함수형 프로그래밍

요약

먼저 개념 정리들을 간단하게 해봅시다.

원칙

함수형 프로그래밍에는 다음과 같은 원칙이 존재합니다.

- 입출력이 순수해야 합니다. (순수함수)
- 부작용(부산물)이 없어야 합니다.
- 함수와 데이터를 중점으로 생각합니다. → 선언적 패턴

사실 위 2개는 같은 소리입니다. 입출력이 순수하다는 것은 **반드시 하나 이상의 인자를 받고, 받은 인자를 처리하여 반드시 결과물을 돌려주어야 한다는 것**입니다. 인자를 제외한 다른 변수는 사용하면 안되고, 받은 인자만으로 결과물을 내어야 합니다. 그리고 이러한 함수를 **순수함수**라고 부릅니다. 하지만 자바스크립트에서는 this라는 개념 때문에 순수함수를 사용하기 힘듭니다. 그래도 최대한 비슷하게 할 수는 있습니다.

두 번째로, 부작용이 없어야 한다는 것은, **프로그래머가 바꾸고자하는 변수 외에는 바뀌어서는 안 된다**는 뜻입니다. **원본 데이터는 항상 불변해야 함**을 기억해야 합니다. 함수형 프로그래밍에서는 프로그래머가 모든 것을 예측하고 통제할 수 있어야 합니다.

예시

대표적인 함수형 프로그래밍 함수로는 map, filter, reduce가 있습니다.

```
let arr = [1, 2, 3, 4, 5]

let map = arr.map(x => x * 2) // [2, 4, 6, 8, 10]
```

처음에 배열(arr)을 넣어(입력), 결과(map)를 얻었습니다(출력). arr도 사용됐지만 arr이 변경되지는 않았고, map이라는 결과를 내고 아무런 부작용도 낳지 않았습니다. 바로 이런 게 함수형 프로그래밍에 적합한 함수, 즉 순수함수 입니다.

다른 예를 보겠습니다.

```
let arr = [1, 2, 3, 4, 5]
let condition = function(x) { return x % 2 === 0; }

let ex = function(array) {
    return array.filter(condition)
}

ex(arr) // [2, 4]
```

여기서 ex를 순수함수라고 볼 수 있을까요? 정답은 아닙니다. 왜냐하면 인자로 받은 array 말고도 condition이라는 변수를 사용했기 때문입니다. 위 ex 함수를 순수함수로 바꾸면 아래와 같아집니다.

```
let ex = function(array, cond) {
    return array.filter(cond)
}

ex(arr, condition)
```

이렇게 하면 쉽게 에러를 추적할 수 있습니다. 인자가 문제였거나, 함수 내부가 문제였거나 둘 중 하나이기 때문이죠.

또한 함수형 프로그래밍에서는 반복문도 사용해서는 안됩니다. 예를 들어 1~10까지 더한다고 가정하면

```
let sum = 0

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

이렇게 하겠죠. 하지만 함수형 프로그래밍에서는 다음과 같이 작성해야 합니다.

```
function add(sum, cnt) {
    sum += cnt;

    if (cnt > 0) {
        return add(sum, cnt - 1)
    } else {
        return sum
    }
}

add(0, 10)
```

이렇게 함수형으로 표현하게 되면 코드의 재사용성이 매우 높아진다는 장점이 있습니다. 한번 함수로 만들어놓으면 언제든지 add 함수를 다시 쓸 수 있게 됩니다.

물론 reduce 함수를 쓰는게 좀 더 함수형 프로그래밍 다운 방법입니다.

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
arr.reduce((prev, cur) => prev + cur)
```

좀 더 깊게 가볼까요?

여기서부터는 다시 처음부터 제대로 세세하게 얘기해보겠습니다.

함수형 프로그래밍을 알아야 하는 이유가 뭘까요?

프로그래머는 항상 자신이 짜는 코드가 좋은 코드이기를 바랍니다. 여기서 좋은 코드에 대한 명확한 기준이 있지는 않지만 적어도 **좋은 설계를 가지고 있을수록 좋은 코드가 된다**는 것 정도는 알고 있습니다.

조금 더 직관적이게 말해보자면 기존에 만들어 둔 코드가 내 발목을 잡는다면 나쁜 코드고 아니라면 좋은 코드라고도 할 수 있지 않을까요? 프로그램이 작을 때는 이에 대한 구분이 잘 되지 않습니다. 오히려 나쁜 코드가 생산성이 더 높기 때문입니다. 하지만 프로그램이 커지면 커질수록 설계가 없는 코드는 점점 생산성이 떨어집니다. 그렇기에 우리는 좋은 설계를 유지하려는 노력이 필요합니다.

우리가 익히 알고 있는 객체지향 프로그래밍 패러다임은 **객체를 중심으로 사고하고 프로그램을 작성하는 것**입니다.

반면 **데이터를 함수로 연결하는 것을 중심으로 사고하고 프로그래밍을 하는 것을 함수형 프로그래밍** 이라고 부릅니다.

함수형 프로그래밍은 프로그램을 이해하는 새로운 관점을 제공합니다.

함수형 프로그래밍은 객체지향 프로그래밍보다 더 단순하게 그리고 간결하게 프로그램을 바라볼 수 있도록 도와줍니다. 그러나 반드시 객체지향 프로그래밍보다 나은 것은 아니죠. 오히려 대부분의 언어가 객체지향으로 되어 있으며 순수함수형 언어가 극소수에 불과하다는 것이 이를 반증합니다.

그러나 자바스크립트는 함수형 프로그래밍 기반 위에 객체지향 언어의 껍데기를 씌운 언어입니다. 이렇게 다소 실험적으로 탄생한 이 언어는 **객체지향에 함수형 프로그래밍을 적당히 섞으면 훨씬 더 좋다**는 것을 개발자들에게 알려주었고 이후 탄생한 다른 언어에도 영향을 끼치며 **객체지향 언어에 함수형을 결합하는 형태의 멀티 패러다임**의 근간을 마련해 주었습니다.

우리가 하는 자바스크립트 언어는 멀티 패러다임 언어

위에서 말했듯이 자바스크립트는 함수형 패러다임을 기반으로 하면서 객체지향의 문법을 쓰는 독특한 언어입니다. 결국 우리가 쓰는 자바스크립트를 가장 잘 쓰기 위해서는 **객체지향스럽게 작성을 하면서도 함수형 프로그래밍 패러다임으로 개발하는 것이 가장 좋다**는 것입니다.

함수형 프로그래밍은 모르지만 Array Method, Promise, Event Listener, setTimeout, Redux 등등 우리가 쓰고 있는 대부분의 JS 라이브러리나 API에 함수형 패러다임이 녹아 있기에 멀리 있는 개념도 아닙니다.

자바스크립트는 멀티 패러다임의 언어이기 때문에 어느 개념이든 원하는 대로 가져다 쓸 수 있고, 심지어는 체계 없이도 어쨌든 돌아가기는 잘 돌아가는 코드를 만들기에 너무 좋습니다. 하지만 이 말은 즉, 나쁜 코드를 작성하기도 정말 쉽다는 의미입니다.

결국 **이 둘의 패러다임을 잘 구분해서 쓸 수 있어야 자바스크립트로 좋은 코드를 작성할 수 있다**는 의미이며, 프론트엔드 개발을 잘 하기 위해서는 **함수형으로 사고하는 패러다임을 잘 이해할 필요가 있습니다.**

함수형 코딩

재밌는 것은 함수형 프로그래밍은 자바스크립트의 근간이 되는 하나의 큰 축인데 반해 이에 대한 개념적인 내용들이 잘 정리된 내용들은 많지 않습니다. Redux로 인해 프론트엔드에서도 함수형 프로그래밍이 한 차례 유행을 했지만 함수형 프로그래밍에 대한 개념 및 이론들은 아직까지도 파편화 되어 있습니다.

함수형 프로그래밍 용어 다시쓰기

함수형 프로그래밍에서는 `순수함수` `불변성` `선언적 패턴` 이라는 3가지 요소가 굉장히 중요합니다. 하지만 이는 개발자 입장에서 혼란이 올 수 있으니 아래와 같이 얘기해보겠습니다.

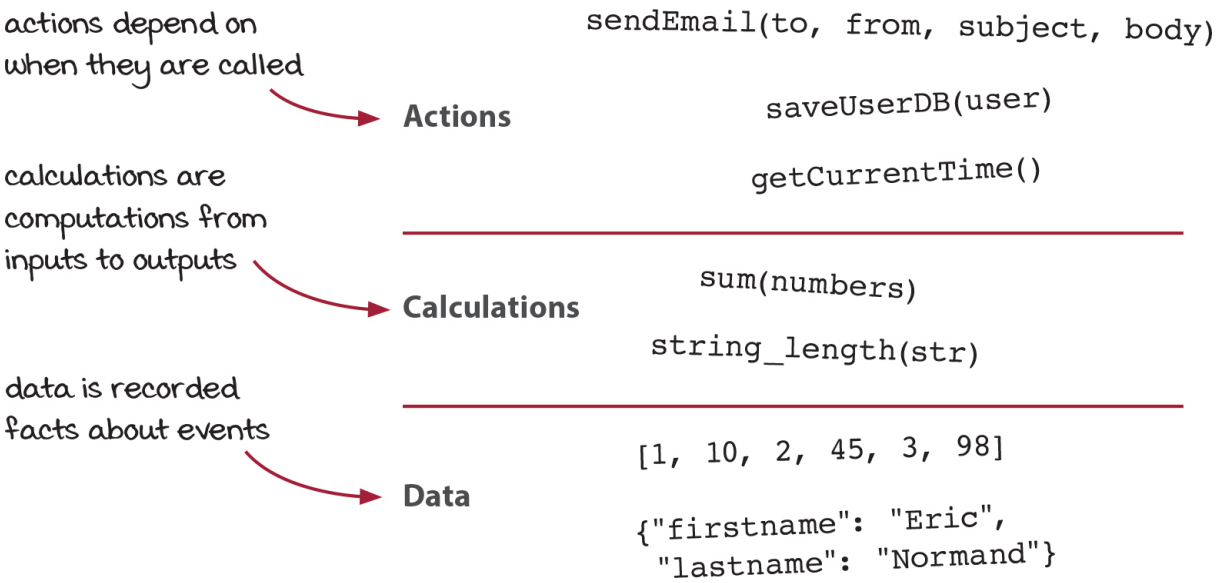
- 순수함수 : 코드를 액션과 계산, 데이터로 분리하자. 특히 액션에서 계산을 분리하는 코드를 작성하자.
- 불변성 : 카피온라이트와 방어적 복사를 이용해 불변성을 유지하자
- 선언적 패턴 : 계층형 설계와 추상화 벽을 이용해 무엇과 어떻게를 구분해 좋은 설계를 유지하자.

액션, 계산, 데이터

함수형 프로그래밍에서는 부수효과와 가변된 상태를 멀리하고 순수함수로 프로그래밍을 하자고 하지만 **대부분의 프로그램의 목적은 부수효과**에 있습니다. 서버에서 데이터를 조회하고 화면을 변경하고 로그를 남기고 등의 행위를 하지 않는 프로그램은 의미가 없습니다. 순수함수만 가지고는 우리가 만들고자 하는 응용 프로그램이 되지 않습니다. 순수함수에서는 이들을 사용하지 말자고 하거든요.

그렇기에 잘못된 오해를 불러 일으키는 순수함수 / 부수효과 이런 용어 대신 **액션, 계산, 데이터 이 3가지로 나눠 구분하여 프로그래밍 하는 것**이라고 다시 정의를 하겠습니다. 그리고 프

로그램을 다음과 같은 액션, 계산, 데이터로 구분을 해보도록 하겠습니다.



각각의 핵심적인 특성은 다음과 같습니다.

- 액션 : 호출 시점과 실행 함수에 의존합니다.
- 계산 : 입력과 출력으로 이루어져 있습니다.
- 데이터 : 이벤트에 대한 사실입니다.

프론트엔드 관점에서 정리해보는 액션, 계산, 데이터

간단하게 카운터를 만들어보겠습니다.

```
<button id="button">0</button>

<script>
document.getElementById("button").onclick = function() {
    button.innerText++
```

```
}
</script>
```

여기서 액션, 계산 데이터를 봅시다.

액션 : 실행 시점이나 횟수에 의존합니다. 즉, **사용자가 버튼 클릭시 숫자가 1 커지는 것**은 언제하느냐에 따라서 또 여러 번 할 수록 다른 결과가 만들어지기에 액션입니다.

계산 : 입력값을 통해 출력값을 만들어 내는 것입니다. 이 프로그램에서는 클릭을 하면 기존에 있는 **숫자에 1을 더해 새로운 숫자를 만들어내는 계산**을 하고 있습니다.

데이터 : 이벤트에 대한 사실입니다. 여기선 **숫자**가 되겠죠. 데이터는 화면에 보여줄 수 있습니다.

프로그램에서 각자의 관계는 **액션이 발생하면 미리 정의된 계산에 의해 데이터가 바뀌게 됩니다.**

함수형 프로그래밍의 관점으로 다시 코드를 작성해보겠습니다.

```
function App() {
    // 데이터
  const [count, setCount] = useState(0)

    // 계산
    const increase = (value) => value + 1

    // 액션
    const onClick = () => setCount(increase(count))

    // 선언적 패턴
    return <button onClick={onClick}>{count}</button>
}
```

함수형 프로그래밍이라더니... 많이 익숙합니다. 사실 함수형 프로그래밍은 현대 UI 프로그래밍에 잘 맞으며 알게 모르게 쓰게 되어 있습니다. 그리고 여기서 주목할 점은 바로 **increase 를 별도의 함수로 만들었다는 것입니다.**

액션에서 계산을 분리해내자!

```
// 잘못 분리된 계산함수
const increase = () => {
    ...
    setState(count + 5)
}

const onClick = () => {
    ...
    increase()
}
```

계산은 반드시 입출력으로 이루어져야 하며 같은 입력에 대해서는 항상 같은 출력값을 내놓아야 합니다. 계산은 여러 번 실행이 되어도 외부 세계에 영향을 주지 않아야 합니다. 위 예시에서 쓰인 increase 함수는 실행 횟수에 따라 시점에 따라 달라지므로 계산처럼 보이지만 계산이 아니라 액션입니다.

> **함수형 프로그래밍의 핵심은 액션과 계산을 확실히 분리해서 액션을 최소화하고 계산함수를 많이 만들어서 관리를 하는 것을 목표로 합니다.**

액션 함수를 계산 함수로 변경하는 방법

함수는 언제나 입출력이 존재합니다. 자바스크립트 코드는 매우 자유롭기 때문에 명시적인 인자와 리턴값 외에도 암묵적인 입출력이 존재할 수 있습니다.

```
const increase = () => {
  ...
  // count는 함수 외부에서 왔으므로 암묵적 입력입니다.
  const result = count + 1 // 1이라는 값은 변경할 수 없는 암묵적 입
  setState(result) // 함수 외부에 있는 값을 변경하는 암묵적 출력입니다
  ...

  // result는 명시적 출력입니다.
  return result
}

const action = () => {
  ...
  increase()
}
```

위와 같이 **하나의 함수에 암묵적 입출력과 명시적 입출력이 섞여 있다면 계산이 아니라 액션입니다. 액션은 실행 횟수와 시점에 의존하므로 테스트하기 어렵습니다.** 액션과 계산이 섞여 있는 함수라면 계산 부분을 분리해야 합니다.

우선 암묵적인 입출력을 제거합니다.

```
const increase = (count, offset) => {
    const result = count + offset // count와 offset을 명시적 입
    // setState와 같은 명시적 출력은 사용하지 않습니다.
    return result
}

// 암묵적인 입출력은 별도의 action에 모아줍니다.
const action = () => {
    setState(increase(count, 1)) // 외부에서 필요한 모든 입력을 넣어
}
```

이렇게 만들어진 계산은 이제 독립적입니다. 언제든 재사용이 가능하며 테스트하기에도 용이합니다. 계산은 조립을 해도 언제나 같은 결과를 만들어내기 때문에 조합에 의한 폭발적인 테스트 시나리오를 만들지 않도록 도와줍니다.

계산은 명시적인 입출력만을 가지며 어떠한 부수효과도 만들어내지 않습니다. 같은 입력에 대해서는 언제나 같은 결과만을 만들어내야 합니다.

> **그리고 이 계산이 바로 순수함수가 됩니다.**

액션 - 계산 - 데이터 정리

프로그램은 곧 데이터의 변화이며 즉, 액션에 의해 변하는 데이터입니다. **데이터가 변하는 방법은 따로 계산으로 독립적으로 만들어두어 액션과 계산과 데이터를 함수를 통해 연결하여 작성하는 개념이 바로 함수형 프로그래밍**인 것입니다.

불변성 - copy on right / 방어적 복사

계산은 여러번 실행해도 외부의 영향에 값을 변경하지 않아야 합니다. 하지만 함수에서 숫자나 문자열이 아닌 객체나 배열을 사용한다면 자바스크립트는 기본적으로 `pass by reference` 방식을 사용하기 때문에 언제든 외부에서 값이 수정되거나 함수 내부에서 외부에 영향을 미칠 수 있습니다. 그리고 그렇지 않기 위해 객체나 배열을 `pass by value` 의 형태로 변경하는 방식을 알아야 합니다.

Copy on Write

앞서 만든 카운터 프로그램에서 요구사항을 살짝 변경해보겠습니다. 버튼을 클릭할때마다 숫자를 더하는게 아니라 배열을 만들어두고 배열의 값을 하나씩 늘려가는 형태로 만들어간다고 생각해보겠습니다.

```
// 1씩 증가하는 계산 함수
const increase = (value) => value + 1

// 1씩 증가하는 값을 배열에 넣는 함수 ex) [1] -> [1,2] -> [1,2,3]
const increase = (arr) => {
  const value = arr[arr.length - 1]
  arr.push(value + 1)
  return arr
}
```

이렇게 만들어진 코드는 계산이 아니라 액션이 됩니다. 자바스크립트에서는 Object나 Array와 같이 덩치가 큰 값을 다룰 때는 pass by reference 라는 방식을 통해 원본을 그대로 전달하고, 원본을 직접 수정할 수 있도록 하는 방식을 통해 효율적으로 값을 조작할 수 있도록 합니다.

하지만 함수형 프로그래밍에서는 조심해야 합니다. 계산(순수함수)은 함수의 동작이 외부 세계에 영향을 끼치지 않아야 하고 실행함수와 시점과는 무관해야 한다고 정의하였기에 함수에서 원본값을 직접 수정하게 된다면 메모리상으로는 효율적이겠지만 외부 세계에 영향을 끼치지 말아야 한다는 제약 조건을 깨게 됩니다.

그렇다면 어떻게 해야 할까요? `pass by value` 방식과 같이 값을 복사해서 수정을 한다면 원본을 건드리지 않고도 원하는 계산이 가능합니다.

```
const increase = (arr) => {
  arr = arr.slice() // array를 조작하기 전에 복사해서 사용한다.
  const value = arr[arr.length - 1]
  arr.push(value + 1)
  return arr
}
```

```
const increase = (arr) => [...arr, arr[arr.length - 1]]
```

이와 같이 값을 조회하고 변경하여 출력값을 만들어야 할 때는 원본의 값을 복사해서 수정하면 되는 것이고 이런 방식을 **Copy on Write 혹은 얕은 복사**라고 합니다. 우리는 Copy on Write 방식을 통해 액션을 계산으로 만들 수 있습니다.

Object 또한 마찬가지 방법으로 변경된 값을 원본 수정없이 출력 가능합니다.

```
const setObjectName = (obj, value) => {
  return {
    ...obj
    name: value
  }
}

const setObjectName = (obj, name) => ({...obj, name})
```

방어적 복사

위에서 배운 Copy on Write 방식으로 액션을 계산으로 변경할 수도 있겠지만 만약 해당 액션이 우리가 수정할 수 없는 라이브러리라면 어떻게 할가요? 계산 함수에 액션이 하나라도 존재한다면 그 함수는 액션입니다. 그리고 그렇게 만들어진 액션들은 코드 전체에 퍼져나가 코드를 어렵게 만듭니다.

```
import someActionLibrary from 'lib'

const someCalculation = (obj, value) => {
    someActionLibrary(obj, value) // obj 값을 변경해서
```

```
    return obj // 출력하면 이 함수는 계산일까요?
}
```

이런 특수한 경우 혹은 mutation 함수를 이용해야만 하는 경우에는 **방어적 복사**라는 기법을 이용할 수 있습니다.

```
const someCalculation = (obj, value) => {
    const clone = structureClone(obj)
    someActionLibrary(clone, value) // clone을 변경해도 원본은 변
    return clone
}
```

이렇게 중첩된 모든 구조를 복사하는 방식을 **깊은 복사**라고 합니다. 자바스크립트에서는 원래 해당 기능이 없었지만 structuredClone()이라는 api가 Native 기능이 되어 해당 API를 사용하면 됩니다.

> 결론은... **카피온라이트**와 **방어적 복사**를 이용해 불변성을 유지하자!!
> 는 것입니다.

선언적 패턴과 계층형 구조

위와 같이 각 영역을 함수로 구분 짓다보면 자연스럽게 좋은 구조를 만들어 낼 수가 있습니다. 그리고 이렇게 코드를 작게 분리할 때 좋은 점은 아래와 같습니다.

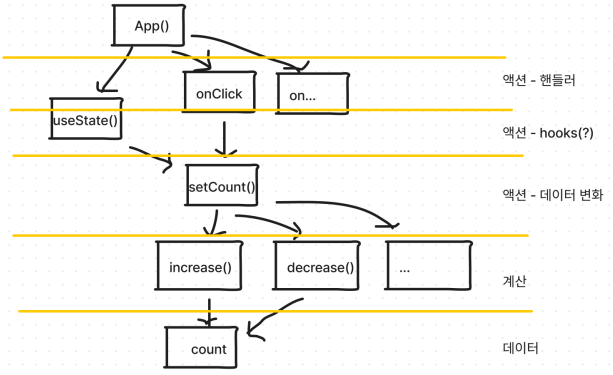
1. 재사용하기 쉽다.
2. 유지보수가 쉽다.
3. 테스트가 쉽다.

그리고 우리는 이렇게 분리한 코드를 조합하는 과정에서 자연스레 함수간 계층이 생긴다는 것을 알 수 있게 됩니다.

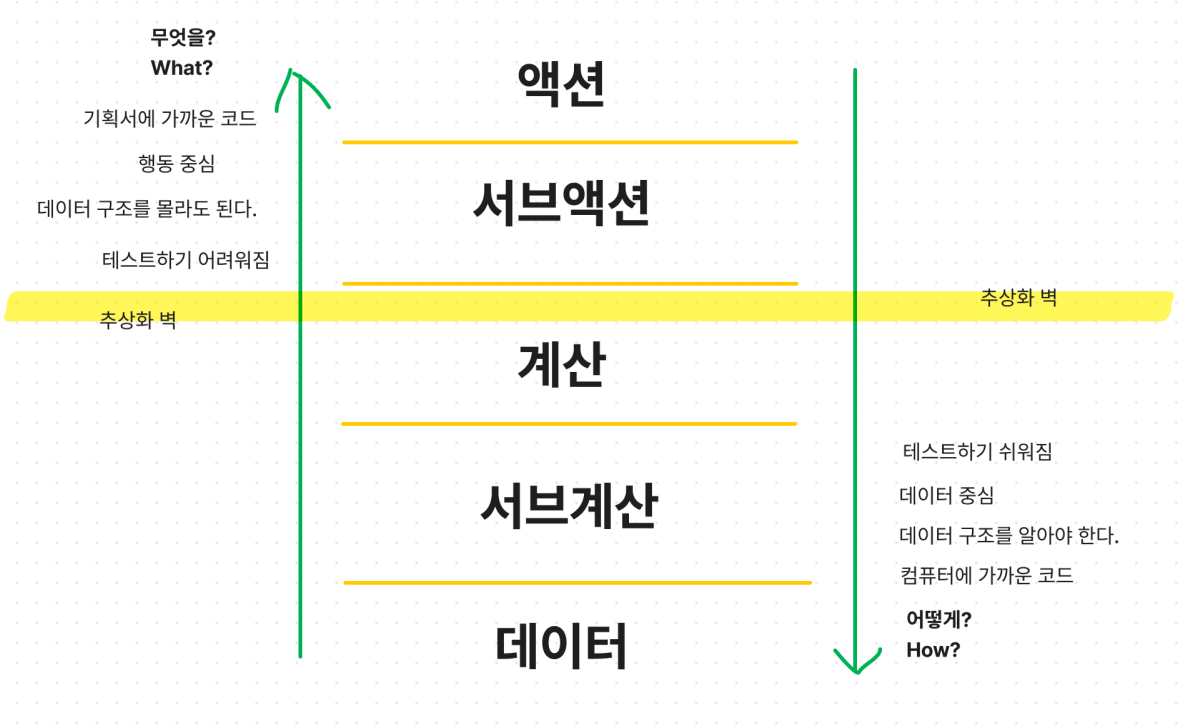
계층적 구조

아래는 처음에 예를 들었던 코드와 그 함수들의 계층입니다.

```jsx
// 함수형 프로그래밍 관점에서 분리해보자.
function App() {

  // 데이터
  const [count, setCount] = useState(0)

  // 계산
  const increase = (value) => value + 1

  // 액션
  const onClick = () => setCount(increase(count))

  // 선언적 패턴
  return <button onClick={onClick}>{count}</button>
}
```



이렇게 액션, 계산, 데이터로 코드를 구분하고 계층을 만들고, 계층을 넘나들지 않는 코드를 짜다보면 자연스럽게 좋은 코드의 구조를 만들 수 있고, **계산의 비중을 높여가고 계층을 넘나들지 않도록 코드를 쪼개다보면 좋은 설계와 리팩토링에 대한 좋은 근거가 될 수 있습니다.**



액션으로 갈수록 코드의 형태는 무엇을 하는 것인지 행동을 기반한 기획서에 가까운 코드가 만들어지며 데이터 구조를 몰라도 되는 형태의 코드를 작성하게 됩니다. 반면 계산과 데이터에 가까워질수록 데이터 중심적인 코드를 작성하게 되고 상대적으로 재사용성이 높고 테스트를 하기 좋은 코드 형태를 갖추게 됩니다.

이런식으로 계층을 나누고 각 계층을 침범하지 않도록 코드를 작성하다보면 자연스레 `추상화 벽` 이라는 것이 만들어지게 되면서 벽 상단으로의 코드 변화가 하단에 영향을 미치지 않고 하위의 코드 변화가 상위에 영향을 주지 않도록 할 수 있습니다.

이렇게 계층이 견고해지는 구조로 작성을 하게 되면 유연하면서도 변화에 국지적인 좋은 설계를 가지게 됩니다. 상위에는 **기획서에 가까운 무엇을 해야할지만 기술을 하는 선언적 패턴**으로 코드를 가져갈 수 있고 하위에는 테스트가 쉬운 코드 조각들로 구성이 되는 좋은 설계 방향의 코드가 만들어지게 됩니다.

> 선언적 패턴 : 계층형 설계와 추상화 벽을 이용해 무엇과 어떻게를 구분하여 좋은 설계를 유지해야 합니다.

근데 왜 JavaScript에서 잘 안쓰나요?

자바스크립트는 멀티 패러다임 언어이지만 순수하게 함수형 프로그래밍 언어는 아니기 때문에 copy 등의 문법을 지원하지는 않습니다. 그래서 아래와 같은 복잡한 구조가 나올 수 밖에 없죠

```
// 액션
const someMutationAction = (obj) => {
    obj.foo.bar.baz = 200
}

// 계산
const someCalculation = (obj, value = 200) => {
    return {
        ...obj,
        foo: {
            ...obj.foo,
            bar: {
                ...obj.foo.bar,
                baz: value
            }
        }
    }
}
```

이런식으로 복잡한 코드를 쓰는 대신 기타 라이브러리의 도움을 받을 수도 있습니다. 예를 들면 immutable.js 같은 것이죠. 그리고 이러한 라이브러리들은 엄청나게 많습니다.

물론 처음에는 이런 형식들을 한 곳에 모으는 라이브러리들도 존재했지만 점점 사용해야 할 함수들이 많아지면서 오히려 더 어렵다는 의견이 생기기 시작했고 결국 하나의 목적을 위해서 함수형 개념만 접목시키는 Redux와 같은 라이브러리들이 나오게 되었습니다.

즉, 현재 자바스크립트 문법의 한계로 함수형 프로그래밍을 하는데 한계가 있으며 그에 대한 대안으로 여러 라이브러리들을 사용할 수 밖에 없게 되었다는 것입니다.

물론 함수형 프로그래밍이 나쁜게 아닙니다. 위에 글들을 보면서 느끼셨겠지만 정말 엄청날 정도로 좋은 개념입니다. 그렇기 때문에 자바스크립트에서 함수형 패러다임을 쓰지 말자는 게 아니라 함수형 패러다임의 사고를 가지고 코드를 작성하다 보면 분명 좋은 코드, 좋은 설계가 될 것이라는 것입니다.