Devon Coates (48554966)

Foundations of Modern Computing Final Report

Overview of Final Program Steps:

1. The program begins and prompts the user for different options, for the sake of this report we will only analyze the startCodeAssistant command due to the other ones being unnecessary for the program.
2. If it is run, the program then prompts the user to use FlowScript or a basic loop. If the user chooses FlowScript, it opens a file that uses FlowScript specifically designed for debugging the C++ program.
3. The interpreter then interprets this FlowScript to create jobs in any order you were given. Unfortunately, I do not have loop compatibility though so it must be manually coded in.
4. If the user chooses a loop, the program creates a compileJob that runs and creates errors from reading the code file "file1" in the "CodeToTest" folder of the "Code" folder.
5. This then saves the errors in a errors.json file in the Data folder. As long as there are errors, the code continues.
6. After this, the program creates a promptJob that reads in the errors along with the original source code to create a specific prompt. Because this is going to OpenAI and under-the-hood processing occurs, the prompt does not need to be too complicated. This job simply sends the source code along with the errors and asks for the code to be corrected without any extra comments.
7. Then, a customJob is run that sends this prompt to OpenAI's gpt-3.5-turbo model. This customJob then gets the response and outputs it into the file that is being corrected.
8. Then, another compileJob is run to check for any errors, if there are, it loops and returns to step 5. This continues until there are no more errors in the code.

Prompts:

- This program does not necessitate in-depth prompts or multiple attempts a majority of the time. This is because I am interfacing with OpenAI and using gpt-3.5-turbo. I reuse the same prompt each time.
- I played around with different prompts on ChatGPT to get the right prompt but did not actually try it through my program until I got one that I liked.
- My first prompt included the text:
    - "You are going to roleplay as a professional C++ developer. I will give you a file with errors in it as well as a JSON file with a list of those errors. I may also give you a chat history of previous prompts and your response. Your job is to correct the errors and add any comments to the code of what you changed. You will then provide the code alone with no additional text."
    - "Source Code: …"
    - "Errors: …"
    - "Chat History: …"

- The reason I told it to roleplay was to get more accurate probabilities specific to the field of C++ developers. I then clearly outlined what I was going to give it and then the jobs I wanted it to do in direct order so that it could be more clear.
- My second prompt simply removed the history because it was confusing the LLM. This solved most problems and ended up consistently correcting the code so I did not continue any further.
- To create the prompt, I used a Python file that would simply read the prompt by reading in an entire file that holds the prompt (where the promptJob wrote the prompt to) and send it to OpenAI's server using their Python library.
- To output any responses, I did the same thing. I used OpenAI's Python library to output the response I got from OpenAI to a specific file using a write function from Python. I did not parse it as it did not need to be parsed a majority of the time because of the prompt. The prompt would tell it not to return any text other than the code.

All Jobs Used:

- Compile Job
    - The compile job uses a system command and a pipe to compile the file1.cpp (or any file that is being corrected) file and read any errors. This then compiles all of the errors into a JSON file that is outputted into errors.json. This does not take any inputs or give any outputs. All jobs communicate with one another via reading/writing to files, not passing data
- Prompt Job
    - The prompt job creates a prompt string that is immediately filled with the beginning part of the prompt. It then reads the raw source code from the file the program is correcting. Then it reads in any errors from that code that would have been created by a compile job. This is technically not taking any inputs or giving any outputs by value but instead uses reading/writing to files to take any errors from a compile job and send the prompt to a python file.
- Custom Job
    - The custom job simply runs a python file that reads in a prompt created by the prompt job. It takes this and then sends it to OpenAI. The response given by OpenAI is then outputted to file1.cpp (or any file that is being corrected) to be later compiled by a compile job.
- All of the above jobs communicate with one another via reading/writing to files.
    - The compile job takes in file1.cpp to output errors.json.
    - The prompt job takes in errors.json and file1.cpp to output prompt.txt
    - The custom job does not take in anything but executes sendPrompt.py which takes in prompt.txt to output file1.cpp.

How the Jobs are Created and Executed:

- The jobs are either created and executed via two methods, a simple loop or FlowScript
- Loop
    - The loop uses a basic main function to create a job, name it, and then queue that job. Once the job is queued, the program waits until it is no longer running and then completes it. Once it is completed, it waits 1 second to begin the next job to allow any writing to occur before reading begins.
    - This happens every time during the loop whenever a job is created and executed
- FlowScript
    - FlowScript uses the interpreter to make jobs via the dot language. If it sees the shape of a circle, it recognizes it as a job and then reads the information from the label. The name is the very first letter. It passes this all in JSON format to the job system API to create and name the job. Whenever a -> is detected, it first queues and executes the left-hand side, then waits for it to be done running, completes it, waits for it to be completed, waits 2 seconds, then continues with the right-hand side. This happens every time a -> is seen and allows any reading/writing to occur between jobs.

From first prompt to final code:

```
You are going to roleplay as a professional C++ developer. I will give you a file with errors in it as well as a JSON file with a list of those errors. Your job is to correct the errors and add any comments to the code of what you changed. You will then provide the code alone with no additional text.

Raw code:
//
// Created by Devon Coates on 10/1/23.
//
#include <iostream>

int main(){
   for(int i = 0; i < 10; i++){
      std::cout << i;
   }
   std::cout << endl
   int x = 3.14;
   return 0;

   there are more errors here?;
   what will you do here? will you remove the text?;
}


Errors Json:
{
        "File1_Errors_and_Warnings": [
                {
                        "file_path": "./Code/JobSystemCode/CodeToTest/file1.cpp",
                        "line_number": 10,
```

```
                "column_number": 22,
                "error_description": " expected ';' after expression",
                "code": "        std::cout << i;\n    }\n    std::cout << endl\n    int x = 3.14;\n    return 0;\n"
            },
            {
                "file_path": "./Code/JobSystemCode/CodeToTest/file1.cpp",
                "line_number": 10,
                "column_number": 18,
                "error_description": " use of undeclared identifier 'endl'; did you mean 'std::endl'?",
                "code": "        std::cout << i;\n    }\n    std::cout << endl\n    int x = 3.14;\n    return 0;\n"
            },
            {
                "file_path": "./Code/JobSystemCode/CodeToTest/file1.cpp",
                "line_number": 11,
                "column_number": 13,
                "error_description": " implicit conversion from 'double' to 'int' changes value from 3.14 to 3 [-Wliteral-conversion]",
                "code": "    }\n    std::cout << endl\n    int x = 3.14;\n    return 0;\n\n"
            },
            {
                "file_path": "./Code/JobSystemCode/CodeToTest/file1.cpp",
                "line_number": 14,
                "column_number": 5,
                "error_description": " unknown type name 'there'",
                "code": "    return 0;\n\n    there are more errors here?;\n    what will you do here? will you remove the text?;\n}\n"
            },
            {
                "file_path": "./Code/JobSystemCode/CodeToTest/file1.cpp",
                "line_number": 14,
                "column_number": 14,
                "error_description": " expected ';' at end of declaration",
                "code": "    return 0;\n\n    there are more errors here?;\n    what will you do here? will you remove the text?;\n}\n"
            },
            {
                "file_path": "./Code/JobSystemCode/CodeToTest/file1.cpp",
                "line_number": 15,
                "column_number": 5,
                "error_description": " unknown type name 'what'",
                "code": "'\n    there are more errors here?;\n    what will you do here? will you remove the text?;\n}\n"
            },
            {
                "file_path": "./Code/JobSystemCode/CodeToTest/file1.cpp",
                "line_number": 15,
                "column_number": 14,
                "error_description": " expected ';' at end of declaration",
                "code": "'\n    there are more errors here?;\n    what will you do here? will you remove the text?;\n}\n"
            }
        ]
}
```

I know what you're thinking, it was supposed to start with FlowScript. Unfortunately, my interpreter was not compatible with loops and thus is not capable of running this program. Instead, I skipped right to correcting the code with the OpenAI. I would create the capability but I have about 4 other finals to do so am unable to spend another 6 hours fixing that. However, this would eventually output the following final code.

```cpp
#include <iostream>

int main() {
   for (int i = 0; i < 10; i++) {
       std::cout << i;
   }
   std::cout << std::endl;
   int x = 3.14;
   return 0;
}
```

FINAL CAPABILITIES

This agent is capable of reading in C++ code, use FlowScript or a simple loop to create jobs that will read and compile the C++ code, parse and output the errors, create a prompt that will communicate with OpenAI, update the C++ code, then repeat until the C++ code compiles properly with no errors.