TASK 2: Documentation

- **Introduction**

This document provides a high-level overview of the IPPeCode interpreter, which is a Python script that interprets and executes IPPeCode in XML format. The interpreter handles various instructions and provides a flexible, extensible structure for adding new instructions as needed. The document covers the libraries used, the issues faced during development, and the solutions employed to resolve those issues.

- **Libraries Used**

The following libraries are used in the interpreter:

1. 'lxml.etree': This library is used for parsing XML files, as it offers a higher performance compared to the built-in xml.etree.ElementTree.
2. 'argparse': This library is utilized to manage command-line arguments, making it easy to specify input and output files.
3. 'sys': This library is used for handling system-level functions, such as reading from standard input or exiting the program with a specific error code.

- **Functions**

1. parse_arguments(): A helper function that processes the command-line arguments, utilizing the argparse library to ensure proper usage of the interpreter
2. parse_xml(file_path): A function that takes an XML file path as input and returns a parsed XML tree using the lxml.etree library.
3. execute_instruction(instruction, input_file, output_file): A function that takes all the instructions processed by 'parse_xml', input file and output file and executes all of them coherently. This function uses internal functions where the behavior of each instruction is defined.

- **Issues and Solutions**

Issue 1: Handling XML Parsing Errors

During the development of the interpreter, I encountered issues related to XML parsing errors. To handle these errors gracefully and provide appropriate error codes, I used exception handling techniques.

I employed try-except blocks to catch specific exceptions like ET.ParseError and FileNotFoundError. This allowed me to exit the program with the appropriate error code, providing meaningful feedback to the user.

Issue 2: Debugging and Tracing Execution

Initially, it was challenging to debug the interpreter and trace its execution due to the complexity of the code and the various instructions it had to handle.

I used the PRINT instruction to output intermediate results during the execution of the program. This allowed me to identify issues and gain insights into the interpreter's internal state. In addition, I added comments to the code for improved readability and maintainability.

Issue 3: Handling Labels and Jumps

The interpreter had to deal with labels and jumps, which required maintaining the program counter (PC) and handling cases where labels were missing or duplicated.

I created a separate labels dictionary in the interpreter's internal state to store label positions. This enabled me to process labels before executing instructions and exit the program with appropriate error codes if duplicate or missing labels were encountered. I also adjusted the PC accordingly when performing jumps and calls.

Issue 4: Making the program to work

Definitely, maintaining the program counter and coming up with an idea to make the program work, was the most challenging part by far. After a lot of research I found out that the best way to achieve it was with the use of lists and dictionaries. When the program was iterating through the instructions, each of them were saved in a dictionary (*inst)* along with its necessary parameters and their values. Furthermore, these dictionaries were stored in a list (*instructions)* and that list was eventually used by the execution of the program itself. These two features added with the dictionary (*state*) gave the project the structure it needed.

- **Tests**

Inside the folder of the project, besides the script “taci.py” there are some additional documents. These documents are tests to verify that the program works well. Four of them are taken from the supporting files for the project (ex1.xml, ex2.xml, ex3.xml and ex3err.xml with its input). The rest of them were created for this task. Test 7-9 should print an error message through stderr: 20, 21 and 30 respectively.

- **Conclusion**

The IPPeCode interpreter is a robust and extensible Python script capable of interpreting and executing IPPeCode in XML format. Through the use of appropriate libraries and well-structured code, the interpreter handles various instructions and provides a flexible foundation for future enhancements.