ЛАБОРАТОРНАЯ РАБОТА № 2
«Исследование функционального подхода к программированию с использованием JavaScript.»

1. Цель работы

Изучение особенностей декларативных парадигм разработки ПО, получение практических навыков разработки ПО с использованием JavaScript.

2 Краткие теоретические сведения

Декларативная парадигма программирования — это подход к разработке программного кода, основанный на описании желаемого результата, а не на определении шагов, необходимых для его достижения. Вместо того, чтобы явно задавать последовательность команд, программист выражает свои намерения и ожидания от программы.

Одним из основных принципов декларативного программирования является использование деклараций или спецификаций, которые описывают, что нужно сделать, а не как это нужно делать. Это позволяет разработчику сосредоточиться на самой задаче, а не на деталях реализации.

Ярким примером декларативной парадигмы является SQL — язык структурированных запросов. Вместо того, чтобы программист задавал шаги, каким образом получить данные из базы данных, используется SQL-запрос, который описывает, какие данные нужны. База данных самостоятельно оптимизирует выполнение запроса и предоставляет результат.

Преимущества декларативной парадигмы заключаются в более высоком уровне абстракции, что делает код более понятным и поддерживаемым. Кроме того, декларативный код обычно более компактный и лаконичный, что упрощает его написание и чтение.

Однако декларативное программирование не всегда возможно или эффективно во всех случаях. В некоторых сценариях требуется более детальное управление над выполнением программы, и в таких случаях императивный подход может быть предпочтительным. Но в задачах, где акцент сделан на результате, а не на способе достижения его, декларативная парадигма оказывается очень полезной и эффективной.

Основные черты:

– задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат;

– отсутствует явное указание на то, как именно следует решать данную задачу – это решение принимается компилятором/интерпретатором языка.

Главное достоинство:

– Позволяет сконцентрироваться на понимании задачи, а не на деталях ее решения, оставляя детали реализации машине. («Я не тактик, стратег!»). При определенных условиях это позволяет легче избегать побочных эффектов и эффективнее оптимизировать код.

Главный недостаток:

– Сложность стыковки с реальным миром:
  – Императивность нижележащих вычислительных средств;
  – Синхронизация с пользователями и внешними процессами.

Развитие декларативной парадигмы

– Функциональная и логическая парадигма:
  – LISP -> Common Lisp, Scheme, Clojure;
  – Prolog -+-> Erlang;
  – ML -> Standard ML, OCAML, Haskell, F#, etc;
  – Scala, C# – мультипарадигменные языки.

- Языки запросов:
  - SQL, XQuery, XSLT, etc.
- Языки разметки:
  - HTML, XAML.
- Различные DSL.

Рассмотрим более подробно функциональную парадигму.

Функциональная парадигма программирования возникла из декларативной парадигмы программирования, которая описывает, что должно быть сделано, а не как это должно быть сделано. В функциональной парадигме программа представляет собой набор функций, которые принимают аргументы и возвращают результаты. Функции в функциональном программировании рассматриваются как математические функции, которые не имеют состояния и не изменяют свои аргументы. Это позволяет писать более декларативный и выразительный код, который легче читать и поддерживать.

Основное отличие функциональной парадигмы от процедурной заключается в том, что в функциональной парадигме данные рассматриваются как неизменяемые (immutable), а изменения состояния программы минимизируются или отсутствуют вовсе. Это делает функциональное программирование более предсказуемым и устойчивым к ошибкам, так как отсутствие изменяемого состояния упрощает отладку и тестирование программ.

Еще одним отличием функциональной парадигмы от процедурной является использование рекурсии вместо циклов. В функциональном программировании рекурсия является основным механизмом повторения, что делает код более компактным и выразительным.

Одним из первых языков программирования, который активно использовал функциональный подход, был язык Lisp, разработанный в 1950-х годах. Он внес значительный вклад в развитие функционального программирования и стал основой для многих других языков, таких как Scheme, Clojure, Haskell и другие.

Процедурная парадигма программирования включает в себя использование процедур, функций и подпрограмм для организации кода. Она сосредоточена на выполнении последовательности инструкций для достижения определенной цели. В процедурном программировании данные могут быть переданы в параметрах функции или использоваться глобально.

Особенности функциональной парадигмы:

- Функции высшего порядка:
  - Функции как first-class citizens;
  - Каррирование (карринг).
- Чистые функции (без побочных эффектов):
  - Дает возможность ряда оптимизаций (параллельность вычислений, мемоизация, редуцирование графа вычислений).
- Рекурсия заменяет циклы:
  - Не от хорошей жизни (нет состояния – нет условия проверки/счетчика цикла);
  - Хвостовая рекурсия оптимизируется обратно в цикл.
- Ленивое вычисление:
  - Бесконечные последовательности и другие.

Проблемы функциональной парадигмы:

- Иммутабельность данных повышает потребление памяти;
- Проблемы с вводом-выводом (не является чистой функцией);
- Сложность моделирования систем;
- Сложность адаптации программистов.

Рассмотрим разницу процедурной и функциональной парадигм на примере:

Простая процедурная функции на JavaScript:

```
function calculateSum(a, b) {
  return a + b;
}
let result = calculateSum(3, 5);
console.log(result); // Вывод: 8
```

В данном примере объявлена функция calculateSum, которая принимает два параметра a и b. Внутри функции происходит сложение параметров и возвращается результат. Затем происходит вызов функции с аргументами 3 и 5 и сохраняется результат в переменную result, которая затем выводится в консоль.

Функциональная парадигма программирования, с другой стороны, сосредоточена на использовании функций как основных строительных блоков программы. В функциональном программировании функции рассматриваются как значения, которые могут быть переданы в другие функции или использованы для создания новых функций. Функциональное программирование обычно использует неизменяемые данные и избегает изменения состояния.

Пример функционального подхода на JavaScript:

```
const add = (a, b) => a + b;
const multiplyByTwo = (number) => number * 2;
const result = multiplyByTwo(add(3, 5));
console.log(result); // Вывод: 16
```

В данном примере мы объявляем две функции add и multiplyByTwo, которые манипулируют данными и возвращают результат. Затем мы используем функцию add для сложения 3 и 5, а затем передаем результат функции multiplyByTwo для умножения на 2. Результат сохраняется в переменной result, которую затем выводим в консоль.

Рассмотрим пример с задачей поиска дублирующихся элементов в одномерном массиве.

Процедурный подход применяет последовательность команд для выполнения задачи:

```
// Процедурный подход
function findDuplicatesProcedural(arr) {
  var duplicates = [];
  for (var i = 0; i < arr.length; i++) {
    for (var j = i + 1; j < arr.length; j++) {
      if (arr[i] === arr[j] && duplicates.indexOf(arr[i]) === -1) {
        duplicates.push(arr[i]);
      }
    }
  }
  return duplicates;
}

var inputArray = [1, 2, 3, 4, 4, 5, 6, 7, 7, 8, 9];
var duplicateElements = findDuplicatesProcedural(inputArray);
console.log(duplicateElements); // Вывод: [4, 7]
```

Функциональный подход:

```
// Функциональный подход
function findDuplicatesFunctional(arr) {
  return arr.filter(function(value, index, self) {
    return self.indexOf(value) !== index && self.lastIndexOf(value) === index;
  });
}
var inputArray = [1, 2, 3, 4, 4, 5, 6, 7, 7, 8, 9];
var duplicateElements = findDuplicatesFunctional(inputArray);
console.log(duplicateElements); // Вывод: [4, 7]
```

Оба решения дадут нам массив с дублирующимися элементами [4, 7]. В процедурном подходе используются циклы для обхода массива и проверки дубликатов, а затем происходит добавления их в новый массив. В функциональном подходе используется функция filter, чтобы отфильтровать элементы, у которых индекс первого вхождения не совпадает с индексом последнего вхождения.

Пример работы с матрицей:

Задача найти дубли элементов побочной диагонали матрицы и вынести их в отдельный массив.

Процедурный подход:

```
function findDuplicates(matrix) {
  let diagonal = [];
  let duplicates = [];

  // Получаем элементы побочной диагонали матрицы
  for (let i = 0; i < matrix.length; i++) {
    diagonal.push(matrix[i][matrix.length - 1 - i]);
  }

  // Поиск дубликатов
  for (let i = 0; i < diagonal.length; i++) {
    for (let j = i + 1; j < diagonal.length; j++) {
      if (diagonal[i] === diagonal[j]) {
        if (!duplicates.includes(diagonal[i])) {
          duplicates.push(diagonal[i]);
        }
      }
    }
  }
  return duplicates;
}
```

```
// Пример использования
const matrix = [
  [6, 2, 3, 4],
  [5, 6, 0, 8],
  [9, 8, 7, 6],
  [0, 4, 3, 2]
];
console.log(findDuplicates(matrix));
```

Функциональный подход:

```
function findDuplicates(matrix) {
  const diagonal = matrix.map((row, index) => row[matrix.length - index - 1]);
  const duplicates = diagonal.filter((value, index, array) => {
    return array.indexOf(value) !== index;
  });
  return duplicates;
}
const matrix = [
  [6, 2, 3, 4],
  [5, 6, 0, 8],
  [9, 8, 7, 6],
  [0, 4, 3, 2]
];
const result = findDuplicates(matrix);
console.log(result);
```

Процедурная парадигма часто акцентирует внимание на изменяемых состояниях и может применять переменные как контейнеры для хранения данных. Например, в процедурном стиле программирования можно создать функцию для поиска дублей и изменять внешний массив, добавляя найденные дубли в него.

В функциональном стиле возвращаются дубликаты без изменения исходной матрицы или создания глобальных массивов. Функции играют роль математических функций и преобразуют данные внутри себя без изменения внешних состояний.

3 Порядок выполнения

3.1 Изучить основные средства языка JavaScript для разработки программ с использованием функциональной парадигмы.

3.2 Выполнить две задачи из варианта на языке JavaScript согласно своему варианту.

3.3 Разработать тестовые примеры.

3.4 Выполнить отладку программ.

3.5 Сформулировать выводы, проанализировав разницу разработки программ с использованием декларативных и императивных парадигм.

3.6 Оформить отчет по проделанной работе.

4 Варианты заданий

Для каждого варианта требуется создать одноместный и двухмерный массивы. Для однокамерного массива нужно добавить минимум 10 элементов, для матрицы 16 элементов.

Вариант 1

Найти сумму всех элементов в массиве.

Удалить все строки матрицы, в которых есть отрицательные элементы.

Вариант 2

Найти наибольший элемент в массиве.

Проверить, содержит ли матрица магический квадрат (сумма элементов всех строк, столбцов и диагоналей одинакова).

Вариант 3

Найти индекс первого вхождения определенного элемента в массив.

Найти среднее арифметическое элементов в каждой строке матрицы.

Вариант 4

Посчитать количество четных элементов в массиве.

Поменять местами две заданные строки матрицы.

Вариант 5

Отсортировать массив по возрастанию.

Найти сумму всех элементов в двумерном массиве.

Вариант 6

Изменить порядок элементов массива на обратный.

Отсортировать строки матрицы по возрастанию суммы их элементов.

Вариант 7

Удалить все дубликаты из массива.

Посчитать сумму элементов каждого столбца и сохранить результаты в одномерный массив.

Вариант 8

Найти среднее арифметическое всех элементов массива.

Посчитать количество строк матрицы, в которых есть хотя бы один отрицательный элемент.

Вариант 9

Проверить, является ли массив палиндромом.

Найти наименьший элемент в двумерном массиве.

Вариант 10

Найти сумму элементов на нечетных позициях массива.

Найти сумму элементов главной диагонали матрицы.