LLM Code Analyzer

0.1 Assumptions

To come up with my prompt, I've made some assumptions about the model that I believe will get the best results. Firstly, my goal is to get the model to write out JSON, while will be parsed back. This is less complex than prompting the model twice and storing the output, but I believe the results will be similar. Firstly, I assume that the model is aware of JSON to a degree that it can write it out. I don't believe it is necessary to teach it the formatting of JSON (as it takes up sensitive context space).

0.2 Prompting Styles

Language models are generally trained to be a completion engine, rather than a direct answering engine. This influences my prompting style, such that I can approach it in two different ways. I could command the language model to perform a task, or I could provide a description and coax the model into completing it with my desired output.

I originally tried prompting as an instruction, but quickly fell into issues, whether the model would simply refuse to output anything, or repeat my prompt over and over. Due to that, I went with coercing the model into completing my output and not relying on any RL the model underwent to make it more conversational.

0.3 Output Generation Tweaks

When initial experimentation occurred, I noticed that the model would stop generating text abruptly, leaving me with malformed JSON output. The issue stemmed from the fact that the generation length, even though it is unlimited by default, was causing the model to stop generating after 30 or so tokens. To fix this, I made the generation length 99999, which is significantly more than would realistically be required, but ensures that the model natural finishes its language generation.

Additionally, reducing the temperature (from the default of 1), significantly improved the generation quality and the reduced the number of the situations where the model would attempt to ”answer” me, instead of writing the plain JSON output.

0.4 Job I/O

The Job, which uses the LLM, named Correct, takes in the output from context_add directly and uses the error description and chunk (containing context) for the error. This is fed into the LLM, while being wrapped around by a template. The job, then returns a JSON object, with a schema that contains an array of compiler errors and a linker error (null, if one was not thrown).

0.5 Attempts

1.
   ```
   format!(
           r#"Given the code: "{}" and the message: "{}". Write out a JSON response
            ↪  with a field called "message" that explains the error and another field
            ↪  called "fix" which has fixed code"#,
       chunk, compiler_msg
   )
   ```

   This the typical style I would use for a robust conversational LLM, such as ChatGPT. Unfortunately, I ran into a few issues with this one, where the model would include filler at the beginning, like: ”Okay” or ”Here you go: ”.

2.
   ```
   format!(
           r#"Given the code: "{}" and the message: "{}". ONLY Write out a JSON
            ↪  response with a field called "message" that explains the error and
            ↪  another field called "fix" which has fixed code. Do NOT write anything
            ↪  besides the output JSON"#, chunk, compiler_msg
   ```

```
)
```

This worked as well, but still would sometimes result in the model adding filler unneeded text at the end of its responses.

3.
```
format!(
        r#"The code chunk: "{}" causes the error: "{}". A fully JSON style response
         ↪  with the requirements: NO additional plain text or trailing characters,
         ↪  JSON schema only contains a "message" field, explaining the error (in
         ↪  the context of the code), and "fix" field, with an updated code chunk:
         ↪  "#, chunk, compiler_msg
)
```

This prompt applies the strategies I used in the previous prompt, but it uses a completion-style phrasing, so that the model does not need to be fine-tuned on conversation to still have success. While this does perform significantly better on other models, Orca had near perfect performance, producing only a JSON output as well as providing good fixes.

4.
```
format!(
        r#"The code chunk: "{}" causes the error: "{}". A fully JSON style response
         ↪  with the requirements: NO additional plain text or trailing characters,
         ↪  JSON schema only contains a "message" field, explaining the error (in
         ↪  the context of the code), and "fix" field, with an updated code chunk
         ↪  which ONLY fixes the specified error: "#, chunk, compiler_msg
)
```

This prompt is a minor update to the previous prompt, where the fixes would ocassionally be different (or contain extra changes) from the interpreted error. This adds an extra requirement, which instructs the model to only focus on the error at hand and to not attempt to fix other issues with the code.

5.
```
let post_prompt = r#"A fully JSON response with the schema: {"fix": string,
 ↪  "message": string} and no additional plaintext characters. The message field
 ↪  explains the error (in the context of the code). The "fix" field contains the
 ↪  full code chunk with updated changes, which ONLY fix the specified error. The
 ↪  JSON object: "#;

format!(
        r#"The code chunk: "{}" causes the error: "{}". {}"#, chunk, compiler_msg,
         ↪  post_prompt
)
```

Ocassionally the tested models would only print the random line instead of the entire code chunk to replace. This fix instructs the model to keep the entire code chunk. Additionally, this fix also better depicts the schema for the model to use, since there were sometimes extranneous fields in the JSON.

6.
```
let post_prompt = r#"A fully JSON response with the schema: {"message": string,
 ↪  "fix": string} and no additional plaintext characters. The message field
 ↪  explains what the linker error means. The "fix" field contains an action that a
 ↪  programmer could do to directly resolve the error. The JSON object: "#;

format!(
        r#"The following linker error: "{}" with the related symbols: "{}". {}"#,
         ↪  linker_msg, sym_messages, post_prompt
);
```

This update keeps the same prompt for the compiler, but additionally adds a prompt for fixing linker errors. It is heavily based on the compiler prompt, and only adds 2 fields for explaining the actual issue and a fix that a programmer could make to resolve it.

0.6 Models used

The models used for evaluation were Mistral OpenOrca and Mistral Instruct from GPT4All. Both exhibited good performance at providing fixes, while also providing near perfect performance in providing correct JSON output.

0.7 Model Performance Examples

1. Broken Code:

   ```
   int main() {
     struct lol {};
     struct lol = struct Meme{};
   }
   ```

   OpenOrca output:

   ```
   {
     "fixes": [
       {
         "fix": "int mein() {\n struct lol {};\n struct Meme{};}\n",
         "message": "The given code has a syntax error due to missing semicolon after
         ↪  'struct Meme{}'."
       }
     ]
   }
   ```

   While the error message is not correct and there isn't a 'correct' fix for the code, the model does produce an acceptable answer. The fixed code from the model is:

   ```
   int main() {
     struct lol {};
     struct Meme{};
   }
   ```

2. Broken Code:

   ```
   int main() {
     int n = 43
   }
   ```

   Mistral Instruct output:

   ```
   {
     "fixes": [
       {
         "fix": "int main() {\n int n = 43;\n}",
         "message": "The semicolon is missing after the variable declaration."
       }
     ]
   ```

This is a perfect reponse from the model
The fixed code from the model is:

```
int main() {
  int n = 43;
}
```

3. Broken Code:

```
int main() {
  struct Monkey {
    int fury;
  };

  struct Monkey m = {
    .furee = 3
  };
}
```

Mistral Instruct output:

```
{
  "fixes": [
    {
      "fix": "int main() {\n struct Monkey {\n int fury;\n };\n}",
      "message": "The variable name 'furee' should be corrected to 'fury'. This is
      ↪  because in C programming, the identifier 'furee' does not exist and
      ↪  therefore causes an error. To fix this issue, change the variable name
      ↪  from 'furee' to 'fury', which will make it a valid field in the struct
      ↪  Monkey."
    }
  ]
```

The issue with the code is most likely a misspelling, however the model takes the safe call and removed the invalid struct allocation.
The fixed code from the model is:

```
int main() {
  struct Monkey {
    int fury;
  };
}
```

4. Broken Code: main.cpp

```
#include <iostream>
// string dsa() {
//   let d = {
//     das auto;
//   }
// }

int min() {
  std::cout << "This is a demo program\n";
  return 0;
}
```

something.cpp

```
int test() {
}
```

OpenOrca output:

```
{
  "compiler_fixes": [
    {
      "fix": "int test() {\n    return 0;\n}",
      "message": "non-void function does not return a value [-Wreturn-type]"
    }
  ],
  "linker_fixes": {
    "fix": "Add a main() function in your source code and ensure that you are
    ↪  correctly linking all necessary libraries.",
    "message": "The linker encountered an issue while trying to create the final
    ↪  executable, as it couldn't find the 'main' function which is required for
    ↪  starting execution."
  }
}
```

5. Broken Code: main.cpp

```
#include <iostream>
// string dsa() {
//   let d = {
//     das auto;
//   }
// }

int min() {
  std::cout << "This is a demo program\n";
  return 0;
}
```

something.cpp

```
int test() {
}
```

Mistral Instruct output:

```
{
  "compiler_fixes": [
    {
      "fix": "int test() {\n    // Add a return statement to return an integer
      ↪  value\n    return 42;\n}",
      "message": "The function 'test()' does not return a value."
    }
  ],
  "linker_fixes": {
    "fix": "Ensure that you have defined and implemented the main function in your
    ↪  source code.",
    "message": "The linker command failed with exit code 1 (use -v to see invocation)"
  }
}
```