A · P · U
ASIA PACIFIC UNIVERSITY
OF TECHNOLOGY & INNOVATION

INDIVIDUAL ASSIGNMENT

CT006-3-3-APLC

ADVANCE PROGRAMMING LANGUAGE CONCEPTS

APU3F2311SE

HAND-OUT-DATE: 18/12/2023

HAND-IN-DATE: 31/1/2024

NAME: HOOI KAI JUN

TP No: TP060766

Table of Contents

Introduction

Programming paradigm refers to the fundamental styles or an approach to solve an existing problem using some programming languages (geeksforgeeks, 2023). Alternatively, we might define it as a methodical approach to problem solving that makes use of the resources available to us, under the direction of a certain approach (geeksforgeeks, 2023). Each programming paradigm establishes the frame of mind and organizing principles that programmers should follow when planning and structuring their code (Liyan, 2023). They influence how programs are written, how problems are solved, and how the overall design philosophy is developed (Liyan, 2023). Every paradigm has advantages and disadvantages, and choosing the right paradigm for a given task can have a significant impact on how effective, maintainable, and scalable a program is (Liyan, 2023). The diagram below shows the several types of programming paradigms:

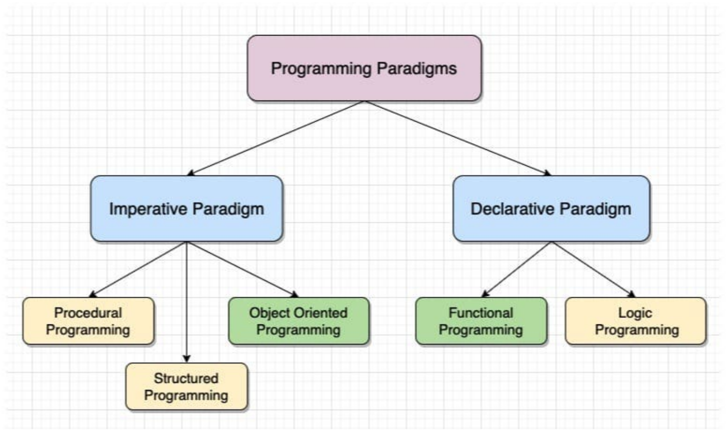
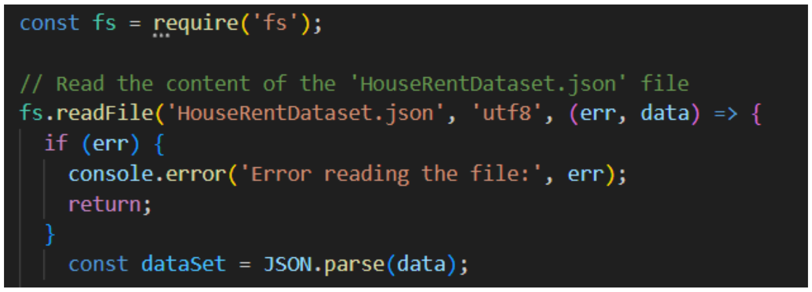


Figure 1: Programming Paradigms

Hence, this assignment focuses on analyzing any data-intensive imperative software program's relationships, structure, and method execution is part of the review process. Using a multi-paradigm design approach that incorporates higher-order functions, side effects, and purity—all fundamental concepts in functional programming—is essential to improving data parallelism. Parallel processing is made possible by using operations like filter, sort, map, and reduce. This increases productivity and evens out the effort. This report should highlight applied programming concepts while documenting both new and current solutions.

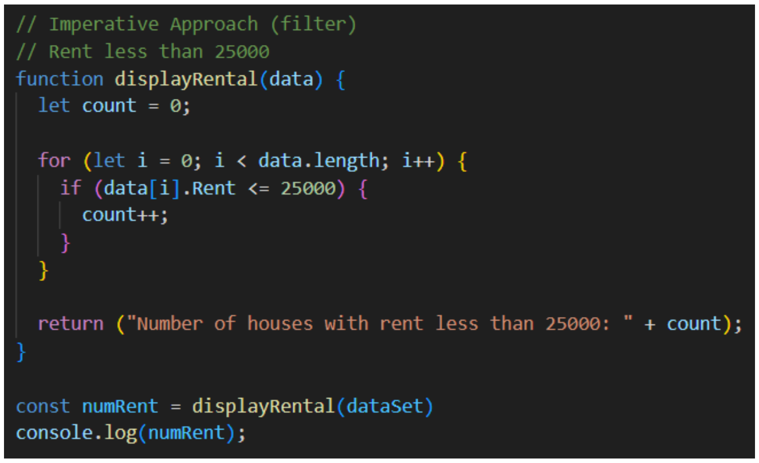
Review analysis of the existing imperative software program



```
const fs = require('fs');

// Read the content of the 'HouseRentDataset.json' file
fs.readFile('HouseRentDataset.json', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading the file:', err);
    return;
  }
    const dataSet = JSON.parse(data);
```

Figure 2: Essential modules

Based on the code snippet above, it utilizes the Node.js “fs” module to read the contents of the file named “HouseRentDataset.json”. First, it imports the 'fs' module, which is a pre-installed Node.js module that helps with file system functions. The readFile function is then used by the code, which assumes that the target file, "HouseRentDataset.json," is encoded in UTF-8. The asynchronous file reading procedure is managed by a callback function. The code examines this callback for errors; if any are found, it logs the information to the console and ends the function. If everything runs without a hitch, the code opens the file and parses the contents, which it assumes are in JSON format. This code reads a JSON file, handles errors, and transforms the content of the file into a JavaScript object that may be used for other programmatic tasks.

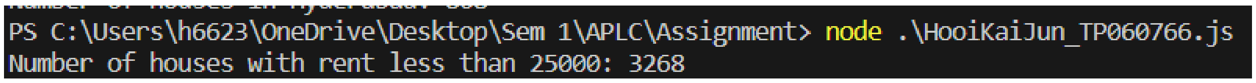


```javascript
// Imperative Approach (filter)
// Rent less than 25000
function displayRental(data) {
  let count = 0;

  for (let i = 0; i < data.length; i++) {
    if (data[i].Rent <= 25000) {
      count++;
    }
  }

  return ("Number of houses with rent less than 25000: " + count);
}

const numRent = displayRental(dataSet)
console.log(numRent);
```

Figure 3: filter (Imperative)

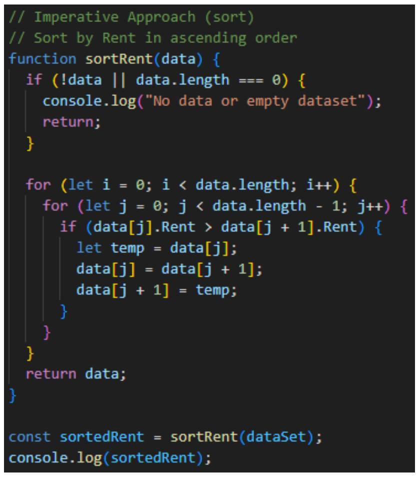
Based on the code snippet above, `displayRental` function is the imperative way of "filter" as it displays the number of houses in Hyderabad. The function uses an array of house data as its parameter. Through a for loop, the function determines how many homes have rents of less than or equal to $25,000 and outputs the result as a string. This method receives a dataset (dataSet), logs the result to console, and stores it as numRent variable; ultimately, utilizing an unambiguous yet practical approach--it efficiently tallies and presents: all homes within the dataset with rents under $25,000.

<u>Output:</u>



```
PS C:\Users\h6623\OneDrive\Desktop\Sem 1\APLC\Assignment> node .\HooiKaiJun_TP060766.js
Number of houses with rent less than 25000: 3268
```

Figure 4: Output

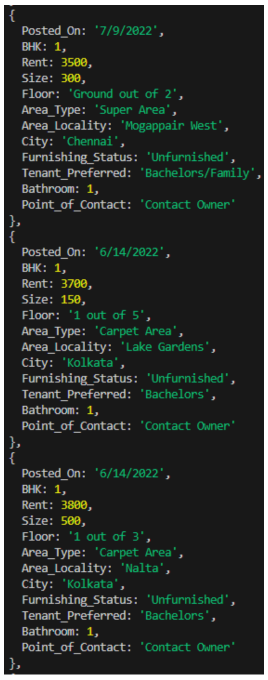


```javascript
// Imperative Approach (sort)
// Sort by Rent in ascending order
function sortRent(data) {
  if (!data || data.length === 0) {
    console.log("No data or empty dataset");
    return;
  }

  for (let i = 0; i < data.length; i++) {
    for (let j = 0; j < data.length - 1; j++) {
      if (data[j].Rent > data[j + 1].Rent) {
        let temp = data[j];
        data[j] = data[j + 1];
        data[j + 1] = temp;
      }
    }
  }
  return data;
}

const sortedRent = sortRent(dataSet);
console.log(sortedRent);
```

Figure 5: sort (imperative)

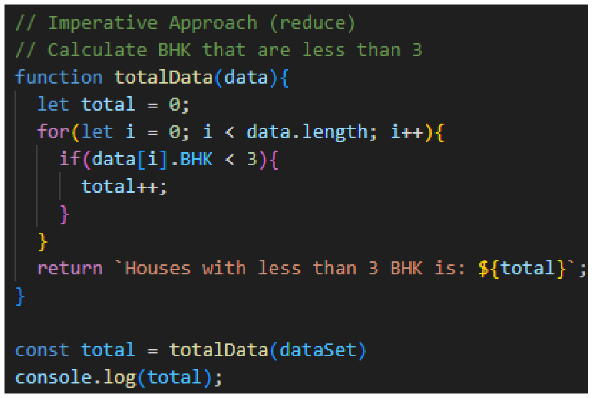
The code snippet above employs the `sortRent` function as an imperative method for sorting; it arranges house rentals in ascending order. This function, utilizing a house data array as its parameter, applies the bubble sort algorithm: specifically ordering based on ascending 'Rent' property values. After providing sortedRent with the ordered dataset--the resulting outcome is recorded directly onto the console. To conclude, the method adeptly manages empty or undefined datasets: it sorts the dataset based on ascending rental values.

<u>Output:</u>



```
{
  Posted_On: '7/9/2022',
  BHK: 1,
  Rent: 3500,
  Size: 300,
  Floor: 'Ground out of 2',
  Area_Type: 'Super Area',
  Area_Locality: 'Mogappair West',
  City: 'Chennai',
  Furnishing_Status: 'Unfurnished',
  Tenant_Preferred: 'Bachelors/Family',
  Bathroom: 1,
  Point_of_Contact: 'Contact Owner'
},
{
  Posted_On: '6/14/2022',
  BHK: 1,
  Rent: 3700,
  Size: 150,
  Floor: '1 out of 5',
  Area_Type: 'Carpet Area',
  Area_Locality: 'Lake Gardens',
  City: 'Kolkata',
  Furnishing_Status: 'Unfurnished',
  Tenant_Preferred: 'Bachelors',
  Bathroom: 1,
  Point_of_Contact: 'Contact Owner'
},
{
  Posted_On: '6/14/2022',
  BHK: 1,
  Rent: 3800,
  Size: 500,
  Floor: '1 out of 3',
  Area_Type: 'Carpet Area',
  Area_Locality: 'Nalta',
  City: 'Kolkata',
  Furnishing_Status: 'Unfurnished',
  Tenant_Preferred: 'Bachelors',
  Bathroom: 1,
  Point_of_Contact: 'Contact Owner'
},
```

Figure 6: Output

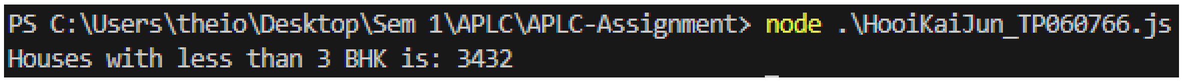


```javascript
// Imperative Approach (reduce)
// Calculate BHK that are less than 3
function totalData(data){
  let total = 0;
  for(let i = 0; i < data.length; i++){
    if(data[i].BHK < 3){
      total++;
    }
  }
  return `Houses with less than 3 BHK is: ${total}`;
}

const total = totalData(dataSet)
console.log(total);
```

Figure 7: reduce (imperative)

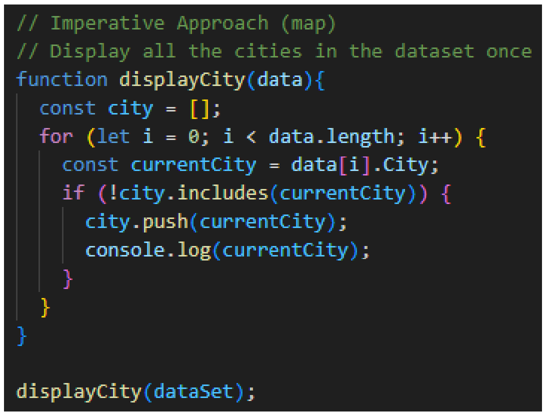
Based on the code snippet above, `totalData` function is the imperative way of "reduce" as it displays the houses less than 3 BHK. The function uses an array of house data as its parameter. The method iterates through the dataset using a for loop, increasing a counter for each home with fewer than 3 BHK. The number of such houses is returned as a string in the result. To sum up, the code effectively ascertains and conveys the quantity of residences with fewer than three bedrooms within the given dataset.

<u>Output:</u>



```
PS C:\Users\theio\Desktop\Sem 1\APLC\APLC-Assignment> node .\HooiKaiJun_TP060766.js
Houses with less than 3 BHK is: 3432
```

Figure 8: Output

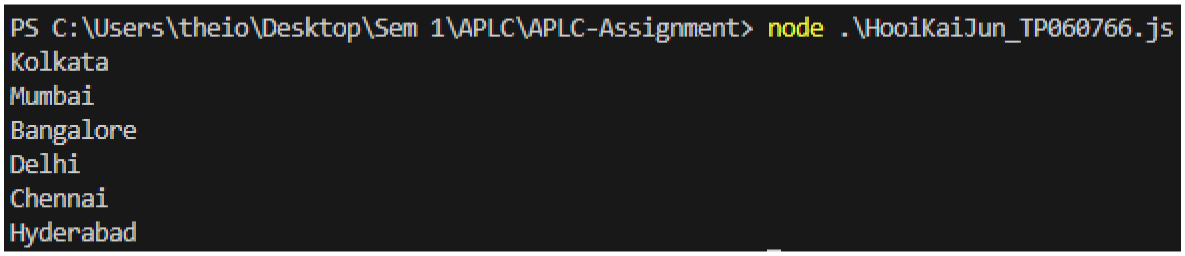


```
// Imperative Approach (map)
// Display all the cities in the dataset once
function displayCity(data){
  const city = [];
  for (let i = 0; i < data.length; i++) {
    const currentCity = data[i].City;
    if (!city.includes(currentCity)) {
      city.push(currentCity);
      console.log(currentCity);
    }
  }
}

displayCity(dataSet);
```

Figure 9: map (imperative)

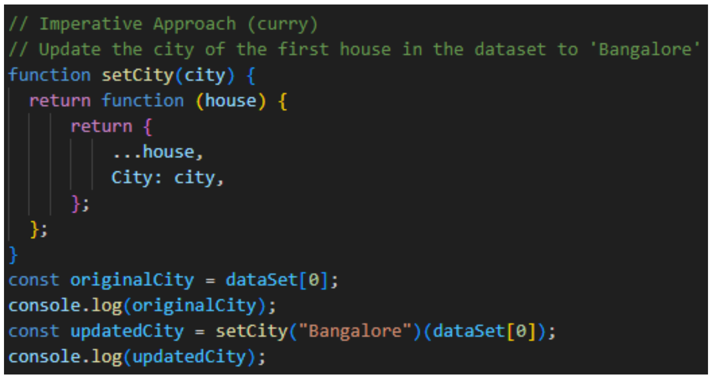
Based on the code snippet above, `displayCity` function is the imperative way of “map” as it displays all the cities in the dataset once. The function uses an array of house data as its parameter. The program loops through the dataset using a for loop, logging the current city if it is not already present in an array called "city." To put it briefly, the code effectively recognizes and shows distinct city names from the dataset, enhancing readability in the console output.

<u>Output:</u>



```
PS C:\Users\theio\Desktop\Sem 1\APLC\APLC-Assignment> node .\HooiKaiJun_TP060766.js
Kolkata
Mumbai
Bangalore
Delhi
Chennai
Hyderabad
```

Figure 10: Output

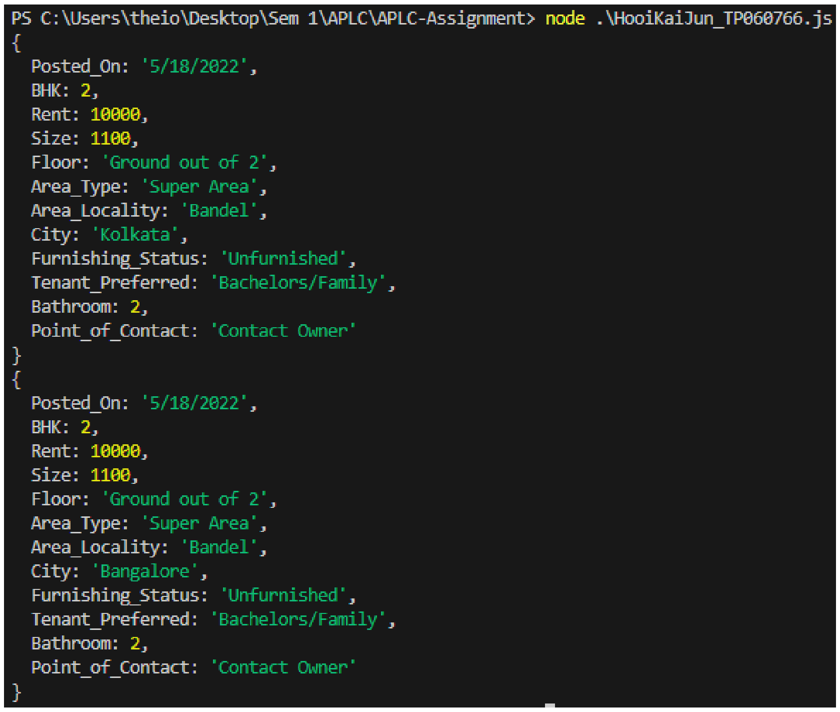


```javascript
// Imperative Approach (curry)
// Update the city of the first house in the dataset to 'Bangalore'
function setCity(city) {
  return function (house) {
    return {
      ...house,
      City: city,
    };
  };
}
const originalCity = dataSet[0];
console.log(originalCity);
const updatedCity = setCity("Bangalore")(dataSet[0]);
console.log(updatedCity);
```

Figure 11: curry (imperative)

Based on the code snippet above, `setCity` function is the imperative way of “curry” as it updates the city of the first house in the dataset to Bangalore. The function uses an array of house data as its parameter. It returns a new function after receiving a city as input. When a house object is passed as an input to the returning function, a modified copy of the house object is created with the supplied city specified in its 'City' property. With this method, the city can be dynamically set for various dwelling types. To put it briefly, the code makes it easier to create changed home objects with a particular city value quickly and efficiently.

Output:

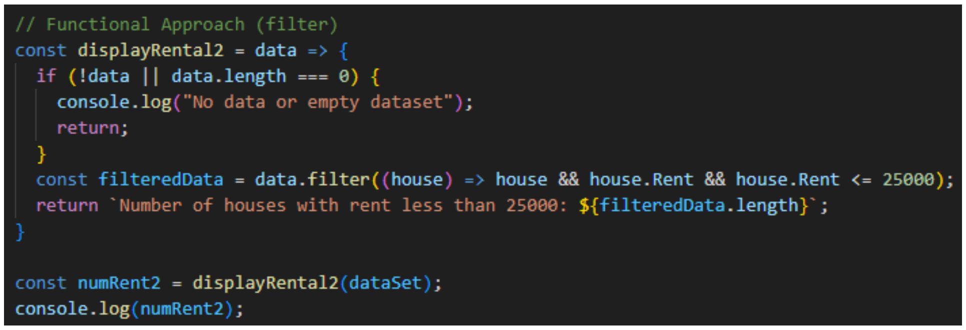


```
PS C:\Users\theio\Desktop\Sem 1\APLC\APLC-Assignment> node .\HooiKaiJun_TP060766.js
{
  Posted_On: '5/18/2022',
  BHK: 2,
  Rent: 10000,
  Size: 1100,
  Floor: 'Ground out of 2',
  Area_Type: 'Super Area',
  Area_Locality: 'Bandel',
  City: 'Kolkata',
  Furnishing_Status: 'Unfurnished',
  Tenant_Preferred: 'Bachelors/Family',
  Bathroom: 2,
  Point_of_Contact: 'Contact Owner'
}
{
  Posted_On: '5/18/2022',
  BHK: 2,
  Rent: 10000,
  Size: 1100,
  Floor: 'Ground out of 2',
  Area_Type: 'Super Area',
  Area_Locality: 'Bandel',
  City: 'Bangalore',
  Furnishing_Status: 'Unfurnished',
  Tenant_Preferred: 'Bachelors/Family',
  Bathroom: 2,
  Point_of_Contact: 'Contact Owner'
}
```

Figure 12: Output

New multi-paradigm solution design with programming concepts

Filter

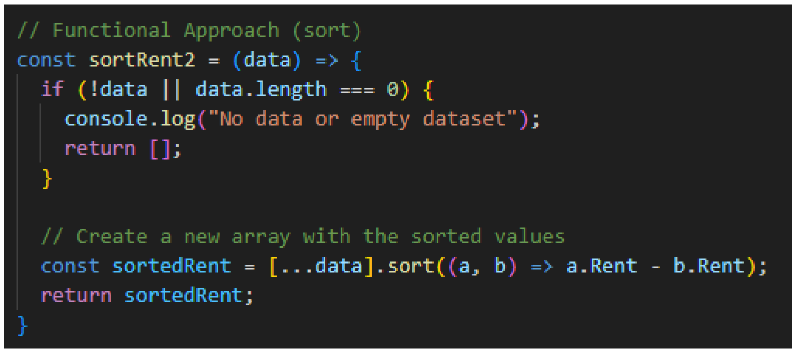


```javascript
// Functional Approach (filter)
const displayRental2 = data => {
  if (!data || data.length === 0) {
    console.log("No data or empty dataset");
    return;
  }
  const filteredData = data.filter((house) => house && house.Rent && house.Rent <= 25000);
  return `Number of houses with rent less than 25000: ${filteredData.length}`;
}

const numRent2 = displayRental2(dataSet);
console.log(numRent2);
```

Figure 13: filter (Functional)

Based on the code snippet above, it uses the arrow function syntax to define a function called `displayRental2`. This function logs a message if it finds that the input dataset is empty or undefined. After that, the dataset is filtered so that only homes with a 'Rent' property equal to or less than 25,000 are included. A string representing the number of such houses is the outcome. In conclusion, the code effectively employs short arrow function syntax and filtering algorithms to calculate and convey the number of houses with rent below 25,000 in the provided dataset. Refer to Figure 4: Output for output.

Sort

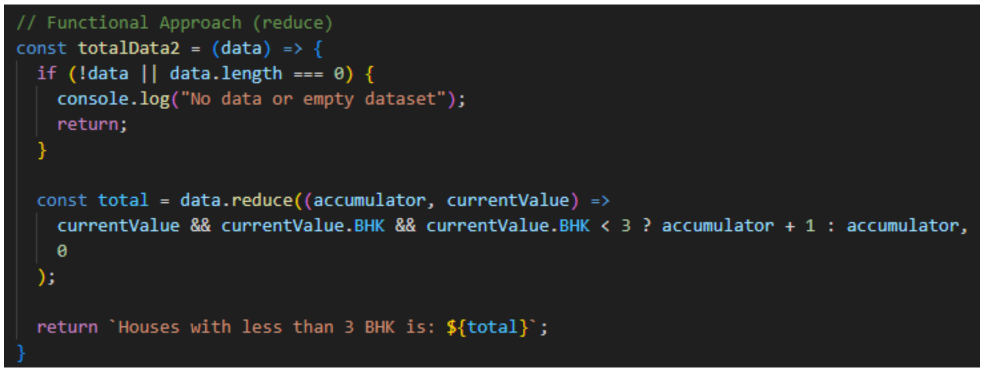


```
// Functional Approach (sort)
const sortRent2 = (data) => {
  if (!data || data.length === 0) {
    console.log("No data or empty dataset");
    return [];
  }

  // Create a new array with the sorted values
  const sortedRent = [...data].sort((a, b) => a.Rent - b.Rent);
  return sortedRent;
}
```

Figure 14: sort (Functional)

An arrow function named `sortRent2` is defined to sort a dataset of properties in ascending order according to their rental values. The function actively verifies for an empty or undefined dataset, logging a notice and returning an empty array if such conditions are met. It then leverages the spread operator along with the sort method incorporating a comparator function to construct a new array embedded with sorted data. The operation concludes by returning this resultant, sorted array. In essence, the method returns a sorted array; sorts the original dataset based on rental values--all while efficiently managing empty or undefined datasets. By employing three key components--the sort method, spread operator and arrow function syntax—it enables an implementation that is not only lucid but also easily comprehensible. Refer to Figure 6: Output for output.

Reduce

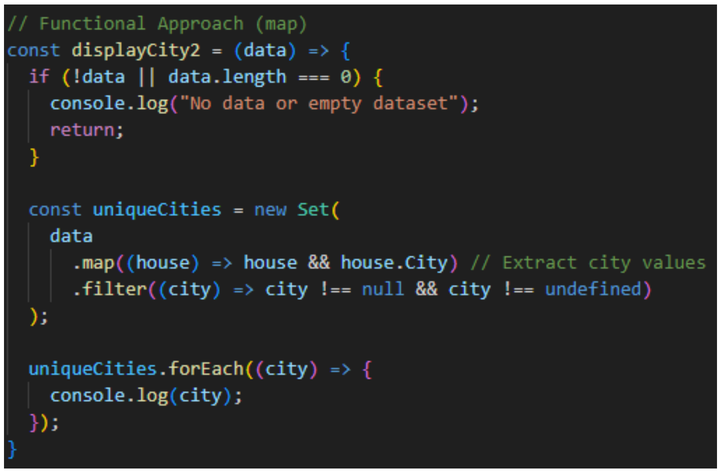


```
// Functional Approach (reduce)
const totalData2 = (data) => {
  if (!data || data.length === 0) {
    console.log("No data or empty dataset");
    return;
  }

  const total = data.reduce((accumulator, currentValue) =>
    currentValue && currentValue.BHK && currentValue.BHK < 3 ? accumulator + 1 : accumulator,
    0
  );

  return `Houses with less than 3 BHK is: ${total}`;
}
```

Figure 15: reduce (Functional)

The `totalData2` arrow function determines the number of BHK (less than 3 bedrooms) in a given dataset; it incorporates a verification for null or empty datasets--reporting an outcome, and returning whether the result is accurate. By employing the {reduce} method, this function tallies homes with values of BHK less than three. Indeed, the result returns a string that represents the number of such houses. In summary: using the 'reduce' method and arrow functions syntax—clearly implemented—it effectively determines and communicates how many residences within our given dataset have fewer than three bedrooms; this approach makes for an easily comprehensible implementation. Refer to Figure 8: Output for output.

Map



```
// Functional Approach (map)
const displayCity2 = (data) => {
  if (!data || data.length === 0) {
    console.log("No data or empty dataset");
    return;
  }

  const uniqueCities = new Set(
    data
      .map((house) => house && house.City) // Extract city values
      .filter((city) => city !== null && city !== undefined)
  );

  uniqueCities.forEach((city) => {
    console.log(city);
  });
}
```

Figure 16: map (Functional)

An arrow function called `displayCity2` is defined to extract and show unique city names from a provided dataset. It involves reporting a message, returning true if it's true, and checking for an empty or undefined dataset. The function combines 'map' and 'filter' to extract valid city values, eliminating any null or undefined items. It then retains the distinct cities in a Set; subsequently employing the `forEach` method logs each unique city name onto the console. Consequently, when presented with an empty or undefined dataset, this code effectively identifies and displays distinct city names from it. By employing Set operations and arrow function syntax, we enhance the clarity and readability of an implementation. Refer to Figure 10: Output for output.

Curry

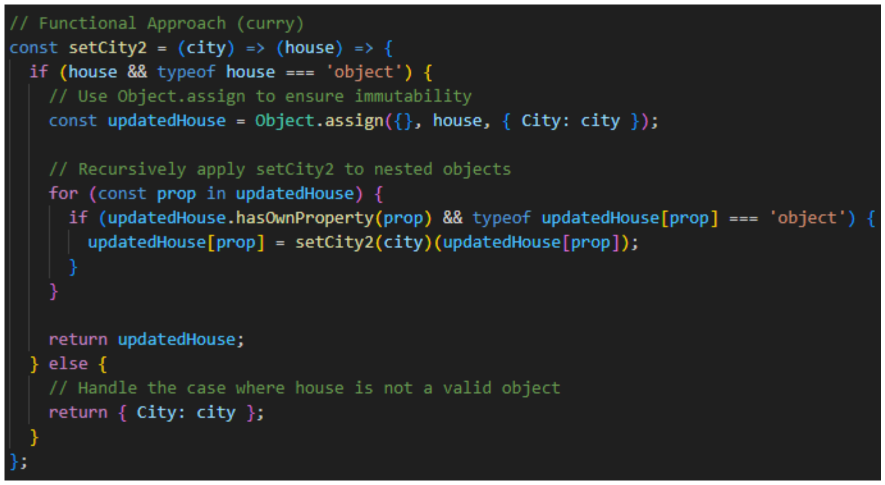

```
// Functional Approach (curry)
const setCity2 = (city) => (house) => {
  if (house && typeof house === 'object') {
    // Use Object.assign to ensure immutability
    const updatedHouse = Object.assign({}, house, { City: city });

    // Recursively apply setCity2 to nested objects
    for (const prop in updatedHouse) {
      if (updatedHouse.hasOwnProperty(prop) && typeof updatedHouse[prop] === 'object') {
        updatedHouse[prop] = setCity2(city)(updatedHouse[prop]);
      }
    }

    return updatedHouse;
  } else {
    // Handle the case where house is not a valid object
    return { City: city };
  }
};
```

Figure 17: curry (Functional)

An arrow function called `setCity2` is defined; it accepts a city as input and returns a different function. Invoking this function again with a home object guarantees immutability by generating a new object with the supplied city as its 'City' property. Recursive logic is also used in the code to apply the setCity2 function to nested items that are part of the home object. Furthermore, it verifies that the input is a legitimate object and deals with instances in which it isn't by generating a new object and setting its 'City' property to the supplied city. To sum up, the code effectively enables the production of altered home objects with a given city value, maintaining immutability, and managing different situations. Refer to Figure 12: Output for output.

Solution design for the existing and new solutions

Explanation of programming paradigm and concepts used in your solution.

As stated in the previous pages, map, sort, reduce, filter, and currying are the few examples of functional programming concepts that can be used to enhance imperative software programs. As to why functional programming paradigm is preferable when compared to imperative software development, it is because functional programming focuses on these core values first class functions, recursion, immutability, pure functions, and higher order functions which allow the software to be more efficient and effective. In particularly and most used, Higher order function (HOF) which accepts 1 or more functions as arguments or returns a function as its result (Prasad, 2023). HOF is more concise and modular which allows dynamic styles of code to be implemented (Prasad, 2023). Moreover, arrow functions are also commonly used in functional programming as it makes the syntax more concise and allows for shorter syntax to create Lambda Expressions (Koinz, 2023). It does not have its own "this" value, but it inherits the "this" value from the surrounding code (lexical scope) making it handy when handling callbacks, event handlers, methods, or functional programming paradigms (Jha, 2023).

In addition, basic ideas of currying, recursion, and partial application are demonstrated in the `setCity2` function. The setCity2 function exemplifies currying, a technique in which a function is changed to take parameters one at a time. Because this method is curried, it can be used in part by first providing the city and then applying it to an object that represents a dwelling. As a byproduct of currying, partial application makes it possible to create functions that are more flexible and reusable by addressing certain arguments prior to execution. The code actively employs recursion - a fundamental programming concept where a function invokes itself: indeed, this recursive method is purposefully applied. It utilizes the setCity2 function within nested objects of a home object; not only does it manage layered structures effectively but also creates an expressive and modular code style in functional programming. By demonstrating its utility through adaptable use cases--sophisticated solutions can indeed emerge from the grounding principles deeply rooted within functional programming ideology.

Conclusion

In conclusion, this report explains the overall programming paradigm, imperative approach of designing software and comparing it against functional programming. Functional programming concepts like first class functions, recursion, immutability, pure functions, and higher order functions which allow the software to be more efficient and effective. As demonstrated and explained with code, map, sort, reduce, filter, and currying are the few examples of functional programming that can be used to enhance imperative software programs. Hence the report provides an insight of how it's necessary to give data parallelism serious consideration in the context of data-intensive applications. This method involves splitting large datasets into manageable chunks and processing each chunk simultaneously on several CPUs or GPUs. This strategy can greatly improve the effectiveness and performance of data manipulation jobs. The assessment goal is to assess any essential software designed to manage massive amounts of data and create a software architecture that incorporates functional programming ideas using a multi-paradigm strategy.

References

geeksforgeeks. (2023, October 31). *geeksforgeeks*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/introduction-of-programming-paradigms/

Jha, L. (2023, August 22). *LinkedIN*. Retrieved from LinkedIN: https://www.linkedin.com/pulse/difference-between-regular-arrow-function-lokesh-jha/

Koinz. (2023, August 13). *LinkedIN*. Retrieved from LinkedIN: https://www.linkedin.com/pulse/functional-programming-101-what-you-need-know-koinzapp/

Liyan, A. (2023, August 11). *Medium*. Retrieved from Medium : https://medium.com/@Ariobarxan/what-is-a-programming-paradigm-ec6c5879952b

Prasad, S. (2023, January 3). *freeCodeCamp*. Retrieved from freeCodeCamp: https://www.freecodecamp.org/news/higher-order-functions-in-javascript-explained/#:~:text=A%20higher%20order%20function%20is%20a%20function%20that%20takes%20one,functions%20like%20map%20and%20reduce.