1. **함수형 프로그래밍, 절차지향 프로그래밍, 객체지향프로그래밍이란 무엇인가?**

* **절차지향 프로그래밍**

일이 진행되는 순서대로 프로그래밍을 하는 방법이다.

장점 : 코드를 따라가며 읽기가 쉬우며 작성하기에도 쉬운 편이다. 컴퓨터의 처리구조와 비슷해 실행 속도가 빠르다.

단점 : 각각의 코드가 순서에 민감하게 연결되어 있어 유지보수가 쉽지 않다. 프로그램을 분석하기가 어렵다.

* **객체지향 프로그래밍**

객체를 먼저 작성하고 함수를 작성하는 형태이다. : 객체 간의 상호작용이다.

데이터(객체)를 먼저 디자인하고 데이터에 맞는 메소드를 구현하는 방식이다. : 상태를 바꾼다. 조작한다. 데이터형에 메소드가 종속된다. 비순수 함수들을 순수객체로 묶는다.

장점 : “(주어)는 (동사)다.” 처럼 사람이 생각하는 방식과 닮아있다. 즉, (객체)는 (메소드)다. 코드의 재사용이 가능하다. 분석과 설계의 전환이 쉽다.

단점 : 처리 속도가 상대적으로 다소 느리다. 설계의 많은 시간이 소요되며 설계를 잘못하면 다시 처음부터 짜야 한다. 테스트가 어렵다.

* **함수형 프로그래밍**

함수를 먼저 작성하는 형태다. 함수를 먼저 만들고 함수에 맞는 데이터를 세팅하는 방식이다. : 상태가 없다.

장점 : 사이드이펙트를 미연에 방지한다. 객체지향보다 코드가 간결하다. 비절차형이라 평가 시점이 중요하지 않다. 테스트가 쉽다.(1회만으로 신뢰성 보장) 데이터형에 구애받지 않는다.

단점 : 상태를 조작할 수 없다.

출처 : <https://susu91.tistory.com/105>

1. **함수형 프로그래밍이 갖는 장점과 단점은 무엇인가.**

함수형 프로그래밍은 순수한 함수를 작성하고, 공유된 상태와 변경 가능한 데이터 및 부작용을 피하여 소프트웨어를 작성하는 프로세스이다.

함수형 프로그래밍은 명령형이 선언형이며, 애플리케이션의 상태는 순수한 함수를 통해 전달된다.

**함수형 프로그래밍의 장점**

함수형 코드는 명령형이나 객체지향 코드보다 간결하고 예측하기 쉬우며, 이에 따라 테스트가 더 쉬워진다.

* 부작용(Side effect)이 없다.
* Thread-safe하다.
* 테스트하기 편리하다.
* 로직에만 집중할 수 있다.

**Pure functions**

함수형 프로그래밍에서 중요한 핵심은 함수는 부작용이 없어야 하며, 외부의 상태에 종속되지 않아야 한다. 즉, 함수는 입력을 받고 출력을 반환함에 있어 외부의 값에 접근하지 않아야 한다.

example)

/\* pure function \*/

const double = num => (num \* 2);

/\* impure function \*/

const operand = 2;

const multiple = num => (num \* operand);

double의 입력이 2라면, 해당 함수의 반환값은 항상 4이다.

반면, multiple의 입력이 2로 고정되어도 함수 외부의 operand가 다른 2가 확실하지 않다면 반환 값을 4라고 예상하기 힘들다.

**Higher Order Functions**

HOC(Higher Order Functions)는 다른 함수를 인자로 사용하거나 함수를 반환하는 함수, 또는 두 가지 특징을 모두 가진 함수이다.

고차함수는 다음과 같은 경우 주로 사용된다.

* 콜백함수, 프로미스, 모나드 등을 사용하여, 동작과 효과 또는 비동기 흐름 제어를 추상화하거나 격리한다.
* 다양한 데이터 유형에 대해 작동할 수 있는 유틸리티를 생성한다.
* 함수를 부분적으로 인수에 적용하거나 재사용 또는 함수 조합을 위한 커리함수를 작성한다.
* 함수 목록을 가져오고, 해당 입력의 조합을 반환한다.

example)

const isEven = x => ! ( x % 2);

const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

const evenNumbers = numbers.filter(isEven);

console.log(evenNumbers); // 2, 4, 6, 8, 10

isEven은 배열을 처리하는 로직이 필요없으며, 배열을 순환하면서 재사용된다.

**Currying**

커링은 여러 인자를 받는 함수를 인자 하나씩 사용하여 함수 순서를 실행할 수 있도록 변환하는 것을 말한다.

이는 람다 표현식과 클로저를 통해 쉽게 구현할 수 있다.

example)

/\* not curried function \*/

function sumOfThreeThings(x, y, z) {

return x + y + z;

}

sumOfThreeThings(1, 2, 3); // 6

/\* curried function \*/

const sumOfThreeThings = x =>

y =>

z =>

x + y + z;

sumOfThreeThings (1) (2) (3); // 6

**Auto-currying**

lodash와 Ramda는 curry 메소드를 가지고 있다. 이는 여러 개의 인수를 가지는 함수를 커링된 함수로 만들어준다.

example)

// use lodash or ramda

const curry = \_.curry || R.curry;

// given function that add 2 parameters

const add = (x, y) => x + y

// transfired function

const curriedAdd = curry(add);

// results

curriedAdd(1) (2) // 3

curriedAdd(1) // (y) => 1 + y

curriedAdd(1, 2) // 3

**Function composition**

합성함수는 말그대로 두 가지 이상의 함수가 합성되었음을 뜻한다. 두 함수 f와 g가 있고 f(g(x))와 같이 사용된다고 할 때, 이를 수식 f ∘ g(x)과 같이 표현할 수 있다.

example)

const compose = (f, g) => x => x => f(g(x));

const add = x => y => x + y;

const pow = x => y => y \*\* x;

const add2 = add(2);

const square = pow(2);

const add2ThenSquare = compose(square, add2);

add2ThenSquare(10); // 144

**Point free notation**

Point free는 함수를 작성할 때, 매개변수를 정의하지 않는 것이다.

이는 함수를 더욱 간결하게 해준다.

example)

const map = fn => list => list.map(fn);

// or use lodash/fp

// import { map } from ‘lodash/fp’;

const add = x => y => x + y;

/\* function without point-free \*/

const incrementAll = numbers => map(add(1))(numbers);

/\* function with point-free \*/

const incrementAllWithPointFree = map(add(1));

incrementAll([1, 2, 3]); // [2, 3, 4]

incrementAllWithPointFree([1, 2, 3]); // [2, 3, 4]

불필요한 매개변수를 사용하지 않으므로 매개변수에 대한 이름들에 대해 생각하지 않아도 되며, 코드가 훨씬 간결해진다.

**Recursion**

재귀 함수는 어느 조건을 만족할 때까지 자기 자신을 호출하는 함수이다. 자바스크립트는 반복을 할 때, for, while 등의 반복문을 사용하지만, 함수형 프로그래밍에서는 반복문 대신 재귀를 사용한다.

아래 예제는 동일한 루프 회수와 값을 얻게 되는 반복문과 재귀함수이다.

기존의 어떤 값을 수정하는 것이 아니라, 기존 값으로 새로운 값을 계산하고 이를 사용한다.

example)

/\* sum 1..10 with loop \*/

let sum = 0;

for (let i = 1; i <= 10; i++) {

sum += i + 1;

}

console.log(sum); // 55

/\* sum 1..10 with recursion \*/

sumRange = ({ start, end }) =>

(sum) => {

if (start > end) {

return sum;

}

return sumRange({ start: start + 1, end })(sum + start);

};

sumRange({ start: 0, end: 10 })(0);

**함수형 프로그래밍의 단점**

* 상태가 없다.
* 어렵다.
* 인력이 부족하다.

출처 : <https://medium.com/korbit-engineering/%ED%95%A8%EC%88%98%ED%98%95-%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D%EC%9D%B4%EB%9E%80-e7f7b052411f>

<https://www.slideshare.net/ssusere9cffb/ss-84864214>

1. **순수함수란 무엇인가.**

**순수함수**는 입력이 같다면 출력도 항상 같은 값이 나오는 것이다. 외부의 상태를 변경하지 않는 함수(=> 평가 시점이 중요하지 않다.)

순수함수의 예

function add(a, b) {

return a + b;

}

=>

1. 동일한 값이 항상 리턴된다.

2. 외부상태를 바꾸지 않는다.

const c = 10;

function add2(a, b) {

return a + b + c;

}

=>

1. 함수 외부에 존재하는 c의 값이 변하지 않으므로 순수함수다. (만일 c의 값에 변경의 여지가 있다면 순수함수가 아님.)

2. 외부 상태(c)를 함수가 바꾸지 않는다.

var obj1 = { val: 10 };

function add5(obj, b) {

return { val : obj1.val + b }

}

console.log(obj1.val); // 10

var obj2 = add5(obj1, 20);

console.log(obj1.val); // 10

console.log(obj2.val); // 30

=>

1. 함수 외부에 존재하는 obj1의 값을 바꾸지 않으므로 순수함수다.

출처 : <https://susu91.tistory.com/106?category=788018>