50.051 Programming Language Concepts

Contents Page

I. Personal Finance Manager

II. Working with Constraints

1. Input files
    a. Account.json
    b. Expenses.json
2. Output files
    a. Main Page
    b. User Bar Graph
    c. Account Overall Expenses
    d. Account Overall Yearly Expenses
3. Parser
    a. Validation
    b. Parsing
4. Compilation
    a. Compiling Input FSM
    b. Compiling Output FSM
5. Finite State Machine (FSM)
    a. Input FSM
    b. Output FSM

III. Challenges

1. Parsing of json file
2. Plotting of Graph
3. Integrating HTML code into C Programming
4. Opening of HTML Page
5. Graph and Categorization Table integration in HTML Page
6. Combining HTML Pages for Output FSM

IV. Breakdown of Contribution

V. Learning Points

I. Personal Finance Manager

In today's diverse landscape of income sources and expenditure types, individuals face challenges in effectively managing their finances without the aid of a platform. The absence of a tool for handling income, expenses and financial goal tracking can lead to difficulties in financial management. To address this issue, our focus will be on the development and optimization of a personal finance management system. This system aims to assist individuals in visualizing their cash flow and expense categories spent in different bank accounts, enabling better control and utilization of money.

System Flow:

The system operates through two principal finite state machines (FSMs): the input FSM and the output FSM. Input to the system consists of two JSON files: "accounts.json" and "expenses.json". The "accounts.json" file includes fields such as "account_id", "user_id", "name", "default_currency", and "balance". On the other hand, "expenses.json" contains fields like "account_id", "date", "description", "currency", and "amount_spent". Within the system, there exist functions within the "parserAccounts" and "parserExpenses" modules to parse these datasets. The input FSM *(Figure 1)* ensures the correct parsing of these files, and any incorrect input files lacking necessary fields in the JSON format will result in an error.

Subsequently, utilizing the parsed data, the system generates an expenses table and expense graphs. The expenses table categorizes expenses into food, transport, shopping, and others, both for overall expenses across accounts and yearly expenses based on the inputted account ID. In parallel, the expense graphs depict expenditure patterns, including overall expenses for all accounts within a user ID, expenses for a single account ID, and yearly expenses for a specific account ID.

The system comprises three primary pages: the first page presents the overall expenses for accounts within a specific user ID, the second page displays the overall expenses for a designated account ID, and the third page exhibits the overall yearly expenses for a specified account ID. The output FSM *(Figure 2)* navigates through these pages sequentially based on user input, initially prompting for the user ID, then the account ID, and finally the desired year.

II. Working with Constraints

1. Input files

In our data management system, we have two distinct input files: accounts.json and expenses.json. This division facilitates efficient processing and management of both user information and expenditure records, optimizing data handling procedures.

a. Accounts.json:

This file serves as a repository for user account details, encapsulating vital information crucial for financial management. This file encompasses the following attributes:

account_id*:* A unique numerical integer identifier assigned to each account, facilitating precise identification and association.

user_id: An integer value designating the specific user to whom the account belongs, ensuring seamless linkage between user profiles and their respective accounts.

name: A string attribute providing users with the flexibility to assign distinctive and recognizable names to their accounts, enhancing organizational clarity. For instance, users may label their accounts as "Savings Account" or "Checking Account" based on their intended usage.

default_currency: This attribute denotes the default currency associated with the account, employing a standardized 3-letter string format. Presently configured to Singapore Dollar (SGD), it enables uniformity in currency representation across the system.

balance: Reflecting the financial standing of each account, this numerical value, which may be expressed as an integer or a floating-point number, signifies the available funds within the account.

By meticulously recording such granular details, accounts.json furnishes a comprehensive overview of users' financial holdings, ensuring transparency and facilitating informed decision-making.

b. Expenses.json:

Complementing the account-centric data in accounts.json, expenses.json captures detailed records of user expenditures, providing invaluable insights into spending patterns and financial outflows. This file encompasses the following attributes:

account_id: Serving as a pivotal linkage, this numerical integer identifier establishes a direct association between individual expenses and their corresponding accounts, enabling precise tracking and attribution.

date: A string attribute denoting the date on which each expenditure occurred, facilitating temporal analysis and chronological organization of transactions.

description: Offering contextual clarity, this string field encapsulates succinct descriptions of each expenditure, elucidating the nature and purpose of the transaction. From identifying the merchant to specifying the rationale behind the purchase, this attribute enriches the dataset with actionable insights.

currency: Employing a standardized 3-letter string format (e.g., SGD, USD, EUR), this attribute signifies the currency denomination in which each expenditure was transacted, accommodating diverse financial contexts and international transactions.

amount_spent: This numerical field, capable of representing both integer and floating-point values, quantifies the financial magnitude of each expenditure, providing a precise record of monetary outflows.

Through the meticulous aggregation of these essential attributes, expenses.json furnishes a comprehensive ledger of user spending activities, empowering stakeholders with the actionable intelligence necessary for prudent financial management and strategic decision-making.

In summary, the segregation of data into distinct input files facilitates a structured and systematic approach to data management, ensuring efficiency, clarity, and comprehensiveness in our financial record-keeping endeavors.

2. Output files

In our data management system, we have four distinct output files: Main Page, User Bar Graph page, Account Overall Expenses page, and Account Overall Yearly Expenses page. The last three pages are our main output files which display graphs and tables containing information based on the input entered by the user (e.g., user ID, account ID, and the year).

a. Main Page

The main page is a mock up of the website's login page, which contains information about our program as well as its features.

b. User Bar Graph

The second page displays the user bar graph which shows the total expense in the default currency (SGD) for all accounts held by the user.

The user bar graph page functions by first processing expenses from multiple accounts, converting expenses in USD and EUR to SGD, and then calculating the total expense in SGD for each account associated with a specified user ID. It utilizes parsed data from Expenses.json and Accounts.json files to retrieve relevant expense and account information. The program employs Plotly library from JavaScript to generate interactive bar graphs within an HTML page, visually representing the total expenses in SGD for each account. The code employs a structured data type called ExpenseTotalsSGD, with expenseTotalsSGD serving as an instance of this struct. Accessing expenseTotalsSGD[account_id].totalExpenseInSGD provides the total expenses in SGD for a specific account_id. This struct encapsulates and organizes expense data in a manner that facilitates efficient retrieval and analysis of spending information.

This visualization aids in analyzing spending patterns and managing accounts effectively. The HTML page includes CSS styling for formatting and JavaScript code to render the bar graphs dynamically. The code to write the HTML onto the .html page is written in C inside the main C file.

The user bar graph page offers a concise and visually intuitive overview of expense distribution across different accounts, facilitating informed financial decision-making.

c. Account Overall Expenses

Categorization Table for Accounts:

Expenses from accounts are systematically categorized into four distinct categories: food, transport, shopping, and others, based on the associated currency. To facilitate this categorization process, we utilize a dictionary named "shops," which maps each shop name to its corresponding category. For instance, establishments like Pizza Hut are categorized under "Food," while stores specializing in home decor are classified under "Shopping."

Within our system, we employ a structured data type called AccountExpenses, which encapsulates cumulative expenditure totals across different categories and currencies. This struct comprises individual fields for each category and currency combination, such as totalFoodSGD, totalTransportUSD, and totalOthersEUR. Each entry in the expenses.json file is processed to extract relevant information, including the account ID, description, currency, and amount spent. Subsequently, the expenditure amounts are updated within the AccountExpenses struct based on the associated category and currency, ensuring accurate tracking and aggregation of expenses for each account. Finally, the categorized expense data is presented in a tabular format on the HTML page, offering stakeholders clear insights into spending patterns and allocation across different expense categories.

Yearly Expenses Graph:

The yearly expenses graph illustrates the expenditure trends over five years (2020 to 2024) for each account within our system. Each account's total expenses are represented by a scatter plot, with the x-axis denoting the year and the y-axis indicating the expense amount. The graph utilizes different color lines to distinguish between expenses in different currencies: USD, SGD, and EUR. The code utilizes three arrays, namely totalExpensesSGD, totalExpensesUSD, and totalExpensesEUR, each sized to accommodate the maximum number of years. As we iterate through the expenses, we extract the year of each expense and record the corresponding expenditure amount. By accessing totalExpensesSGD[expenseYear – Min_year], for instance, where Min_year represents the earliest year, we can conveniently retrieve the total expense in SGD for a specific year, such as 2021, associated with a particular account ID. This approach ensures systematic organization and easy access to expense data, facilitating comprehensive analysis of spending trends over the years.

This visualization offers a comprehensive overview of each account's spending behavior over time, facilitating analysis of financial patterns and trends. By plotting the total expenses for each currency against the corresponding year, stakeholders gain valuable insights into how expenditure fluctuates across different accounts and currencies throughout the specified period.

d. Account Overall Yearly Expenses

Yearly Categorization Table for Accounts:

Expanding upon the categorization framework, we introduce a yearly perspective to analyze expenditure trends over time. In addition to the existing categorization criteria, this enhanced analysis incorporates an additional input field: the year extracted from the date field of the expenses.json entries. Each expense entry is first evaluated to determine if it corresponds to the specified year of interest. Subsequently, the shop name and category are examined to update the AccountExpensesYear struct accordingly.

Similar to its counterpart, the AccountExpensesYear struct features distinct fields for each category and currency combination, augmented with a "Year" suffix, such as totalFoodSGDYear, totalTransportUSDYear, and totalOthersEURYear. By leveraging this extended categorization mechanism, stakeholders gain deeper insights into annual spending patterns, enabling informed decision-making and strategic financial planning. The resulting yearly categorization table provides a comprehensive overview of expenditure distributions across different categories and currencies, facilitating easier analysis and interpretation of financial data trends over time.

Monthly Expense Graph:

The monthly expenses graph provides a detailed depiction of expenditure trends over a specific year, catering to the user's input. Each account's total expenses are visualized through a scatter plot, where the x-axis denotes the months of the year, and the y-axis represents the corresponding expense amounts. Similar to the yearly graph, different colored lines are utilized to differentiate expenses in various currencies: USD, SGD, and EUR.

To achieve this visualization, the code utilizes three arrays—totalExpensesSGD, totalExpensesUSD, and totalExpensesEUR—each sized to accommodate all twelve months. It iterates through the expense data, extracting the year and if the year matches the year inputted by the user it records the expense amount accordingly for all months of that year. Accessing totalExpenseSGD[expenseMonth], for example, provides the expense in SGD for a particular month of the specified year.

This approach ensures a systematic organization of expense data, facilitating a comprehensive analysis of spending patterns and trends across different months and currencies for the selected year.

Similar to the yearly graph, the monthly graph offers stakeholders valuable insights into each account's spending behavior over time, aiding in financial analysis and decision-making processes.

3. Parser

Before proceeding with parsing the data and storing it, a crucial step in our data processing pipeline involves data validation. This validation mechanism meticulously scrutinizes the input type of each field to ensure adherence to predefined data formats and standards. We have developed a manual parser tailored to our specific requirements. This parser is designed to seamlessly ingest accounts.json and expenses.json, extracting pertinent information and organizing it into a structured format within a designated data structure.

a. Validation

Data validation constitutes a pivotal aspect of our parsing process, serving to maintain data integrity and consistency throughout our system. There are two types of validation done – first we ensure that the keys of each field are in the correct format and order, and second, we ensure that the value of each field is in the required format. The validation criteria for each field are meticulously crafted to enforce strict adherence to predefined data types and constraints.

During the parsing process, thorough validation of keys for each field is essential to ensure data integrity and accuracy. This validation involves checking not only the order of fields but also their formats. The isValidKey function employs **strcmp** to compare the expected key with the actual key encountered during parsing. This comparison ensures that each field appears in the correct order within the JSON object. If a mismatch occurs, indicating an unexpected key, the function raises an error, signaling a deviation from the expected JSON structure. Additionally, this function tracks the position of the next character after the key, facilitating seamless parsing of the corresponding value.

Let's delve deeper into the validation requirements for the values of each field:

account_id & user_id (Integer): These fields must strictly adhere to integer data type specifications. Additionally, account_id is constrained to a numerical range between 1 and 90, while user_id is limited to values ranging from 1 to 50. Any deviation from these constraints such as inputting a String or value outside of the range will trigger an error, signaling potential data inconsistencies.

name & description (String): Fields designated as strings must not only conform to string data type requirements but also undergo scrutiny to ensure they are not erroneously inputted as integers or empty strings. Any violation of these constraints will prompt the parser to raise an error, signaling the need for corrective action.

default_currency & currency (3-Letter String): The currency fields are expected to adhere to a standardized 3-letter string format (e.g., SGD, USD, EUR). Any deviation from this format such as inputting an Integer, empty string or String longer than 3 letters will trigger a validation error, prompting users to rectify the formatting inconsistencies.

balance & amount_spent (Integer or Float): Numerical fields such as balance and amount_spent must adhere to either integer or floating-point data type specifications. Any attempt to input invalid numerical data or non-numeric characters will result in a validation error, highlighting potential data inaccuracies.

date (String in YYYY-MM-DD format): The date field must adhere strictly to the YYYY-MM-DD format to ensure uniformity and consistency in date representation. Any deviation, such as inputting dates in DD-MM-YYYY format, providing an empty string or Integer, will prompt the parser to raise an error, necessitating corrective measures.

By enforcing validation criteria across all pertinent fields, our parsing process ensures the integrity, accuracy, and reliability of the parsed data, thereby enhancing the robustness of our data management framework. Through these meticulously crafted validation mechanisms, we uphold the standards of data quality and consistency, empowering stakeholders with trustworthy and actionable insights for informed decision-making.

b. Parser

The parsing and validation mechanisms in the provided code are designed to ensure that the data extracted from the JSON files adheres to predefined standards and formats by traversing the JSON content, validating each key-value pair against predefined criteria to ensure data integrity and consistency. Through detailed error reporting and systematic validation mechanisms, the parser enables stakeholders to identify and address data inconsistencies, ultimately enhancing the reliability and quality of the parsed data.

The parser begins by skipping leading whitespace characters and verifying that the JSON content starts with an opening square bracket [, indicating the beginning of an array. It then iterates through each element of the array while ensuring that the JSON objects start with a curly brace {. Within each object, the parser checks for valid keys by comparing them with the expected keys. If a mismatch is detected, an error is raised to signal a formatting issue.

After confirming the validity of the keys, the parser moves on to validate the corresponding values. It checks each value against predefined criteria to ensure adherence to data type specifications and constraints. For instance, integer values must fall within specified ranges, strings must not be empty or contain numerical characters, and numeric fields must be either integers or floats.

Integral to the parser's robust functionality is its sophisticated mechanism for tracking line numbers throughout the parsing process. As the parser traverses each line of the JSON input, it meticulously increments a line number variable, synchronizing it with the parser's progress. This systematic approach imbues each line with a unique identifier, facilitating precise error localization in the event of validation discrepancies. Furthermore, the parser seamlessly integrates line number tracking into its error reporting mechanism, embedding the pertinent line number within error messages

4. Compilation

To ensure seamless compilation of our files with the standard flags -Wall, -Werror, -ansi, and -pedantic, we perform a thorough post-execution check. This entails compiling the code again with all the specified flags to catch any potential issues.

Our systematic approach involved:

1. **Checking Macro Definitions**: We ensure that macro definitions are correctly formatted, verifying that no other symbols precede comments within the definitions.
2. **Adding Semicolons**: If necessary, we add semicolons to the end of structure or union definitions to adhere to C syntax standards.
3. **Using C Style Comments**: We replace C++ style comments (//) with C style comments (/* ... */) where applicable to ensure compatibility and consistency.
4. **Rearranging Declarations**: We move declarations to the beginning of code blocks or place code after declarations to comply with C language requirements.
5. **Removing Unused Variables**: Any unused variables are promptly removed to enhance code clarity and efficiency.
6. **Ensuring Structure Integrity**: We verify that structures contain all required members, addressing any discrepancies to maintain the integrity of the data model.
7. **Avoiding Inline Variable Declarations**: We refrain from declaring variables in the initialization part of for loops, promoting readability and adherence to coding conventions.

By diligently addressing these issues, we not only resolve compilation errors but also uphold code quality standards.

a. Compiling Input FSM

To rectify errors in our input Finite State Machine (FSM) and its associated files, we meticulously address each issue based on the error messages encountered. Below is a screenshot of successfully compiling the input FSM with all the flags.

```
PS C:\Users\wangyongjie\term6\PLC\50.051-Personal-Finance-Manager> gcc -o FSM_input  -ansi -pedantic -Wall -Werror  FSM_Input.c Parse
rAccounts/AccountsParser.c ParserExpenses/ExpenseParser.c
PS C:\Users\wangyongjie\term6\PLC\50.051-Personal-Finance-Manager> ./FSM_input
account.json file and expenses.json file are successfully read.
All accounts fields are successfully parsed and validated.
All expenses fields are successfully parsed and validated.
Successfully processed ParserAccounts/Accounts.json and ParserExpenses/Expenses.json.
```

a. Compiling Output FSM

The type of error we encountered is similar to what we encountered while compiling input FSM.

Below is a screenshot of successfully compiling the output FSM with all the flags.

```
PS C:\Users\wangyongjie\term6\PLC\50.051-Personal-Finance-Manager> gcc -o FSM_output  FSMoutput.c HTML_output/Page2-UserBarGraph.c HT
ML_output/Page3-AccOverallExpenses.c HTML_output/Page4-AccYearlyExpenses.c ParserExpenses/ExpenseParser.c ParserAccounts/AccountsPars
er.c  -ansi -pedantic -Wall -Werror
PS C:\Users\wangyongjie\term6\PLC\50.051-Personal-Finance-Manager> ./FSM_output
Please enter 'exit' to terminate the Finance Overview Session at any time.
Enter the user ID: 3
```

To ensure seamless execution of all files across our diverse team, comprising both Mac and Windows users, we've meticulously tested each file individually. Should any errors arise preventing successful execution on any platform, we've identified the root causes and made necessary modifications. Our efforts revealed two primary issues stemming from platform differences. Firstly, in the `Page2-UserBarGraph.c` file, the failure to initialize all elements of the `accountsPerUser` array to zero caused errors specifically on Windows systems, while it worked flawlessly on macOS. Secondly, we identified a case sensitivity issue when referencing header files, where Windows allowed incorrect casing without error, whereas macOS flagged it as an error. To address these discrepancies and ensure uniform functionality across all platforms, we've taken corrective actions by explicitly initializing the array to zero and correcting the casing inconsistencies. These measures guarantee that every file runs successfully on every team member's computer, fostering seamless collaboration and productivity.

5. Finite State Machine (FSM)

Our project employs two Finite State Machines (FSMs): an input FSM, which manages the parsing and validation of expense and account JSON files, and an output FSM, responsible for integrating four output files and determining the appropriate page based on user inputs. Serving as the program's entry file, the output FSM orchestrates the flow of our application, ensuring seamless user interaction and data processing.

a. Input FSM

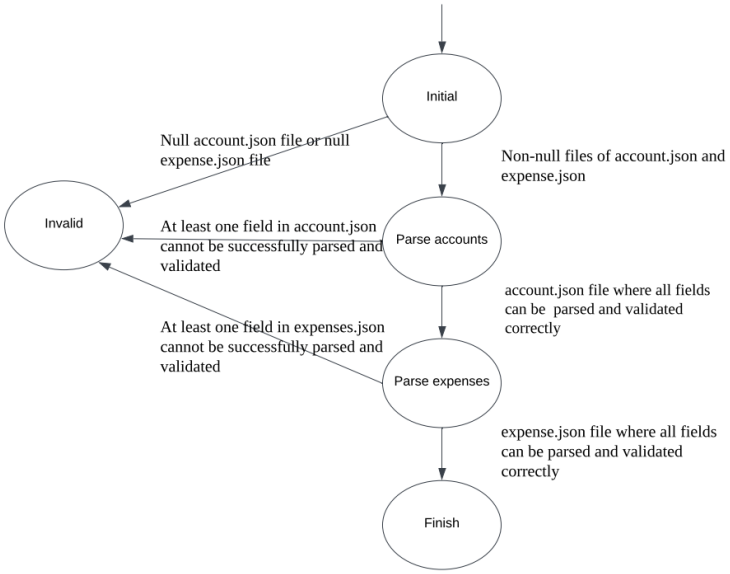

Figure 1. Input FSM

The input FSM transitions through multiple states based on the outcome of operations like file reading and JSON parsing.

Initially, the FSM is set to the **Initial state** where it attempts to open and confirm access to the specified account and expense JSON files. If either file is null, the FSM transitions to an **Invalid state**, indicating a failure in the operation. If successful, it progresses to **Parse accounts state**.

In the **Parse accounts state**, the FSM reads and parses the contents of the account.json file. It calls the parse_accountsjson function. If the parsing is successful and all fields are validated without errors, the FSM moves to the **Parse expenses state**. Otherwise, it transitions to **Invalid**.

During the **Parse expenses state**, a similar process is executed for the expense file. The file's content is read, parsed, and validated. If successful, the FSM reaches the **Finish state**, indicating that both files have been successfully processed and validated. If an error occurs during this phase, the FSM again transitions to **Invalid**.

Overall, this FSM efficiently manages the flow of reading, parsing, and validating structured JSON data, handling errors and ensuring that each step must be completed successfully before moving to the next.

b. Output FSM

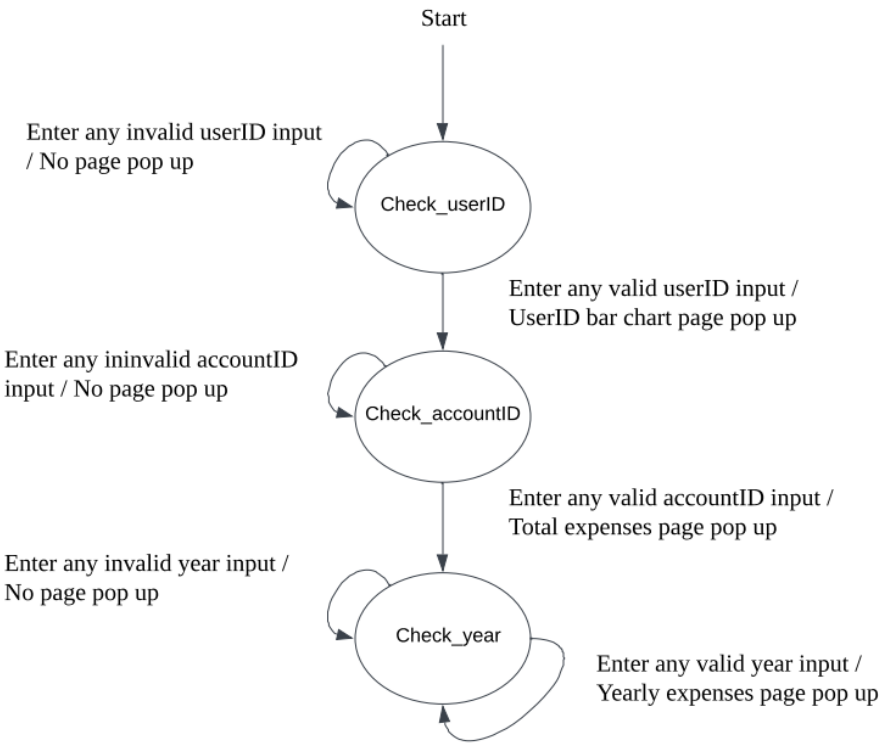

Figure 2. Output FSM

This output FSM, a core component of the system, responds dynamically to user inputs at various stages of the financial overview process.

When initiated, the FSM starts in the **Check_userID state**, checks the validity of a user ID entered by the user, and, if valid, generates and pops up the User Bar Graph HTML page. It then transitions to the **Check_accountID state** upon successful validation or prompts the user to re-enter a correct user ID if validation fails.

In the **Check_accountID state**, it validates an entered account ID entered by the user, and if correct, produces a detailed Overall Expenses for that account, generating and popping the Account Overall Expenses HTML page. On failure, the user is prompted to retry.
Finally, the **Check_year state** checks if the entered year is within a valid range and if the year corresponds to any recorded expenses for that account. If successful, the Account Overall Yearly Expenses html page will be generated and displayed.

The flow between states allows for an organized progression through different levels of financial analysis, from broad user overview down to specific account details for a chosen year. Each state transition is dependent on successful input validation, ensuring that the system's output remains reliable and relevant to the user's input.

In addition to this, since the output FSM will be used as the initial point of entry into the program to interact with the user at the terminal, we also implement the code to handle the following three rules: 1) Only integers can be used as inputs to the FSM. Therefore, before feeding the user input into the FSM, it will first check if it is an integer in the main function. 2) When the user types 'exit', the user will not be asked to continue typing input and the program will stop checking and sending the input to the FSM. 3) The main page will pop up once the user executes the output file.

III. Challenges

1. Parsing of json file

In writing the parser function, the primary challenges include handling the complexity of the JSON structure. This function is responsible for parsing and validating the format of JSON data to ensure it matches the expected structure and data types. This requires a precise understanding of the expected JSON format and checking for it in the code. Another challenge is error handling. During the validation process, if data does not meet the expected format or required fields are missing, errors need to be reported promptly. This requires providing detailed descriptions of errors to help developers or users quickly identify the issue.

Additionally, precise line number tracking is a challenge. The function needs to accurately calculate the position of errors in the JSON data, including line numbers. This is crucial for debugging and resolving issues. Data type checks are also a challenge, requiring verification that each field's data type matches the expected type, such as integer, float, or string. Different data types require different validation methods, and the code needs to correctly recognize and handle them.

Since we have the corresponding print information in the place where the error is returned, not only can it prompt the user to use the program but also helps us to debug. When we change the field in the JSON but don't get the corresponding expected results, we can use the information returned by the print of the number of rows and other information to detect and locate the problem and fix it.

2. Plotting of Graph

To effectively visualize cashflow through expense graphs, we first explored and evaluated various methods before ultimately deciding to utilize the Plotly library from JavaScript. This decision stemmed from Plotly's versatility and robust capabilities, offering an extensive range of charting options tailored to diverse data visualization needs. Then we deliberated about the most suitable graph formats to convey the intended information effectively. For user graphs, bar graphs emerged as the optimal choice, providing a clear breakdown of total expenses in SGD for each account held by the user. Conversely, scatter plots emerged as the preferred visualization method for account-based graphs, such as yearly and monthly expense trends, because they offer the flexibility to distinguish between expenses in different currencies using different colored lines.

With the graph formats delineated, the next challenge entailed devising efficient methods for data storage to facilitate seamless plotting. For user graphs, the code harnessed a struct data type called ExpenseTotalsSGD, with each instance representing aggregated expenses in SGD for a specific account. For yearly graphs, the visualization strategy incorporated distinct color lines to differentiate expenses denominated in different currencies, namely USD, SGD, and EUR. To enable this visualization, the code employed three arrays—totalExpensesSGD, totalExpensesUSD, and totalExpensesEUR—each dimensioned to accommodate the maximum number of years. Through iterative processing of expense data, the code systematically recorded expenditure amounts for each currency, leveraging array indexing to facilitate seamless retrieval of expense data based on year and currency. Similarly, for monthly graphs, the code employed a parallel approach, utilizing three arrays to record expenses in SGD, USD, and EUR across all twelve months. By iterating through expense data and filtering entries based on the year input by the user, the code meticulously logged expenditure amounts for each month, enabling precise visualization of spending trends over time.

Having formulated the data visualization strategies, the final challenge involved seamlessly integrating the Plotly code into HTML files. This necessitated invoking the Plotly JavaScript library within the HTML code structure and dynamically writing the HTML code onto the .html page inside the C file to generate the graphs.

3. Integrating HTML code into C Programming

The integration of HTML code into C programming posed several challenges, prompting us to explore various approaches to achieve seamless integration. Initially, we experimented with incorporating HTML alongside JavaScript functions within C, aiming to leverage the combined functionalities. However, we encountered difficulties in effectively calling C functions from within the HTML environment, hindering the smooth execution of our intended functionalities. Subsequently, we explored the utilization of WebAssembly, a promising technology for compiling C code into a binary format that can be executed within web browsers. While this approach showed potential, the process of compiling multiple C functions into WebAssembly proved to be troublesome and time-consuming. Seeking an alternative, we delved into client-server architecture using Node.js, which necessitated recompilation of C code using node-gyp commands. Unfortunately, we faced challenges with proper execution of C functions within the HTML environment, leading us to seek alternative solutions.

Finally, we settled on an approach known as file-based HTML generation in C, which involves utilizing a C program to dynamically generate HTML content and write it to a file. This method is commonly employed for generating static HTML pages or dynamically generating HTML content that doesn't require real-time updates. It offers a straightforward implementation without the need for complex web server setups or additional dependencies. By adopting this approach, we were able to seamlessly integrate HTML and C functionalities, effectively overcoming the challenges encountered with previous methods. The generated HTML file can be easily opened in a web browser, providing a convenient means of viewing the dynamically generated content. This method proved to be a pragmatic solution, facilitating the integration of HTML and C code while ensuring efficiency and simplicity in implementation.

4. Opening of HTML Page

Upon successfully integrating HTML into our C programming environment, we encountered a subsequent challenge with the manual opening of HTML pages. The need to individually click on each HTML file generated by our C program proved to be both troublesome and inefficient. Especially since we were dealing with multiple pages spread across different folders. This manual process not only consumed valuable time but also lacked a professional touch, detracting from the overall user experience. Recognizing the need for a more streamlined approach, we sought a solution to automatically open the HTML files upon completion of the C functions.

After exploring various options, we devised a method to automate the opening of HTML files across different operating systems. Leveraging system commands such as "start" for Windows, "xdg-open" for Linux, and "open" for Apple systems, we implemented a mechanism to trigger the automatic opening of HTML files. With this enhancement, the generated HTML pages are now seamlessly launched in the default web browser upon completion of the C functions, eliminating the need for manual intervention and enhancing the overall efficiency of our workflow. This automated process not only improves productivity but also enhances the professionalism of our application, providing users with a smoother and more intuitive experience when interacting with the generated HTML content.

5. Graph and Categorization Table integration in HTML Page

To effectively integrate both the categorization table and expense graph into an HTML page, we first had to ensure there is no code overlap between the two functionalities. The categorization table is generated based on the expenses categorized by the shop dictionary, updating total values for each category and displaying them in a tabular format. On the other hand, the expense graph calculates the total expense in SGD, USD, and EUR for each account over the years or for a single year and displays it in a scatter plot.

We had to ensure that the HTML formatting is such that the table precedes the graph, maintaining clarity and readability for the user. Headers and descriptions are appropriately formatted, avoiding any overlaps. CSS styling is applied to ensure consistency with the overall application's design and layout, ensuring a seamless user experience.

The categorization process involves iterating through expenses and matching them with predefined shop categories. Total expenses are then updated based on the currency and category, ensuring accurate accounting. Similarly, the expense graph iterates through expenses to calculate total expenses for each year and currency, which are then plotted on the graph using Plotly library.

Finally, the HTML page is generated with the necessary elements for both the categorization table and expense graph and with its style and format consistent with the overall application.

6. Combining HTML Pages for Output FSM

After setting up the framework for the FSM and implementing functions for different pages, we encountered errors when compiling all three output files together to generate the FSM. These errors included naming conflicts and repeated definitions. Since different individuals wrote the functions for the output files, there were instances of using the same variable names or structures across multiple header and source files. To resolve these conflicts, we consolidated commonly used structs or variables into a single common.h file.

V. Learning Points

During the project we delved into various aspects of programming and project development. One notable area of focus has been our exploration of Finite State Machines (FSMs) and their integration into projects. FSMs serve as powerful tools for modeling complex systems with multiple states and transitions, offering structured approaches to manage control flow, user interfaces, and other dynamic processes.

In parallel, we've honed our skills in manual parsing of JSON files and the extraction of pertinent information, enabling us to organize data into structured formats efficiently. This proficiency equips us with the ability to handle diverse data sources seamlessly within our projects.

Furthermore, we've embraced the fusion of HTML code with C programming, expanding our comfort zone to create dynamic and interactive user interfaces. This integration enhances the usability and visual appeal of our applications, elevating the overall user experience.

Moreover, our exploration of the Plotly library has empowered us to visualize data through insightful graphs, enriching our projects with compelling visual representations. These graphical aids facilitate data analysis and interpretation, enhancing the effectiveness of our financial manger.

As we apply input and output FSMs in our projects, we deepen our understanding of their practical utility and expand our capabilities in developing scalable solutions to address real project requirements. When considering the flow of a project, having a clear global understanding of the project structure is essential. In implementing FSMs, high levels of mutual understanding and collaboration among team members are also crucial. Open communication channels ensured a smooth integration of code and cohesive development across the team. This collaboration helps avoid duplication of effort and potential conflicts, fostering a more productive work environment.

Learning how to effectively use FSMs within this project benefited both the project and team collaboration. By working together, we shared insights, provided feedback, and collectively troubleshoot challenges, fostering continuous improvement.

Collectively, these skills in manual JSON parsing, HTML integration, Plotly utilization, and FSM implementation equipped us with a comprehensive toolkit for developing robust, user-friendly applications that meet the demands of real-world projects.