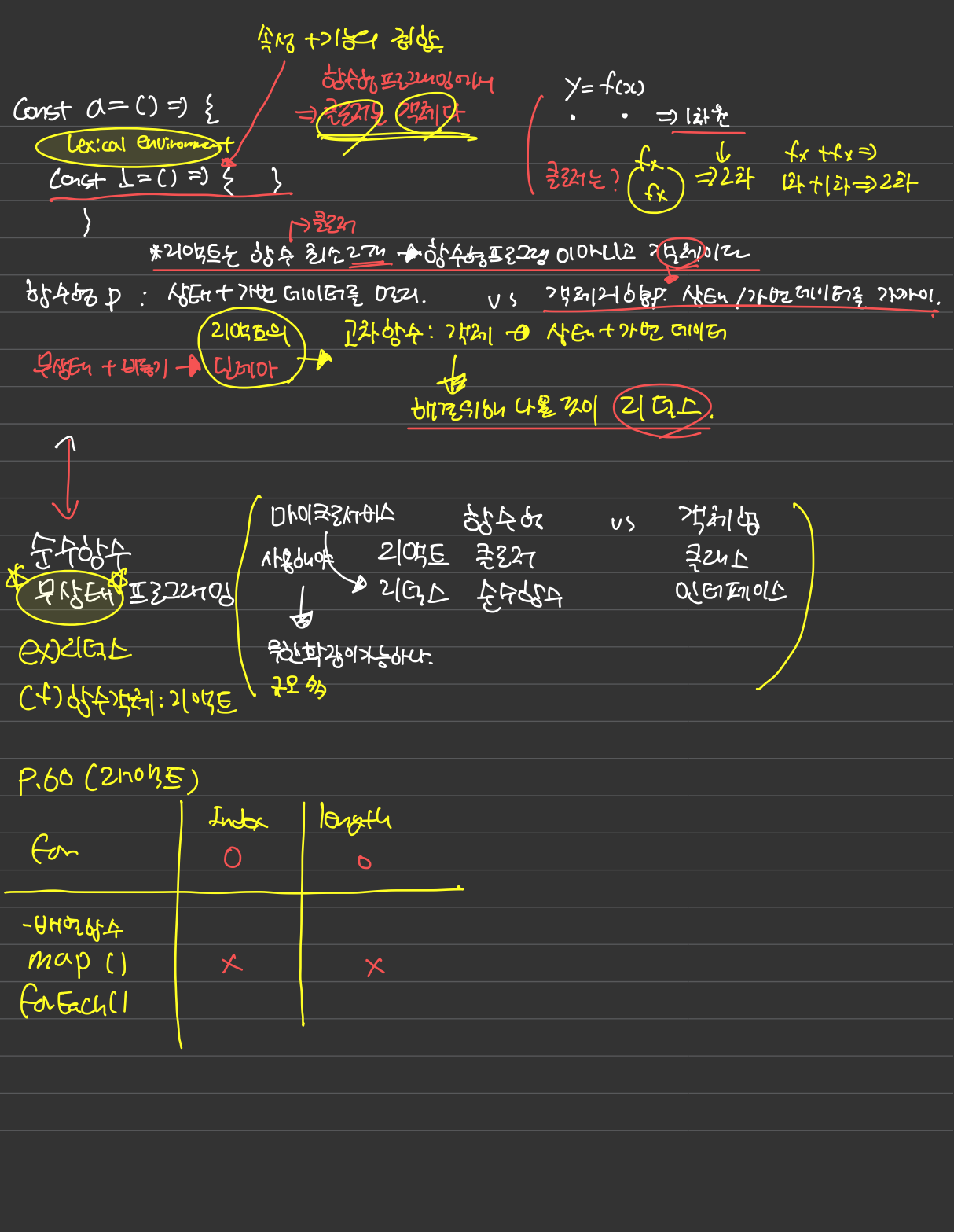


P.60 (리액트)

	Index	length
for	0	0
-내부함수		
map()	X	X
forEach()		

원하는 만큼 함수적이 되어라.

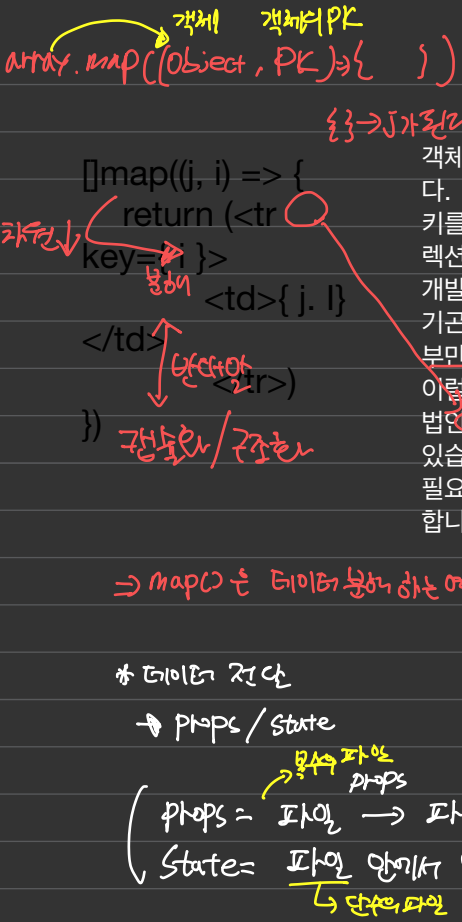
함수형 프로그래밍(functional programming)은 본질적으로 프로그래밍을 수학으로 간주하는 것이다. 함수형 프로그래밍의 많은 개념들은 알론조 처치(Alonzo Church)의 람다 대수(Lambda Calculus)가 기초가 되었는데 이는 현대 컴퓨터의 개념과 조금이라도 흡사했던 그 어떤 것보다 앞섰다. 하지만 컴퓨터의 실제 역사는 다르게 흘러간다. 컴퓨터가 처음 발명될 무렵, 존 폰 노이만(John von Neumann)의 아이디어는 처치(Church)보다 더욱 중요하게 여겨졌고 그에따라 초기 컴퓨터들의 설계에서부터 현재까지 많은 영향을 미쳤다. 폰 노이만의 생각은 "프로그램이란 명령들을 실행하기 위해 설계된 기계장치에서 실행되는 명령들의 목록"이라고 확고했다.

그렇다면 함수형 프로그래밍을 "수학으로 간주되는 프로그래밍"이라고 여기는 것에 대한 의미는 무엇일까? 폰 노이만은 수학자였고, 모든 프로그래밍의 근원지는 수학에서 찾을 수 있다. 함수형 프로그래밍이 수학적이라는 것은 무슨 의미이고 어떠한 수학과 관련되어 있다는 것일까?

사실 특정한 수학 분야와 관련되어 있다는 의미는 아니다. 람다 대수가 집합론(set theory)이나 논리수학, 범주론 등의 특정한 분야의 수학과 밀접한 관련이 있는 것은 사실이다. 하지만 프로그래밍의 기초가 되는 초등학교 수학의 수준부터 접근해보자. 다음은 자주 봤던 코드일 것이다.

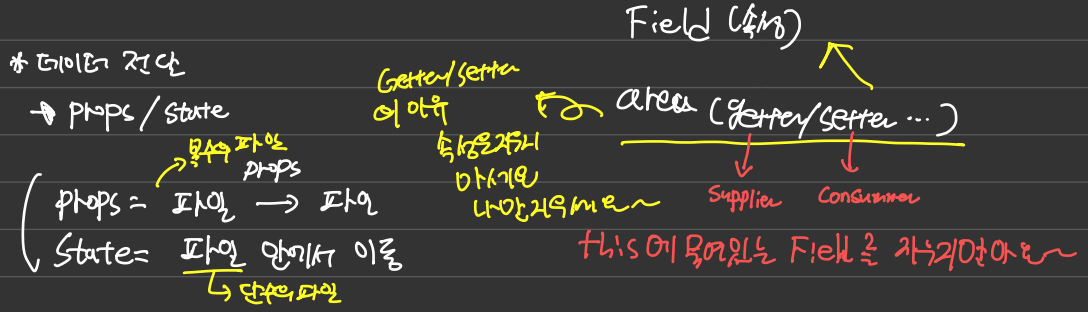
Streams는 종종 함수형 언어와 연관되어 있다. 기본적으로 느릿느릿 평가되는 긴(거의 무한한) 리스트들이다. 즉, 문자열의 요소가 필요한 경우에만 평가된다는 의미다. Maps는 리스트의 모든 요소에 함수를 적용하여 (이러한 목적을 위해)전문화된 리스트인 스트림을 포함한 새로운 리스트를 리턴한다. 이는 엄청나게 유용한 기능이다. 반복문(loop)을 작성할 필요 없이, 그리고 심지어 데이터를 얼마나 가지고 있는지조차 모르는 상태에서 반복문을 작성하는 최고의 방법이다. 또한 스트림 요소를 아웃풋에 패스할지 여부를 선택하는 "filter"를 만들고 maps와 filter를 함께 연결할 수도 있다. Unix pipeline이 생각난다면 맞다. Streams, maps, filters와 이 모든 걸 함께 연결하는 것은 함수형 언어와 Unix shell이 연관되어 있듯이 마찬가지로 연관이 있다.

반복문을 피하는 또 다른 방법은 파이썬의 기능인 "comprehensions(컴프리헨션)"를 사용하는 것이다. List comprehension(리스트 컴프리헨션)과 익숙해지기 쉽다. 컴팩트하고 하나하나의 오류를 제거하며 탄력적이다. 컴프리헨션이 전통적인 반복문에서는 컴팩트한 표기법처럼 보일 수 있지만, 사실 집합론에서 비롯되었으며 가장 가까운 컴퓨터적 "친척"은 함수형 프로그래밍이 아닌 관계형 데이터베이스(relational database)다. 다음은 리스트의 모든 요소에 함수를 적용하는 컴프리헨션이다.

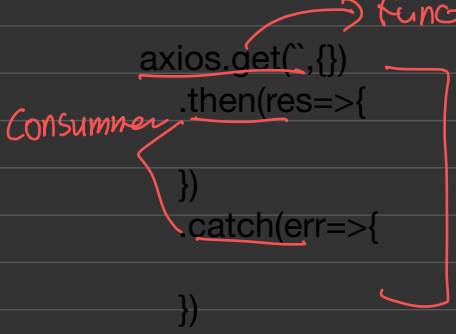


객체와 배열은 자바스크립트에서 가장 많이 쓰이는 자료 구조입니다.
키를 가진 데이터 여러 개를 하나의 엔티티에 저장할 땐 객체를, 컬렉션에 데이터를 순서대로 저장할 땐 배열을 사용하죠.
개발을 하다 보면 함수에 객체나 배열을 전달해야 하는 경우가 생기곤 합니다. 가끔은 객체나 배열에 저장된 데이터 전체가 아닌 일부만 필요한 경우가 생기기도 하죠.
이럴 때 객체나 배열을 변수로 '분해'할 수 있게 해주는 특별한 문법인 구조 분해 할당(destructuring assignment) 을 사용할 수 있습니다. 이 외에도 함수의 매개변수가 많거나 매개변수 기본값이 필요한 경우 등에서 구조 분해(destructuring)는 그 진가를 발휘합니다.

⇒ map()은 데이터 분해 하는 역할



props 변경X → 변경위해선 state에 담아줘야함
state 자식간 이름 X → 이름위해선 props에 담아줘야함



```
const SeoulCCTV = () => {

  const [items, setItems] = useState([])

  const list = () => {
    axios.get('/data/SeoulFloatingPopulation.json')
    .then(res=>{
      setItems(res.data.DATA)
    })
    .catch(err=>{
        alert(err)
    })
  }

  return ( <>)
```

function

대기

상황을 묶어줘야 한다.

```
useEffect(() => {
    fetchList();
}, []);
```

함수를 파라미터 / 리턴 X

↓

컨슈머로,

함수안에 함수

클로저

예) useEffect() → 호출
useEffect() {} → 선언

· 함수 사용하는 순서

return 하고 / const [ ] = useState 작동
그리고 대기 상태

바디가 있으면 "선언" / 없으면 "호출"

⇒ 움직여줘야함 ⇒ useEffect
(실행시켜줘야한다) 함수

⇒ useEffect() 로 시작

↓

컨슈머

Redux toolkit

{value:0} JSON ↑ initial state — {()=>{}} ↑ reducer → 3개가 필요하다. — Key값 ↑ name, string

Creating a slice requires a string name to identify the slice, an initial state value, and one or more reducer functions to define how the state can be updated. Once a slice is created, we can export the generated Redux action creators and the reducer function for the whole slice.
Redux requires that we write all state updates immutably, by making copies of data and updating the copies. However, Redux Toolkit's `createSlice` and `createReducer` APIs use Immer inside to allow us to write "mutating" update logic that becomes correct immutable updates.

Localstorage.setItem("k", "j") → 단 하나 전역이다 Static 자바
sessionstorage

⇒ 저장공간↓ → Store를 만들어야한다
→ only one!

구조.

```
import { createSlice } from '@reduxjs/
toolkit';

const CounterSlice = createSlice({})

export const {} = CounterSlice.actions;
export default CounterSlice.reducer;
```