click Functionality
        # Button Background Set to File Hierarchy Background (Maintenance As It Always Blends In Unless Changed)
        fileHierarchyObject = tk.Button(FileHierarchyBar, text = fileName, font = ButtonFontStyle, width = 200, bord
        fileHierarchyObject.pack(side = 'top', pady = 10) # 10 Aesthetically Suitable for Vertical Separation
        # Top is Ideal for All Files as This Enables Uniformity, Which is Aesthetically Pleasing
displayFileHierarchy() # Calls It Upon Start-Up To Display Existence of Default Empty New File
```

Next, we shall implement the functionality for the 'New File' file operation. This is **related to 'Analysis'** by the criterion of 'New File' leading to the creation of an empty, new file, **justified** as the function is triggered upon the click of the 'New File' button. This is **related to Problem Breakdown** by all processes in Algorithm D, **justified** as this is the designed algorithm for the creation of a new file, which is what is being implemented. To **explain**, newFileClick() is connected to the 'New File' button's command, meaning it is triggered on click. It then clears the text editor (re-inserting the

first line number), and updates the file hierarchy with '***.txt' to reflect the creation of a new text file. To **justify**, the file's default extension is '.txt' as this is the only file type allowable in the IDE, so it will not cause issues during maintenance (as GCSE OCR Pseudocode is unlikely to incorporate image processing, for example). To **justify** the clearing of the text editor, previously Algorithm D specified the opening of the File Explorer, though in principle this would have the same implementation as 'Import File' with a few extraneous steps. To avoid this, 'New File', more logically, creates an empty text file, and the clearing of the text editor reflects the fact that the user is now working on a new, previously unedited file.

```
# Function Triggered Upon the Click of the New File Button in the File Operation Bar
# Creates an Empty New File and Updates the File Hierarchy Bar to Reflect the New, Unsaved File
def newFileClick():
    global textEditorFileLocation # Maintenance: This Will Avoid Scope Errors in the Future if Assignment is Added
    global fileHierarchyObjects # Also Avoids Scope Errors, if Implementation of File List is Changed to Stack, e.g.
    TextEditor.delete('1.0', tk.END) # Clears Contents of Text Editor (as New File Is By Default Empty)
    TextEditor.insert('1.0', '1 ') # Re-Inserts the First Line Number To Prevent Unwanted Logical Deletions
    fileHierarchyObjects.append('***.txt') # ***.txt is the Placeholder for Any New,Unsaved File
    displayFileHierarchy() # Updates Displayed File Hierarchy to Reflect Addition of New File
    textEditorFileLocation += 1 # New File is in the Text Editor, so this File Location Must Update to It
```

Next, we shall implement the functionality for the 'Import File' file operation. This is **related to 'Analysis'** by the criterion of 'Import File' opening a text file via File Explorer, **justified** as the File Explorer route is the mechanism by which we shall import existing files. This is **related to Problem Breakdown** by the entirety of Algorithm E, **justified** as importing a file will require File Explorer and updates to both the text editor and file hierarchy. To **explain**, the file path is obtained using File Explorer, which is stored in the list of file names and displayed in the file hierarchy to reflect the update of a newly imported file. The text editor is also cleared and reinstated with the file contents (after the addition of in-text line numbers) via the helper function insertFileContents(). To **justify**, the helper function is used to reduce the bulk of importFileClick() which improves readability and maintainability. For further justification, it may be used in other functions in which the file must be updated (or during testing for the quick loading of placeholder file contents), and this avoids code repetition. To **justify** the addition of in-text line numbers, line numbers are necessary to avoid accidental logical/syntax errors as these are always removed when passed to the interpreter, but they are not to be saved in the on-device text files as this hinders backwards compatibility with pre-written pseudocode programs (not built using the IDE).

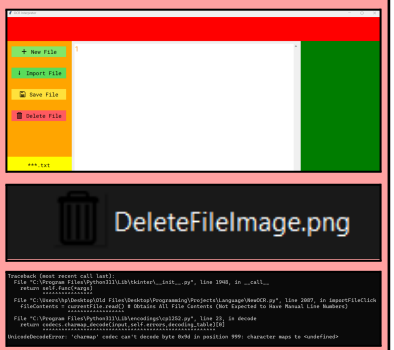
```
# Updates Text Editor With the Contents of a Text File (Pre-Validated to Ensure It Is a Text File)
def insertFileContents(fileContents):
    TextEditor.delete('1.0', tk.END) # Clears Text Editor First So It Can Be Reloaded with New File Contents
    fileLines = fileContents.split('\n') # Splits File Into Lines (Necessary as Line Numbers Are Re-Instated On a Line-By-Line Basis)
    for lineNumber in range(len(fileLines)): # Goes Through Each Line (lineNumber as This is Useful For Text Editor Line Numbers)
        fileLines[lineNumber] = str(lineNumber + 1) + ' ' + fileLines[lineNumber] # Updates Lines to Include Line Number
    fileContents = '\n'.join(fileLines) # Rejoins File Contents Separating Lines With Newline Characters
    TextEditor.insert('1.0', fileContents) # Re-Instates Text Editor With New File Contents

# Uses File Explorer to Obtain a File Existing on the User's Device, Updating the Text Editor and File Hierarchy to Reflect the Update
def importFileClick():
    global textEditorFileLocation # Maintenance: This Will Avoid Scope Errors in the Future if Assignment is Added
    global fileHierarchyObjects # Also Avoids Scope Errors, if Implementation of File List is Changed to Stack, e.g.
    # fileLocation Contains the Complete File Path From the Starting Drive of the File (Necessary for Location Identification in Save)
    # Validation: *.txt Implies Only Text Files Can Be Opened, Necessary as Only Text is Supported by the OCR Syntax
    fileLocation = filedialog.askopenfilename(initialdir = '/Language', title = 'Import File', filetypes = (('Text Files', '*.txt'), ))
    if not fileLocation: # Validation: If The File At That Location Does Not Exist, Further Processing Should Not Occur
        return # Halts Further Processing to Avoid Python FileErrors, Thus Avoiding Accidental Halting
    with open(fileLocation, 'r') as currentFile: # Location Identification Enables the File Being Imported to be Encapsulated
        fileContents = currentFile.read() # Obtains All File Contents (Not Expected to Have Manual Line Numbers)
        insertFileContents(fileContents) # Updates Text Editor to Contain Contents of Imported File
        fileHierarchyObjects.append(fileLocation) # Adds File Location to the File Hierarchy As It Is a Newly Imported File
        displayFileHierarchy() # Updates File Hierarchy to Reflect Change of Newly Imported File
        textEditorFileLocation += 1 # As This is an Additional File to be Moved To, the "Current File Index" Must Be Incremented (Append)
```

Prototype 1 Testing

Test Data	Expected Output	Actual Output
'New File' button clicked	Current contents of text editor cleared, except for first line	Current contents of text editor cleared, except for first line

	number, and ***.txt appended to file hierarchy	number, and ***.txt appended to file hierarchy ✅  

‘Import File’ button clicked with text file attempted to be opened	File explorer opened, with the name of the selected file appended to the file hierarchy and the file contents appearing in the text editor	File explorer opened, with the name of the selected file appended to the file hierarchy and the file contents appearing in the text editor ✅
		 
		`Var1 = 10`
		`for count i = 0 to 10`
		`print( "Hello" )`
		`next`
		 

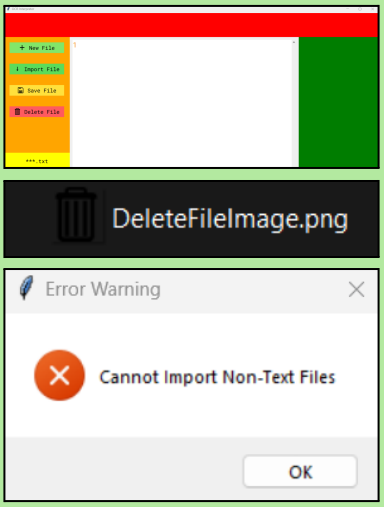
‘Import File’ button clicked, with PNG file attempted to be opened	PNG is not imported, and a warning box displayed stating that only text files can be opened	There are no PNG options, which is correct, but if a PNG file type in the directory is typed manually, it accepts the file and throws an error upon file.read() ❌ 

Prototype 1 Remedial Action

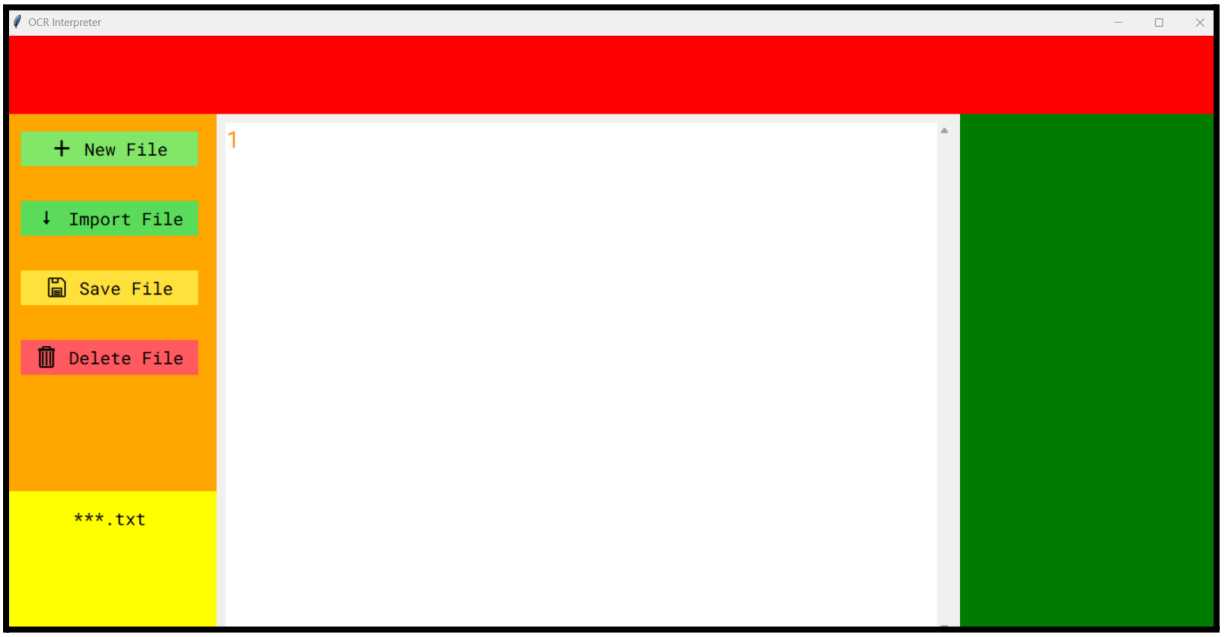
The final test failed as instead of a window pop-up displaying an error warning to the user, a Python error was thrown, which led to program halting. This occurred as though there was built-in text file validation when calling the File Explorer function, the user can get around this by typing in the PNG file's full name (including file extension) manually, thus leading to the File Explorer function returning the path of the PNG file which threw an error when file.read() was called (because read() is not defined for non-text files). Though this error is unlikely to be encountered in practice due to files usually being selected by clicking instead of typing, for robustness this issue should still be fixed. The action that will be taken is to perform file type validation for non-text files after the File Explorer function returns the file path. If a non-text file is found, a warning message will be displayed (fulfilling the success criterion) and further processing stopped to prevent accidental Python errors. This is **justified** particularly because Python errors often lead to the accidental halting of the program, which is undesirable as this makes the user experience less smooth. File type validation immediately after the file path is returned is also ideal as there is no processing in between, thus reducing the likelihood of further errors being caused.

```
if fileLocation.split('.')[-1] != 'txt': # Validation: Non-Text Files Should Not Be Imported in the IDE
    messagebox.showerror('Error Warning' , 'Cannot Import Non-Text Files') # Displays a Warning Box to Notify the User of This
    return # Halts Further Processing to Avoid Python FileErrors, Thus Avoiding Accidental Halting
```

For **full justification**, the previously failed test has now been completed successfully:

'Import File' button clicked, with PNG file attempted to be opened	PNG is not imported, and a warning box displayed states that only text files can be opened	PNG is not imported, and a warning box displayed states that only text files can be opened ✅ 

Prototype 1 Evidence



Prototype 1 Review

At Stage 3 Prototype 1, we created the mechanisms for 'New File' and 'Import File', the former clearing the text editor contents and updating the file hierarchy with an empty, unsaved file and the latter opening a pre-existing, on-device file while updating the file hierarchy and text editor. By implementing each in separate functions, we utilised modularity, which was useful as the separate buttons could access these functions independently without extra conditionals (in determining which button was pressed) and was also useful during unit-testing of each function separately. We also included display functions, which displayed both the initial status of the buttons (called once as they are static) and one that continually updated the file hierarchy, useful as this was used multiple times, thereby greatly reducing code repetition. Overall, considering all test cases passed, including robust test cases that checked for file importing errors (as its functionality is context-dependent unlike 'New File' due to the case of importing non-text files), there is sufficient evidence to state that Stage 3 Prototype 1 has been completed successfully, particularly with the presence of relatively robust validation that occurred to check for non-existent files, erroneous file types, and protected against IndexOutOfBounds (by preventing over-decrementation), meaning we can move on from 'New File' and 'Import File' until post-development testing.

Prototype 2: Save File + Delete File + New File + Import File

First, we shall implement the functionality for the 'Save File' file operation. This is **related to 'Analysis'** by the criterion of 'Save File' saving the file currently in the text editor, **justified** as the functionality will be triggered upon the click of the 'Save File' button. This is **related to Problem Breakdown** by the entirety of Algorithm F, **justified** as the conditional split between 'Save As' and 'Save' will need to be accounted for. To **explain**, masterSaveClick() is a redirection function which activates the 'Save As' or 'Save' pathway depending on whether the text editor file exists or not (whether it has been saved). The 'Save' pathway just updates the on-device file with the text editor contents, and the 'Save As' pathway creates a new on-device file using File Explorer, updating its contents with text editor contents, while changing the file hierarchy to reflect the newly saved file (by changing ***.txt to the file name). To **justify**, a split between 'Save As' and 'Save' must occur as if 'Save' is clicked for a non-existing file, there may be a Python error (unwanted halting) due to attempting to access a file that does not yet exist and if 'Save As' is clicked on an existing file, there would be file overwriting with files that have the same name (or the creation of duplicate files).

Though the latter is error-free, it would be extraneous, thereby being tedious to the user. The justification for using File Explorer, to reiterate, is that it is a recognisable standard and it is consistent with the previous file importing scheme. As such, it is a logical choice that extends upon current functionality, such that the user should easily understand how these operations work.

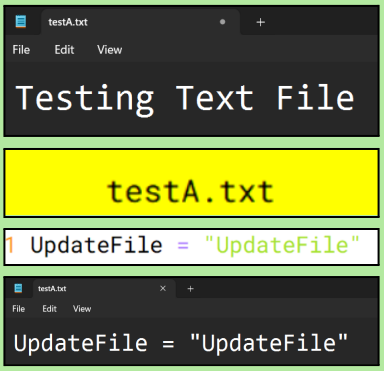
```
# Redirection Function Which Starts Save As or Save Functionality Depending on Saved Status of Text Editor File
def masterSaveClick():
    global textEditorFileLocation # Maintenance: This Will Avoid Scope Errors in the Future if Assignment is Added
    global fileHierarchyObjects # Also Avoids Scope Errors, if Implementation of File List is Changed to Stack, For Example
    if fileHierarchyObjects[textEditorFileLocation] == '***.txt': # Validation: Avoids FileErrors if the Wrong Save
        # Functionality is Applied on a File of a Differing Save Status
        saveAsFileClick() # Unsaved Files Must Be Saved Using 'Save As', As This Leads to the Creation of Name/Path
    else:
        saveFileClick() # Already Saved File Must Just Have Contents and Not Name/Path Updated
```

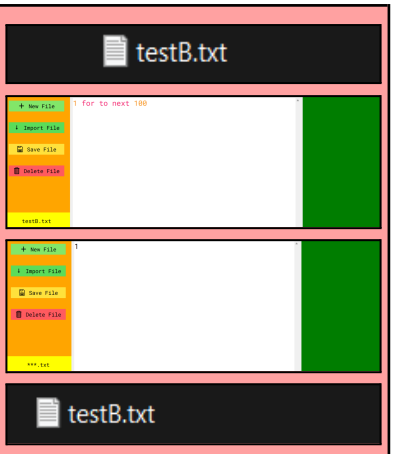
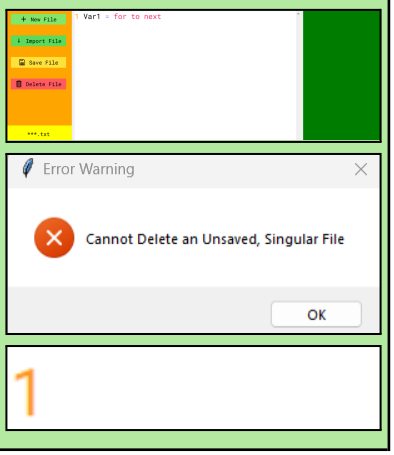
```
# Function that Updates File Contents of a Pre-Saved File With That of the Text Editor
def saveFileClick():
    global textEditorFileLocation # Maintenance: This Will Avoid Scope Errors in the Future if Assignment is Added
    global fileHierarchyObjects # Also Avoids Scope Errors, if Implementation of File List is Changed to Stack, For Example
    fileLocation = fileHierarchyObjects[textEditorFileLocation] # Obtains Location of File In Text Editor (Being Saved)
    with open(fileLocation, 'w') as currentFile: # Write Mode as Contents Will Be Updated (From Scratch)
        fileLines = TextEditor.get('1.0', tk.END).split('\n')[:-1] # Avoids Last Line Becase Splicing May Lead to Index Errors (Halting)
        for lineNumber in range(Len(fileLines)): # lineNumber As This Is Needed for List Indexing To Update Values
            fileLines[lineNumber] = fileLines[lineNumber][2:] # Removes Line Number of Each Line Before Writing (Backwards Compatability)
        currentFile.write('\n'.join(fileLines)) # Writes All Contents after Joining Lines by Newline Character (Visually Consistent)

# Function That Copies Text Editor Contents to New On-Device File and Updates File Hierarchy With New Name
def saveAsFileClick():
    global textEditorFileLocation # Maintenance: This Will Avoid Scope Errors in the Future if Assignment is Added
    global fileHierarchyObjects # Also Avoids Scope Errors, if Implementation of File List is Changed to Stack, For Example
    # fileLocation Contains the Complete File Path From the Starting Drive of the File (Necessary for Location Identification in Save)
    # Validation: *.txt Implies Only Text Files Can Be Saved, Necessary as Only Text is Supported by the OCR Syntax
    fileLocation = filedialog.asksaveasfilename(defaultextension = '.txt', initialdir = '/Language', title = 'Save File', filetypes = (('Text Files',
    if not fileLocation: # Validation: If The File At That Location Does Not Exist, Further Processing Should Not Occur
        return # Halts Further Processing to Avoid Python FileErrors, Thus Avoiding Accidental Halting
    with open(fileLocation, 'w') as currentFile: # Write Mode as Contents Will Be Updated (From Scratch) (Also Creates File)
        fileLines = TextEditor.get('1.0', tk.END).split('\n')[:-1] # Avoids Last Line Becase Splicing May Lead to Index Errors (Halting)
        for lineNumber in range(Len(fileLines)): # lineNumber As This Is Needed for List Indexing To Update Values
            fileLines[lineNumber] = fileLines[lineNumber][2:] # Removes Line Number of Each Line Before Writing (Backwards Compatability)
        currentFile.write('\n'.join(fileLines)) # Writes All Contents after Joining Lines by Newline Character (Visually Consistent)
        fileHierarchyObjects[textEditorFileLocation] = fileLocation # As the Path is Updated, the File List Should Be Updated With That Path
        displayFileHierarchy() # Updates File Hierarchy to Reflect Change of Newly Saved File
```

Next, we shall implement the functionality for the Delete File' file operation. This is **related to 'Analysis'** by the criterion of 'Delete File' deleting the file currently opened in the text editor, **justified** as we shall implement 'Delete File' relatively only to the text editor file. This is **related to Problem Breakdown** by the entirety of Algorithm G, **justified** as this includes all removals from the file hierarchy and the text editor as well as the deletion of the on-device file. To **explain**, generally decrementation of the file hierarchy occurs with the deletion of the final element (representing the file currently stored in the text editor), with that file being removed from the device (given its location stored in fileHierarchyObjects) and the text editor cleared to represent the file being removed. However, when there is a single file in the text editor, decrementation does not occur, instead either leading to a warning box if the deletion of a singular, unsaved file is attempted, or a conversion from a file name to a '***.txt' in the file hierarchy if it is saved (and the standard deletion procedure described above). To **justify**, decrementation does not occur when there is a single file as otherwise the index would fall to -1 (may lead to Python IndexOutOfBounds Errors, so unwanted halting, or the deletion of the incorrect file down the line, which may lead to the loss of sensitive work). The validation for the unsaved file is necessary as if attempting to delete a file that does not exist on the device, Python FileErrors may lead to unwanted halting, worsening the user experience.

```
# Function That Deletes Text Editor File If It Exists, Updating the
def deleteFileClick():
    global textEditorFileLocation # Maintenance: This Will Avoid Scope Errors in the Future if Assignment is Added
    global fileHierarchyObjects # Also Avoids Scope Errors, if Implementation of File List is Changed to Stack, For Example
    TextEditor.delete('1.0', tk.END) # Clears Text Editor to Reinstate With Empty File (As Current One Is Being Deleted)
    TextEditor.insert('1.0', '1 ') # Adds First Line Number: Necessary to Avoid Syntax Errors During Interpretation
    if len(fileHierarchyObjects) == 1: # If Only 1 File Exists, This Case Is Handled Differently (Without Decrementation)
        if fileHierarchyObjects[0] == '***.txt': # Validation: Cannot Delete an Unsaved, Singular File
            messagebox.showerror('Error Warning', 'Cannot Delete an Unsaved, Singular File') # Displays Warning Box to User
            return # Halts Further Processing To Avoid Python Errors (With Indexing)
        fileLocation = fileHierarchyObjects[0] # 0 == textEditorFileLocation In This Case (Optimisation)
        fileHierarchyObjects[0] = '***.txt' # If The File Was Saved, Then Can Clear Its Name By Making it ***.txt (Default Empty)
        os.remove(fileLocation) # Removes the File Permanently From the Device (Use With Caution)
        displayFileHierarchy() # Updates File Hierarchy to Reflect Change of Newly Deleted File
        return # Halts Further Processing To Avoid Index Decrementation (and Python OutOfBoundsErrors)
    if fileHierarchyObjects[textEditorFileLocation] == '***.txt': # Non-Singular, Unsaved Files Can Be Removed From Hierarchy
        textEditorFileLocation -= 1 # Decrements Index to Point to Previous File
        fileHierarchyObjects.pop() # Deletes Empty Text File (pop() Removes Final Index)
        return # Halts Further Processing To Avoid Further Decrementation of Index
    textEditorFileLocation -= 1 # Decrements Index to Point to Previous File
    fileHierarchyObjects.pop() # Deletes File Name From Hierarchy (pop() Removes Final Index)
```

Prototype 2 Testing

‘Save File’ button clicked with pre-saved file in the text editor	When the on-device file is opened, the contents are updated to the state when the button was clicked	When the on-device file is opened, the contents are updated to the state when the button was clicked ✅ 
‘Save File’ button clicked with unsaved file in the text editor	File explorer opened, and a file with identical contents to that in the text editor is stored in the input location, with ***.txt updated	File explorer opened, and a file with identical contents to that in the text editor is stored in the input location, with ***.txt updated ✅ 
‘Delete File’ button clicked with pre-saved file in the text editor	On-device version of the file in the text editor is removed from its location and file hierarchy cleared, and the text editor is cleared	File hierarchy and text editor is cleared, but on-device version of the file is not removed from its location ❌

		
‘Delete File button clicked with unsaved file in the text editor	No file is deleted. A warning box is displayed stating that an unsaved file cannot be deleted, but the text editor is cleared	No file is deleted. A warning box is displayed stating that an unsaved file cannot be deleted, but the text editor is cleared ✅ 

Prototype 2 Remedial Action

The third test failed as instead of deleting the on-device file from its location, the file remained after clicking the ‘Delete File’ button. This occurred as though we managed to remove the file from the file hierarchy, such that it was no longer referenced anywhere in the IDE, we did not explicitly remove the file from the computer system. The action that will be taken is to use the ‘os’ module, given the known file location, to remove the file permanently from the system. This is **justified** as it completes the functionality mentioned in the Success Criteria and desired by stakeholders, which is that ‘Delete File’ produces permanent on-device deletion. There naturally are security concerns (e.g. loading an important file and deleting it using the IDE accidentally or maliciously), but there is validation (for example, the “Only text files” rule) that prevents the deletion of non-text files, which may be important for system operations).

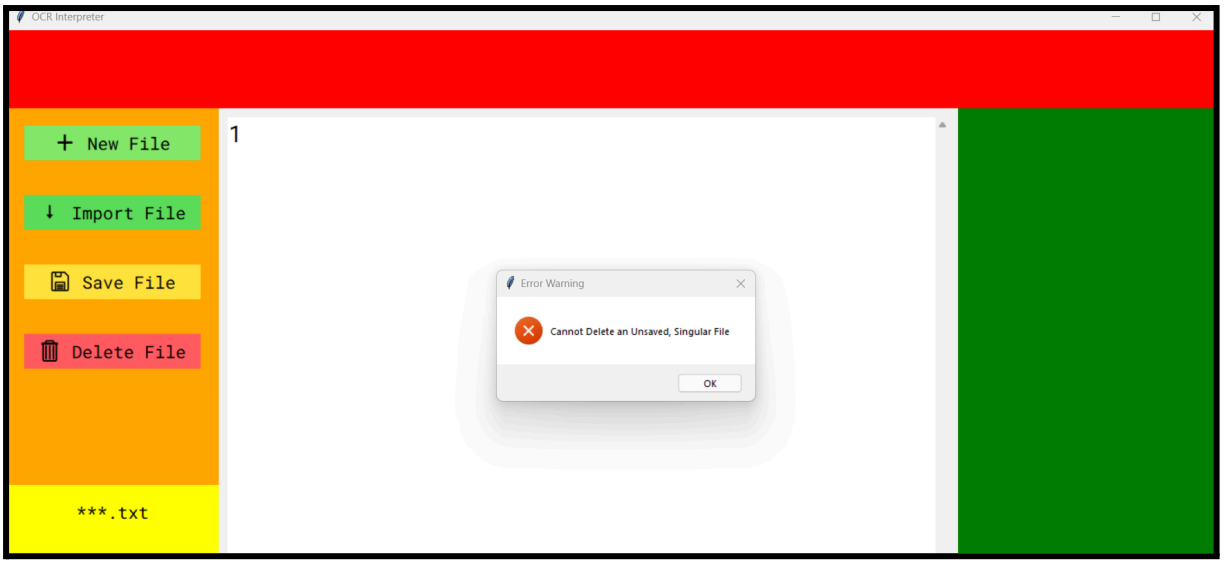
```
fileLocation = fileHierarchyObjects[textEditorFileLocation] # For a Non-Singular, Saved File, Gets Its Location
os.remove(fileLocation) # Permanently Removes That File From the Device
```

For **full justification**, the previously failed test has now been completed successfully:

‘Delete File’ button clicked with pre-saved file in the text	On-device version of the file in the text editor is removed	File hierarchy and text editor is cleared, but on-device version

editor	from its location and file hierarchy cleared, and the text editor is cleared	of the file is not removed from its location ✅ 

Prototype 2 Evidence



Prototype 2 Review

At Stage 3 Prototype 2, we created the mechanisms for 'Save File' and 'Delete File', the former being separated into two functions being the 'Save' pathway, which just updated the contents of the file of known location with that of the text editor, and the 'Save As' pathway, which updated the contents of a newly created on-device file (because an unsaved file was being saved) with that of the text editor, while updating the file hierarchy to reflect the new name of the file and the file hierarchy list to reflect the new location of the file, and the latter ('Delete File) which cleared text editor contents, removed the final element from the file hierarchy list (representing the file currently open in the text editor) and moved the index to the appropriate previously open file. Furthermore, there was successful validation in both parts, such as file deletion not occurring for a singular, unsaved file (as this could lead to over-decrementation), ensuring robustness, while the modularity of separate functions for each operation type once again enabled buttons to avoid extraneous conditionals and improved readability. Overall, considering all test cases passed successfully, there is sufficient evidence to state that Stage 3 Prototype 2 was successful, and we can move on to Stage 4 (and return for post-development testing).

Stage 3 Evidence of Validation for All Key Elements

New File + Import File Validation

Type Check: Ensures that if attempting to import a non-text file, no further processing occurs, the file is not imported, and a warning message is displayed so the user knows of the error. Validation occurs inside importFileClick().

```
if fileLocation.split('.')[-1] != 'txt': # Validation: Non-Text Files Should Not Be Imported in the IDE
    messagebox.showerror('Error Warning' , 'Cannot Import Non-Text Files') # Displays a Warning Box to Notify the User of This
    return # Halts Further Processing to Avoid Python FileErrors, Thus Avoiding Accidental Halting
```

Presence Check: Ensures that if the file attempted to be imported does not actually exist in that location (e.g. if manually typing a file incorrectly), further processing halts, thus avoiding Python FileErrors (and unwanted halting). Validation occurs inside importFileClick().

```
if not fileLocation: # Validation: If The File At That Location Does Not Exist, Further Processing Should Not Occur
    return # Halts Further Processing to Avoid Python FileErrors, Thus Avoiding Accidental Halting
```

Type Check: Pre-validation for importing files, ensuring that all files clickable (those that are displayed to the user) are only text files, meaning that the only way to attempt to import a non-text file is via manual typing, thus reducing its likelihood. Validation occurs inside importFileClick().

```
# Validation: *.txt Implies Only Text Files Can Be Opened, Necessary as Only Text is Supported by the OCR Syntax
fileLocation = filedialog.askopenfilename(initialdir = '/Language', title = 'Import File', filetypes = (('Text Files', '*.txt'), ))
```

Save File + Delete File Validation

Format Check: Ensures that file hierarchy objects with value '***.txt' are handled using 'Save As' as they are unsaved. This avoids Python FileErrors in accessing non-existent files at invalid locations. Validation occurs inside masterSaveClick().

```
if fileHierarchyObjects[textEditorFileLocation] == '***.txt': # Validation: Avoids FileErrors if the Wrong Save
    # Functionality is Applied on a File of a Differing Save Status
    saveAsFileClick() # Unsaved Files Must Be Saved Using 'Save As', As This Leads to the Creation of Name/Path
```

Presence Check: Ensures that if attempting to save an unsaved file in a non-existent location (e.g. entering a non-existent location in the path), further processing halts, thus avoiding Python FileErrors (and unwanted halting). Validation occurs inside saveAsFileClick().

```
if not fileLocation: # Validation: If The File At That Location Does Not Exist, Further Processing Should Not Occur
    return # Halts Further Processing to Avoid Python FileErrors, Thus Avoiding Accidental Halting
```

Format Check: Ensures that if there is a singular file with name '***.txt' in the file hierarchy (representing an unsaved file) and 'Delete File' is clicked, no further processing occurs (to avoid over-decrementation of index of files and Python OutOfBounds errors). Validation occurs inside deleteFileClick().

```
if fileHierarchyObjects[0] == '***.txt': # Validation: Cannot Delete an Unsaved, Singular File
    messagebox.showerror('Error Warning', 'Cannot Delete an Unsaved, Singular File') # Displays Warning Box to User
    return # Halts Further Processing To Avoid Python Errors (With Indexing)
```

Stage 3 Summary Review

At Stage 3 Prototype 1, we created the mechanisms for 'New File' and 'Import File', the former clearing the text editor contents and updating the file hierarchy with an empty, unsaved file and the

latter opening a pre-existing, on-device file while updating the file hierarchy and text editor. By implementing each in separate functions, we utilised modularity, which was useful as the separate buttons could access these functions independently without extra conditionals (in determining which button was pressed) and was also useful during unit-testing of each function separately. We also included display functions, which displayed both the initial status of the buttons (called once as they are static) and one that continually updated the file hierarchy, useful as this was used multiple times, thereby greatly reducing code repetition. Overall, considering all test cases passed, including robust test cases that checked for file importing errors (as its functionality is context-dependent unlike 'New File' due to the case of importing non-text files), there is sufficient evidence to state that Stage 3 Prototype 1 has been completed successfully, particularly with the presence of relatively robust validation that occurred to check for non-existent files, erroneous file types, and protected against IndexOutOfBounds (by preventing over-decrementation), meaning we can move on from 'New File' and 'Import File' until post-development testing.

At Stage 3 Prototype 2, we created the mechanisms for 'Save File' and 'Delete File', the former being separated into two functions being the 'Save' pathway, which just updated the contents of the file of known location with that of the text editor, and the 'Save As' pathway, which updated the contents of a newly created on-device file (because an unsaved file was being saved) with that of the text editor, while updating the file hierarchy to reflect the new name of the file and the file hierarchy list to reflect the new location of the file, and the latter ('Delete File) which cleared text editor contents, removed the final element from the file hierarchy list (representing the file currently open in the text editor) and moved the index to the appropriate previously open file. Furthermore, there was successful validation in both parts, such as file deletion not occurring for a singular, unsaved file (as this could lead to over-decrementation), ensuring robustness, while the modularity of separate functions for each operation type once again enabled buttons to avoid extraneous conditionals and improved readability. Overall, considering all test cases passed successfully, there is sufficient evidence to state that Stage 3 Prototype 2 was successful, and we can move on to Stage 4 (and return for post-development testing).

Stage 4: Console + File-Handling Window + Text Editor + Interpreter

Prototype 1: Input + Output

First, the 'Run' button must be loaded and its click attached to a function that executes the text editor code (as input to the interpreter). This is **related to 'Analysis'** by the requirement of the console having input/output capabilities, **justified** as the 'Run' button provides the only link between the interpreter and the console (and therefore the console's input and output mechanisms). This is **related to Problem Breakdown** by the 'Load Buttons' process of Algorithm A, **justified** as the 'Run' button is being loaded. To **explain**, the 'Run' button's positioning is carried out using helper functions created previously in Stage 3, such as getFileImage(). During runClick(), which is activated upon the click of the 'Run' button, a local 'Execute File' is created which stores the text editor contents after removal of the in-editor line numbers and the whitespace surrounding opened/closed brackets. The 'Execute File' is then used by the Lexer, which creates the token list, and moves on to the Parser and Node Interpreter for final execution. To **justify**, using helper functions during button positioning reduces the bulk of the code, improving readability, while applying the same button design leads to aesthetic consistency, which may improve the user experience. To **justify** the replacement of the bracket-surrounding whitespace, the Lexer does not disregard this whitespace as irrelevant due to the way it handles brackets (expecting a character), thus this avoids an infinite loop, thus improving the smoothness of the IDE experience.

```
# Function that Runs the Text Editor Content on the Interpreter
def runClick():
    with open('execute.txt', 'w') as currentFile: # Write Mode as Contents Will Be Updated (From Scratch) (Creates Execution File)
        fileLines = TextEditor.get('1.0', tk.END).split('\n')[:-1] # Avoids Last Line Becase Splicing May Lead to Index Errors (Halting)
        for lineNumber in range(len(fileLines)): # lineNumber As This Is Needed for List Indexing To Update Values
            if '\t' in fileLines[lineNumber]:
                tempContents = fileLines[lineNumber][1:]
                tempContents = tempContents.replace('\t', '  ')
            else:
                tempContents = fileLines[lineNumber][2:] # Removes Line Numbers From Lines Before Loading to Execute File
            tempContents = tempContents.replace('( ', '(') # Function/Procedure Calls Do Not Accept Whitespace Before and After
            tempContents = tempContents.replace(' )', ')') # Open and Closing Brackets, So the Execution File Must Remove Them
            fileLines[lineNumber] = tempContents # Removes Line Number of Each Line Before Writing (Backwards Compatability)
        currentFile.write('\n'.join(fileLines) + '\n') # Writes All Contents after Joining Lines by Newline Character (Visually Consistent)
    tokenList = Lexer('execute.txt').getTokenList() # Begins the Lexing Process for Node Interpretation of Text Editor Input
```

```
RunImage = getFileImage('RunImage.png') # Maintenance: Global as Tkinter Imaging Requires a Global Reference to Avoid Garbage Collection
def packRunButton():
    # Button Associated With Run Click to Begin Execution as Soon as It Is Pressed
    # Background Consistent With Light Mode Scheme (In Case Colour Themes are Not Updated on Start-Up)
    RunButton.pack(side = 'left', pady = 20, expand = False, padx = 200) # expand = True to Centre Button Horizontally
packRunButton() # Calls the Function to Load Button on Screen (Called Once Because Button is Static)
```

Next, we shall handle input functionality. This is **related to 'Analysis'** by the requirement of the console having input/output functionality, **justified** as the implementation of the input() function will be here. This is **related to Problem Breakdown** by the 'Accept User Input' process of Algorithm M, **justified** as input() will accept input from what is written in the console post-prompt. To **explain**, the console is initially disabled (only readable), but is activated when input() is called by the Node Interpreter, where the prompt is validated and set up, so the user may see the prompt before entering the input. The input is then entered after a Zero-Width-Character, and after the key release of Return/Enter, handleConsoleInput() is called which accepts the input and re-sends it to the Node Interpreter (by restarting execution, this time with an actual value of mainConsoleInput) where the string value is then passed to the parent node of the input function call so further processing can occur with the string. To **justify**, the console is initially disabled to prevent accidental functionality of an input being accepted without input() being called (avoiding Python errors and program halting). To **justify** the Zero-Width-Character, this allows the splitting of the input into the prompt and the inputted value, from which the inputted value can be obtained for further processing without being intrusive as the Zero-Width-Character is not visible.

```
mainConsoleInput = None # Used for Conditional Input Functionality (Either Setting Up Prompt or Returning Input)
# Function Called Upon Return-Release in Console to Signify that a Value Has Been Input
def handleConsoleInput(event):
    global mainConsoleInput # Maintenance: As Assignment is Done, Necessary to Avoid Local Python Assumption
    Console.configure(foreground = ConsoleStandardFontColour) # Resets Font Colour to Black in Case an Error was Ouput Immediately Prior
    if Console.cget('state') == tk.DISABLED: # Validation: Only Allows the Acceptance of Input if the Console is Enabled
        return # Halts Further Processing (visitInput Should Have Left This Enabled, Alternative is Incorrect Functionality)
    consoleInput = Console.get('1.0', '2.0')[:-1].split(u'\u200c')[1][1:] # Zero-Width Character Used to Differentiate Between
    # Prompt and Inputted Value (This Obtains Input to Use for Processing)
    mainConsoleInput = consoleInput # Input Set Globally so Input Node Visit Function Can Receive it For Return to Parent Node
    Console.delete('1.0', tk.END) # Clears Text Editor After Input Received as There Is No Further Interaction Necessary Here
    Console.configure(state = tk.DISABLED) # Disables Console as After Input is Complete, No More Interactions in Pipeline
    runClick() # Resets Execution, But With Input Pre-Set So Program Continues Past Input With Value
Console.bind('<KeyRelease-Return>', handleConsoleInput) # Binds Input Handling to Return (User Presses Enter to Enter Input)
```

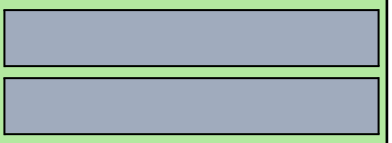
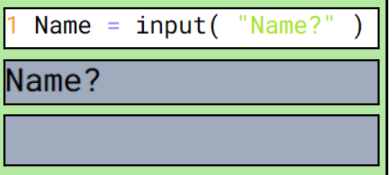
```
# Visits FunctionCallNode to Execute Function, Returning Value
def visitFunctionCallNode(self, node):
    global mainConsoleInput # Maintenance: As Assignment is Done, Must Be Defined Global, Otherwise Assumed Local
    if node.nameToken.value == 'input': # Checks if Built-In Function input() is Chosen
        if mainConsoleInput is None: # Handles Situation When Input Has Just Been Activated (Value Not Yet Inputted)
            if len(node.parameterNodeList) > 1: # Validation: Input Can Only Have One Argument
                return Error('n/a', 'TypeError', f'Invalid Argument Count') # No Line Number as Interpreter Does Not Have Access
            if len(node.parameterNodeList) == 0: # Validation: Only Calls Visit on Node and Not Empty None Object
                prompt = '' # Therefore Avoids Python Error as Visit is Not Defined for None Type Objects
            else: # Handles Case When a Node Exists for the Input Parameter (Singular)
                prompt = self.visit(node.parameterNodeList[0]) # Obtains prompt Value By Visiting What Should Return String
            if type(prompt).__name__ != 'String': # Validation: If the Input prompt is Non-String, This is Against Syntax
                return Error('n/a', 'TypeError', f'Expects String Argument') # TypeError Thrown as a String is Expected
            Console.configure(state = tk.NORMAL) # Enables Console so Input Can Be Received
            Console.configure(foreground = ConsoleStandardFontColour) # Sets Foreground to Black (To Indicate Normal, Non-Error Input Function)
            Console.delete('1.0', tk.END) # Clears Console to Remove Previous Program Outputs (Unrelated)
            Console.insert('1.0', prompt.value + u'\u200c' + ' ') # Zero-Width Character to Enable Splitting of Input Line
            return
            # To Distinguish Between Prompt and Inputted Value to Obtain Input for Further Functionality
        else:
            returnString = mainConsoleInput # mainConsoleInput Now Will Contain Value by handleConsoleInput() (Post-Return)
            mainConsoleInput = None # Resets mainConsoleInput so Future Input Statements Work From Scratch (Correctly)
            return String(returnString) # Returns String of Inputted Value For Processing in Rest of Program
```

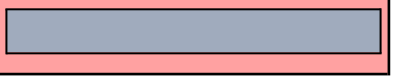
Next, we shall handle print functionality. This is **related to 'Analysis'** by the criterion of the console having input/output capabilities, **justified** as outputting to the console is the function of print(). This

is **related to Problem Breakdown** by the 'Display Output On Console' process of Algorithm M, **justified** as when print() is called (given parameters), the parameters will be displayed on the console section of the GUI. To **explain**, the output is built up by iterating through the parameters one-by-one, then clearing the console and reinstating it with the output built up during the print() function processing in the Node Interpreter. To **justify**, the output is built up one-by-one as each parameter is a separate node and the methods available to the Interpreter class are done on a node-by-node basis. Further **justification** is that it enables validation for each node, which is useful for error identification. The **justification** for clearing the console is that the previous output will be irrelevant to the current output (for example, the output is separate to the input to a previous input() function), so it acceptable, while clearing space to avoid long strings of simultaneous print(), input(), and error messages, thus being aesthetically suitable and simple (for usability as simplicity promotes understandability).

```
# Visits ProcedureCallNode, Called Standalone in the Node Interpreter
def visitProcedureCallNode(self, node):
    if Console.cget('state') == tk.NORMAL:
        return
    if node.nameToken.value == 'print': # Checks for the Built-In Function print() For Console Output
        printString = '' # printString is Initially Set to Empty as It Will Be Built By Iterating Over Parameters
        for arg in node.parameterNodeList: # Goes Through Each Parameter In Order
            nodeValue = self.visit(arg) # This Returns the Value of Item to Output (1 Parameter)
            if type(nodeValue).__name__ == 'Error': # Validation: Checks If There Was An Error In Processing a Parameter
                return Error('n/a', 'SyntaxError', 'Invalid Parameter Format') # If So, Program Halter to Avoid Accidental Halting
            printString += nodeValue.toString().value # If Valid, Adds String to printString For Output to Console
        Console.configure(state = tk.NORMAL) # Enables Console so Error Can Be Ouput On Screen
        Console.configure(foreground = ConsoleStandardFontColour) # Sets to Black In Case Error Was Previously Output
        Console.configure(state = tk.DISABLED) # Disables Console To Prevent Accidental Deletion of Program Output
        return # Halts Further Processing to Not Mistake print() For a Custom Function
```

Prototype 1 Testing

Test Data	Expected Output	Actual Output
Attempt to type when console is not in Input mode	Nothing should be written to the console	Nothing was written to the console ✅ 
Name = input("Name?")	"Name?" should be output to console, and after typing in the input, the console should clear	"Name?" output to console, and after typing "Name", console was cleared ✅ 
print("Hello")	"Hello" should be output to console	Nothing output to console ❌ 
print(1)	"1" should be output to console	Nothing output to console ❌ 
Name = input("Name?")	Output should be "Hello" +	Nothing output to console ❌

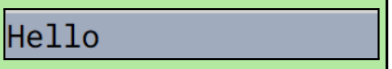
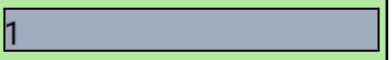
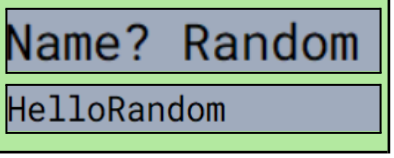
print(“Hello ” + Name)	input to ‘Name? ‘	

Prototype 1 Remedial Action

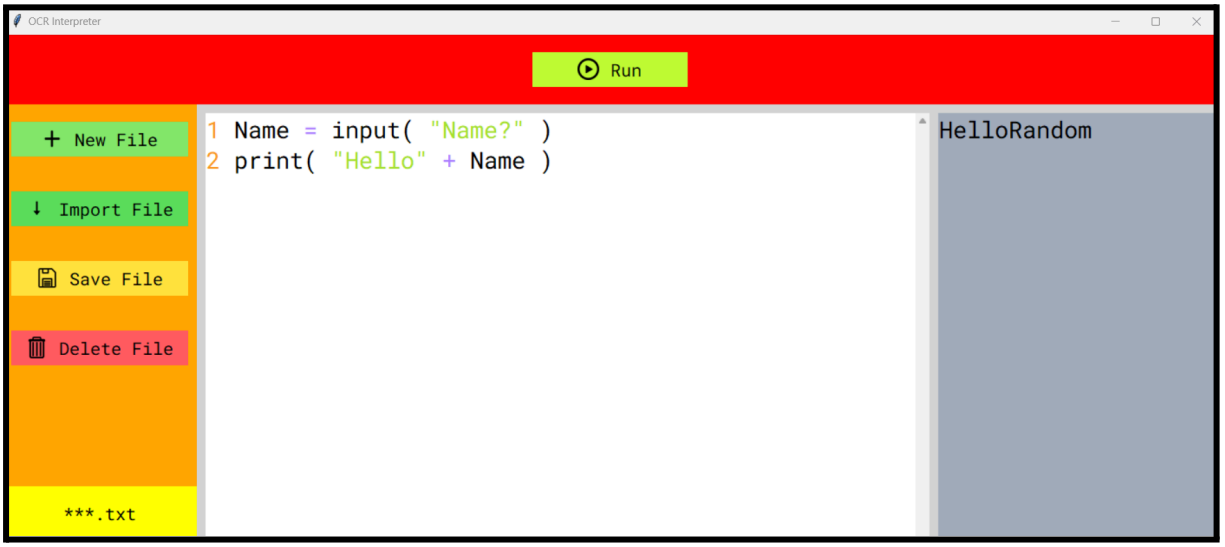
Tests 3, 4, and 5 failed as instead of outputting “Hello”, “1”, and (Input to “Name?”) respectively, nothing was output to the console, including no error messages in the terminal indicating that there was an interpreter-discovered error between start-up and print() execution. This occurred as during the print() section of visitProcedureCallNode(), though the console was cleared and formatting applied for displaying printString (string to be printed), the insert method on the Console object was not called. The action that will be taken is to therefore call the insert() method, ideally at tk.END as this enables the stacking of print outputs, meaning that multiple outputs of print statements can be seen simultaneously. This action is **justified** as insert() is the standard text insertion method of Tkinter text objects, and is therefore reliable. The action of stacking print outputs is further **justified** as there can conceivably be multiple print statements in a program (e.g. printing in a for loop), so multiple print call outputs should be supported.

```
if Console.cget('foreground') != 'black': # Enables Print Stacking Except For Errors (Should Not Exist Simultaneously)
    Console.delete('1.0', tk.END) # Clears Console As Previous Interactions are Irrelevant (Unrelated)
Console.insert(tk.END, printString + '\n') # Inserted at End as The Final Print Call Will Be Printed Finally
```

For **full justification**, the previously failed tests have now been completed successfully:

print(“Hello”)	“Hello” should be output to console	“Hello” output to console ✅ 
print(1)	“1” should be output to console	“1” output to console ✅ 
Name = input(“Name?“) print(“Hello ” + Name)	Output should be “Hello” + input to ‘Name? ‘	“Random” output to console after inputting “Random” for “Name?” ✅ 

Prototype 1 Evidence



Prototype 1 Review

At Stage 4 Prototype 1, we created the mechanisms for console inputting/outputting (for valid statements), the former accepting input and returning it to the node interpreter and the latter being an independent operation which just outputted to the console. We did maintain modularity by implementing each operation inside separate functions, particularly with a complicated but readable conditional within the input() function which required re-interpreting to continue with, but this was successful under robust test cases. This modularity was useful for unit-testing, and will prove useful during maintenance (for debugging purposes and general readability). Overall, considering all test cases were passed successfully, with the test cases themselves being suitably robust as they tested output for multiple data types within print (going beyond just strings), thus also testing string conversion operations, with further foundation being provided by strict validation to ensure that console interactions for input/output are highly limited (only to their respective function calls), thus avoiding accidental deletion and unexpected user-based errors, there is sufficient evidence that Stage 4 Prototype 1 has been completed successfully and we can move on from it until post-development testing.

Prototype 2: Error Output + Input + Output

Now, we shall implement the functionality for error outputs (when errors are discovered during the interpretation process). This is **related to 'Analysis'** by the criterion of the console producing error messages with type and line number, **justified** as these are components of an Error object (as defined during Stage 1) and will be contained in its string representation. This is **related to Problem Breakdown** by the 'Display Error on Console' process of Algorithm M, **justified** as the string representation of the error is ultimately output on the console. To **explain**, if there is an error during the Lexing stage, it does not return a valid token list, but an Error object including the line number in which the error occurred and the type of error. As such, this information is checked at the end of Lexing, and if there is an error, the line number is decremented and the console cleared, updated, and re-disabled. In the case of the interpreter, each visit function has error checking at points of assignment where other visit methods are used. To **justify**, the line number is decremented as due to the additional newline added before the Lexer (to check first-line indentation), line numbers connected to errors, by default, are one too high (so they must be lowered by 1). To justify only validating for visit methods, this is because each visit method is an independent function, so just returning the error upwards will not necessarily return it back to the main runClick() function, so in-place error checking must occur.

```
AST = Parser(tokenList).parseProgram()
if type(AST).__name__ == 'Error': # Validation: Only Passes AST to Interpreter if No Errors During Parsing Stage
    if isinstance(AST.lineNumber, int): # Validation: Only Reduces Line Number if Valid Line Number Available
        AST.lineNumber -= 1 # Due to Initial Newline, lineNumber is Always Overshot by 1
    errorString = repr(AST) # Error Message Pre-Defined as String Representation of Error Class
    Console.configure(state = tk.NORMAL) # Enables Console to Allow Error Message to be Output
    Console.configure(foreground = ConsoleErrorFontColour) # Sets Font Colour to Error Colour as Defined in GUI
    Console.delete('1.0', tk.END) # Clears Console as Errors are Standalone (Not Paired With Valid Output)
    Console.insert('1.0', errorString) # Displays Error Message to User For Debugging Purposes
    Console.configure(state = tk.DISABLED) # Disables Console To Avoid Accidental Deletion of Error
    return # Halts Further Processing as Errors Are Unlikely to Be Processed Validly by Interpreter
else:
    Interpreter().visit(AST) # Interpreter Error Validation Occurs Inside Each Visit Method
```

```
nodeValue = self.visit(arg) # This Returns the Value of Item to Output (1 Parameter)
if type(nodeValue).__name__ == 'Error': # Validation: Errors are Handled Seldom to Avoid Accidental Halting
    Console.configure(state = tk.NORMAL) # When Accessing Non-Existent Error Attributes
    Console.configure(foreground = ConsoleErrorFontColour) # Sets Foreground to Error Colour if Non-Errors Output Before
    Console.delete('1.0', tk.END) # Clears Console As Previous Interactions Are Not Part of This Error
    Console.insert('1.0', nodeValue) # Inserts Error Message to Screen for User Debugging
    Console.configure(state = tk.DISABLED) # Disables Text Editor Afterwards to Prevent Accidental Deletion of Error
    return # Halts Further Processing to Avoid Accidental Halting When Accessing Error Attributes Incorrectly
```

Prototype 2 Testing

Test Data	Expected Output	Actual Output
Var1 ! 3	SyntaxError	Console Not Updated ❌
3 = Var1	SyntaxError	SyntaxError ✅ SyntaxError at Line Number 1: Assignment Syntax Invalid
Var1 = 2 + True	TypeError	TypeError ✅ TypeError at Line Number n/a: Integer & Boolean Invalid Operands For +
Var1 = Var2 + 3	NameError	NameError ✅ NameError at Line Number n/a: Unrecognised Identifier

Prototype 2 Remedial Action

The first test failed as instead of outputting a Syntax Error (due to the presence of the invalid token '!'), nothing was output to the console. This occurred as during error checking while interpreting, though this was done post-parsing and mid-node-interpretation, it was not done post-lexing. The action that will be taken therefore is to add the same sequence of decrementing the error line

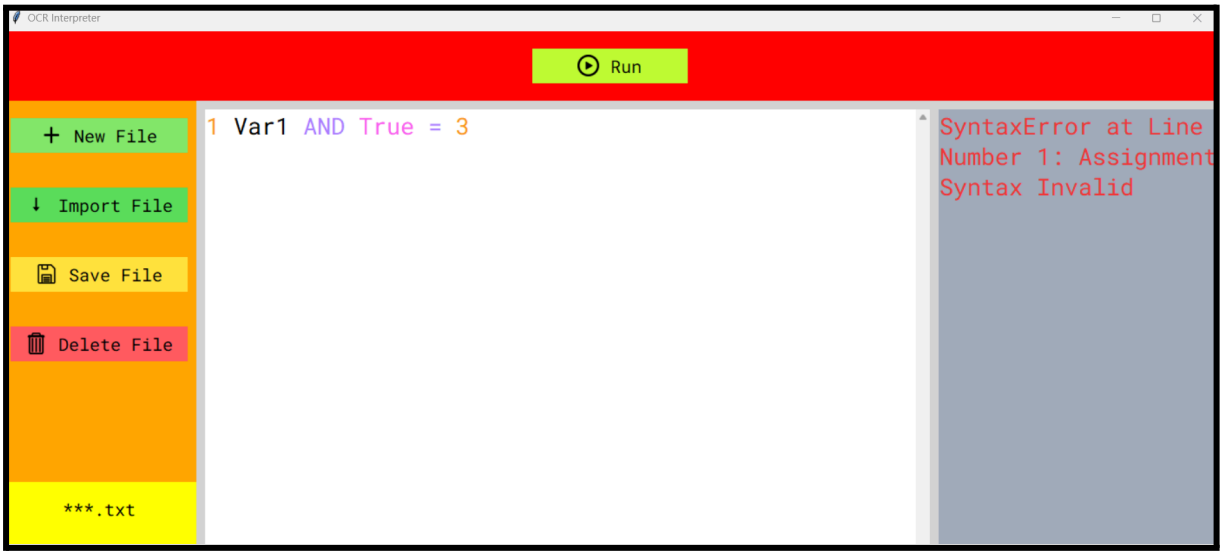
number post-conditional to check if there is an error post-lexing, and outputting that error to the console, then disabling it to prevent further user interaction. This action is **justified** as it is highly readable, considering all error outputs for the Lexer needs to only happen at a single location (post-lexing) and is therefore maintainable. The **justification** for disabling the console afterwards is to prevent the accidental deletion of error messages (as they are unrecoverable in this case), thus enabling proper debugging and satisfying that stakeholder preference.

```
tokenList = Lexer('execute.txt').getTokenList() # Begins the Lexing Process for Node Interpretation of Text Editor Input
if type(tokenList).__name__ == 'Error': # Validation: Only Passes tokenList to AST if No Errors During Lexing Stage
    if isinstance(tokenList.lineNumber, int): # Validation: Only Reduces Line Number if Valid Line Number Available
        tokenList.lineNumber -= 1 # Due to Initial Newline, lineNumber is Always Overshot by 1
    errorString = repr(tokenList) # Error Message Pre-Defined as String Representation of Error Class
    Console.configure(state = tk.NORMAL) # Enables Console to Allow Error Message to be Output
    Console.configure(foreground = ConsoleErrorFontColour) # Sets Font Colour to Error Colour as Defined in GUI
    Console.delete('1.0', tk.END) # Clears Console as Errors are Standalone (Not Paired With Valid Output)
    Console.insert('1.0', errorString) # Displays Error Message to User For Debugging Purposes
    Console.configure(state = tk.DISABLED) # Disables Console To Avoid Accidental Deletion of Error
    return # Halts Further Processing as Errors Are Unlikely to Be Processed Validly by Interpreter
else:
```

For **full justification**, the previously failed test has now been completed successfully:

Var1 ! 3	SyntaxError	SyntaxError ✅ InvalidTokenError at Line Number 1: ! Unrecognised Token

Prototype 2 Evidence



Prototype 2 Review

At Stage 4 Prototype 2, we created the mechanism for error outputting to the console, which involved the transfer of Error objects upwards through both the Lexer and Parser (and internally for the Node Interpreter within visit() functions) and final outputting based on the discovery of errors post-conditional. Once again, code blocks of error handling (processing for error outputting) were found in clusters, thus improving readability as all similar processing occurred in a single location, where readability improves maintainability. There was also strict validation in interaction with errors (mainly the disabling of the Console text box) to prevent the deletion of error messages after they

are outputted (ideal as they are unrecoverable unless re-run, thus aiding debugging). Overall, considering all test cases were passed successfully, with the test cases being robust in that they tested errors produced in all three interpreting stages (Lexer, Parser, and Node Interpreter), there is sufficient evidence that Stage 4 Prototype 2 has been completed successfully, meaning we can move on from it until post-development testing, and move on to Stage 5.

Stage 4 Evidence of Validation for All Key Elements

Output + Input Validation

Presence Check: Only enables input to be accepted (and thus the handle input functionality to run) if the console is disabled, as this could lead to accidental input prompts that are not interactable. Validation occurs inside handleConsoleInput().

```
if Console.cget('state') == tk.DISABLED: # Validation: Only Allows the Acceptance of Input if the Console is Enabled
    return # Halts Further Processing (visitInput Should Have Left This Enabled, Alternative is Incorrect Functionality)
```

Type Check: Ensures that if a parameter of print is deemed invalid (i.e. throws an error during processing), processing of the print() function does not occur (to avoid unwanted functionality of printing Error objects as normal parameters). Validation occurs inside visitProcedureCallNode().

```
if type(nodeValue).__name__ == 'Error': # Validation: Checks If There Was An Error In Processing a Parameter
    return Error('n/a', 'SyntaxError', 'Invalid Parameter Format') # If So, Program Halter to Avoid Accidental Halting
```

Length Check: Ensures that if there is more than one parameter to the input function (because at maximum it has one string input), processing of the input() function does not occur (potential Python errors and unwanted halting if invalidly continuing with input()). Validation occurs inside visitFunctionCallNode().

```
if len(node.parameterNodeList) > 1: # Validation: Input Can Only Have One Argument
    return Error('n/a', 'TypeError', f'Invalid Argument Count') # No Line Number as Interpreter Does Not Have Access
```

Error Output Validation

Type Check: Ensures line number is only attempted to be decremented in AST if it is an integer (non-integer cases when line number is unknown, like during system errors where it is a string ‘n/a’), thus avoiding accidental halting due to incompatible operations on strings. Validation occurs inside the runClick() function.

```
if isinstance(AST.lineNumber, int): # Validation: Only Reduces Line Number if Valid Line Number Available
    AST.lineNumber -= 1 # Due to Initial Newline, lineNumber is Always Overshot by 1
```

Type Check: Ensures line number is only attempted to be decremented in tokenList if it is an integer (non-integer cases when line number is unknown, like during system errors where it is a string ‘n/a’), thus avoiding accidental halting due to incompatible operations on strings. Validation occurs inside the runClick() function.

```
if isinstance(tokenList.lineNumber, int): # Validation: Only Reduces Line Number if Valid Line Number Available
    tokenList.lineNumber -= 1 # Due to Initial Newline, lineNumber is Always Overshot by 1
```

Type Check: Ensures that if an error is encountered, not only is the console updated, but further processing halts as error objects are generally incompatible with interpreter operations (e.g. binary/unary operations), likely being a potential cause for Python errors (unwanted halting). Validation here occurs inside the visitFunctionCallNode() function.

```
if type(prompt).__name__ == 'Error': # Validation: Error Types are Handled to Avoid Accidental Halting with Attributes
    Console.configure(state = tk.NORMAL) # Enables Console So Content Can be Written To (Error)
```

Stage 4 Summary Review

At Stage 4 Prototype 1, we created the mechanisms for console inputting/outputting (for valid statements), the former accepting input and returning it to the node interpreter and the latter being an independent operation which just outputted to the console. We did maintain modularity by implementing each operation inside separate functions, particularly with a complicated but readable conditional within the input() function which required re-interpreting to continue with, but this was successful under robust test cases. This modularity was useful for unit-testing, and will prove useful during maintenance (for debugging purposes and general readability). Overall, considering all test cases were passed successfully, with the test cases themselves being suitably robust as they tested output for multiple data types within print (going beyond just strings), thus also testing string conversion operations, with further foundation being provided by strict validation to ensure that console interactions for input/output are highly limited (only to their respective function calls), thus avoiding accidental deletion and unexpected user-based errors, there is sufficient evidence that Stage 4 Prototype 1 has been completed successfully and we can move on from it until post-development testing.

At Stage 4 Prototype 2, we created the mechanism for error outputting to the console, which involved the transfer of Error objects upwards through both the Lexer and Parser (and internally for the Node Interpreter within visit() functions) and final outputting based on the discovery of errors post-conditional. Once again, code blocks of error handling (processing for error outputting) were found in clusters, thus improving readability as all similar processing occurred in a single location, where readability improves maintainability. There was also strict validation in interaction with errors (mainly the disabling of the Console text box) to prevent the deletion of error messages after they are outputted (ideal as they are unrecoverable unless re-run, thus aiding debugging). Overall, considering all test cases were passed successfully, with the test cases being robust in that they tested errors produced in all three interpreting stages (Lexer, Parser, and Node Interpreter), there is sufficient evidence that Stage 4 Prototype 2 has been completed successfully, meaning we can move on from it until post-development testing, and move on to Stage 5.

Stage 5: Complete GUI

Prototype 1: Light Mode

First, we shall display the ‘Change Theme’ button, attaching the relevant ‘Light Mode’ and ‘Dark Mode’ drop-down selection feature. This is **related to ‘Analysis’** by the criterion of ‘Change Theme’ opening a drop-down for ‘Light Mode’ and ‘Dark Mode’, **justified** as the button will create a drop-down of clickable objects for both themes. This is **related to Problem Breakdown** by the ‘Load Buttons’ process of Algorithm A, **justified** as ‘Change Theme’ is ultimately a clickable button. To **explain**, a Tkinter Menu object was created with drop-down commands labelled ‘Light Mode’ and ‘Dark Mode’ respectively, also clickable connected to functions to be implemented associated with the colour setting expected by each theme function, which is connected to the button. The image associated with the button is also pre-formatted using a separate helper function. To **justify**, the Tkinter Menu object connected to the button is ideal as the menu functionality is necessary (as ‘Light Mode’ and ‘Dark Mode’ are hierarchically lower options), but it also enables the ‘Change Theme’ button to remain a static object (the display text does not change depending on the option selected, **justified** as this was intended in the GUI Structure section). To **justify** using a separate helper function (instead of the pre-defined function used for the file operation buttons) to obtain the image, this is because the image had to be scaled larger (because a paintbrush is a more complicated icon, there were issues with a low resolution if (25, 25) was used).

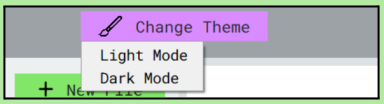
```
# Function Which Obtains and Formats the Button Image for the Change Theme Button
def getChangeThemeImage():
    fileName = 'ChangeThemeImage.png' # Maintenance: For Publication, May Want to Change Local Reference to Reliable Global, Internet-Hosted Reference
    fileImage = Image.open(fileName) # Opens the File Given Its Location
    if fileImage is None: # Validation: Only Returns Valid Image if File Exists at That Location
        return # This Provides None for Image, Thus Preventing an Error Message (Unwanted Halting)
    fileImage = fileImage.resize((30, 30)) # (30, 30) Pixels Is Ideal Given the Standard Height of Tkinter Buttons Aesthetically
    fileImage = ImageTk.PhotoImage(fileImage) # Converts the Image into a Tkinter-Compatible Format For Placement Into Button
    return fileImage # Returns the Fully Formatted Image
ChangeThemeImage = getChangeThemeImage() # Obtains Formatted Image for Change Theme button
# Button Which is Pressed to Open a Drop-Down With Options Light Mode and Dark Mode For Theme Selection
ThemeButton = tk.Menubutton(TopBar, text = ' Change Theme', image = ChangeThemeImage, compound = tk.LEFT, borderwidth = 0, width = 225, font = ButtonFontStyle, bac
# Maintenance: Background Set to Light Mode Colour as Light Mode is Default In Case Light Mode Setting Does Not Occur Upon Start-Up Or Called Prior
DropFontStyle = ('Roboto Mono', 14, 'normal') # Slightly Smaller Font for Menu Options As These are Hierarchichally Lower

# Function Which Formats and Attaches Menu Options to Change Theme Button and Displays Button On Screen
def packChangeThemeButton():
    ThemeMenu = tk.Menu(ThemeButton, tearoff = False) # Tearoff For Options Unnecessary as Clicking NULL Objects Unnecessary
    ThemeButton.configure(menu = ThemeMenu) # Attaches Menu to Button So Dropdown Is Created Upon Click (Expected Functionality)
    ThemeMenu.add_command(label = 'Light Mode', font = DropFontStyle, command = setLightMode) # Light Mode Attached to setLightMode Function (Colour Switching)
    ThemeMenu.add_command(label = 'Dark Mode', font = DropFontStyle, command = setDarkMode) # Dark Mode Attached to setDarkMode Function (Colour Switching)
    ThemeButton.pack(side = 'left', pady = 20, expand = False, padx = 100) # expand False Not Ideal as Run Should Be Central (Not Shared With Run)
packChangeThemeButton() # Function Called to Place Button on Screen (Unchanging Because Button Itself is Static)
```

Next, we shall implement 'Light Mode' functionality. This is **related to 'Analysis'** by the criterion of 'Change Theme' opening a drop-down for 'Light Mode' and 'Dark Mode', **justified** as 'Light Mode' extends the functionality (by click command) of the 'Light Mode' button. This is **related to Problem Breakdown** by the 'Apply Light Mode Design Parameters' process of Algorithm C, **justified** as the relevant foreground and background colours will be changed as per the Colour Palette Structure section. To **explain**, all visible foreground/background colours were changed in-function (which was activated upon 'Light Mode' click), except for the foregrounds of the file hierarchy and console, which were set by assignment to be applied in the console and file hierarchy functions implemented previously. To **justify**, most colours being changed in-function maximises readability as colours are determined in a shared location, while generally optimising as otherwise each prior location would need to be checked one-by-one over a relatively large file, increasing the likelihood of missing elements (reducing productivity). To **justify** the need for assignment in the console and file hierarchy, this is because these text elements are dynamic in the sense that their foregrounds are constantly updated at points beyond the 'Change Theme' click (in displayFileHierarchy() and visitProcedureCallNode()). As such, they need a permanently stored value to reference in these cases, hence the assignment to globally accessible variables.

```
# Function Which Sets All GUI Elements to Light Mode Colours (Defined in Colour Palette Structure) Upon Selecting 'Light Mode' in 'Change Theme'
def setLightMode():
    global ConsoleStandardFontColour # Maintenance: Global Variable Necessary as Python Assumes Local Declaration
    global ConsoleErrorFontColour # These Are Necessarily Set Here as Font Colours Are Dynamically Updated During Execution, Adding Files, and Text-Editing
    global FileHierarchyFontColour # Maintenance: As Such, They Will Change Post-Change-Theme-Click, And They Will Need to Know Font Colour Then
    window.configure(background = '#dce1e3') # Maintenance: Colours the Gaps in the Text Editor and Console (Due to Padding)
    TopBar.configure(background = '#9da2a6') # Top Bar Houses the Run Button and Change Theme Button
    FileHierarchyBar.configure(background = '#EDF0F1') # This and the Below Option Coloured Consistently in Colour Palette Structure
    FileOperationBar.configure(background = '#EDF0F1') # This and the Above Option Coloured Consistently in Colour Palette Structure
    TextEditor.configure(background = '#FFFFFF') # Pure White to Maximise Contrast With Dark Font (Readability Important While Text-Editing)
    Console.configure(background = '#CCCCCC') # Darker Than Text Editor To Provide Differentiation Mechanism for Windows
    TextEditor.configure(foreground = '#282828') # Dark to Contrast White Background
    ConsoleStandardFontColour = '#282828' # Dark to Contrast Light Background
    ConsoleErrorFontColour = '#F33A3A' # Red to Indicate Something Went Wrong (Colour Association)
    FileHierarchyFontColour = '#282828' # Dark to Contrast Light Background
    ThemeButton.configure(foreground = '#282828') # Dark to Contrast Light Background
    RunButton.configure(foreground = '#282828') # Dark to Contrast Light Background
    NewFileButton.configure(foreground = '#282828') # Dark to Contrast Light Background
    ImportFileButton.configure(foreground = '#282828') # Button Colours for Font Consistent as Consistency May be More Aesthetic
    SaveFileButton.configure(foreground = '#282828') # Dark to Contrast Light Background
    DeleteFileButton.configure(foreground = '#282828') # Dark to Contrast Light Background
    ThemeButton.configure(background = '#D98CFF') # Background Relatively Light to Contrast Dark Font as Light Mode
    RunButton.configure(background = '#C0FA33') # Background Relatively Light to Contrast Dark Font as Light Mode
    NewFileButton.configure(background = '#82E769') # Background Relatively Light to Contrast Dark Font as Light Mode
    ImportFileButton.configure(background = '#5CDE5D') # Different Green to Differentiate With Green of Run Button
    SaveFileButton.configure(background = '#FFE240') # Background Relatively Light to Contrast Dark Font as Light Mode
    DeleteFileButton.configure(background = '#FF5C62') # Background Relatively Light to Contrast Dark Font as Light Mode
    if len(Console.get('1.0', tk.END)) > 1: # Validation: Only Changes Font Colour if Text Already Exists in the Console
        Console.configure(foreground = '#282828') # Avoids Unnecessary Operations, Thus Optimising
```

Prototype 1 Testing

Test Data	Expected Output	Actual Output
Open Program	Initial colour palette should match that of Light Mode (default) exactly	Colour palette matched Light Mode except for the file hierarchy (yellow background and grey font as declared during testing) ❌

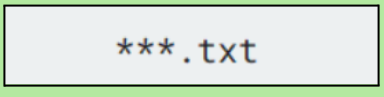
		
Click ‘Change Theme’	Drop-down of ‘Light Mode’ and ‘Dark Mode’ should become available	Drop-down of ‘Light Mode’ and ‘Dark Mode’ was made available ✅ 
Click ‘Light Mode’ after clicking ‘Change Theme’	Colour palette should match that of Light Mode exactly	Colour palette matched Light Mode except for the file hierarchy (yellow background and grey font as declared during testing) ❌ 

Prototype 1 Remedial Action

Test 1 and 3 failed as instead of displaying the File Hierarchy with ‘Light Mode’ characteristics, the background colour was yellow and the font colour grey as defined arbitrarily during testing. This error occurred as displayFileHierarchy() was called only once upon start-up (where its background and foreground were set to the testing values) and not called at ‘Change Theme’ click, particularly important because the file hierarchy is dynamic as described prior. The action that will be taken therefore is to call displayFileHierarchy() every time ‘Light Mode’ (and ‘Dark Mode’) is clicked. This action is **justified** as the alternative would be to change the initial values to those of the ‘Light Mode’ default. Though this would work upon start-up, it would not work upon changing from ‘Light Mode’ to ‘Dark Mode’ or vice versa as this is a static change for an element that is fundamentally dynamic.

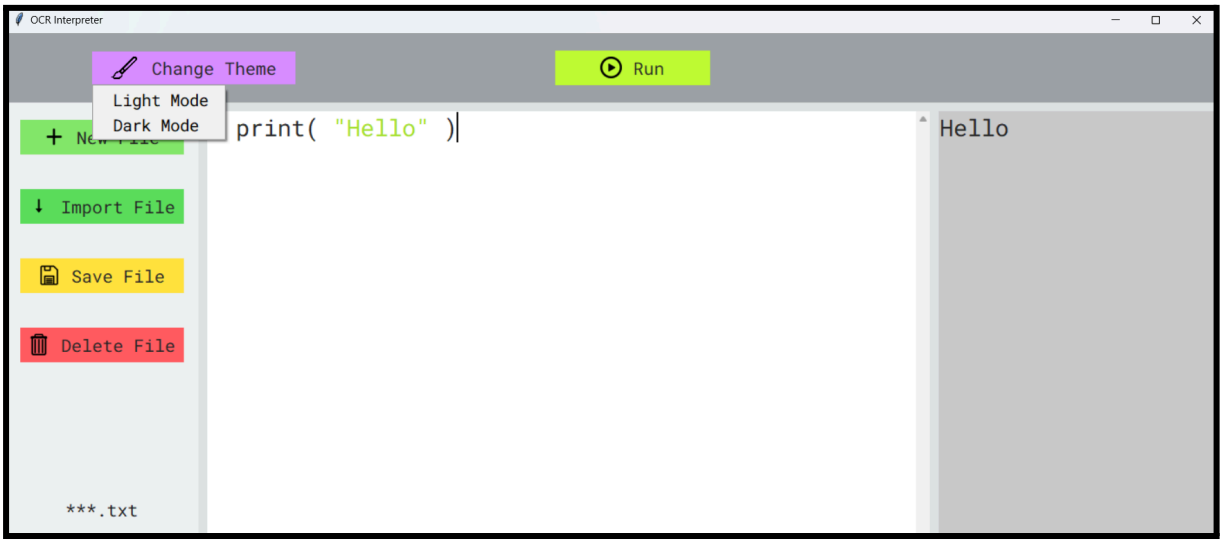
```
DeleteFileButton.configure(background = '#FF5C62') # Background Relatively Light to Contrast Dark Font as Light Mode
displayFileHierarchy() # Maintenance: Necessary as the File Hierarchy is Dynamic
if len(Console.get('1.0', tk.END)) > 1: # Validation: Only Changes Font Colour if Text Already Exists in the Console
```

For **full justification**, the previously failed tests have now been completed successfully:

Open Program	Initial colour palette should match that of Light Mode (default) exactly	Initial colour palette matched that of Light Mode exactly ✅ 

Click ‘Light Mode’ after clicking ‘Change Theme’	Colour palette should match that of Light Mode exactly	Colour palette matched that of Light Mode exactly ✅ ***.txt

Prototype 1 Evidence



Prototype 1 Review

At Stage 5 Prototype 1, we created mechanisms for loading the ‘Change Theme’ menu-button and for ‘Light Mode’, which set background/foreground colours based on the Colour Palette Structure section. We maintained modularity by using helper functions for ‘Change Theme’ including for the formatting of the image and packing for display, which enabled unit-testing and improved the readability of the code (operations associated closely with function name). We also optimised by ensuring that colour changes occurred, where possible, in the setLightMode() function, thus ensuring that colour changes were localised around a shared location, thus improving readability and maintainability, as this content could easily be changed in the future. Overall, considering all test cases were passed successfully, with the test cases themselves being robust in their assessment of ‘Light Mode’ under different situations, there is sufficient evidence that Stage 5 Prototype 1 has been completed successfully (for testing at the post-development stage).

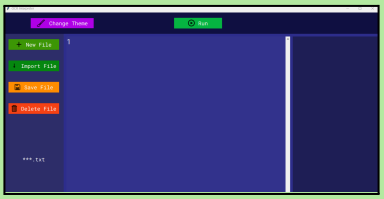
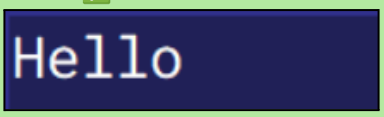
Prototype 2: Dark Mode + Light Mode

Now, we shall implement ‘Dark Mode’ functionality. This is **related to ‘Analysis’** by the criterion of ‘Change Theme’ opening a drop-down for ‘Light Mode’ and ‘Dark Mode’, **justified** once again as ‘Dark Mode’ extends the functionality (by click command) of the ‘Dark Mode’ button. This is **related to Problem Breakdown** by the ‘Apply Dark Mode Design Parameters’ process of Algorithm C, **justified** as the foreground and background colours will be altered as defined in the Colour Palette Structure section. To **explain**, all visible foreground/background colours were changed inside the setDarkMode() function (which was activated upon ‘Dark Mode’ click), except for the foregrounds of the file hierarchy and console, which were set by assignment to be applied in the console and file hierarchy functions implemented previously. To **justify**, most colours being changed inside setDarkMode() maximises readability as colours are determined in a shared location, while optimising as otherwise each prior location would need to be checked one-by-one over a relatively large file, increasing the likelihood of missing elements (reducing productivity). To **justify** the need

for assignment in the console and file hierarchy, this is because these text elements are dynamic in the sense that their foregrounds are constantly updated at points beyond the 'Change Theme' click (in displayFileHierarchy() and visitProcedureCallNode()). As such, they need a permanently stored value to reference in these cases, hence the assignment to globally accessible variables.

```
# Function Which Sets All GUI Elements to Dark Mode Colours (Defined in Colour Palette Structure) Upon Selecting 'Dark Mode' in 'Change Theme'
def setDarkMode():
    global ConsoleStandardFontColour # Maintenance: Global Variable Necessary as Python Assumes Local Declaration
    global ConsoleErrorFontColour # These Are Necessarily Set Here as Font Colours Are Dynamically Updated During Execution, Adding Files, and Text-Editing
    global FileHierarchyFontColour # Maintenance: As Such, They Will Change Post-Change-Theme-Click, And They Will Need to Know Font Colour Then
    window.configure(background = '#30308c') # Maintenance: Colours the Gaps in the Text Editor and Console (Due to Padding)
    TopBar.configure(background = '#111145') # Top Bar Houses the Run Button and Change Theme Button
    FileHierarchyBar.configure(background = '#2E2E6C') # This and the Below Option Coloured Consistently in Colour Palette Structure
    FileOperationBar.configure(background = '#2E2E6C') # This and the Above Option Coloured Consistently in Colour Palette Structure
    TextEditor.configure(background = '#34348E') # Dark to Maximise Contrast With Light Font (Readability Important While Text-Editing)
    Console.configure(background = '#202057') # Darker Than Text Editor To Provide Differentiation Mechanism for Windows
    TextEditor.configure(foreground = '#FDFDFD') # Light to Contrast Dark Background
    ConsoleStandardFontColour = '#FDFDFD' # Light to Contrast Dark Background
    ConsoleErrorFontColour = '#F33A3A' # Red to Indicate Something Went Wrong (Colour Association)
    FileHierarchyFontColour = '#FDFDFD' # Light to Contrast Dark Background
    ThemeButton.configure(foreground = '#FDFDFD') # Light to Contrast Dark Background
    RunButton.configure(foreground = '#FDFDFD') # Light to Contrast Dark Background
    NewFileButton.configure(foreground = '#FDFDFD') # Light to Contrast Dark Background
    ImportFileButton.configure(foreground = '#FDFDFD') # Button Colours for Font Consistent as Consistency May be More Aesthetic
    SaveFileButton.configure(foreground = '#FDFDFD') # Light to Contrast Dark Background
    DeleteFileButton.configure(foreground = '#FDFDFD') # Light to Contrast Dark Background
    ThemeButton.configure(background = '#B203E7') # Background Relatively Dark to Contrast Light Font as Dark Mode
    RunButton.configure(background = '#07B742') # Background Relatively Dark to Contrast Light Font as Dark Mode
    NewFileButton.configure(background = '#3C9805') # Background Relatively Dark to Contrast Light Font as Dark Mode
    ImportFileButton.configure(background = '#07870F') # Different Green to Differentiate With Green of Run Button
    SaveFileButton.configure(background = '#FF8E00') # Background Relatively Dark to Contrast Light Font as Dark Mode
    DeleteFileButton.configure(background = '#F64214') # Background Relatively Dark to Contrast Light Font as Dark Mode
    displayFileHierarchy() # Maintenance: Necessary as the File Hierarchy is Dynamic
```

Prototype 2 Testing

Test Data	Expected Output	Actual Output
Click 'Change Theme' and then 'Dark Mode' after program loads	Colour palette should match that of 'Dark Mode' exactly	Colour palette matched that of 'Dark Mode' exactly ✅ 
Click 'Dark Mode' and then 'Run' after typing: print("Hello")	Colour palette of console should match that of 'Dark Mode's standard console output font colour	Colour palette of console matched that of 'Dark Mode's standard console output font colour ✅ 
Click 'Run' after typing: print("Hello") During 'Light Mode' and then click 'Dark Mode'	Colour palette of console should match that of 'Dark Mode's standard console output font colour (after clicking 'Dark Mode' should be 'Light Mode's colour before that)	Colour palette of console did not switch to 'Dark Mode's standard console output font colour ❌ 

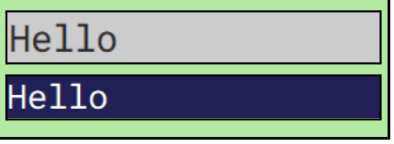
Prototype 2 Remedial Action

The final test failed as instead of the console output font colour changing after 'Dark Mode' was clicked, it remained in the 'Light Mode' setting it was in prior to the click. This error occurred as

setDarkMode() did not include a console configuration for its foreground, which is particularly important as the console text is dynamic and needs to be regularly updated (upon 'Run' clicks). The action that will be taken is to create a console font colour configuration after validating for the presence of an existing console output. The **justification** for the action of validating for the presence of an existing output is to ensure that colours are not changed if there is nothing present for the colour to change to, which is ideal as this avoids unnecessary operations, thus optimising for time.

```
if len(Console.get('1.0', tk.END)) > 1: # Validation: Only Changes Font Colour if Text Already Exists in the Console
    Console.configure(foreground = '#FDFDFD') # Avoids Unnecessary Operations, Thus Optimising
```

For **full justification**, the previously failed test has now been completed successfully:

Click 'Run' after typing: print("Hello") During 'Light Mode' and then click 'Dark Mode'	Colour palette of console should match that of 'Dark Mode's standard console output font colour (after clicking 'Dark Mode' should be 'Light Mode's colour before that)	Colour palette of console did switch to 'Dark Mode's standard console output font colour ✅ 

Prototype 2 Evidence

Prototype 2 Review

At Stage 5 Prototype 2, we created the mechanism for 'Dark Mode', which worked based on the second column of the Colour Palette Structure section. setDarkMode() was consistent with setLightMode(), thus improving readability (as rehauls for themes could easily find reflective changes between the two available themes), thereby also improving maintainability (these rehauls become easier). We also maintained modularity by keeping 'Dark Mode' switches within setDarkMode(), excluding dynamic changes in the file hierarchy and console, though this was handled easily by global assignments (acceptable as descriptive variable names de-risk overwriting). Furthermore, there was stringent validation for the length of the console, thus optimising performance as unnecessary Tkinter object manipulation operations were avoided. Overall, considering all test cases were passed successfully, with the test cases themselves being robust in that they tested similar cases to Prototype 1, while looking at the behaviour of the dynamic console, there is sufficient evidence that Stage 5 Prototype 2 has been completed leaving post-development.

Stage 5 Evidence of Validation for All Key Elements

Light Mode Validation

Presence Check: Ensures that the image for the 'Change Theme' button is only processed if the image file (named 'ChangeThemeImage.png') exists, thus avoiding accidental halting due to Python Errors if the file has been deleted. Validation occurs inside getChangeThemeImage().

```
if fileImage is None: # Validation: Only Returns Valid Image if File Exists at That Location
    return # This Provides None for Image, Thus Preventing an Error Message (Unwanted Halting)
```

Length Check: Ensures that the console font colour is only changed to 'Light Mode' settings if there is currently text in the console (works because this would be called independent of 'Run'), thus avoiding unoptimised extra operations. Validation occurs inside setLightMode().

```
if len(Console.get('1.0', tk.END)) > 1: # Validation: Only Changes Font Colour if Text Already Exists in the Console
    Console.configure(foreground = '#282828') # Avoids Unnecessary Operations, Thus Optimising
```

Dark Mode Validation

Length Check: Ensures that the console font colour is only changed to 'Dark Mode' settings if there is currently text in the console (works because this would be called independent of 'Run'), thus avoiding unoptimised extra operations. Validation occurs inside setDarkMode().

```
if len(Console.get('1.0', tk.END)) > 1: # Validation: Only Changes Font Colour if Text Already Exists in the Console
    Console.configure(foreground = '#FDFDFD') # Avoids Unnecessary Operations, Thus Optimising
```

Stage 5 Summary Review

At Stage 5 Prototype 1, we created mechanisms for loading the 'Change Theme' menu-button and for 'Light Mode', which set background/foreground colours based on the Colour Palette Structure section. We maintained modularity by using helper functions for 'Change Theme' including for the formatting of the image and packing for display, which enabled unit-testing and improved the readability of the code (operations associated closely with function name). We also optimised by ensuring that colour changes occurred, where possible, in the setLightMode() function, thus ensuring that colour changes were localised around a shared location, thus improving readability and maintainability, as this content could easily be changed in the future. Overall, considering all test cases were passed successfully, with the test cases themselves being robust in their assessment of 'Light Mode' under different situations, there is sufficient evidence that Stage 5 Prototype 1 has been completed successfully (for testing at the post-development stage).

At Stage 5 Prototype 2, we created the mechanism for 'Dark Mode', which worked based on the second column of the Colour Palette Structure section. setDarkMode() was consistent with setLightMode(), thus improving readability (as rehauls for themes could easily find reflective changes between the two available themes), thereby also improving maintainability (these rehauls become easier). We also maintained modularity by keeping 'Dark Mode' switches within setDarkMode(), excluding dynamic changes in the file hierarchy and console, though this was handled easily by global assignments (acceptable as descriptive variable names de-risk overwriting). Furthermore, there was stringent validation for the length of the console, thus optimising performance as unnecessary Tkinter object manipulation operations were avoided. Overall, considering all test cases were passed successfully, with the test cases themselves being robust in that they tested similar cases to Prototype 1, while looking at the behaviour of the dynamic console, there is sufficient evidence that Stage 5 Prototype 2 has been completed leaving post-development.

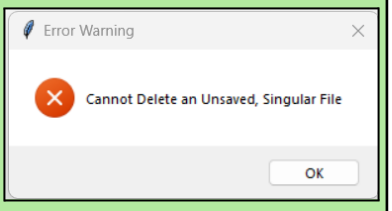
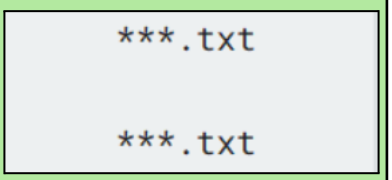
Evaluation

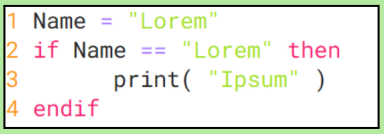
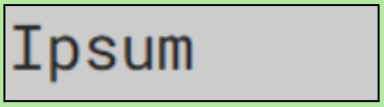
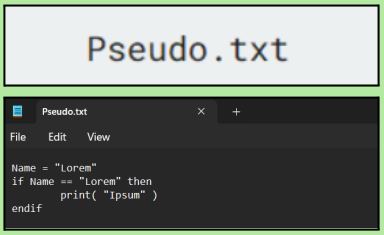
Annotated Post-Development Testing

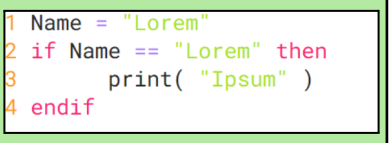
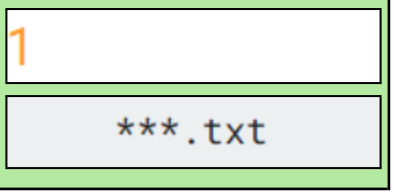
Testing for Function

To test for function, the program must display valid behaviour for expected/common test data (i.e. with common inputs expected from a normal user). To best simulate this, testing for function will be accomplished primarily by beta testing with each interacting stakeholder, where each stakeholder will be told to use the IDE as they would generally intend to use it with their work (therefore avoiding the unusual testing of extreme or purposefully invalid test data) in a single continuous session, which is logical as one criteria I hope to be assessed is how smooth transitioning from one action to another is while using the IDE. They will also be asked to record their test data inputs and their output after completing that input, from which an annotation will be made determining whether that test was successful or not, with additional qualitative comments from the stakeholder.

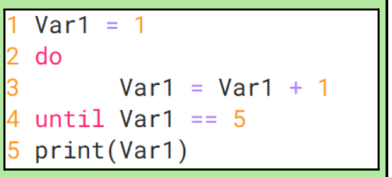
Shao Shen Kuar's Functional Testing

Test Data	Actual Output	Annotation
Press 'Delete File' button	Pop-up window warning of deleting an unsaved file	Pop-up window warning of deleting an unsaved file is seen, with message 'Cannot Delete...' being particularly useful in describing how the error arose. Non-halting was also appreciated as he would have found that tiring over time. ✅ 
Press 'New File' button, and attempt to create an empty text file	Empty contents of text editor and new '***.txt' appeared in the text editor	This is the expected behaviour from the Analysis section, and Shao Shen appreciated that this remained consistent during development. ***.txt was also a suitable placeholder according to him as no file he created has been named similarly ✅ 
Enter into text editor: Name = "Lorem" if Name == "Lorem" then	Name = "Lorem" if Name == "Lorem" then print("Ipsum")	These were the expected pretty printing colours from the Design section. Shao Shen

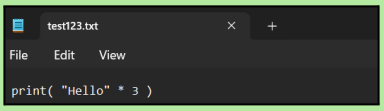
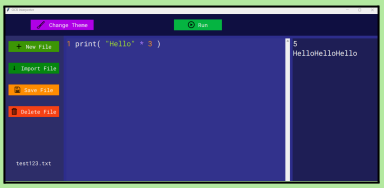
print(“Ipsum”) endif	endif	found that they were suitably contrasting with the background, particularly with the large font size and appreciated the quick response time ✅ 
Press ‘Run’ button	“Ipsum” output on Console	This was the expected output given OCR syntax. Shao Shen stated that the correct output being produced was the most important factor, and cross-referencing with OCR, this he found was correct ✅ 
Press ‘Save File’ button	File explorer opened and he saved the file as ‘Pseudo.txt’, and this was updated in the file hierarchy	This was the expected output from the Analysis stage. Shao Shen stated that though he did not mind if this feature did not exist, he appreciated that it did as file-storing allowed him to save his work and easily transfer it to his teachers ✅ 
Close Program	Program window closed	This was the expected output from Analysis. Shao Shen appreciated that the file was closed without error messages (due to poor garbage collection, for example) and found this first session smooth ✅ 
Press ‘Import File’ button, opening the file worked on just prior	Opened File Explorer in file-opening mode, then loaded the text editor into the state it was in prior	This was the expected output from the Analysis stage. Shao Shen really appreciated that all main file operations (concerning file opening and saving) worked correctly

		(without unexpected errors) ✅ 
Press ‘Delete File’ button	Text editor content cleared and the on-device file was non-existent	This was the expected output from the Analysis stage. Shao Shen appreciated that file deletion was easy and simple (single click operation) and that all relevant features were updated ✅ 

Toby Low’s Functional Testing

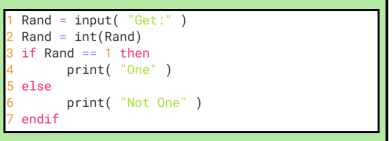
Test Data	Actual Output	Annotation
Enter into text editor: Var1 = 1 do Var1 = Var1 + 1 until Var1 == 5 print(Var1)	Var1 = 1 do Var1 = Var1 + 1 until Var1 == 5 print(Var1)	This was the expected output from the Design stage. Toby appreciated the pretty printing, particularly due to struggles in reading continuous, undifferentiated blocks of text ✅ 
Click the ‘Run’ button	‘5’ output to the Console	This was the expected output from Analysis. Toby appreciated the quick execution times, particularly as pseudocode problems are often short in nature ✅ 
Press ‘Save File’ button	File explorer opened and he saved the file as ‘test123.txt’, and this was updated in the file hierarchy	This was the expected output from Analysis. Toby appreciated the file-storage mechanisms, stating that it was vital that his earlier work could be transferred for execution at speed ✅

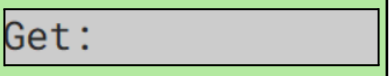
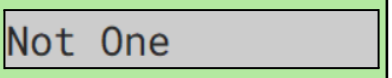
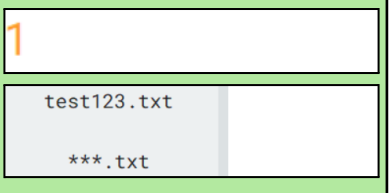
		test123.txt
		`Var1 = 1`
		`do`
		`Var1 = Var1 + 1`
		`until Var1 == 5`
		`print(Var1)`

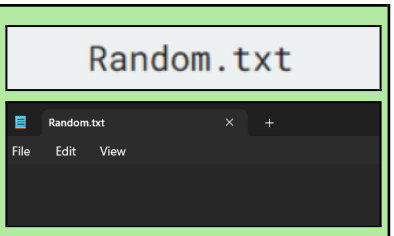
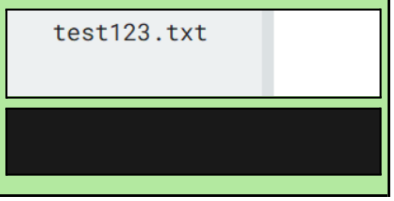
Enter into text editor: print(“Hello” * 3)	print(“Hello” * 3)	This was the expected output from the Design stage. Toby once again appreciated that each element was coloured noticeably differently, describing this as a useful usability feature ✅ `print( "Hello" * 3 )`
Click the ‘Run’ button	‘HelloHelloHello’ output to the Console	This was the expected output from the OCR syntax. Toby appreciated that the niche elements of OCR syntax (including uncommon string operations) were accounted for ✅ `HelloHelloHello`
Click the ‘Save File’ button	Contents of ‘test123.txt’ were changed to reflect new text editor contents	This was the expected output from Analysis. Toby was pleased that file storage was completely functional, working for both saved and unsaved files ✅ 
Click ‘Change Theme’ and then click ‘Dark Mode’	All screen contents were changed to those in the Dark Mode column of the Colour Palette Structure section	This was naturally the expected output from Design. Toby appreciated that ‘Change Theme’ was so simple (few button clicks) and that the GUI looked aesthetically pleasing ✅ 

Tulin Kasimaga's Functional Testing

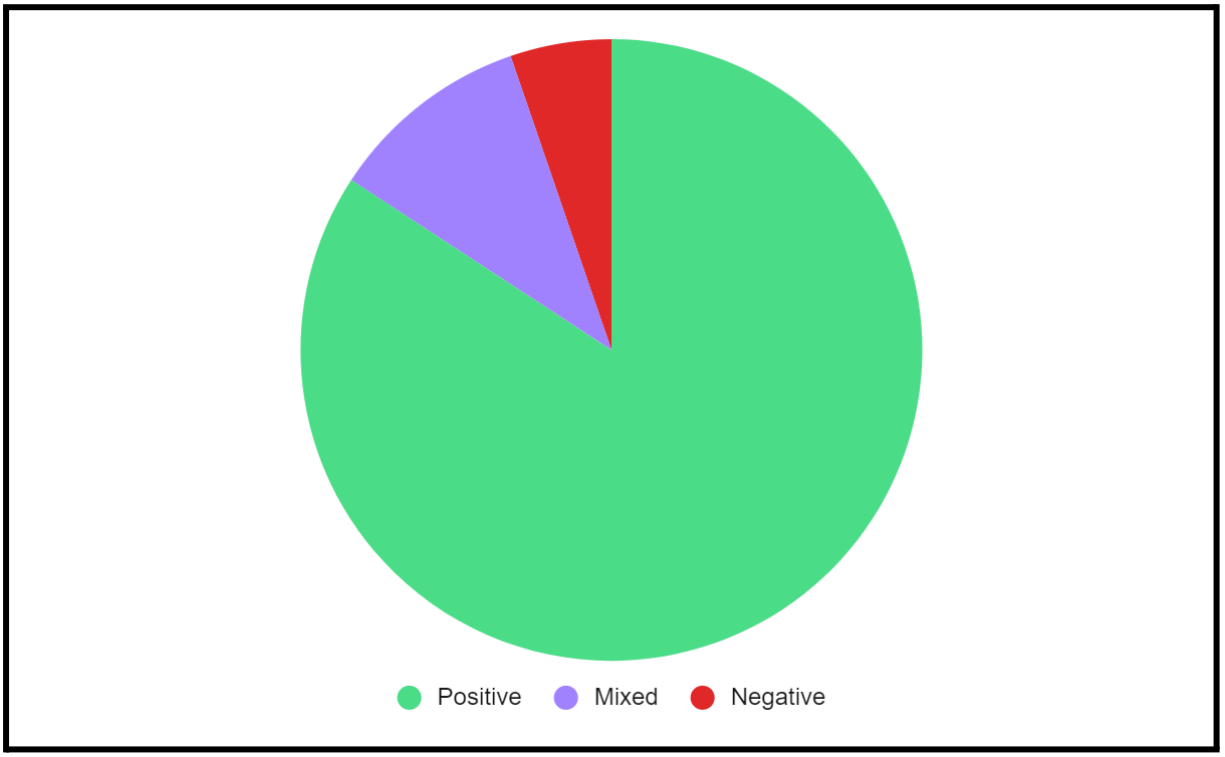
Test Data	Actual Output	Annotation

Enter into text editor:
Rand = input("Get:")
Rand = int(Rand)
if Rand == 1 then
  print("One")
else
  print("Not One")
endif
	Rand = input("Get:")
	Rand = int(Rand)
	if Rand == 1 then
	  print("One")
	else
	  print("Not One")
	endif
		This was the expected output from the Design stage. Ms. Tulin appreciated the pretty printing, nothing that items with similar functions were coloured consistently, which helped her follow along while she coded ✅
		 

Click the 'Run' button	Console outputted "Get: ", allowing her to input data	This was the expected output from Analysis. Ms. Tulin noted that inputting was necessary for her to execute her student's work (which included input()), so this was a useful feature ✅ 
Inputted value '2' in the Console	Console outputted 'Not One'	This was the expected output from the OCR syntax. Ms. Tulin was pleased with conditionals working correctly, particularly as they are a commonly tested paradigm at the GCSE level ✅ 
Clicked the 'New File' button	Text editor cleared and another '***.txt' appeared in the file hierarchy	This was the expected output from Design. Ms. Tulin stated that though multi-file support was not vital, she would find it efficient when marking the work of multiple students ✅ 
Click the 'Save File' button	File explorer opened and he saved the file as Random.txt', and this was updated in the file hierarchy	This was the expected output from Design. Ms. Tulin stated that though she would mainly import, she could use file storage to make changes to the student's work for marking and advising purposes ✅

		
Click the ‘Delete File’ button	‘Random.txt’ was removed from the device and the file hierarchy removed ‘Random.txt’	This was the expected output from Design. Ms. Tulin stated that though she would be cautious of deletion, she would find it useful in organising the IDE (when accidentally importing unintended files) ✅ 

From the test data provided by the interacting stakeholders outlining how they would expect to use the IDE on a day-by-day, standard basis (thus testing functionality and not the robustness of extreme/invalid test data which stakeholders are unlikely to find applicable in real life), we can see that in their experiences, all test cases have passed, which is logical considering we tested fairly robustly during iterative development. The associated annotations also have qualitative comments that help understand the functional user experience. In fact, we can generally assess how positive these comments are on a positive-mixed-negative scale, to see how successful the final product is with the interacting stakeholders.



Here, the feedback was overwhelmingly positive, with 16 positive comments compared to 2 mixed and 1 negative. The negative comment here was a suitable caution from Tulin Kasimaga, stating that file deletion was potentially dangerous, particularly as a single click leads to a full deletion of a file, which is undesirable both in terms of the pseudocode text files currently being worked on and other files that theoretically can be accidentally imported, particularly highly sensitive or system-important ones. That said, this negative comment was mixed in with the notion that file deletion was useful for organising in the IDE, and that overall she was happy with the operation. As such, remedial action of removing the 'Delete File' operation is inadvisable, especially as the other two stakeholders preferred the 'Delete File' operation without critical points. The positive comments also included the notion of smooth transitioning between file operations, and that all required elements of the solution existed which enabled the ideal manipulation and execution of pseudocode files (while maintaining the bare simplicity optimised for relatively simple pseudocode problems). As such, functional testing is complete, and there is sufficient evidence to state that the IDE is functional enough for the final product to be considered successful when used normally.

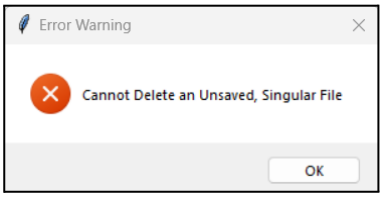
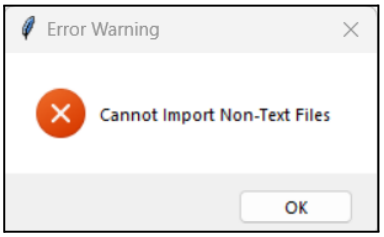
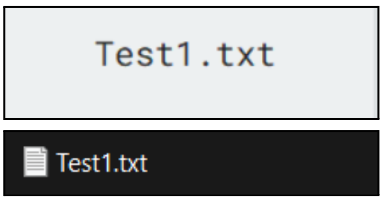
Testing for Robustness

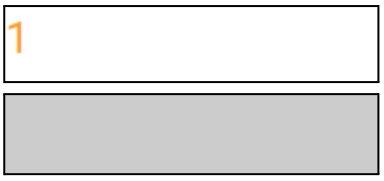
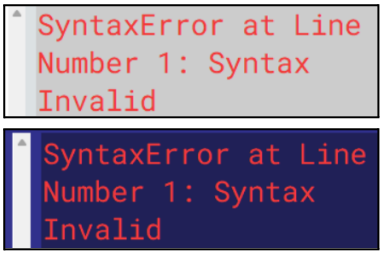
Now that the final product has been deemed sufficiently functional, we shall test for robustness, which will involve a thorough white-box test of each main operation, including the interactions, which are all of the buttons, the 'Ctrl + _' functionality, and different text editor inputs to the interpreter. Another aspect of testing for robustness is the necessity that erroneous/extreme test data is handled in a valid way, particularly as these are prone to being overlooked during iterative development. As such, this section will primarily be composed of erroneous/extreme test data where applicable (justified as the previous stage and iterative development testing has already tested whether normal test data is processed validly and it has been completely successful).

Checklist of Items for Robust Testing

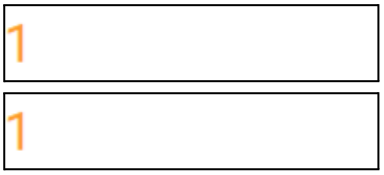
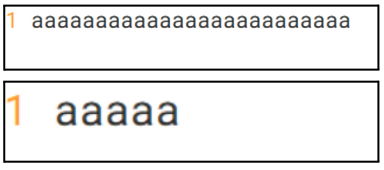
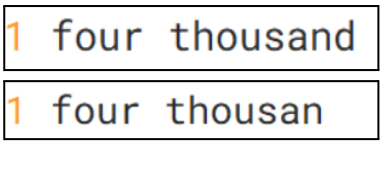
Item	**Robust Test Number**
'Run' Button	-
'Change Theme' Button	-
'New File' Button	-
'Import File' Button	-
'Save File' Button	-
'Delete File' Button	-
Ctrl + Z Functionality (Other Ctrl + _ Already in Tkinter)	-
Loop Structures	-
Conditional Statements	-
Arithmetic Operators	-
Boolean Operators	-
Functions	-
Procedures	-
Identifier Scope	-

Robust Testing of Buttons

Test Data	Actual Output	Annotation
Test 1 - Click 'Delete File' with unsaved, singular file in File Hierarchy	Warning box stating that unsaved, singular files cannot be deleted 	This was the expected output from the Analysis stage. Erroneous test data as files should not be deleted if single and unsaved (to avoid underflow of file hierarchy elements). Message is also useful as it is specific, with the associated cross making obvious that an error was made ✅
Test 2 - Click 'Import File' and attempt to import 'NewFileImage.png'	Warning box stating that non-text files cannot be imported 	This was the expected output from the Analysis stage. Erroneous test data as PNGs are not supported by Tkinter documentation, and Python errors while attempting to interpret unsupported file types would lead to unwanted halting. Message is useful as it specifies that text files are the only supported file type ✅
Test 3 - Click 'New File' twice	Two more '***.txt' appear in the file hierarchy and the text editor remains blank 	This was the expected output from the Analysis stage. This is valid test data, though it has not been tested previously (more than two text files simultaneously in the file hierarchy). The space is suitable for this many files (desirable as otherwise, file names are incomprehensible) ✅
Test 4 - Click 'Save File' while three text files are open, saving as 'Test1.txt'	Only last file in the file hierarchy becomes 'Test1.txt' while the prior two files remain as '***.txt' 	This was the expected output from the 'Analysis' stage. This is valid test data, though saving with more than two files in the text editor has not been tested. The change only occurred to the last file in the text editor, which is ideal as it shows independence in the multi-file system, which is desirable to avoid corruption ✅
Test 5 - Click Run when the text editor is empty	Nothing occurs in the Console (no halting of the entire program) and the text editor and file hierarchy contents remain the same	This was the expected output from Analysis. Erroneous test data as if 'Run' is executed on an empty text file, there is the possibility of Python errors occurring (due to unavailable statement types after the first

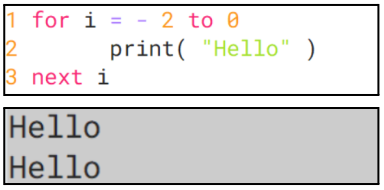
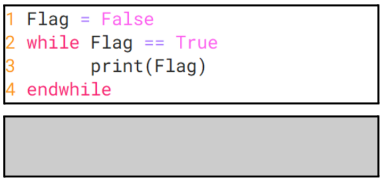
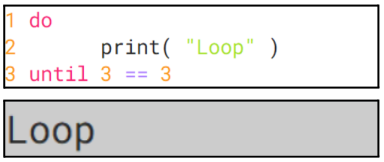
		new line). The fact that no accidental halting or terminal errors occurred (and non-empty situations were tested), means that the 'Run' button is robust ✅
Test 6 - Type 'for to next' in the Text Editor, which should produce an error message, and then press 'Change Theme', then 'Dark Mode'	SyntaxError in Console, and the colour remains the same after 'Dark Mode' is clicked 	This is the expected output from Design. Valid test data, though changing themes when an error is in the console has not been tested prior. The fact that the colour remains constant is ideal, as it contrasts both backgrounds well and retaining red is proper as the notion that it is an error should remain. As 'Change Theme' for non-errors has already been tested, 'Change Theme' is now evidently robust ✅

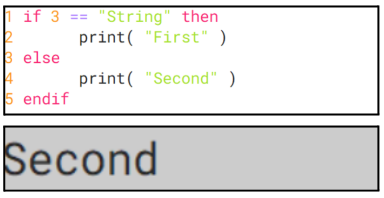
Robust Testing of 'Control + Z' Functionality

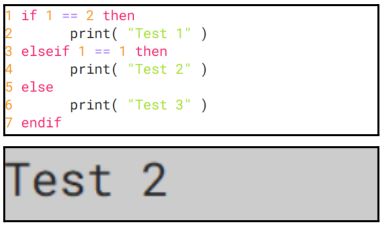
Test Data	Actual Output	Annotation
Test 7 - 'Ctrl + Z' clicked once before anything in text file is interacted with	Nothing changes in the text editor (no removals or additions) 	This is the expected output from Analysis. Extreme test data, as if there is nothing in the text editor state list, over-decrementation should not occur and Python errors should be avoided (unwanted halting) ✅
Test 8 - 'Ctrl + Z' clicked 25 times after 25 characters are spelled out in continuity	After clicking 'Ctrl + Z' 25 times, 20 characters were removed (by undoing) and 5 remained 	This is the expected output from Analysis. Erroneous test data, as due to the text editor state list size limit of 20, undoing 25 times should not lead to 25 deletions, as is seen here. The fact that each undo was smooth (one character per key press) is ideal as this avoids unwanted deletions ✅
Test 9 - 'Ctrl + Z' clicked once after a character is removed via 'Backspace'	After clicking 'Ctrl + Z', the character that was removed after pressing 'Backspace' was reinstated 	This was the expected output from the Analysis stage. This is valid test data, though testing post-character-deletion was not tested prior. As character deletion can be undone, this is a useful feature in case content is accidentally deleted and needs to be reinstated (particularly if the work is sensitive). This was also

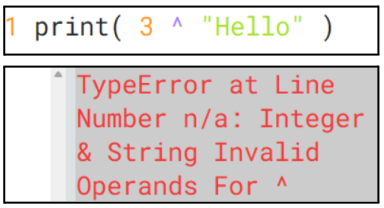
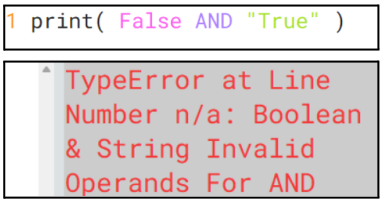
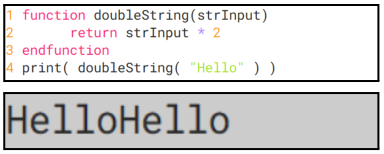
	1 four thousand	smooth (one character per key press), which is ideal as this works long term. Now, considering deletion, limits, emptiness, and normal undoing has been tested, 'Ctrl + Z' is evidently robust ✅

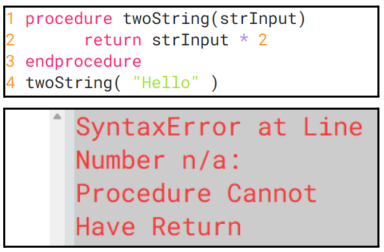
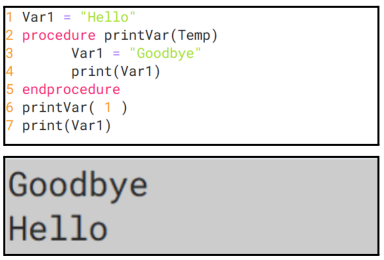
Robust Testing of Interpreter

Test Data	Actual Output	Annotation
Test 10 - Enter into text editor: for i = -2 to 0 print("Hello") next i And press 'Run'	Outputted 'Hello' twice in separate lines in the Console 	This was the expected output from Analysis and the OCR documentation. Valid test data, though negative numbers were previously untested (and generally unary operators) in for loops, but this still performed the required 2 iterations which is ideal ✅
Test 11 - Enter into text editor: Flag = False while Flag == True print(Flag) endwhile And press 'Run'	Outputted nothing to the Console after clicking the 'Run' button 	This was the expected output from the OCR documentation. Valid test data, though while loops that are never entered due to the initial condition failing has not been tested yet, though now they are robust as unentered and entered while loops have both been tested (here and during iterative development) ✅
Test 12 - Enter into text editor: do print("Loop") until 3 == 3 And press 'Run'	Outputted 'Loop' a single time to the Console after clicking the 'Run' button 	This was the expected output from the OCR documentation. Valid test data, though do until loops that fail their condition after the first pass have not been tested yet, though now they are robust as both multi-iteration and single-iteration do until loops have been tested (here and during functional testing) ✅

Test 13 - Enter into text editor:
if 3 == "String" then
  print("First")
else
  print("Second")
endif
And press 'Run'
	Outputted 'Second' to the Console after clicking the 'Run' button
	 
		This was the expected output from the OCR documentation. Valid test data, though if statements where the condition involved an equality between mismatching types have not been tested yet, though now they are robust as both standard and non-matching comparisons have been tested successfully (here and during iterative development) ✅

Test 14 - Enter into text editor:
if 1 == 2 then
  print(“Test 1”)
elseif 1 == 1 then
  print(“Test 2”)
else
  print(“Test 3”)
endif
And press ‘Run’
	Outputted ‘Test 2’ to the Console after clicking the ‘Run’ button
	 
	
		This was the expected output from the OCR documentation. Valid test data, though exiting the middle ‘elseif’ branch has not been tested yet, though now if statements are robust as exiting at the ‘if’, ‘elseif’, and ‘else’ branches (complete list) have all been tested successfully (here and during iterative development) ✅

Test 15 - Enter into text editor: print(3 ^ “Hello”) And press ‘Run’	Outputted TypeError in the Console, with message specifying that the operands are invalid  	This was the expected output from the OCR documentation. Erroneous test data, testing whether an error would be identified (not a halting Python error but an interpreter-produced error) when the power symbol is surrounded by a number and a string. As a TypeError was correctly produced, arithmetic operators are robust in that all other standard arithmetic operators were tested during iterative development ✅
Test 16 - Enter into text editor: print(False AND “True”) And press ‘Run’	Outputted TypeError in the Console, with message specifying that the operands are invalid  	This was the expected output from the OCR documentation. Erroneous test data, testing whether an error would be identified (again not a halting Python error) when AND has an operand of a string with value “True”, thus also whether strings are too type-flexible, which they are not, which is ideal. Boolean operators have been tested with both erroneous and valid test data, so they are robust ✅
Test 17 - Enter into text editor: function doubleString(strInput) return strInput * 2 endfunction print(doubleString(“Hello”)) And press ‘Run’	Outputted ‘HelloHello’ (result of strInput * 2) in the Console after clicking the ‘Run’ button  	This was the expected output from the OCR documentation. Valid test data, though functions inside procedures (print) have not been tested previously, nor the doubling of contiguous brackets. Now that functions have been tested for use cases in normal assignment (iterative development) and in value-returning to procedures, they are robust ✅
Test 18 - Enter into text editor: procedure twoString(strInput) return strInput * 2 endprocedure	Outputted SyntaxError in the Console after clicking the ‘Run’ button, with message specifying that procedure	This was the expected output from Analysis. Erroneous test data, testing whether the interpreter would identify the

twoString("Hello") And press 'Run'	cannot have 'return'  	error (and not a halting Python error) when a return was found inside a procedure (invalid, only functions have return statements). The error message is also sufficiently specific here, which is ideal and will aid in user debugging. Procedures are now robust as they have been tested normally (iterative development) and with invalid test data ✅
Test 19 - Enter into text editor: Var1 = "Hello" procedure printVar(Temp) Var1 = "Goodbye" print(Var1) endprocedure printVar(1) print(Var1) And press 'Run'	Outputted 'Goodbye' in the first line of the Console and 'Hello' in the second after clicking the 'Run' button  	This was the expected output from the OCR documentation. Valid test data, though printing identifiers with the same name in the same program (though in different scopes) has not been tested yet. Evidently, the inside-procedure scope Var1 was printed with its associated value and the global scope Var1 was printed with its associated value. As both local and global scopes have been tested, local/global scope functionality is robust ✅

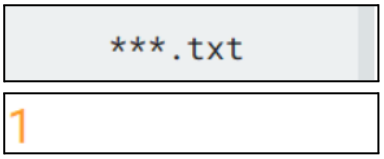
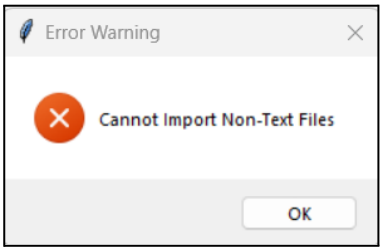
Item	**Robust Test Number**
'Run' Button	5
'Change Theme' Button	6
'New File' Button	3
'Import File' Button	2
'Save File' Button	4
'Delete File' Button	1
Ctrl + Z Functionality (Other Ctrl + _ Already in Tkinter)	7, 8, 9
Loop Structures	10, 11, 12
Conditional Statements	13, 14
Arithmetic Operators	15
Boolean Operators	16
Functions	17
Procedures	18
Identifier Scope	19

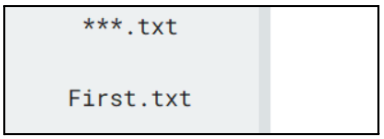
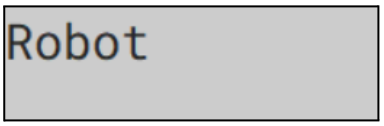
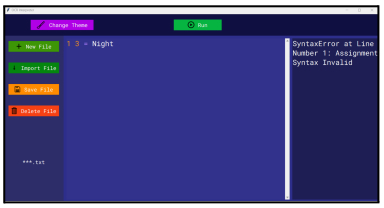
As all dynamic features (involving interactions that trigger some level of functionality) have evidently been robustly tested (successfully), with justification of robustness provided in the annotation column, we have sufficient evidence to state that the IDE and all of its developed features are robust (and capable of handling erroneous and extreme data). As such, 'Testing for Robustness' has been completed successfully, and we can move on from post-development testing for functionality and robustness.

Annotated Evidence For Usability Testing

Evidence for usability testing will be split into objective and subjective testing, the former of which will test the usability features of the IDE that can be tested in an objective manner (such as whether a button click performs its pre-specified, intended action), and the latter of which will gather qualitative data (such as whether the 'Dark Mode' colours are aesthetically pleasing and not harsh on the eyes) for analysis. The justification for the separation between objective and subjective features is that objective testing can most efficiently be performed by the developer (as if opinions are not involved, there will be no variation between data from different users, and this will lead to duplication) and with subjective data, different people can offer different insights into whether something is aesthetically pleasing, for example, due to a natural variation in opinions.

Objective Testing of Usability Features

Test Data	Actual Output	Annotation
'New File' button is clicked when IDE is opened	After one button is clicked, the text editor remains blank and a blank file is added to the file hierarchy 	This shows that only one button needs to be clicked for file creation, which is a useful usability feature (also showing that 'New File' works as intended in Analysis). The creation of an obviously blank file is also useful (new files should be created without placeholders due to variation) ✅
'Import File' button is clicked, importing a PNG file	After one button is clicked, the importing functionality begins executing (excluding File Explorer). Error warning stating that non-text files cannot be opened pops up 	This shows that only one button needs to be clicked for file importing, which is a useful usability feature (also showing that 'Import File' works as intended in Analysis). The prevention of importing a PNG is necessary, as a stated usability feature is that only text files are handled (due to their fit with pseudocode and lack of complexity in dealing with pixels, etc.). The error message is also specific (no other information needs to be conveyed) ✅
'Save File' button is clicked, on '***.txt', saving it as 'First.txt.'	After one button is clicked, the 'Save As' functionality begins executing. 'First.txt' opens on the device (blank due to text editor being blank) and file hierarchy updates to show 'First.txt'	This shows that only one button needs to be clicked for file saving, which is a useful usability feature (also showing that 'Save File' works as intended in Analysis). The updating of the file hierarchy is

		a useful usability feature (always seeing the file you are working on avoids confusion) ✅
'Delete File' button is clicked on 'First.txt'	After one button is clicked, 'First.txt' is removed from the file hierarchy and the device 	This shows that only one button needs to be clicked for file deletion, which is a useful usability feature (also showing that 'Delete File' works as intended in Analysis). The removal of the file hierarchy is a useful usability feature (showing that the previous file has been returned and that you are not working on a non-existent file is logical) ✅
Click 'Run' after typing: print("Robot") In the text editor	After one button is clicked, 'Robot' is output to the Console 	This shows that only one button needs to be clicked for file execution, which is a useful usability feature (also showing that 'Run' works as intended in Analysis). The quick execution time, where 'Robot' popped up on screen practically instantaneously shows that the interpreter is sufficiently optimised for short programs ✅
Click 'Run' after typing: 3 = Night In the text editor	After one button is clicked, a Syntax error indicating that the assignment syntax is valid is displayed 	This shows only one button needs to be checked for error identification. The usability features required an error type, line number, and error message for efficient error relaying (improves user debugging). Here, 'SyntaxError', '1', and 'Assignment Syntax Invalid' fulfil this criterion, thus improving usability ✅
Click 'Change Theme' and then click 'Dark Mode'	After two buttons are clicked ('Change Theme and 'Dark Mode'), the colour palette changes from 'Light Mode' to 'Dark Mode' 	This shows that only two buttons need to be clicked for theme changing (better than one as having light mode and dark mode uncoupled would take more screen space and be less logical). This also shows that 'Change Theme' works as intended in the Analysis section ✅

From objective testing, we can see that all buttons function correctly (specific outcome determined during the Analysis and Design stages, so they are measurable on a functional basis), that there are indeed two theme options for users ('Light Mode' and 'Dark Mode'), that warning boxes appear

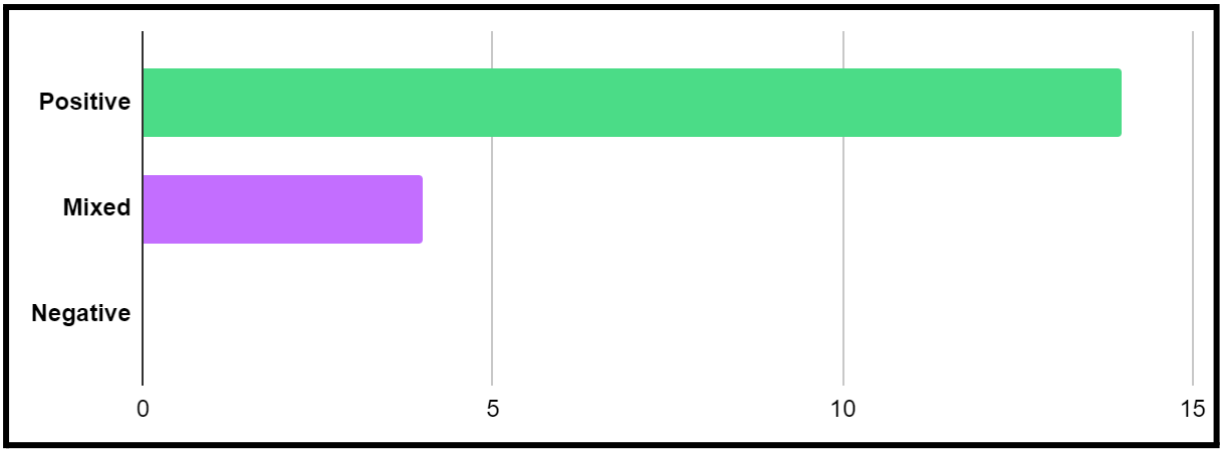
upon non-interpreter system errors with specific error messages outlining the problem, and that interpreter-produced error messages consist of the structured determined during Analysis (error type, line number, and error message). As all objective usability features have been tested and all test cases have passed successfully, there is sufficient evidence to state that these final objective usability features do not require revisiting post-development and are suitable in their current form.

Subjective Testing of Usability Features

For the subjective usability testing section, the interacting stakeholders will use their copy of the software to answer pre-set questions which will cover the remaining aspects of usability not tested in the object usability testing section. Their annotations will generally be assessed and categorised on a positive-mixed-negative scale, and the responses aggregated with analysis performed thereafter. To justify this approach, attempting to quantify the qualitative data (perform counts of generally categorised opinions) will enable the data to be represented more concisely, such that decision-making to determine whether the implemented usability features (those currently in the program) meet an acceptable standard can be completed.

Question	Annotations		
	Shao Shen Kuar	**Toby Low**	**Tulin Kasimaga**
Q1 - Do the buttons have a sufficiently large size?	Yes, they are clearly visible on the screen, and this is further improved by their colours contrasting with the background ✅	Yes, I think they are more than large enough. I would prefer 'Run' to be larger, not necessarily because of visibility but because of its relative importance, but this is minor 🟣	Yes, the buttons are completely large enough. They are entirely readable and I believe others users would agree ✅
Q2 - Do the buttons have a sufficiently large font size?	Yes, the font takes a respectable amount of space in the button, and they are large enough, so I believe this is the case ✅	Relative to the buttons, I think the font size is perfect. It is entirely readable to me and there is enough space aesthetically for the other elements ✅	I suppose I would prefer if the font size was slightly larger, but I think it is entirely readable and this is just a matter of aesthetic preference 🟣
Q3 - Regarding the button symbols, do you think they are appropriate?	I think each symbol illustrates what you want it to illustrate, and I cannot really think of alternatives, so I would say so ✅	I think the symbols, aesthetically, are one of the highlights of the IDE. They fit their function well and I would not change them ✅	The symbols are perfectly fine, especially in combination with the colours. Noting the 'Garbage' symbol with the 'Red' background, any user would understand the consequences of file deletion ✅
Q4 - Is the window size suitable?	Now that I think about it, I feel like full-screen mode would be my personal preference, though I acknowledge this changed from	I think this is the ideal window size. Like I said, you can have multiple windows open simultaneously, such as the syntax website for coding	The window size matches what we discussed previously, and looking at the final product, I think this was the right choice, particularly

	previously. This is just because file importing avoids the need for me to constantly cross-reference, but this window size is also great 🟣	aid, while it being large enough to be easily readable ✅	because it is large and enables multiple windows to be opened simultaneously ✅
Q5 - The current UI sections are divided into the file window, the text editor, and the console. Is this your ideal combination?	I think there being few UI elements makes perfect sense for a pseudocode IDE, as clutter can be avoided, thus making the user experience more intuitive, while avoiding extraneous features because pseudocode is a simple language ✅	Simplicity is key here, and so I believe that this is the ideal combination. I would not want more as this would lead to screen cluttering, and the limited functionality of pseudocode does not require extra elements ✅	Yes, I think minimising actions to only the essentials maximises the space you have for each, allowing the large buttons and text editor font size, for example. Also, pseudocode does not require extensive debugging or other over-the-top features ✅
Q6 - Do you believe the colour palette chosen for 'Light Mode' and 'Dark Mode' is suitable?	Yes, 'Light Mode' has the simplicity generally associated with light IDE themes and 'Dark Mode', with the overarchingly dark blue palette, provides aesthetic contrast, and they both do actually, leading them to both be functional and be aesthetically pleasing ✅	I would say that 'Dark Mode' looks really great, and the blues in particular contrast the buttons well and it feels conducive to productivity. 'Light Mode' also provides great contrast, but I think grey colours can be somewhat dull, relatively speaking. That said, that is a personal preference and for a light theme, it looks perfectly fine 🟣	I prefer 'Light Mode' due to its simplicity, as I do with other IDEs, but I think there is a demonstrable contrast with both themes that makes the IDE quite aesthetically pleasing. As there is contrast in both, and there certainly is a benefit with options, I would agree that the colour palettes are suitable ✅



After subjective testing, we can see that a majority of reactions to all subjective usability features are positive. Specifically, this is calculated to be approximately 78%, which is sufficiently high as there were no entirely negative reactions and the mixed reactions themselves had a positive flavour. This

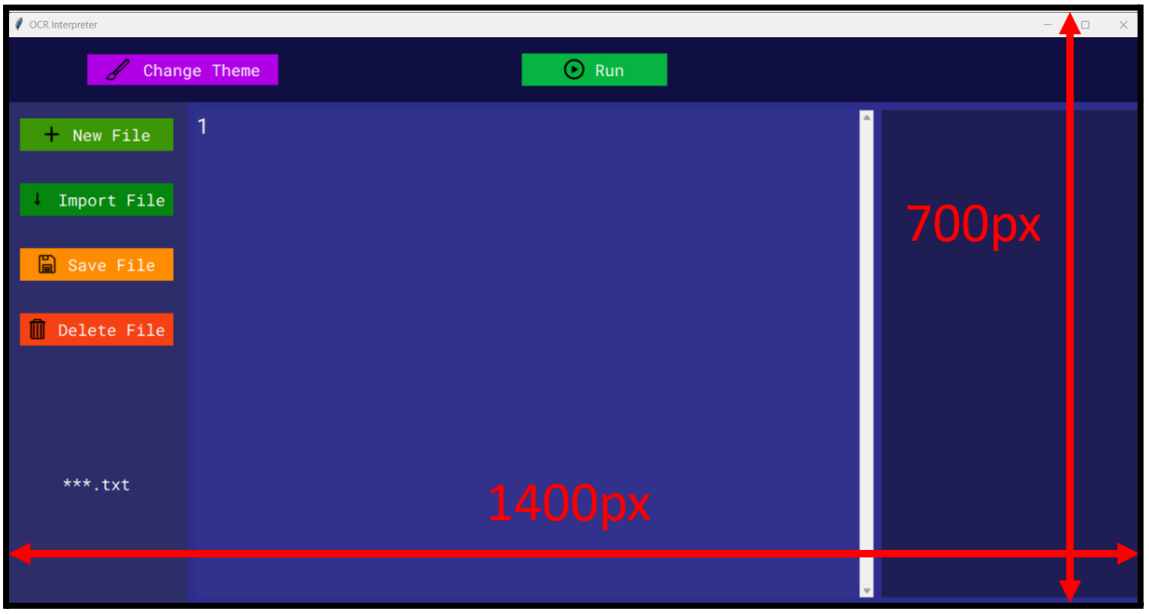
is statistically significant as all mixed reactions stated that the corresponding feature was suitable in its current state, but that their preferences leaned one way in a non-extreme manner. As such, this variation can be seen as being person-dependent (as these are subjective questions), and stemming from what is a relatively small, opportunistic sample due to the nature of the project. That said, each positive reaction was significantly positive and reaffirms the notion that the final product is aesthetically and functionally intuitive and suitable, and that the usability features, being the displayed features such as the font and element size, the levels of contrast between overlapping UI elements, the structure of error messages, and the limitation to text files, have been chosen and implemented successfully.

Cross-Referencing With Success Criteria

For this section, we shall determine the extent to which each success criterion has been met by going through each of them as specified in the Analysis section one-by-one, producing the required test evidence (if implemented correctly) as specified in the 'How to Evidence' column and cross-referencing thereafter, and by cross-referencing with the test data from the 'Annotated Post-Development Testing' section, particularly in the case of checking whether the interpreter is working up to standard as the large volume of those tests can be used as useful evidence (as they cover many areas). We shall begin with the first success criterion and increment from there.

Success Criterion 1	
1400px by 700px GUI	Screenshot of 1400px by 700px Tkinter window GUI after opening IDE + Screenshot of section of code in which 1400px by 700px GUI is implemented

Test Evidence for Cross-Referencing with Success Criterion 1




```
window.geometry('1400x700') # As Per the Success Criteria, 1400x700 is Preferred
```

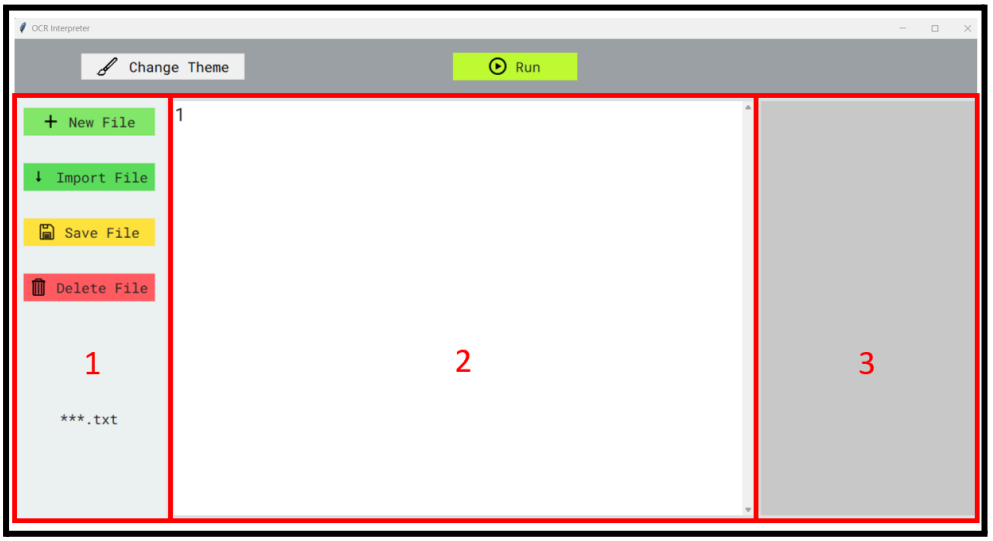
Test 7 from 'Toby Low's Functional Testing' also displays the entire GUI, and when measured, the GUI comes out to be 1400px by 700px in size, which matches Success Criterion 1.

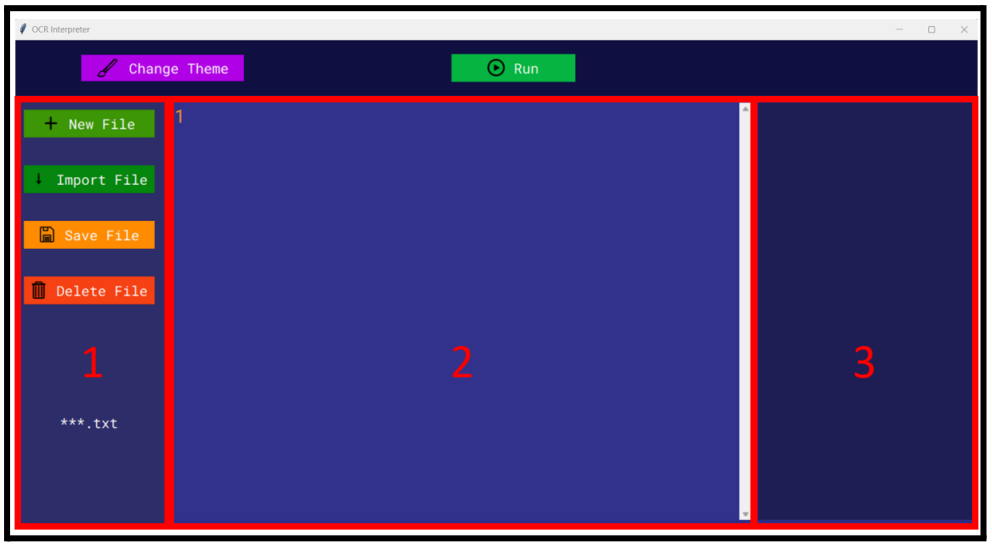
Evaluation and Explanation of Success Criterion 1

Success Criterion 1 required both a screenshot of the 1400x700px GUI and a screenshot of the section of code in which that was implemented. Both of the required testing features were provided in the test evidence, and to **explain/justify** this evidence being sufficient, the GUI being a fixed size means that it is static (the window size does not change based on any other interactions, and this is evidenced by the window.geometry setting only appearing a single time in the code). As such, showing that the GUI is 1400x700px at a given point in time is equivalent to showing that it remains that size throughout the session (until the IDE is closed). As such, there is sufficient test evidence to state that Success Criterion 1 has been **fully met**.

Success Criterion 2	
GUI divided into three primary, different-coloured, vertically divided sections (excluding top bar)	Screenshots of three primary vertically divided sections of different colours for each available theme

Test Evidence for Cross-Referencing with Success Criterion 2





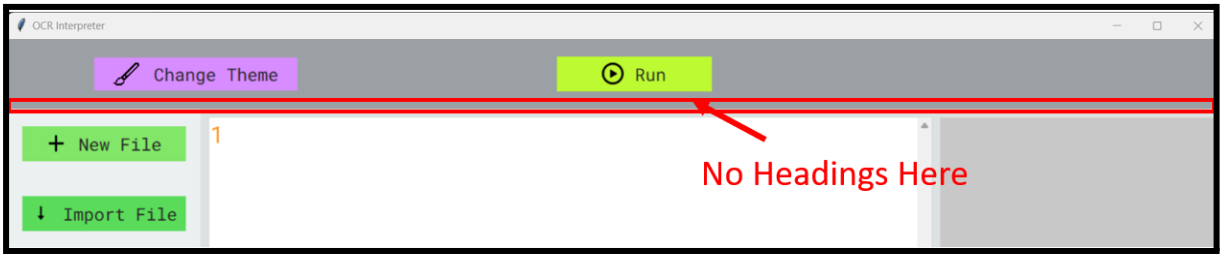
Test 7 from 'Toby Low's Functional Testing' also displays the entire GUI, where the three primary sections can be seen vertically divided in 'Dark Mode', thus partially completing Success Criterion 2. The 'Light Mode' counterparts of the primary window elements can be seen in Test 14 of the 'Testing for Robustness' section in the case of the text editor, Test 17 of the 'Testing for Robustness' section in the case of the console, and Test 2 from 'Shao Shen Kuar's Functional Testing' in the case of the file window. As such, all three primary windows can be seen in 'Dark Mode' and 'Light Mode' (vertically divided), thus completing Success Criterion 2 successfully.

Evaluation and Explanation of Success Criterion 2

Success Criterion 2 required a screenshot of three primary, vertically divided sections in the available colour themes ('Light Mode' and 'Dark Mode' here). By showing each of the primary sections ('File Window', 'Text Editor', 'Console'), as seen within the red-outline rectangles, we have fulfilled this requirement, and to **explain/justify** this evidence being sufficient, once again it is due to the static nature of the IDE UI, where the only major UI shift is seen with the addition of elements to the file hierarchy and with changes to the colour themes, though both of these actions do not change the size or location of each of the three primary windows, nor removing or adding extra sections. As such, this static nature makes equivalent taking the screenshots as shown above and screenshots taken at later times within the session. As such, there is sufficient test evidence to state that Success Criterion 2 has been **fully met**.

Success Criterion 3	
Each primary section has headings 'Files', 'Text Editor', 'Console' (from left to right)	Screenshot of three section headings being 'Files', 'Text Editor', 'Console' (from left to right)

Test Evidence for Cross-Referencing with Success Criterion 3



Test 7 from 'Toby Low's Functional Testing' displays the full GUI in 'Dark Mode' and when viewing the screenshot, there is once again a lack of headings in the UI at the post-development stage.

Evaluation and Explanation of Success Criterion 3

Success Criterion 3 required a screenshot of the headings of each section, specifically 'Files', 'Text Editor', and 'Console' from left to right. These should be persistent (static as the UI structure generally is), and so taking a screenshot at a point would be equivalent to evidencing it via a screenshot at any point in the session. However, as **explained/justified** in the above image and from the functional test data, we see that there are no visible headings. As such, we can state that Success Criterion 3 is **not met.**

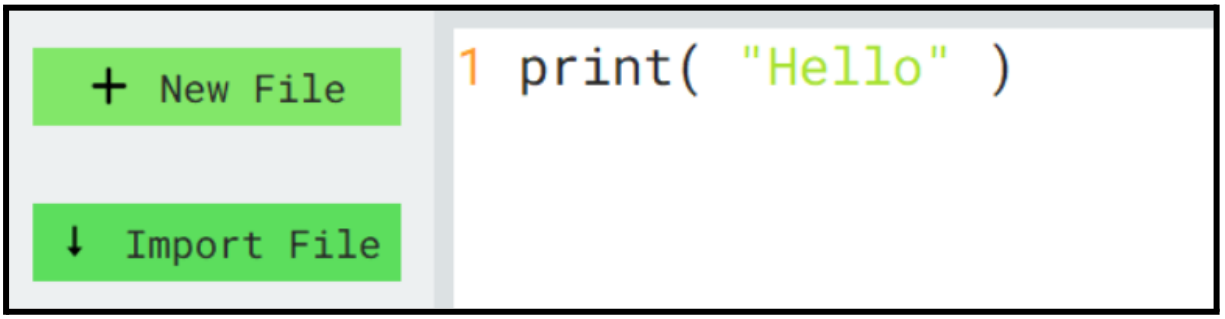
Addressing Success Criterion 3 in Further Development

To **justify**, the reason that Success Criterion 3 (adding headings to the primary windows) was not met is due to the timeline of development. I had already set up the GUI elements in fixed locations with internal elements, some of which moved dynamically with file interactions also in quite a fixed manner (due to the specific use of pixel values to determine object attributes like height, width, and padding, thus leading to some hard-coding in displaying elements). Due to the 'Complete GUI' stage being the fifth and final stage, when I attempted to add the headers, there was a downwards shift of all primary window elements (excluding the top bar, though ideally the top bar would also have shrunk, thus further changing the button size parameters of 'Change Theme' and 'Run'), and this downwards shift made the IDE aesthetically unsatisfactory. In fixing this downwards shift, there would need to be a near-complete restructuring of GUI elements (specifically, their height, width, padding, and font size) and this was unachievable due to time restrictions.

To **address Success Criterion 3 in further development**, I would first complete the necessary GUI restructuring by reducing the heights of the primary window elements (specifically, in code, the FileOperationBar, FileHierarchyBar, TextEditor, and Console), reducing the heights and widths of the internal window elements (NewFileButton, ImportFileButton, SaveFileButton, DeleteFileButton, and fileHierarchyObject) to leave space for the headers, create the 'Heading Row' by creating a Label object that takes the width of the screen (1400px) and a suitable height to have 'Files', 'Text Editor', and 'Console' in a sufficiently large, readable font, reduce the height of TopBar, thus also reducing the font size and height of ThemeButton and RunButton, and from there, the headings would be completely implemented (no need to worry about size changes, justified as the 'Heading Row' is static). To add, because the issue arose out of the difficulty in changing hard-coded Tkinter padding, height, and width values, I would also change the values from being hard-coded to being a percentage of the screen or window height/width, such that element size attributes are responsive to changes in their parent frames, and that fewer sizes need to be changed before complete GUI restructurings are completed successfully.

Success Criterion 4	
Large font size in text editor and console	Screenshot of text in Text Editor + Console + Statements of acceptance from each interacting stakeholder saying font size in text editor and console is sufficiently large

Test Evidence for Cross-Referencing with Success Criterion 4



Furthermore, there are statements of acceptance from the interacting stakeholders in Q2 and Q5 in the 'Annotated Evidence of Usability Testing' section and from evidence gathered during feedback conversations. There are:

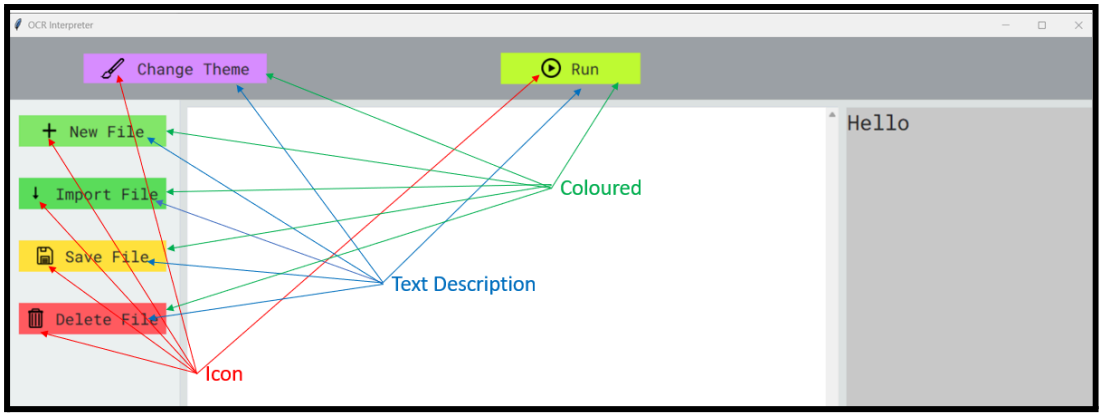
Shao Shen Kuar	Toby Low	Tulin Kasimaga
"The font takes up a respectable amount of space", "The font size in the console and text editor are suitably large"	"Font size is perfect", "Large enough to be easily readable", "The font size in the console and text editor are suitably large"	"The font is entirely readable", "The space is maximised for the font size", "The font size in the console and text editor are suitable large"

Evaluation and Explanation of Success Criterion 4

Success Criterion 4 required screenshots of text both in the text editor and the console, which has been provided in the two images above, and statements of acceptance from the interacting stakeholders, multiple of which are shown in the table above. As such, all of the established requirements have been met and evidenced, and to **explain/justify** the evidence being sufficient, font size being suitably large is a qualitative piece of data, meaning that it can only be reinforced through the opinions of users testing the IDE. As there have been statements of acceptance from all three available interacting stakeholders, where they provide a representative sample due to their involvement with GCSE pseudocode in some manner, this complete acceptance provides sufficient evidence that Success Criterion 4 has been **fully met**.

Success Criterion 5	
Coloured buttons consisting of a text description and a suitable icon	Screenshots of each button, showing the colour, text description, and icon

Test Evidence for Cross-Referencing with Success Criterion 5



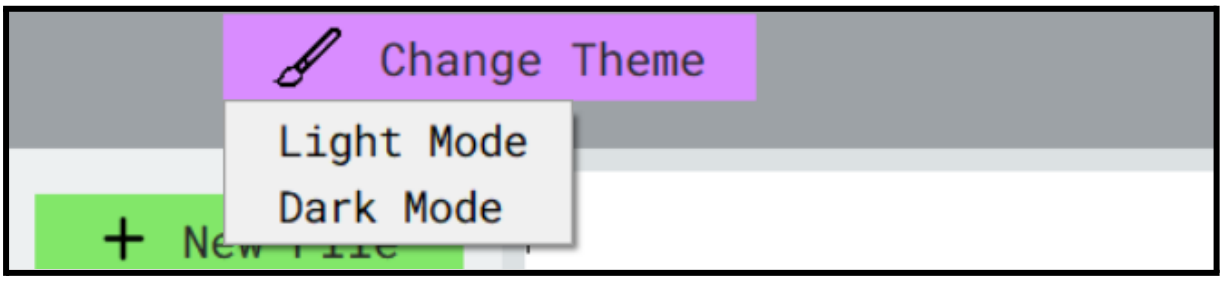
Furthermore, Test 7 in the 'Objective Testing of Usability Features' section displays the full GUI, which shows the buttons in 'Dark Mode' also all with non-dull colours, and the same text descriptions and icons as shown in the above images.

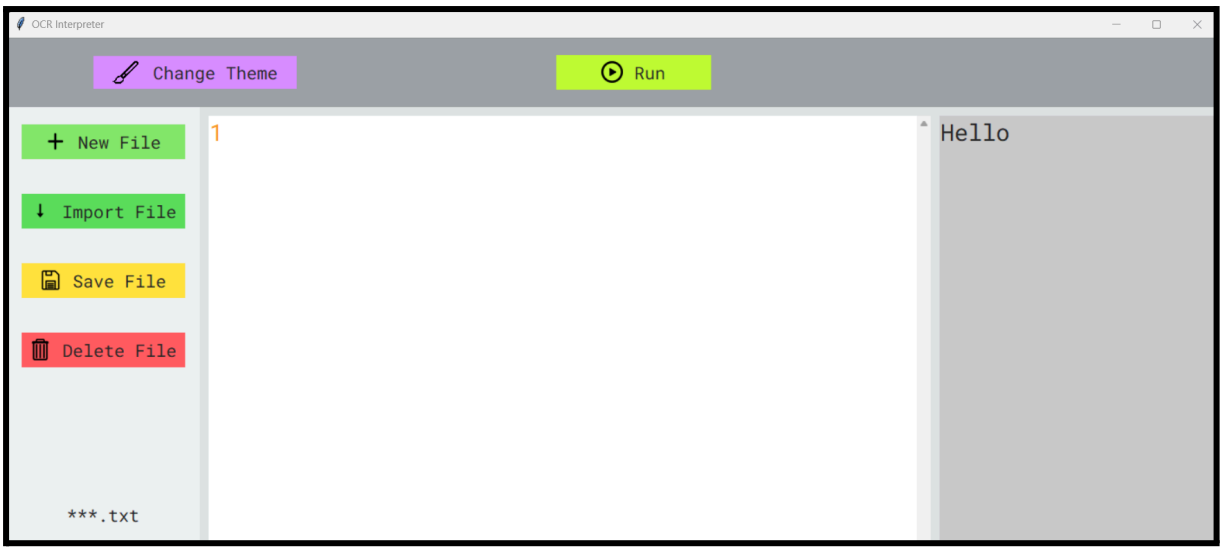
Evaluation and Explanation of Success Criterion 5

Success Criterion 5 required a screenshot showing all of the buttons with their colours, text descriptions, and icons clearly displayed. As the screenshot above shows the buttons, with their colours being marked clearly by the green arrows, their text descriptions being marked clearly by the blue arrows, and the icons being marked clearly by the red arrows, this requirement has been met and evidenced, and to **explain/justify** this evidence being sufficient, the presence of colour, text, and images is an objective test (and thus by viewing it here in the above screenshot, the test has passed successfully) and the buttons are static (they are unchanging on further interactions), so there is sufficient evidence to state that Success Criterion 5 has been **fully met**.

Success Criterion 6	
'Change Theme' button opens a drop-down of themes for 'Dark Mode' and 'Light Mode'	Screenshot of drop-down menu with themes 'Dark Mode' and 'Light Mode' after clicking 'Change Theme' + Screenshot of 'Dark Mode' IDE after clicking 'Dark Mode' + Screenshot of 'Light Mode' IDE after clicking 'Light Mode'

Test Evidence for Cross-Referencing with Success Criterion 6





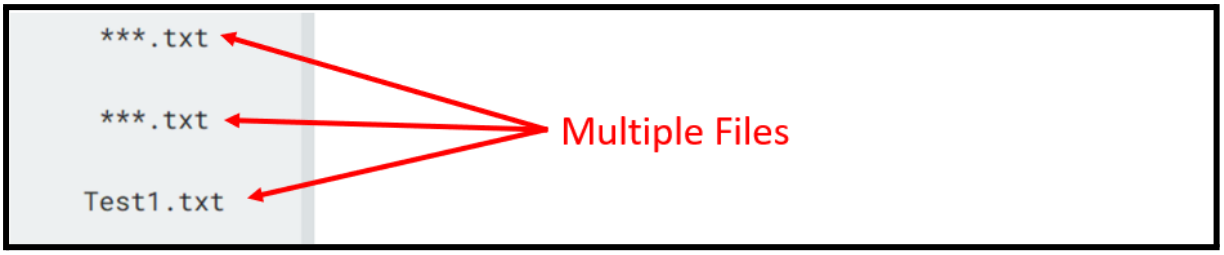
Furthermore, Test 7 in the 'Objective Testing of Usability Features' section displays the full GUI in 'Dark Mode', which occurred after 'Change Theme' and 'Dark Mode' were clicked in sequence, providing further evidence that both 'Change Theme' and 'Dark Mode' work as expected.

Evaluation and Explanation of Success Criterion 6

Success Criterion 6 required a screenshot of the menu after clicking 'Change Theme', which is evidenced by the first screenshot provided, the IDE in 'Dark Mode' after 'Dark Mode' was clicked, evidenced by the second screenshot provided, and the IDE in 'Light Mode' after 'Light Mode' was clicked, which is evidenced by the third screenshot provided. As such, all requirements have been met and evidenced, and to **explain/justify** this evidence being sufficient, both themes are specific in the sense that their colour palette has been pre-defined, and by cross-referencing the intended colour palette with the above screenshots, one can see that they are identical (which is an objective measure), and as such, there is sufficient evidence to state that Success Criterion 6 has been **fully met**.

Success Criterion 7	
File hierarchy consisting of file names (support multiple) that can be clicked to be opened	Screenshot of file hierarchy consisting of file names + Screenshot of file being opened in the text editor after clicking it in the hierarchy

Test Evidence for Cross-Referencing with Success Criterion 7



Furthermore, in Test 2 of 'Shao Shen Kuar's Functional Testing' and Test 4 of 'Tulin Kasimaga's Functional Testing', multiple files can be in the file hierarchy simultaneously, fulfilling the former part of Success Criterion 7.

Evaluation and Explanation of Success Criterion 7

Success Criterion 7 required screenshots of both multiple files in the hierarchy at the same time, which was provided in the first screenshot above (thus showcasing that simultaneous file handling is a successfully implemented feature) (this is sufficient for showcasing for simultaneous file handling, justified/explained as the displayFileAlgorithm is static - it does not change based on any available interactions in the IDE), and the file contents of a file in the file hierarchy in its last open state after clicking the relevant file hierarchy button. To **explain/justify**, the latter screenshot shows that after clicking a populated (non-empty) '***.txt' file, the contents of the text editor did not change (it is unpopulated) and the file currently being worked on did not switch. As such, by providing evidence for one part of Success Criterion 7 and not the other, there is sufficient evidence to state that Success Criterion 7 has been **partially met**.

Addressing Success Criterion 7 in Further Development

To **justify**, the reason that Success Criterion 7 is only partially met is that upon clicking a file in the file hierarchy, the contents of the text editor and the in-system current file variables did not switch to that of the clicked file, which occurred as the fileHierarchyButtonClick functionality was not implemented. This is primarily due to a lack of custom attributes within Tkinter Button objects, thus reducing their flexibility. Ideally, I would store the file location in a custom attribute of the File Hierarchy Button, so that when it is clicked, that location could be retrieved and file-reading could occur (within a few lines of code) to update the text editor with its contents, thus completing the requirement. However, as custom variables do not exist, there was no way to obtain the file location upon button click without creating a file organisation system within the code, which was impractical due to time constraints.

To **address Success Criterion 7 in further development**, provided there were fewer time constraints, I would have to create a file organisation system within the code, which could be implemented by creating a dictionary of Button : location pairs either as a global variable or under a combined class system that involves both elements, such that when a button is clicked, it can be identified and used as a key to the dictionary to obtain the relevant location, from which the same file-reading process could occur (including formatting to re-add line numbers to avoid processing errors and unwanted halting). The use of global variables would lead to maintenance issues, so to avoid that, the combined class system will likely be the best way forward, particularly as this promotes modularity, in which unit-testing is facilitated within this overarching class that surrounds the dictionary.

Success Criterion 8	
‘New File’ button allows an empty (excluding line number), new text file to be created	Screenshot of empty text editor after ‘New File’ is clicked + Screenshot of empty file added to the file hierarchy

Test Evidence for Cross-Referencing with Success Criterion 8

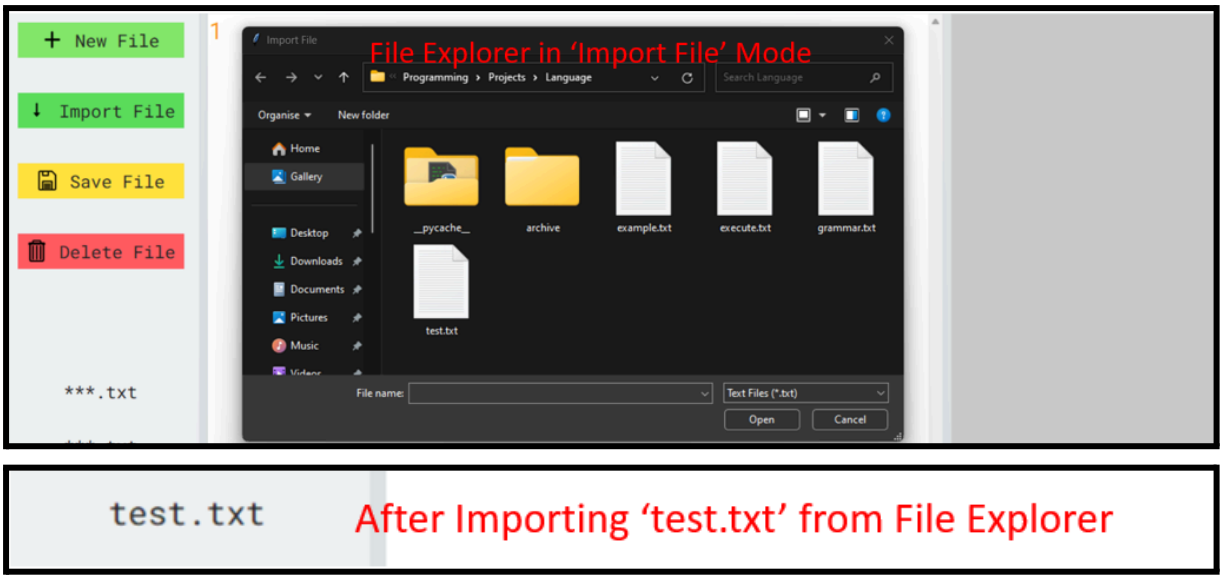
Test 3 of the ‘Testing for Robustness’ section and Test 1 of the ‘Objective Testing of Usability Features’ section also tests the click of ‘New File’ directly, and there you can see both the empty text editor contents and empty file appended to the file hierarchy after clicking ‘New File’, thus fulfilling both of the above requirements.

Evaluation and Explanation of Success Criterion 8

Success Criterion 8 required both a screenshot of the empty text editor and of the empty file (‘***.txt.’) appended to the file hierarchy after clicking ‘New File’. By providing evidence of both aspects in the annotated screenshots above, we can see that both requirements have been met and evidenced, and to **explain/justify** this evidence being sufficient, the displayFileHierarchy() and the NewFileButtonClick() function are static - they do not change based on any available interactions within the IDE, such that the above evidence is equivalent to confirming ‘New File’ works at any point in a session. As such, from the above screenshots, there is sufficient evidence to state that Success Criterion 8 has been **fully met**.

Success Criterion 9	
‘Import File’ button opens File Explorer, allowing an existing text file to be selected	Screenshot of File Explorer being opened after clicking ‘Import File’ + Screenshot of the file appended to the file hierarchy

Test Evidence for Cross-Referencing with Success Criterion 9



Test 8 of ‘Shao Shen Kuar’s Functional Testing’ also shows that after ‘Import File’ is clicked, the file hierarchy is opened allowing a file to be imported, after which the text editor and the file hierarchy is updated to reflect the import, thus satisfying the above requirements.

Evaluation and Explanation of Success Criterion 9

Success Criterion 9 required both a screenshot of File Explorer being opened after ‘Import File’ was clicked, which is evidenced in the first screenshot shown above (specifically, in ‘Import File’ mode as evidenced by the ‘Open’ button in the bottom-right) and a screenshot of the file hierarchy having the appended, imported file (as evidenced in the second screenshot shown above (where ‘test.txt’ was imported successfully). As such, both requirements have been fulfilled, and to **explain/justify** this evidence being sufficient, ‘Import File’ has functionality limited to ensuring that File Explorer is opened in ‘Import Mode’, and that the text editor and file hierarchy is updated (this functionality is static, thereby not changing), so the above screenshots and the reflection of the text editor changing in Test 8 of ‘Shao Shen Kuar’s Functional Testing’ provides sufficient evidence to state that Success Criterion 9 has been **fully met**.

Success Criterion 10	
‘Save File’ button saves file currently opened in the text editor	Screenshot of saved file with changes before and after clicking the ‘Save File’ button respectively

Test Evidence for Cross-Referencing with Success Criterion 10

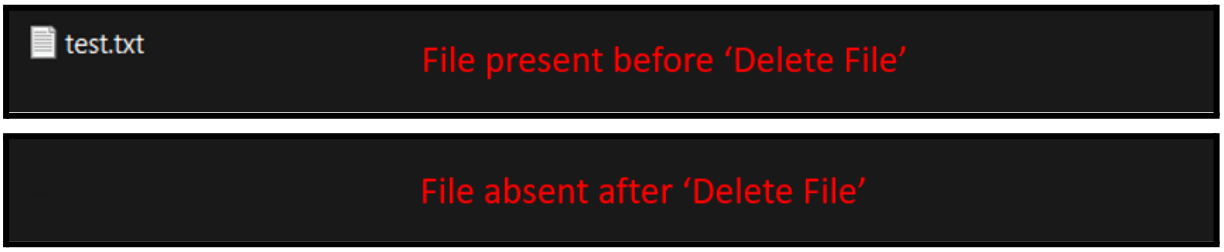
Test 3 of the ‘Testing for Robustness’ section and Test 3 of the ‘Objective Testing of Usability Features’ also showcase that after ‘Save File’ is clicked, depending on the current state of the file (saved or unsaved), the ‘Save As’ functionality or ‘Save’ functionality is executed, and the on-device file contents are updated in either case, thus fulfilling the above requirement.

Evaluation and Explanation of Success Criterion 10

Success Criterion 10 required screenshots of the on-device file contents before and after ‘Save File’ was clicked, ensuring that the file contents were updated with that of the text editor, and this evidence was provided in the before and after screenshots above, thus fulfilling the requirement. To **explain/justify** this evidence being sufficient, ‘Save File’ has functionality limited to updating the on-device file contents, and this functionality is static in that it does not change under available interactions in the IDE. As such, by providing screenshots of the file contents being updated at the above point, this is equivalent to showing ‘Save File’ works more generally at any point in the session. As such, there is sufficient evidence to state that Success Criterion 10 has been **fully met**.

Success Criterion 11	
‘Delete File’ button deletes file currently opened in the text editor	Screenshot of file present and absent in its intended location before and after clicking the ‘Delete File’ button respectively

Test Evidence for Cross-Referencing with Success Criterion 11



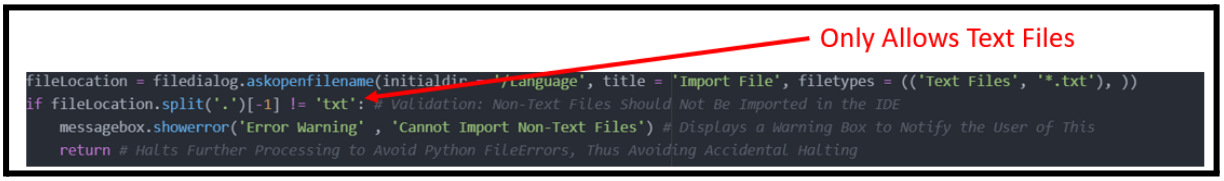
Furthermore, Test 6 of ‘Tulin Kasimaga’s Functional Testing’ and Test 4 of ‘Objective Testing of Usability Features’ showcases that clicking ‘Delete File’ leads to the on-device version of the aforementioned file in the text editor becoming absent where it was once present, thus fulfilling the above requirement.

Evaluation and Explanation of Success Criterion 11

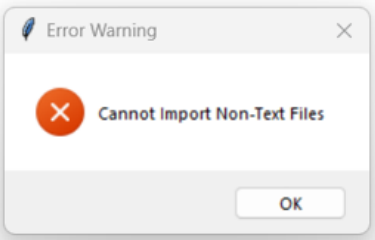
Success Criterion 11 required screenshots of the current text editor file present before ‘Delete File’ was clicked (assuming an uncorrupted file), which was evidenced by the first screenshot shown above, and one of the aforementioned file being absent from its previous location after ‘Delete File’ was clicked (thus evidencing its deletion), which was evidenced by the second annotated screenshot shown above. To **explain/justify** this evidence being sufficient, ‘Delete File’ has functionality limited to removing the on-device version of the current text editor file (and this functionality is static, such that it does not change with the available IDE interactions). As such, the above screenshots showcasing this exact functionality at a given point is equivalent to showcasing that it works more generally throughout an IDE session. As such, there is sufficient evidence to state that Success Criterion 11 has been **fully met.**

Success Criterion 12	
‘Import File’ only enables the opening of text files	Screenshot of section of code where this is implemented + Screenshot of error messages produced when non-text files are opened

Test Evidence for Cross-Referencing with Success Criterion 12



Error message when PNG File is attempted to be opened



Furthermore, Test 2 of 'Testing for Robustness' and Test 2 of 'Objective Testing of Usability Features' showcases that when a non-text file is attempted to be imported, the same error message / warning box appears, thus fulfilling the above requirement.

Evaluation and Explanation of Success Criterion 12

Success Criterion 12 required a screenshot of the section of code in which non-text file validation occurred, which is provided in the first screenshot above (including further annotations with comments) and a screenshot of the warning message that is displayed when one attempts to import a non-text file, which is provided in the second screenshot above (and in further testing in the paragraph above). To **explain/justify** this evidence being sufficient, there is one path of functionality when a non-text file is imported (validated immediately after the file location is obtained), and this is static, meaning it does not change given any available interaction in the IDE. As such, by showcasing that the error warning appears at the point above, this is equivalent to showcasing that the non-text file validation works correctly at any point during an IDE session. As such, there is sufficient evidence to state that Success Criterion 12 has been **fully met**.

Success Criterion 13	
'Run' button executes program, updating console where applicable	Screenshots of console being updated (where applicable) after clicking the 'Run' button

Test Evidence for Cross-Referencing with Success Criterion 13

```
1 print( "Hello World" )
```

```
Hello World
```

Console Updated with Valid Input

```
1 print( % )
```

```
InvalidTokenError at
Line Number 1:
Unrecognised Token
```

Console Updated with Invalid Input

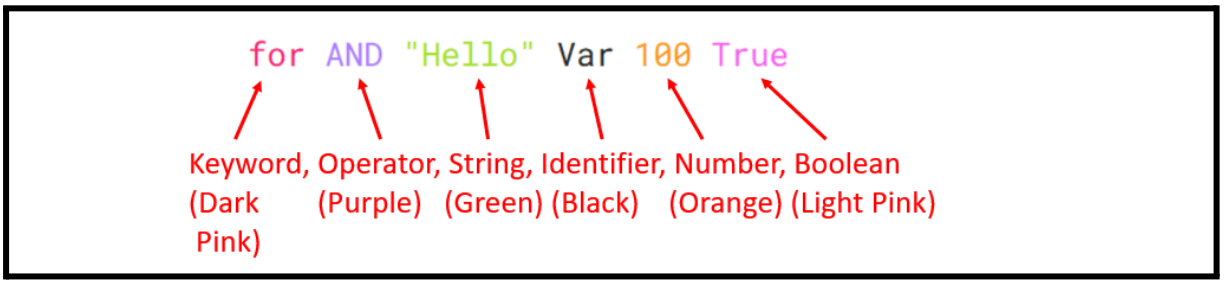
Furthermore, all tests in the 'Robust Testing of Interpreter' section showcase the console being updated to various text editor states (including input, print, and error output modes) after clicking the 'Run' button, thus fulfilling the above requirement.

Evaluation and Explanation of Success Criterion 13

Success Criterion 13 required a screenshot of the console being updated after the 'Run' button was clicked, two of which are provided in the above images (both for valid and invalid input, thus showcasing it working for both modes while displaying the different colour palettes), thus fulfilling the requirement. To **explain/justify** this evidence being sufficient, the console's output is linked in code only to the 'Run' button and this functionality is static, such that it does not change given any available IDE interaction. As such, by providing the screenshots above of the console being updated after 'Run' is clicked, this is equivalent to showcasing that it works generally throughout an IDE session. As such, there is sufficient evidence to state that Success Criterion 13 has been **fully met**.

Success Criterion 14	
Pretty printing in the text editor	Screenshots of each keyword category being highlighted a particular colour

Test Evidence for Cross-Referencing with Success Criterion 14



Furthermore, in all of the tests of 'Robust Testing of Interpreter', the code-execution tests of each stakeholder's functional testing, and Test 7 of the 'Objective Testing of Usability Features' section, images of pretty printing (with colours consistent with the image above) exist for different platforms, thus fulfilling the above requirement.

Evaluation and Explanation of Success Criterion 14

Success Criterion 14 required a screenshot of each keyword category being highlighted a particular colour, and this is provided in the screenshot above (colours are chosen to maximise contrast, with the black of the Identifier - or white in the case of 'Dark Mode' - chosen to express the relative generality of identifiers), thus fulfilling this requirement. To **explain/justify** this evidence being sufficient, the colours themselves are static (unchanging, except for Identifiers, though this is evidenced in the tests provided in the above paragraph), meaning that they are unchanging amidst available IDE interactions. As such, by showcasing that keywords are highlighted with particular colours in the above screenshot, we show that pretty printing works more generally throughout the IDE session, and this is especially true as stakeholder functional testing showed it worked across different devices. As such, we have sufficient evidence to state that Success Criterion 14 has been **fully met**.

Success Criterion 15	
Vertically scrollable text editor with line numbers	Screenshot of text editor with line numbers (incremented by 1 each line) + Screenshot of working vertical scroll bar

Test Evidence for Cross-Referencing with Success Criterion 15

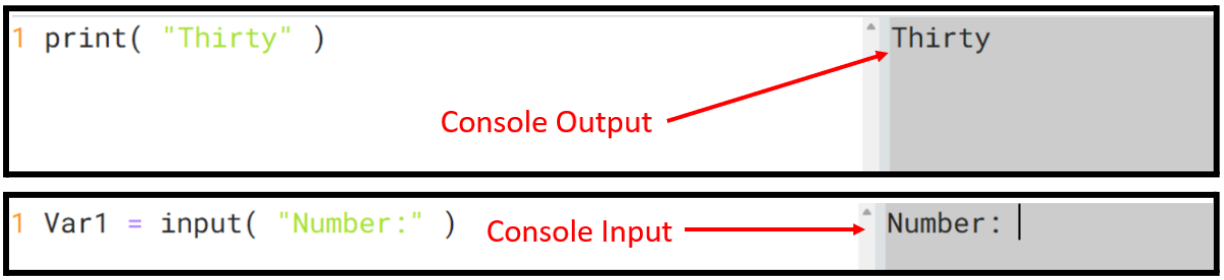
Furthermore, Test 7 from 'Toby Low's Functional Testing' displays the full GUI, in which a line number and the text-editor vertical scroll bar can be seen, thus fulfilling both aspects of the above requirement.

Evaluation and Explanation of Success Criterion 15

Success Criterion 15 required screenshots of the incrementing line numbers, which is evidenced in the first screenshot provided above (incrementing from 1 to 5 as there are only 5 lines in this program) and the vertical scroll bar of the text editor, which is evidenced in the second screenshot provided above (vertical as it moves up and down, as annotated) thus fulfilling both elements of the requirement. To **explain/justify** this evidence being sufficient, the line-number-adding function triggered upon the key release of Return and the Scroll Bar UI element are both static, meaning that they are unchanging with the available IDE interactions. As such, showcasing that line numbers and the scroll bar work at the above point is equivalent to showcasing that they work throughout the session. As such, there is sufficient evidence to state that Success Criterion 15 has been **fully met**.

Success Criterion 16	
Console has input/output capabilities	Screenshot of console output after a program is run by pressing the 'Run' button + Screenshot of console input after a program requiring input is run by pressing the 'Run' button

Test Evidence for Cross-Referencing with Success Criterion 16



Furthermore, Test 2 from 'Toby Low's Functional Testing', Test 4 from 'Shao Shen Kuar's Functional Testing', and the 'Robust Testing of Interpreter' section showcases both console output and input occurring after print() and input() are called respectively and 'Run' clicked, thus fulfilling the above requirement.

Evaluation and Explanation of Success Criterion 16

Success Criterion 16 required screenshots of both console output, which is evidenced by the first screenshot shown above, and input, which is evidenced by the second screenshot shown above, (calling print() and input() respectively) after clicking the 'Run' button, thus providing evidence that the requirement has been fulfilled. To **explain/justify** this evidence being sufficient, we have performed robust input() and output() checks in Stage 1 of Iterative Development, 'Robust Testing of Interpreter', and during the stakeholders' functional testing. Furthermore, there is little variation that can occur with these functions (input(), for example, takes a single string input, allowing us to abstract the variation in those strings, meaning that our testing input("Number:") is equivalent to testing input(ANY STRING) in effect, particularly as there is complete flexibility in code in what those strings are - no hard-coding or pre-specifying values). As such, there is sufficient evidence to state that Success Criterion 16 has been **fully met**.

Success Criterion 17	
The interpreter must produce the correct output for a program consisting of any of the statement types mentioned in the OCR GCSE Pseudocode documentation	Thorough white-box testing for each foreseeable usage of syntax + Faultless beta black-box testing results from the interacting stakeholders

Test Evidence for Cross-Referencing with Success Criterion 17

Thorough White-Box Testing		
OCR Syntax Item	**Number in 'Testing for Robustness'**	**All Tests Passed?**
Loop Structures	10, 11, 12	Yes ✅

Conditional Statements	13, 14	Yes ✅
Arithmetic Operators	15	Yes ✅
Boolean Operators	16	Yes ✅
Functions	17	Yes ✅
Procedures	18	Yes ✅
Identifier Scope	19	Yes ✅

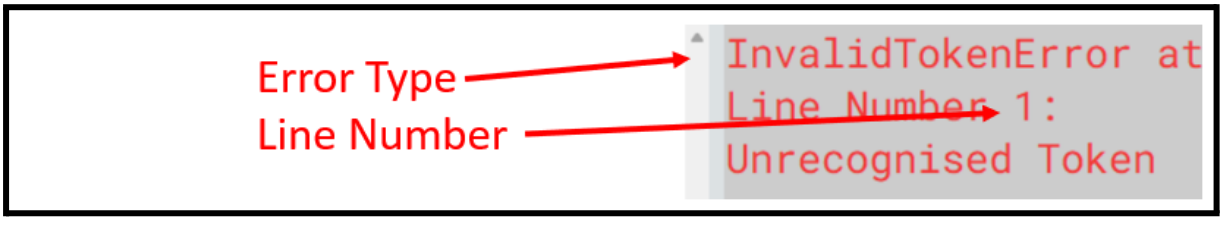
Beta Black-Box Testing		
Stakeholder Name	**Test Quantity in 'Functionality Testing'**	**All Tests Passed?**
Shao Shen Kuar	8	Yes ✅
Toby Low	7	Yes ✅
Tulin Kasimaga	5	Yes ✅

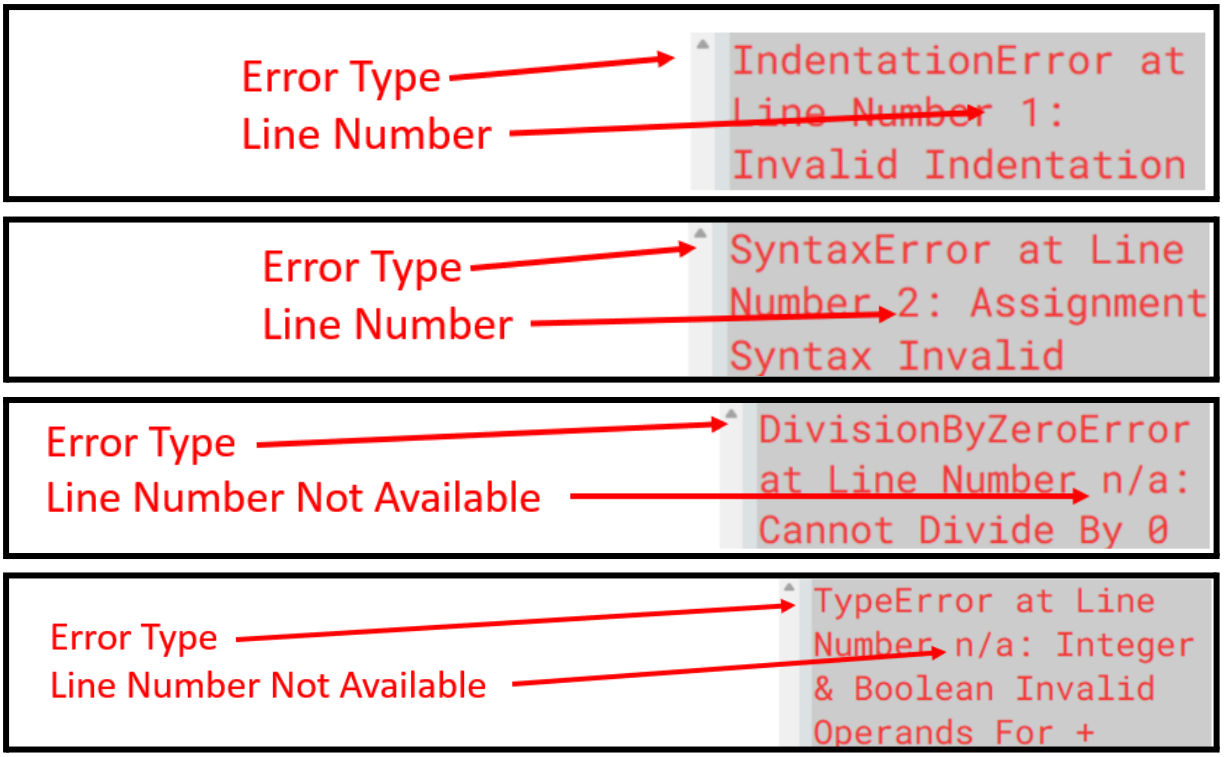
Evaluation and Explanation of Success Criterion 17

Success Criterion 17 required thorough white-box testing of all OCR syntax features within the interpreter, and this is evidenced by the entire 'Robust Testing of Interpreter' section, which is summarised in the white-box testing table shown above, and faultless beta black-box testing from all interacting stakeholders, as evidenced by the black-box testing table shown above (faultless as all test cases that the stakeholders tested passed and all interacting stakeholders are involved as the project began and remained with three - Shao Shen Kuar, Toby Low, and Tulin Kasimaga), thus fulfilling the above requirement. To **explain/justify** this evidence being sufficient, the checklist that was produced during 'Test for Robustness' stage was produced by cross-referencing with the GCSE OCR Pseudocode syntax, thus containing all of the available syntax features as the testing was robust (as the verifiably complete checklist was completed ticked). To add, suitable caution was applied to unexpected functionality (by ensuring that others users tried it, which is particularly robust as this was done with all interacting stakeholders). As such, there is sufficient evidence to state that Success Criterion 17 has been **fully met**.

Success Criterion 18	
The interpreter produces and the console consequently outputs error messages consisting of the error type and line number	Screenshots of various error message types, providing evidence of the error type and line number

Test Evidence for Cross-Referencing with Success Criterion 18





Furthermore, Test 6 from 'Testing for Robustness' showcases an error that contains both the error type and line number fulfilling the requirement, while Test 16 and Test 18 from 'Testing for Robustness' showcase errors that contain error types but not line numbers, thus not fulfilling the requirement here.

Evaluation and Explanation of Success Criterion 18

Firstly, to **explain/justify** the requirements being sufficient, each error type showcased in the above screenshots account for all of the error types implemented in this pseudocode IDE, and due to there only being minor variation between errors of the same type (either identical or having different built-in error messages, which does not change the display of error type), there is sufficient evidence to show that part of Success Criterion 18 has been met - that of displaying the error type across the multiple error types available in the IDE. However, when it comes to displaying the line number, TypeError and DivisionByZero error only consistently lack a useful line number to display (consistent as this is true across the test cases stated in the above paragraph). Considering complete success is only achieved when all error types have line numbers, due to there being exceptions with TypeError and DivisionByZeroError, there is sufficient evidence to state that Success Criterion 18 has been **partially met**.

Addressing Success Criterion 18 in Further Development

To **justify**, the reason that Success Criterion 18 was only partially met is due to the failure of TypeErrors and DivisionByZeroErrors to contain useful, accurate line numbers that stated where the error occurred. This arose due to these being the only two error types that were found in the Interpreter (the other error types were returned in the Lexer and Parser), and at the Interpreter stage, there was no access to stored line-number information as the Lexer and Parser had through the accumulation of this information in tokens (because not all token information was copied over during the transition from the Parser to the Interpreter, and more specifically, the transition from the token list to the abstract syntax tree). In attempting to fix the issue, there would need to be a tedious restructuring of each node type in the abstract syntax tree, and within the interpreter itself in the way it handles each node, and this was unachievable due to time constraints.

To **address Success Criterion 18 in further development**, the most straightforward path would be to complete this restructuring, which would involve creating a token attribute (or a line number attribute from the token, though this is more difficult as not all nodes are at the same logical level as tokens or are multi-lined in nature) to each node type declared for the abstract syntax tree, thus having providing access of line numbers to the interpreter. From there, I would change each return Error() statement in the interpreter, deleting the 'n/a' string and replacing it with the relevant token's lineNumber attribute, thus fixing the issue entirely (though this stage of changing each error, of which many exist due to the robustness of the interpreter, is the most tedious). As such, I would also attempt to further modularise error types to reduce the time for future error restructurings if they occur. I would do this by grouping error types under different subclasses of Error (e.g. SyntaxError, IndentationError), and creating default getter methods that I would call instead of using return Error()). As such, if I wanted to change how errors worked, I would only need to change the text in a few limited locations (the error type subclasses, of which there would be 5, thus greatly improving development efficiency).

After cross-referencing the success criteria with the test data and evaluating them accordingly into buckets of 'Fully Met', 'Partially Met', and 'Not Met', the results can be summarised in a table:

Success Criterion	Status
1400px by 700px GUI	Fully Met ✅
GUI divided into three primary, different-coloured, vertically divided sections (excluding top bar)	Fully Met ✅
Each primary section has headings 'Files', 'Text Editor', 'Console' (from left to right)	Not Met ❌
Large font size in text editor and console	Fully Met ✅
Coloured buttons consisting of a text description and a suitable icon	Fully Met ✅
'Change Theme' button opens a drop-down of themes for 'Dark Mode' and 'Light Mode'	Fully Met ✅
File hierarchy consisting of file names (support multiple) that can be clicked to be opened	Partially Met 🟣
'New File' button allows an empty (excluding line number), new text file to be created	Fully Met ✅
'Import File' button opens File Explorer, allowing an existing text file to be selected	Fully Met ✅
'Save File' button saves file currently opened in the text editor	Fully Met ✅
'Delete File' button deletes file currently opened in the text editor	Fully Met ✅
'Import File' only enables the opening of text files	Fully Met ✅
'Run' button executes program, updating console where applicable	Fully Met ✅
Pretty printing in the text editor	Fully Met ✅
Vertically scrollable text editor with line numbers	Fully Met ✅
Console has input/output capabilities	Fully Met ✅

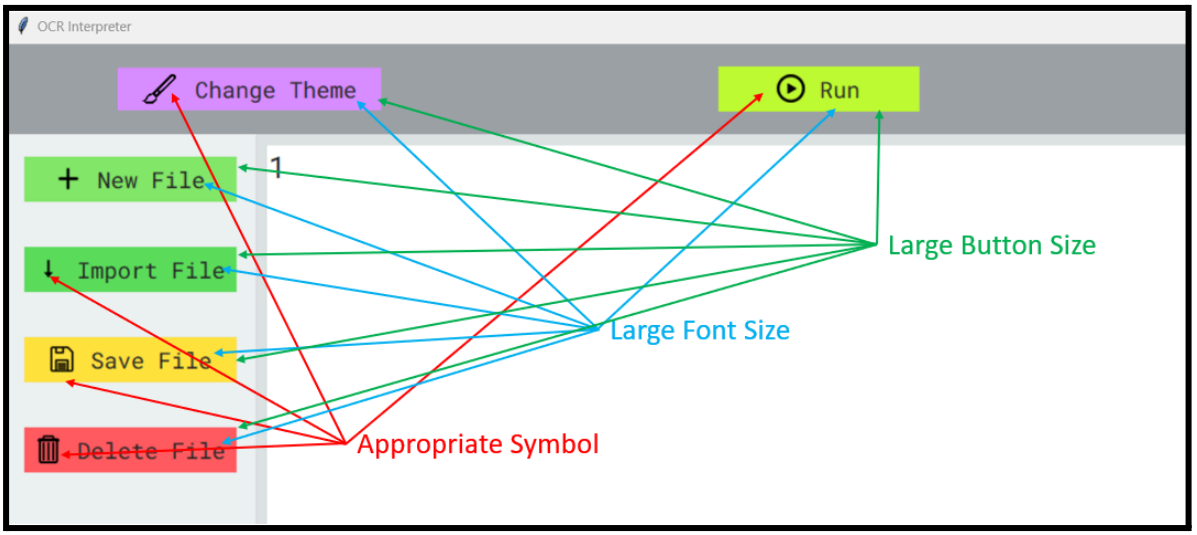
The interpreter must produce the correct output for a program consisting of any of the statement types mentioned in the OCR GCSE Pseudocode documentation	Fully Met ✅
The interpreter produces and the console consequently outputs error messages consisting of the error type and line number	Partially Met 🟣

Evidence of Usability Features

For this section, we shall determine the extent to which each usability feature as specified in the Analysis section has been met by going through them one-by-one, attempting to provide sufficient evidence (and cross-referencing with the 'Annotated Evidence for Usability Testing' section to reinforce the strength of evidence provided), finally **justifying** their success as effective usability features by categorising them as a 'Success', 'Partial Success', or 'Failure'. We shall begin with the first usability feature and increment from there.

Usability Feature 1
Large buttons (with a large font size and an appropriate symbol attached). True for all buttons 'Run', 'Change Theme', 'New File', 'Import File', 'Save File', and 'Delete File'.

Evidence of Usability Feature 1



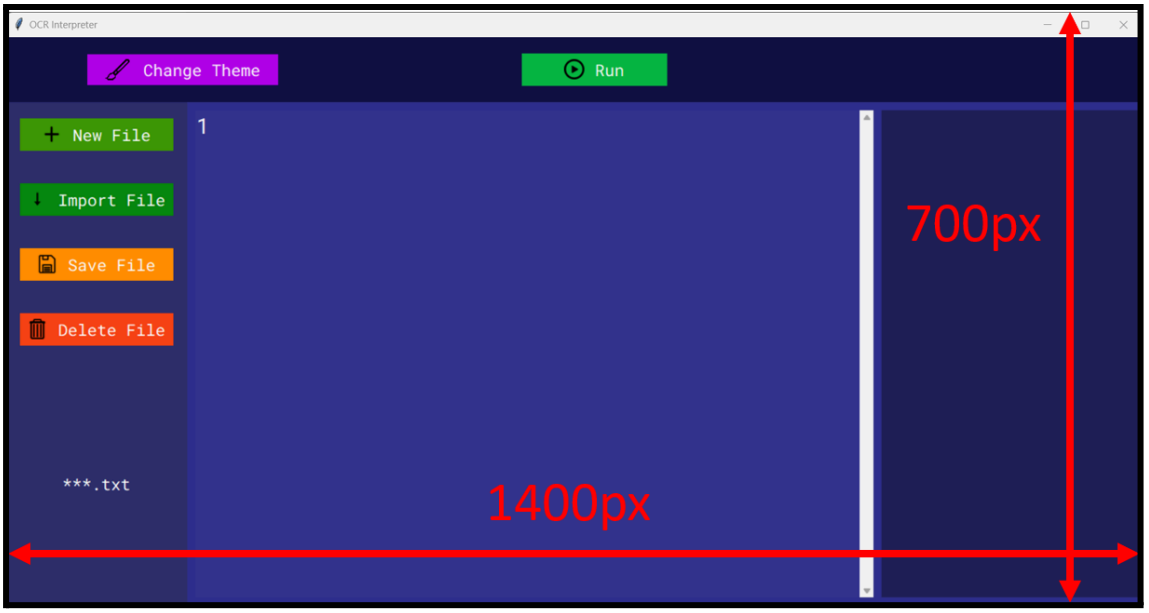
Button Feature	**Sufficiently Positive Comments?**		
	Shao Shen Kuar	**Toby Low**	**Tulin Kasimaga**
Large Button Size	"Clearly visible on the screen" ✅	"They are more than large enough" ✅	"Buttons are completely large enough" ✅
Large Font Size	"Takes a respectable amount of space" ✅	"Font size is perfect" ✅	"Entirely readable" ✅
Appropriate Symbol	"Illustrates what you want it to" ✅	"One of the highlights of the IDE" ✅	"Symbols are perfectly fine" ✅

Justifying Usability Feature 1 as an Effective Usability Feature

The evidence provided was a screenshot of all buttons ('New File', 'Import File', 'Save File', 'Delete File', 'Run', and 'Change Theme') with annotations of their large button size, font size, and appropriate annotations, thus showcasing evidence that the buttons exist and all features specified are fulfilled (with the size at least objectively larger than the default Tkinter font size, thus designed to be generally readable) and a table of sufficiently positive comments from all interacting stakeholders to provide qualitative feedback on the relative sizes of each object. To **justify** this usability feature as effective, the sufficiency of each item's size is subjective and depends on a given user's eyesight strength and reading preferences, with the screenshot being supplemental as they are evidently large enough on first inspection. As all users agreed that the size of the buttons and button font were sufficiently large and the symbols sufficiently appropriate, noting their clear visibility on the screen and the quantity and quality of information that vividly represents the function of the button (particular the 'Red' and the 'Garbage Can' for 'Delete File' indicating the permanent consequence of file deletion), there is sufficient evidence to state that Usability Feature 1 as an effective usability feature is a **Success**.

Usability Feature 2
1400 x 700 Window Size

Evidence of Usability Feature 2



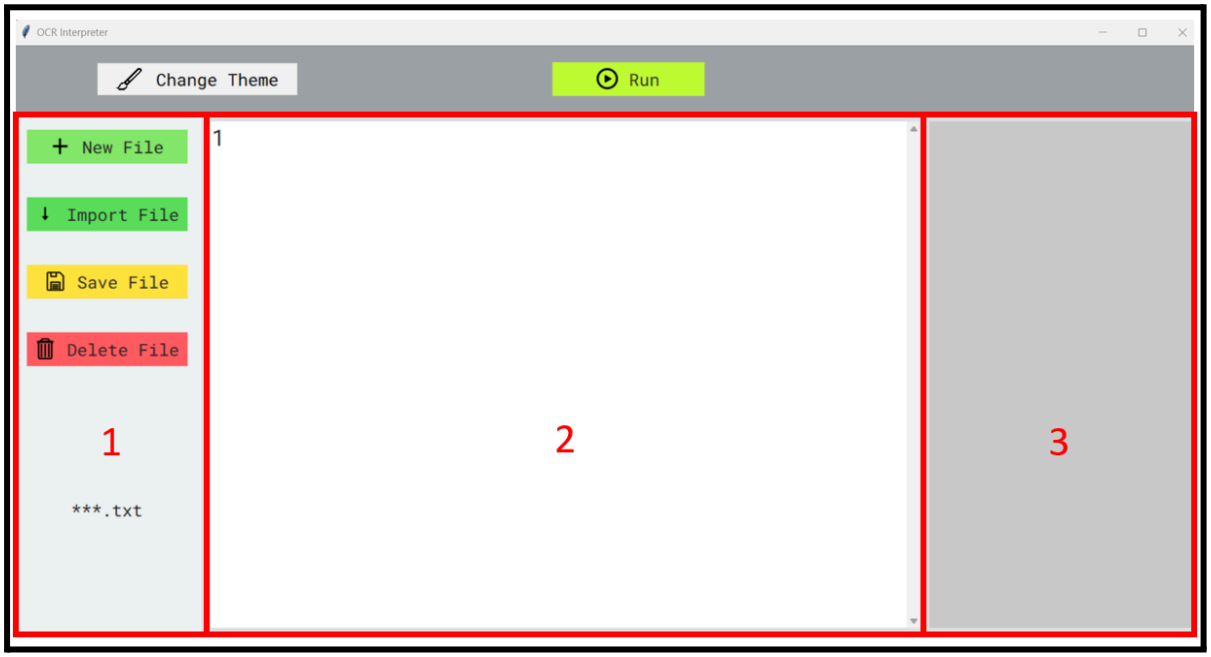
Sufficiently Positive Comments about Window Size?		
Shao Shen Kuar	**Toby Low**	**Tulin Kasimaga**
"This window size is also great" ✅	"Ideal window size", "Large enough to be suitably readable" ✅	"This is the right choice, particularly because it is large" ✅

Justifying Usability Feature 2 as an Effective Usability Feature

The evidence provided was a screenshot which showed that the window size was exactly 1400 by 700 px, which was taken after measurement and further evidenced when evidencing Success Criterion 1, and sufficiently positive comments from all of the interacting stakeholders indicating that the window size, at 1400px by 700px, is preferred and does not require changing. To **justify** this usability feature as effective, screen size is an objective, quantitative variable, and it suffices to provide measurements (and code displays in Success Criterion 1) which verifiably prove that this window size requirement has been met. There is further **justification** for selecting 1400 by 700 px as well with the sufficiently positive comments from all available interacting stakeholders, noting that the window size was large enough for readability without the issues of full-screen borderless (no simultaneous window opening here, unwantedly breaking up coding sessions). As such, there is sufficient evidence to suggest that Usability Feature 2 as an effective usability feature is a **Success**.

Usability Feature 3
Few Primary UI Elements (only the essentials of the file-handling window, console, and text editor)

Evidence of Usability Feature 3



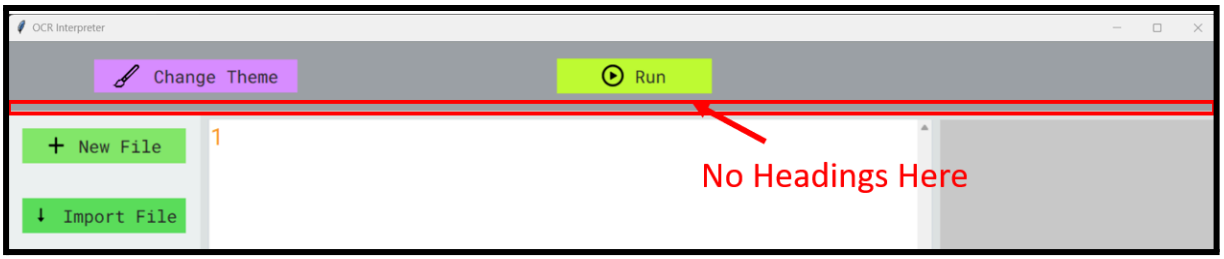
Sufficiently Positive Comments about UI Elements?		
Shao Shen Kuar	**Toby Low**	**Tulin Kasimaga**
“There being few UI elements makes perfect sense” ✅	“I believe that this is the ideal combination of elements” ✅	“Minimising actions to only the essentials maximises the space you have for each” ✅

Justifying Usability Feature 3 as an Effective Usability Feature

The evidence provided was a screenshot showing the division of the GUI into the three primary components (1: File-handling window, 2: Text editor, 3: Console), thus evidencing that the above requirement of these three specific sections being displayed as the only primary components has been met, and a table of positive comments reinforcing the success of achieving this Usability Feature across the three interacting stakeholders. To **justify** this usability feature as effective, there being specific primary components (named as above) is an objective measure and through the annotations in the first screenshot and their internal contents, this has objectively been met. There is further **justification** with there being sufficiently positive comments from all interacting stakeholders (largest available, reliable sample size at the time of obtaining evidence), noting that there were indeed few UI elements (a subjective, qualitative measure that seems to be agreed-upon universally by the current user base) and that the combination is ideal, noting further that by reducing the quantity of UI elements, there is extra space for other features like text editor font size (improves readability and reduces eye strain). As such, there is sufficient evidence to state that Usability Feature 3 as an effective usability feature is a **Success**.

Usability Feature 4
Primary Window Headers ('Files', 'Console', 'Text Editor')

Evidence of Usability Feature 4



Test 7 from 'Toby Low's Functional Testing' displays the full GUI in 'Dark Mode' and when viewing the screenshot, there is once again a lack of headings in the UI at the post-development stage.

Would you prefer for there to be headings?		
Shao Shen Kuar	**Toby Low**	**Tulin Kasimaga**
"I would, particularly as it was a previously declared usability feature", "Would be ideal for those without IDE experience" ❌	"Honestly, I don't mind. Headings would certainly be useful, especially for those who haven't used IDEs before, but it doesn't affect me" 🟣	"A decent amount of my students haven't really coded before, or at least they do in differently styled IDEs. Headings would help with organisation" ❌

Justifying Usability Feature 4 as an Effective Usability Feature

Though no headings were implemented, to **justify** them as effective usability features, headings have an organisational purpose, ideal for naming disparate UI elements (the file-handling window, the text editor, and the console), particularly for those who have not had IDE experience and are unfamiliar with structural paradigms. Because there are few IDE elements, further organisational provisions are not a requirement, but it would have improved usability. Further **justification** is that

the comments of the headings criticised their absence (as evidenced by the lack of headings in the screenshot provided above), noting the point of user experience and the intuitiveness of understanding the IDE layout initially. As such, there is sufficient evidence to suggest that Usability Feature 4 as an effective usability feature is a **Failure**.

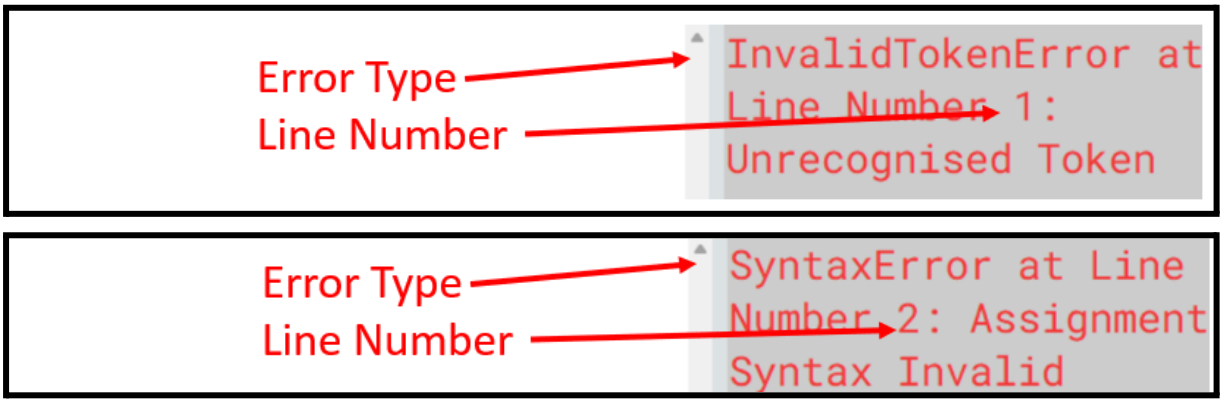
Addressing Usability Feature 4 in Further Development

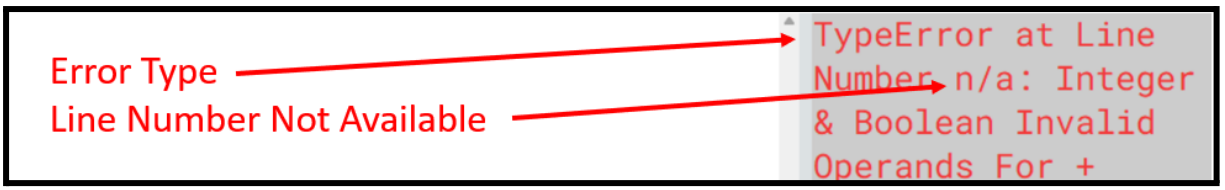
To **address the implementation of Usability Feature 4 in further development**, I would first complete the necessary GUI restructuring by reducing the heights of the primary window elements (specifically, in code, the FileOperationBar, FileHierarchyBar, TextEditor, and Console), reducing the heights and widths of the internal window elements (NewFileButton, ImportFileButton, SaveFileButton, DeleteFileButton, and fileHierarchyObject) to leave space for the headers, create the 'Heading Row' by creating a Label object that takes the width of the screen (1400px) and a suitable height to have 'Files', 'Text Editor', and 'Console' in a sufficiently large, readable font, reduce the height of TopBar, thus also reducing the font size and height of ThemeButton and RunButton, and from there, the headings would be completely implemented (no need to worry about size changes, justified as the 'Heading Row' is static). To add, because the issue arose out of the difficulty in changing hard-coded Tkinter padding, height, and width values, I would also change the values from being hard-coded to being a percentage of the screen or window height/width, such that element size attributes are responsive to changes in their parent frames, and that fewer sizes need to be changed before complete GUI restructurings are completed successfully.

To **address maximising the effectiveness of Usability Feature 4**, the headers would have to be a suitably large font size, which would maximise readability and lead to less eye strain, and considering the text editor font size was evidenced as suitably large, approximately 20 pixels should be sufficient. Furthermore, there would need to be sufficient vertical padding between the text and the edges of the 'Heading Row', as this would lead to avoid the visual effect of collision between disparate items, thus improving the aesthetics of the headings and the overall GUI. Once these are fulfilled, Usability Feature 4 would also be effective.

Usability Feature 5
Logical Formatting for each error message including the error type, an error message, and the line number

Evidence of Usability Feature 5





Sufficiently Positive Comments about Error Format?		
Shao Shen Kuar	**Toby Low**	**Tulin Kasimaga**
"Overall, the error messages are really specific, more so than Python even, but DivisionByZero errors for example never show a line number, which can make error identification difficult" 🟣	"I like that all the required elements are at least partially there - type, line number, message. However, TypeErrors tend to not have line numbers, which is unfortunate because they are common" 🟣	"It's unfortunate that TypeErrors do not have line numbers - they're quite common, but overall, the error format is very descriptive and specific, making debugging easier." 🟣

Justifying Usability Feature 5 as an Effective Usability Feature

To **justify** the current error format (Error type, Line number, Error message) as being an effective usability feature overall, the positive aspect of the above comments described the intended format itself as highly descriptive, specific, and fulfilling the pre-set requirements, where specificity is important to both address the location of the error and describe its nature, thus helping in error identification for faster debugging, thus increasing proportionally the time spent coding new features (improving productivity). However, the only issue with the current implementation is that line numbers are not implemented for TypeErrors and DivisionByZeroErrors due to them being node-interpreter-produced errors, thus not fulfilling the line number requirement of the error format usability feature partially. As such, there is sufficient evidence to suggest that Usability Feature 5 as an effective usability feature is a **Partial Success**.

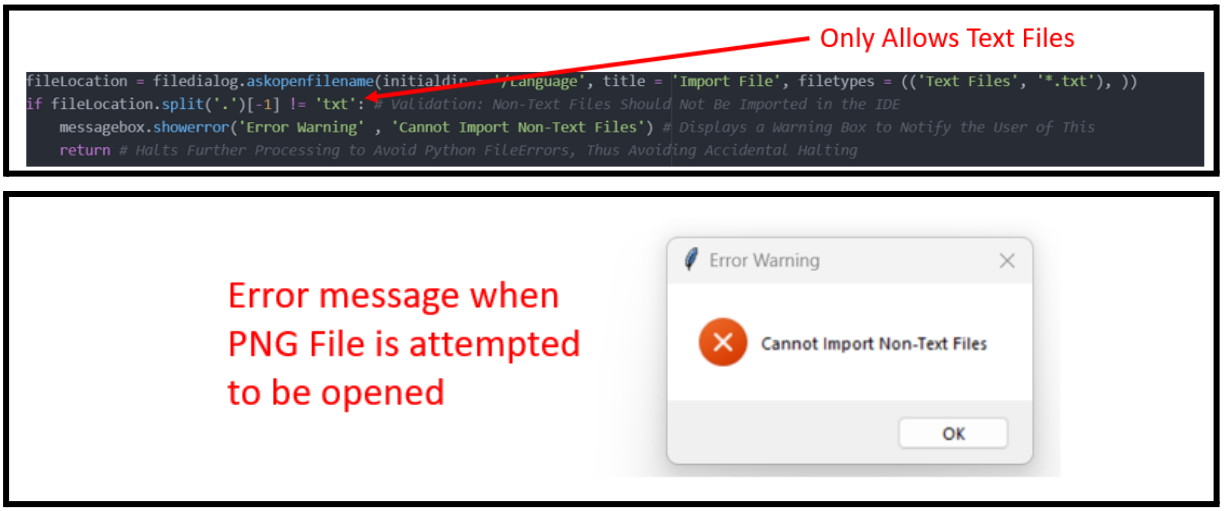
Addressing Usability Feature 5 in Further Development

To **address the implementation of Usability Feature 5 in further development**, the most straightforward path would be to complete the restructuring of the parser-node-interpreter interface, which would involve creating a token attribute (or a line number attribute from the token, though this is more difficult as not all nodes are at the same logical level as tokens or are multi-lined in nature) to each node type declared for the abstract syntax tree, thus having providing access of line numbers to the interpreter. From there, I would change each return Error() statement in the interpreter, deleting the 'n/a' string and replacing it with the relevant token's lineNumber attribute, thus fixing the issue entirely (though this stage of changing each error, of which many exist due to the robustness of the interpreter, is the most tedious). As such, I would also attempt to further modularise error types to reduce the time for future error restructurings if they occur. I would do this by grouping error types under different subclasses of Error (e.g. SyntaxError, IndentationError), and creating default getter methods that I would call instead of using return Error()). As such, if I wanted to change how errors worked, I would only need to change the text in a few limited locations (the error type subclasses, of which there would be 5, thus greatly improving development efficiency).

To **address maximising the effectiveness of Usability Feature 5**, when implementing the line numbers of TypeErrors and DivisionByZeroErrors, validation should be coded to ensure that the tokens contain an integer line number attribute (thus avoiding unwanted printing of extraneous, confusing information) and thorough white-box testing should be completed to ensure that these line numbers accurately reflect the true line number of the error, as inaccurate information may lead to attempting to identify errors in the wrong locations, which would reduce productivity and lead to greater time wasted, decreasing user satisfaction.

Usability Feature 6
File interactions are limited to text files

Evidence of Usability Feature 6



Sufficiently Positive Comments about Text File Limitation?		
Shao Shen Kuar	**Toby Low**	**Tulin Kasimaga**
"I think this is the most practical decision. Enabling non-text files would lead to bulkier validation code when nothing could be done with them (due to not being supported by OCR), which is poor optimisation" ✅	"The error messages are really useful. Obviously, there is no support for non-text files by OCR, so it's good that file hierarchy space is not wasted holding non-interactable PNGs for example" ✅	"This was a wise decision. Text files are the ideal format, being accessible, common, and supporting all of the features of pseudocode and nothing more. This promotes simplicity, which makes the experience more intuitive" ✅

Justifying Usability Feature 6 as an Effective Usability Feature

To **justify** Usability Feature 6 as an effective usability feature, the comments regarding the text file limitation are wholly positive, noting particularly the practicality of avoiding bulkier validation code (and unwanted Python halting errors when attempting to "execute" PNG files, for example), maintaining maximal space for pseudo code text files (thus improving organisation), and the acknowledgement that text files themselves are ideal due to their focus on textual characters (not utilising extraneous UI features that pseudocode does not require), thus reducing wasted storage space and enabling a more simple, intuitive user experience (suitable as a major portion of the target demographic is GCSE students). The sufficiently positive comments and the screenshots above providing evidence of implementation of the Usability Feature (in a low-level manner) lead to there being sufficient evidence to suggest that Usability Feature 6, as an effective usability feature, is a **Success**.

Usability Feature 7
Division of colour themes into ‘Light Mode’ and ‘Dark Mode’

Evidence of Usability Feature 7

Sufficiently Positive Comments about Available Colour Themes?		
Shao Shen Kuar	**Toby Low**	**Tulin Kasimaga**
“Yes, ‘Light Mode’ has the simplicity generally associated with light IDE themes and ‘Dark Mode’, with the overarchingly dark blue palette, provides aesthetic contrast”, “Both functional and aesthetically pleasing” ✅	“I would say that ‘Dark Mode’ looks really great, and the blues in particular contrast the buttons well and it feels conducive to productivity. ‘Light Mode’ also provides great contrast” ✅	“I prefer ‘Light Mode’ due to its simplicity, as I do with other IDEs, but I think there is a demonstrable contrast with both themes that makes the IDE quite aesthetically pleasing” ✅

Justifying Usability Feature 7 as an Effective Usability Feature

To **justify** Usability Feature 7 as an effective usability feature, the comments regarding the available colour themes have been deemed sufficiently positive, noting particularly that 'Light Mode' and 'Dark Mode' both offer suitable contrast (thus improving readability and reducing eye strain), while also being aesthetically pleasing (necessary validation as the sufficiency of the colour themes is qualitative). Furthermore, 'Light Mode' is offered as a standard default, which is logical as those inexperienced with IDEs and the 'Dark Mode' trend may work more productively in this familiar environment, whereas 'Dark Mode' provides a darker background to work against, thus being less harsh on the eyes to maximise usability. As such, there is **justified** space for both themes in the option list. By providing evidence of implementation (and general developer qualitative claims) through the two screenshots above and qualitative opinions on the matter that are wholly (and sufficiently) positive, there is sufficient evidence to state that Usability Feature 7 is a **Success**.

After providing evidence of implementation of the Usability Features, and assessing them on a qualitative basis by the judgement of the tone and content of the opinions of the stakeholders on how well-implemented these Usability Features were, the effectiveness of each Usability Feature was evaluated into being a 'Success', 'Partial Success', or 'Failure'. The results are summarised below:

Usability Feature	Status
Large buttons (with a large font size and an appropriate symbol attached). True for all buttons 'Run', 'Change Theme', 'New File', 'Import File', 'Save File', 'Delete File'	Success ✅
1400 x 700 Window Size	Success ✅
Few Primary UI Elements (only the essentials of the file-handling window, console, and text editor)	Success ✅
Primary Window Headers ('Files', 'Console', 'Text Editor')	Failure ❌
Logical Formatting for each error message including the error type, an error message, and the line number	Partial Success 🟣
File interactions are limited to text files	Success ✅
Division of colour themes into 'Light Mode' and 'Dark Mode'	Success ✅

Maintenance Issues

Now, we shall go through maintenance issues, which are aspects of the program's implementation that may lead to difficulties in adapting the code to meet changing requirements, updates that reflect changes in the OCR syntax, and generally present challenges to new developers who wish to completely understand the project so that they can complete these updates or change the code as Python (the base programming language) updates (addressing maintenance). Below, we shall go through the maintenance issues one-by-one and discuss development schemes to handle these maintenance issues in the future given sufficient time:

Maintenance Issue	Development to Deal With Maintenance Issue and Potential Improvements/Changes
In the handleAlpha() function of the Lexer class, when defining a specialist keyword (e.g. AND,	The primary issue here is the repeating 'elif' statements (internal contents are repeated too)

DIV, for), there is a large quantity of 'if' and 'elif' statements which look nearly identical, thus producing lots of code repetition not only adding to the bulk of the Lexer code (reducing readability), but making the word-token creation process more challenging (because further branches need to be added in an inefficient, code-repeating manner). This is further complicated by the inclusion of TokenTypeContainer custom token types, which are also referenced in handleAlpha(), and are defined elsewhere during the __init__() function of the TokenTypeContainer class. As such, if one wanted to extend the program by adding new keywords (if OCR included an XOR Boolean operator in their pseudocode, for example), then code would have to be added in many locations in a relatively inefficient, repeating manner, thus being a maintenance issue.	in the handleAlpha() function. Due to the similar structures of disparate 'elif' word-identifying statements, the best way forward in extensibility would be to create a single template function which implements each aspect of the code (checking the previous character, checking the next character, returning the syntax error if fields are invalid, and finally returning the token if they are valid), which would take in as argument all of this information (indexing using currentWord - where 'AND' refers to the 'AND' keyword, for example). As such, the handleAlpha function would only consist of three lines, thus being much more readable and understandable, particularly with descriptive comments. To store all of the word-token information, we could use a dictionary of lists, where the key is currentWord, and the corresponding list contains, in order, the previousCharacter string, the nextCharacter string, and the token type. The dictionary may be defined as an attribute of the Lexer class - as the available tokens stay constant throughout Lexing - while maintaining the variable's self-contained nature. We could also move the location of the TokenTypeContainer class closer to the __init__() function of the Lexer class as this would create a locus of token-creation functionality, which is much more readable as they will be in a shared location.
The setLightMode() and setDarkMode() functions, which are activated upon 'Light Mode' and 'Dark Mode' clicks respectively (responsible for changing the GUI theme) look nearly identical, which is a maintenance issue as there is significant code repetition (because each element's foreground + background is changed, and there are many UI elements), thus leading to bulkier code, poorer readability, while making maintenance more difficult. Furthermore, if one wants to create a complete theme restructure by changing both the 'Dark Mode' and 'Light Mode' colour palettes, they would have to do so in two separate locations (as opposed to a single, shared location) twice, which may be time-consuming and particularly confusing for new developers of the IDE. If one wants to add a theme (in addition to the two existing ones), they would also not be able to manipulate a single, space-conservative variable, but have to create a whole new function for that particular theme, linking it back to the new button, thus leading to significant code repetition each iteration. As such, this is a maintenance issue.	As there is code repetition, there is fundamentally a symmetry between setLightMode() and setDarkMode(), and this can be utilised by creating a single template function which generally 'Sets Some Mode', and this can change each foreground + background colour just once in the code, thus reducing code bulk, improving readability and improving maintainability. This also reduces the amount of validation required (as this is also identical) to just once in the code. To obtain the colour information for that particular theme, we can create a separate dictionary which stores each foreground + background colour for a particular element (dictionary as keys can be made highly readable - as strings), where each element for a key in that dictionary is in this case a length-2 list, where index 0 is the colour for 'Light Mode' and index 1 is the colour for 'Dark Mode', where 0 and 1 can be mapped to the theme by using a separate dictionary (e.g. themeIndex['Light Mode'] = 1). This is also highly extensible as if one wants to add more themes, we would just have to increase the length of each list by 1, which is simple as everything would be in a single, shared location. Also, the self-contained nature of these variables can be maintained by enclosing the Tkinter GUI functionality within a class, thus making the dictionary a class

	attribute, enabling unit-testing. As such, the program would become easier to maintain.
There is also the presence of ConsoleStandardFontColour, ConsoleErrorFontColour, and FileHierarchyFontColour, which are all global variables that determine the font colours of various UI elements. Though they are functional, they are challenging to maintain as unit-testing is difficult as they are not held within a container (they are global), they are referenced in multiple locations in the code (which is challenging because if this system is restructured, like if the font colour variables are to be removed due to a changing user requirement of having the default Tkinter font colour instead of a custom font colour for non-keyword text, then multiple variables would need to be deleted, and there may be errors due to accidentally missing a deletion of a variable instance), and there is hard-coding in certain areas (for example, when the font colour is resetted during a print() command, from previously outputting an error, then the conditional checks for the actual hexadecimal red as opposed to ConsoleErrorFontColour), which is especially undesirable if the error font colour wants to be changed (conceivable when adding a 'Red Theme', which is possible, for example)	Firstly, the Tkinter GUI functionality should be held within a class, such that all three variables can be made class attributes as opposed to global variables, enabling unit-testing within the class, which would speed up future debugging speeds (more maintainable). We would also bring the font-changing functionality closer together (by bringing in proximity the functions in which font colours are changed - visitProcedureCallNode() and handleConsoleInput()), thus having a single, relatively shared location for the assignments of these three variables, which is useful as this promotes readability (and reduces the amount of space needed to be traversed to alter UI font functionality). Finally, the hard-coding should be removed, where when there is an explicit colour used in the program, it should be replaced with the appropriate variable - ConsoleErrorFontColour in the case of errors specifically. This will further be aided by the enclosing of the GUI functionality within a class, as class attributes are guaranteed to be defined before the variable is used in assignment, such that Python NameErrors can be avoided (to avoid unwanted halting).
Currently, the console error display system for node-interpreter-produced errors is quite inefficient, as each time a potential error is detected, which is quite regularly considering there is a often a high volume of syntax tree nodes in a single program, the same block of code, in which the error is detected via an if statement, the console is activated, cleared, and reinstated with the error's repr() state, and then disabled once again to avoid the deletion of error messages occurs, thus leading to a large amount of code repetition, which significantly adds to the bulk of the Interpreter class, thus reducing readability to a great extent, leading to difficulty in navigating (which is problematic as if an aspect of the interpreter needs to be changed, such as parsing a new node type post-OCR syntax change, then this would become more difficult), thus leading to a lower level of maintainability. Because this code block takes so much space proportionally, it also sways focus away from the other more intensive logic of the Interpreter, which means that a new developer would struggle more to understand it (lower maintainability).	Because the code blocks used in the console error display system in the Node Interpreter are so similar (with the only difference being the variable - potentially carrying the error - tested), we can create a singular template function which takes as argument this only difference, thus converting the variable-name identifier to a fixed parameter variable, which can then be used to follow the same sequence of error detection, console activation, console clearing, error inserting, and console deactivation. In the node interpreter, each time the code block is found, it can be replaced with the call of the singular function, thus reducing this aspect of the code by a factor of 7, which greatly reduces code bulk and improves readability (and therefore maintainability). The function here would take as argument the variable that may be an error (if it has returned an error from node-visiting). After exiting the function call, there may need to be an extra conditional to ensure that the interpreter is exited (via return) after the error is detected, thus avoiding unwanted halting due to Python errors when mishandling system errors.

Limitations of the Solution

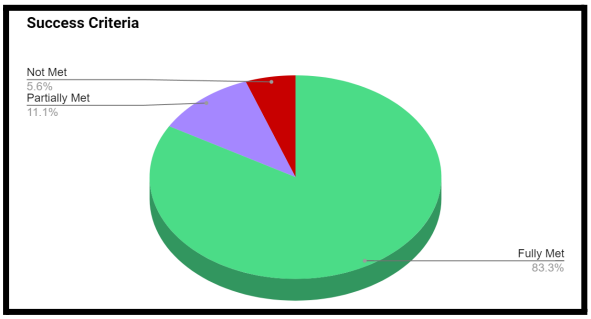
Now, we shall go through the Limitations, which are aspects in which the program is lacking, or features that exist in an incomplete/not optimally desirable manner. As such, we shall identify where specifically the limitation is, noting the ways in which it worsens the user experience and discuss development methods to deal with these limitations such that improvements can be made, improving user satisfaction. Below, we shall go through the limitations one-by-one and discuss development schemes to handle these limitations in the future given sufficient time:

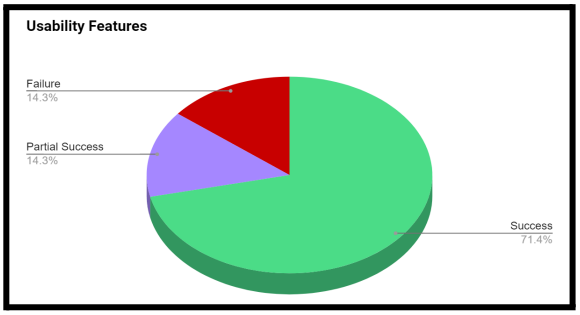
Limitation	Development to Deal With Limitation and Potential Improvements/Changes
The screen size is fixed at 1400 by 700px. This is a limitation as a greater range (by enabling the scaling of the window as can be done with computer windows, for example) would allow for different preferred aspect ratios to be chosen dependent on the user. This might be ideal if a non-standard aspect ratio-computer is used, such as one that is smaller than 1400 by 700px, in which case the entirety of the window would not be visible leading to a potential error or limited usability (this is niche, though it is possible within the target demographic). Here, scaling would enable the entire window to fit on to the screen simultaneously. Also, a greater range would enable switching between full-screen and partial-screen mode, thus gaining the benefits of greater readability and larger font sizes in the former and simultaneous window opening (such as for checking syntax) in the latter within a single session, something that is currently not achievable due to it being fixed in partial screen mode. Due to there being limited flexibility, potential issues with small aspect ratio-computers, and an inability to gain the benefits of greater readability (by scaling up), this fixed screen size is a limitation.	Though the window can be made scalable (perhaps above a lower bound so as to ensure that all contents are fitted in a realistically interactable format), the primary issue is maintaining a consistently aesthetic layout of the UI elements throughout all available aspect ratios upon scaling (i.e. the elements should fit together nicely when scaled up or down). To achieve this, the formatting of each UI element should be not be an absolute pixel value (as here it would not scale with the window), but a percentage of the size parameters of the parent element (for example, the FileHierarchyBar's height could be made to be 80% of the window, and the width of the NewFileButton 90% of the FileHierarchyBar). By basing it off of the parent element (and not always the entire window), there is greater flexibility and we do not have to worry about space clashes between outer and inner elements). Furthermore, if the scaling system in Tkinter requires not just a single percentage declaration (that automatically scales) but a constant re-updating of the size parameters, then we could re-declare the element sizes based proportionally on the current parent element size everytime the window's size is changed (using a bind() function). We could have access to the main parent elements via class attributes, as enclosing them within a class avoids their non-self-contained nature, thus enabling unit-testing and improving maintainability.
The only debugging facility currently provided is the Error, which is output to the console when something erroneous within the code is found during interpretation, and is composed of the type, line number, and message. Though this is useful, debugging facilities could be better (the interacting stakeholders preferred it to avoid too much help, but there may be people in the user base who would benefit) by creating a step-by-step debugger, which would enable each line of code to be executed and paused (with variable information stored temporarily), such that users could view the current state of variables and ensure that the error did not occur on the previously executed	To create a step-by-step debugger to extend debugging facilities to improve the user coding experience, we shall need to change the 'Interpreter' functionality by executing a line, and then pausing execution. To sustain this, we could further divide programs into overall lines (or more accurately, steps) during the Parser stage by creating a list of what constitutes a step and referring to it constantly during parsing and node creation. As such, node visiting in the Interpreter will involve visiting a StepNode and then immediately halting. Progress needs to be stored, as reinstating the interpreter after 'Run' is re-clicked (which is the mechanism by which steps can be iterated

line, until the error is finally detected on a specific line, thus making debugging very straightforward and powerful, enabling one to skip exactly to the error, and find out the program state when it appeared so that functionality can be remedied. This feature would enable faster debugging, greater time spent coding, more productive coding sessions, thus improving the user experience and satisfying stakeholders. Due to the lack of this feature and the current limit of relatively small error messages, our solution is currently limited.	through, or a new set of buttons including 'Step Forwards', 'Run Program', 'Run Until Error' can be defined with functionality identical to the names for readability creation) will need variables to reference to ensure that the program does not miss previously assigned variables (and produce misallocated NameErrors). As such, we can create a Heap class, which will contain a dictionary storing as keys identifier names and values identifier values, which may also be constantly reference, or we can make scopeList a whole-class variable of Interpreter (not only limited to a single instance of the class), thus giving it a greater scope that allows it to remain constant between multiple Interpreter calls.
Currently, there is no ability for the user to add comments to the program (personal statements that aid in development and maintenance, but do not actually change the functionality of the code, as is seen with '#' in Python). This is alright as it is not a major portion of OCR syntax (more so during A-Levels, which is not the target audience of the program, though this extensibility makes sense), but comments would make sense as an extra usability feature as it teaches students (major portion of target demographic) that documenting your code and explaining functionality for future maintenance is important, it would enable teachers to reinforce this lesson if the IDE is used for teaching purposes, and in larger pseudocode problems, which may be extensive pseudocode testing in the NEA, for example, it helps the user to stay on track and in control of all of the work previously done. This is important as there is a large variation in the potential user base (students with varying skill and experience levels), so this level of aid may help one keep organised without offering too much help (does not code the program's logic, for example).	To implement comments, we could use the OCR syntax for maximum precision and consistency, which is '//' beginning the comment, ending at '\n', which can be used as the beginning and termination point while Lexing. Because comments do not affect program functionality, they will need to be ignored during Lexing, and this will be achieved by continuing the standard character-by-character reading until a '//' is reached, iterating without accumulating or checking for characters (thus ignoring the comment) until a '\n' character is found, at which point normal Lexing can be continued. There is potentially an issue with line numbers being skipped over (which may lead to console errors being displayed with inaccurately low line numbers during the Interpreter stage - as line numbers are skipped). This can be avoided by ensuring that comment-skipping halts just before the newline of that line is complete (this is equivalent to the current state, which works as per Testing for Robustness), or by removing comments at once before the getTokenList() function of the Lexer class is called.

Conclusion

After the Evaluation section, cross-referencing with the pre-established Success Criteria + intended Usability Features to determine their level of success post-implementation, while looking at general maintenance issues and limitations to gauge where specifically the project was lacking, we can build a general picture of how successful the project was overall. As evidenced by the below graphs indicating how well met the success criteria and usability features were (original data in their respective sections), the results are predominantly positive:





Proportionally, Success Criteria were **Fully Met** >80% of the time and Usability Features were **Successes** >70% of the time, which is particularly positive when considering that the absolute count of **Partially Met/Partial Successes** and **Not Met/Failures** number less than 4 in each case separately. Beyond statistical analysis, functional/robust post-development testing, in which all test cases were passed successfully, shows that the IDE is a reliable piece of software, justified as the testing was complete keeping all program components in mind and referring to all of the interacting stakeholders (to maximise the accessible variety of opinions, to ensure the IDE is suitable across a varied user base). There were also highly positive qualitative comments, highlighting in particular the smoothness of the experience (in transitioning from one program component to another), the simplicity of the user interface (indicating it reflected the simplicity level of OCR pseudocode accurately and that it was intuitive to understand), and the wise selection of features (noting that all features were useful - the necessities of program execution with the text editor and the console, but also the useful but sufficiently restrained debugging facilities of printed error messages, for example). As such, looking at what was implemented (relative to Success Criteria and Usability Features), there is sufficient qualitative and robust quantitative data (from post development testing) to state that implementations were successful.

Naturally, there were also limitations identified in the final solution, existing primarily due to time constraints over the development cycle or an organisation of iterative development that did not align with that particular feature. However, these limitations are the lack of desirable, yet unnecessary features. For example, comments would be useful and sufficiently restrained, but within the context of a general pseudocode question (quite short in nature), the logic is straightforward and comments to decompose this simple logic would be extraneous (though potentially useful in certain cases). As such, the current solution (implementation of the IDE) is fairly complete in terms of necessary features (also evidenced by Subjective Testing of Usability Features, annotations in Stakeholder Functional Testing, and feedback when evidencing Usability Features).

In conclusion, the current implementation of the IDE is complete in terms of necessary features, fulfilling the Success Criteria well overall (particularly fulfilling the critically functional Success Criteria such as a complete Interpreter) and providing ample Usability Features for users to interact with it evidently smoothly. Though more work could be done to expand the current suite of features, some extensions are slightly extraneous in nature, thus highlighting the ideal simplicity of the current state of the IDE. As such, this project - attempting to create a GCSE OCR Pseudocode IDE - has sufficient evidence overall to be considered a success.

Bibliography

Item Referenced	How It Was Used
Official Tkinter Documentation (https://docs.python.org/3/library/tk.html)	In learning how Tkinter elements were created and formatted
Official OS Documentation (https://docs.python.org/3/library/os.html)	In learning how on-device file deletion can occur from within a Python program
freeCodeCamp.org Tkinter Course (https://www.youtube.com/watch?v=YXPyB4XeYLA)	In learning the basics of Tkinter (opening windows, the different types of UI elements offered, and positions elements with pack() etc.) as I was previously inexperienced with the module
TopTal.com Basic Interpreter Tutorial (https://www.toptal.com/scala/writing-an-interpreter)	Though no code was copied (and their implementation of the interpretation process was far different as can be seen), the lexer-parser-interpreter divide was useful to observe, which is why it acted as the base for the OCR Pseudocode Interpreter
Official OCR A-Level Computer Science Book	Further reading from the syllabus was useful for further understanding this lexer-parser-interpreter divide when studying the stages of compilation, which inspired ideas such as the symbol table for keeping track of variables (though mine was relatively high-level due to the Python base)