Functional Programming

Aspect Oriented Programming (AOP)

AOP ဆိုတာ ဘာလဲဆိုရင် concern(တခုချင်းစီအလိုက် လုပ်ဆောင်ချက်)တွေခွဲထားပီး runtimeမှာမှ အဲ့concernတွေကို ပြန်ပေါင်းပီး လုပ်ဆောင်တဲ့ကောင်မျိုးကို ခေါ်တယ်။ ဥပမာနေနဲ့ဆို security နဲ့ဆိုင်တဲ့ ကုဒ်ကသပ်သပ် loggingနဲ့ဆိုင်တဲ့ ကုဒ်ကသပ်သပ် အဲ့တာကိုမှ securityကုဒ်ထဲကကောင်ကို log ထုတ်ချင်ရင် တခုချင်းလိုက်logမလုပ်နေပဲ runtimeမှာမှ 2ခုကို mergeပီး log လုပ်တာမျိုးကိုခေါ်တယ်၊ Example language နေနဲ့ကတော့ springမှာသုံးတဲ့ aspectJက AOPကိုသုံးထားတာဖြစ်တယ် ဥပမာနေနဲ့ကတော့

```
function logging(obj) {
        for (let prop in obj) {
          if (typeof obj[prop] === "function") {
            console.log("Prop ", prop);
            let oldFunc = obj[prop];
            obj[prop] = function () {
              console.log("Old Function ", prop, " was called");
              oldFunc();
            };
          }
        }
      }

      let obj = {
        hello() {
          console.log("hello");
        },
        greeting() {
```

```
      console.log("Nice To Meet You");
    },
  };

  logging(obj);
  obj.hello();
  obj.greeting();
```

ဒီကုဒ်မှာဆိုရင် logကို တခုချင်းဆီ ထုတ်မနေပဲ concernတခုခွဲထုတ်ပီး ပီးတော့မှ log ထုတ်ချင်တယ်ဆိုရင် runtime မှာမှ loggingကို bindပေးလိုက်တာမျိုးကိုခေါ်တယ်။

Reactive Programming

Reactive Programming ဆိုတာက rxjs လို reactive libraryကိုသုံးထားတဲ့ကောင်မျိုးဖြစ်တယ်။ သူကဘယ်လိုလဲဆိုရင် value တခုရှိတယ်ဆိုပါဆို့ အဲ့ကောင်ကိုသုံးထားတဲ့နေရာက ၅ခုရှိတယ်၊ အဲ့အချိန်မှာ တနေရာရာကသာ value တန်ဖိုးကိုပြောင်းလိုက်ရင် ကျန်တဲ့ ၄နေရာမှာပါချိန်းသွားတာမျိုးကို ပြောတာမြင်သာအောင်ပြောရရင် excelတွေမှာ column valueတွေကို sumလုပ်တဲ့ကောင်လိုမျိုးပေါ့ row 1ခုက တန်ဖိုးပြောင်းသွားတာနဲ့ အောက်က စုစုပေါင်းတန်ဖိုးမှာ သွားပြောင်းသွားတာမျိုးဒါမျိုးကိုခေါ်တာ၊

Event Driven

Event Drivenကကျတော့ Vbတို့မှာ သုံးထားတဲ့ကောင်မျိုးကိုပြောတာ သူက OO concept

တွေကို မစဉ်းစားပဲ ဘယ်button ကို clickလိုက်ရင် ဘာလုပ်မယ်ဆိုပီး သွားတဲ့ကောင်မျိုး။

Declarative Programming

Declarative programmingရဲ့ main conceptက သူက ဘာလိုချင်တာလို့ပဲပြောတာ ဘယ်လိုလုပ်ပါလို့မပြောဘူး Example နေနဲ့ဆိုရင် SQLလိုကောင်မျိုးပေါ့ နောက်ဆုံးသွင်းလိုက်တဲ့ ဒေတာ၃ခုလိုချင်တယ်လို့ပဲပြောတယ် မင်းဘယ်လိုရေးပါလို့မပြောဘူ အဓိကက result ရဖို့ပဲ imperativeနဲ့ကွာတာက declarativeက what to do ကို အဓိကထားပီး imperativeက how to do ကို လုပ်ဆောင်တာဖြစ်တယ်။ declarative programmingက instruction onlyဖြစ်တဲ့တွက်ကြောင့် သူမှာဆိုရင် conditional operator if, thenတို့မပါဘူး။ ပီးတော့ သူက end resultမှာပဲ focus ထားတယ်။ ဥပမာနေနဲ့ဆို အမေက ကြက်ဥဝယ်ခဲ့လို့ပြောတယ် ဘယ်ကဝယ်ပါလို့မပြောဘူး အဲ့တော့ ကိုက deliveryနဲ့ မှာလဲရတယ် ဈေးထဲသွားဝယ်လဲရတယ် supermarket သွားဝယ်လဲရတယ် အဓိကက ကြက်ဥရဖို့ပဲ။ သူ့ကို အဓိကအားဖြင့် databaseတို့ configuration management software တွေမှာတွေ့ရတယ်၊ imperative ကကျတော့ အဲ့လိုမဟုတ်ဘူး ဘယ်လိုလုပ်မယ်ဆိုတဲ့ instructionတွေပေးရတယ် ဥပမာဆိုရင် ကားငှားလိုက်တယ် သွားမယ့်လမ်းကြောင်းကို ကားသမားကမသိဘူး အဲ့တော့ လမ်းကိုဘယ်တိုင်းမောင်း ဘယ်နားရောက်ရင် ဘယ်ကိုကွေ့ အဲ့လိုမျိုးနဲ့ လိုချင်တဲ့resultကို instructionတွေပေးပီး လုပ်တာမျိုးကိုခေါ်တယ်၊

Declarative UI

Declarative UIဆိုတာက declarative programmingနဲ့မတူပါဘူး။ သူကဘာလဲဆိုရင် UI createလုပ်ရင် ကုဒ်တွေကို UIတခုချင်းစာအတွက် လိုက်ရေးနေတာမျိုးမဟုတ်ပဲ stateကနေ create လုပ်ထားတာမျိုးကိုခေါ်တယ်။ stateဆိုတာက user input or business logicတွေကို holdထားတဲ့ variable or objectတခုခု ကိုပြောတာ။ အဲ့တော့ stateတာတခုခု ပြောင်းသွားမယ်ဆိုရင် UIကို ပြန်ပီး လိုက်ရေးနေစရာမလိုပဲ သူ့ဒေတာနဲ့လိုက်တဲ့ UIကို reflect ဖြစ်သွားတာမျိုးကိုခေါ်တယ်။ သူ့ကို react, flutter, etcစတဲ့ကောင်တွေမှာ မြင်နိုင်တယ်။ imperative UI ကျတော့အဲ့လိုမဟုတ်ဘူး ဘာလုပ်ချင်တယ် ဘာလုပ်ရင်ဘာဖြစ်မယ်ဆိုတာကို တခုချင်းစီ ပြန်လိုက်ရေးထားတာမျိုးကိုခေါ်တယ် ဥပမာနေနဲ့ဆိုရင် HTMLမှာ UI event 1ခုခု ဖြစ်ရင် ဘာလုပ်မယ်ဆိုပီး eventတွေကို တခုချင်းပြန်လိုက်ရေးထားတာမျိုးကိုခေါ်တယ်။

Declarative UI Example

```
const [scene, setScene] = useState('button')

     if(scene === 'button' || scene === 'question'){
       return(
           <Button
               blue = {scene === 'button'}
               pink = {scene === 'question' }
               onClick = {() => setScene('question')}>
               {scene === 'question' && 'Chaned Data'}
           </Button>
       );
```

Imperative UI Example

```
const container = document.querySelector('.container');
      const btn = document.querySelector('.btn');

      const lisener1 = () => {
        btn.style.backgroundColor = '#DB2777';
        btn.style.innerText = 'Are you sure?';
        btn.removeEventListener('click', listener1);
        btn.addEventListener('click', listener2);
      }

      const listener2 = () => {
        container.innerHTML = 'ChangeData';
      }

      btn.addEventListener('click', listener);
```

Functional Programming

Functional programming မှာဆိုရင် data immutation တွက် constကိုသုံးတယ်၊ immutable ဖြစ်တဲ့တွက်ကြောင့် ဒေတာတွေက processကို အပြိုင်မလုပ်ဘူး အဲ့တော့ဘာကောင်းသွားလဲဆိုရင် concurrency problemတွေမရှိဘူး concurrency problem တွေကို ဖြေရှင်းပီးသားဖြစ်သွားတယ်။ နောက်တခုက သူက functionတွေပဲသုံးတယ်။ ပီးတော့ side effectမပါတဲ့ pure function တွေကိုပဲသုံးတယ်။ side effect ဆိုတာဘာလဲဆိုရင် function မှာပါတဲ့ arguments တွေကိုပဲသုံးတယ် ကျန်တဲ့ I/Oတွေ အခြားဒေတာကိုပြောင်းလဲစေတယ့်ကောင်တွေကို မသုံးဘူး၊ Error Handling မှာဆိုရင်လဲ

imperativeမှာသုံးတဲ့ try catchလိုမျိုးကို မသုံးပဲ optional, monad လိုကောင်မျိုးကို သုံးတယ်။ FPက mostly data structure နေနဲ့ list, arrayလိုကောင်မျိုးကို filter, mapတို့လိုကောင်မျိုးနဲ့ တွဲပီး သုံးတယ်၊

Imperative နဲ့ functional programmingရဲ့မတူတဲ့အချက်နေနဲ့က

```
Imperative              Functional
----------------------------------------
variable(mutable)       const(immutable)
if, while, do, for      function
impure function         pure function
```

ဥပမာနေနဲ့ age 30ထက်ကြီးတယ် ပီးတော့Maleဆိုရင် Mrလို့ နာမည်ရှေ့မှာတက်ပီး Femaleဆိုရင် Mrs.လို့ prefix လုပ်တာကို declarative and functional styleနဲ့ ရေးကြည့်မယ်။

```
let humans = [
  {
    name: "Mg Mg",
    age: 31,
    gender: "Male",
  },
  {
    name: "Aung Aung",
    age: 21,
    gender: "Male",
  },
  {
    name: "Ma Ma",
    age: 18,
    gender: "FeMale",
```

```
  },
  {
    name: "Hla Hla",
    age: 34,
    gender: "FeMale",
  },
];

// Imperative Design
let result = [];
for (let human of humans) {
  if (human.age > 30) {
    let obj = Object.assign(human);
    if (obj.gender === "Male") {
      obj.name = `Mr ${obj.name}`;
    } else {
      obj.name = `Mrs ${obj.name}`;
    }
    result.push(obj);
  }
}
console.log("Declarative Result", result);

// Functional Design
let adult = (human) => human.age > 30;
let namePrefix = (human) =>
  human.gender === "Male"
    ? { ...human, name: `Mr ${human.name}` }
    : { ...human, name: `Mrs ${human.name}` };

let fpResult = humans.filter(adult).map(namePrefix);
console.log("FP Result ", fpResult);
```

What is Function in FP?

FPမှာ ရှိတဲ့ function တွေက Mathကနေလာတဲ့ function တေွဖြစ်တယ် သူက ပေးလိုက်တဲ့ ဒေတာပေါ်မူတည်ပီး ကျန်တဲ့ ဒေတာတွေမှာ changeမရှိပဲနဲ့ transformလုပ်ပီး result ကိုပြန်ပေးလိုက်တာမျိုး ဥပမာဆိုရင် valueကို double လုပ်တဲ့ functionဆိုပါဆို့ ဝင်လာတဲ့ value ကို ၂ဆပြောင်းပေးတယ် resultပြန်ပေးလိုက်တယ် ဒါပဲ ဝင်လာတဲ့ဒေတာကိုလဲ မချိန်းဘူး အဲ့ဒေတာမှာပဲ မူတည်ပီး process လုပ်လိုက်တယ်။

```
double(x) => x * 2;
```

imperative နဲ့မတူတာက imperative မှာကျတော့ state ရဲ့ တန်ဖိုးတွေကို ပြောင်းလဲစေတာမျိုး side effect တွေရှိတယ်။

Pure function

Functional Programming က function တွေသည် အနည်းဆုံး parameter 1ခုကို လက်ခံရမယ် မဟုတ်ရင် ကျန်တဲ့ ဒေတာကို ယူသုံးပီး side effect ဖြစ်တယ်ဆိုတာ သေချာနေပီ။ နောက်တခုက return ပြန်ရမယ်။ return မပြန်ထားတဲ့ function သည် side effect ဖြစ်နေတယ်ဆိုတာသေချာတယ်၊ Function in FP must have input and return value. နောက်တခုက ဝင်လာတဲ့ parameter ပေါ်မှာပဲ အလုပ်လုပ်ရမယ် အခြား stateတွေကို process လုပ်တာမျိုးမဖြစ်ရဘူး။ နောက်တခုက same input must be same output. ဒီ အချက်လက်တွေ ပြည့်စုံပီး side effectမရှိတဲ့ function ကို pure function လို့ခေါ်တယ်

Referential transparency

Referential transparency ဆိုတာ same input ပေးရင် same output ထွက်ရမယ်။ pure function တွေက RTဖြစ်တယ်၊ ဘာကောင်းလဲဆိုရင် reason ကောင်းတယ်။ ဘာလို့ဆိုတော့ပေးလိုက်တဲ့ ဒေတာနဲ့တင်လုပ်တဲ့အတွက် အဲ့တာကိုကြည့်လိုက်တာနဲ့ ဘာလုပ်ချင်တယ်ဆိုတာသိတယ်။ နောက်တခုက substitution ကောင်းတယ်။ ဘယ်လိုမျိုးလဲဆိုရင် add(2,3)ဆိုရင် 5ရမယ်ဆိုတာသေခြာတယ် အဲ့တော့ဒေတာတွေမှာ ဒီတန်ဖိုးပေးလိုက်ရင် calculation ထပ်လုပ်နေစရာမလိုပဲ သူ့ရဲ့ result valueကို replaceလုပ်ပီး သုံးနိုင်တယ်။ ဒါမျိုးကိုပြောတာ။ နောက်တခုက parallel(concurrency) problems မရှိဘူး။ ဘာလို့လဲဆိုရင် other resources တွေကို ယူမသုံးထားတဲ့တွက် side effectမရှိဘူး ဒီတော့ လုပ်နေတဲ့ အခြား Threadsတွေ, stateတွေကို ယူသုံးမိတယ် ဆိုတာမျိုးမရှိဘူး ဒီတော့ parallel problem ဆိုတာမရှိဘူး။

Pipe Lines and Composable

Pipe Line ဆိုတာက function 1ခုကထွက်တဲ့ outputကို အခြား function 1ခုမှာ input အနေနဲ့ သုံးတာမျိုးကိုပြောတာ၊ composableဆိုတာက multiple functionတွေကို တခုပီး တခုဆက်ခေါ်ပီးသုံးတာမျိုးကိုပြောတာ။

Higher Order Function(HOF)

HOF ဆိုတာ function 1ခုကို parameter အနေနဲ့သော်လည်းကောင်း output အနေနဲ့သော်လည်းကောင်း return ပြန်ပေးတဲ့ functionမျိုးကိုခေါ်တယ်။ Mathsမှာဆို differentiate နဲ့ သွားတူတယ်။ ဥပမာနေနဲ့ဆို

```
function getFunc(func) {
  return func();
}

function returnFunc() {
  return hello;
}

function hello() {
  console.log("Hello HOF");
}

getFunc(hello);
returnFunc()();
```

အပေါ်က getFunc(func) and returnFunc()လို function မျိုးတွေကိုခေါ်တယ်။ HOFကို support လုပ်ဖို့ဆိုရင် First class function လိုအပ်တယ်။ FPရဲ့ main concept က OOP လိုမျိုး state ကို မထိန်းပဲနဲ့ dataကို function တွေက တဆင့် pass လုပ်ပီး transform လုပ်တာဖြစ်တယ်။ တကယ်လို့ state ကို ထိမ်းချုပ်ရင်တော့ closure လိုကောင်မျိုးကို သုံးပီး လုပ်လို့ရတယ်။

Predicate Function

Predicate Function ဆိုတာ Boolean value return ပြန်ပေးတဲ့ function မျိုးကို ခေါ်တာ။

ဥပမာနေနဲ့ဆိုရင်

```
let isEven = (value) => value % 2 === 0;
```

အဲ့လို Boolean တန်ဖိုးပြန်ပေးတဲ့ကောင်ကို predicate function လို့ခေါ်တယ်

Unless Function

Unless Functionမှာဆိုရင် predicate functionရယ် invoke လုပ်မယ့် function ရယ်ပါတယ်။ သူ့main conceptက predicate function ပေါ်မူတည်ပီး function ကို invoke လုပ်မလုပ် ဆုံးဖြတ်တာမျိုးကိုခေါ်တယ် ဥပမာနေနဲ့ဆိုရင်

```
let isEven = (val) => val % 2 === 0;
let unless = (predicate, fn) => {
  !predicate && fn();
};

[1, 2, 3, 4, 5].forEach((element) => {
  unless(isEven(element), () => {
    console.log(element, " is an odd");
  });
});
```

Unary Function

Unary Functionဆိုတာက single argumentကိုပဲ လက်ခံတဲ့ functionမျိုးကို ခေါ်တယ်။

ဘယ်လိုနေရာမျိုးတွေမှာသုံးလဲဆို ဥပမာအနေနဲ့

```
console.log([1, 2, 3, 4].map((x) => x * x));
console.log([1, 2, 3, 4].map(parseInt)); // output [1,Nan,Nan,Nan]
// parseInt(1,0)    1
// parseInt(2,1)    Nan
// parseInt(3,2)    Nan
```

ဒီကောင်မှာဆိုရင် mapမှာက element, index, arrayဆိုပီးပါတယ် parseIntမှာကကျတော့ string, baseဆိုပီးပါတယ် အဲ့တော့ index and baseက map ဖြစ်သွားပီး excepted result ထွက်မလာဘူးဖြစ်သွားတယ် အဲ့လိုနေရာမှာ parseIntကို unary function မျိုး ပြောင်းပီး ဖြေရှင်းနိုင်တယ်။

```
const unary = (fn) => (fn.length === 1 ? fn : (arg) => fn(arg));

console.log([1, 2, 3, 4].map(unary(parseInt)));
```

အဲ့လိုမျိုး param ကို invert လုပ်ပီး ချိန်းလို့ရတယ်

Once Function

once function ဆိုတာ function callကို တခါပဲလုပ်စေချင်တဲ့အခါမျိုးမှာသုံးတယ်။ ဘယ်လိုရေးလဲဆိုတော့ call stateကို closure ကိုသုံးပီး handle လုပ်တယ်။ ဥပမာနေနဲ့ဆို

```
const once = (fn) => {
  let done = false;
  return function () {
    return done ? undefined : ((done = true), fn.apply(this,
arguments));
  };
};
```

```
function api() {
  console.log("api called");
}

let apiCall = once(api);
apiCall();
apiCall();
```

အဲ့လိုမျိုး တကြိမ်ပဲလုပ်စေချင်တဲ့ကောင်မျိုးမှာဆိုရင် သုံးတယ်။

Memoize Function

Memorize function ဆိုတာက process လုပ်ပီးသားဒေတာကို cache လုပ်ပီး သိမ်းထားတာမျိုးကိုပြောတာ။ same data process လုပ်ဖို့လာရင် နောက်တကြိမ်ထပ်မလုပ်ပဲ cached ထဲကနေ ပြန်ပေးလိုက်တာမျိုး။ ဥပမာနေနဲ့ဆိုရင်

```
let memorize = (fn) => {
  let lookupTable = {};

  return (arg) => lookupTable[arg] || (lookupTable[arg] = fn(arg));
};
let factorial = (n) => (n === 0 ? 1 : n * factorial(n - 1));

let fact = memorize(factorial);
fact(5);
fact(4);
```

အဲ့လိုမျိုး သုံးနိုင်ပါတယ်။ ရှာပီးသား value တွေကို cacheအနေနဲ့ ထည့်ထားပီး နောက်တကြိမ် ပြန်ခေါ်ရင် process timeမကုန်ပဲ ရှိပီးသားထဲကနေ ပြန်ပေးလိုက်တဲ့ ပုံစံမျိုးပေါ့။

Compose Function

Compositionဆိုတာက functionတွေကို ဆင့်ကာဆင့်ကာ သုံးတာကိုခေါ်တာ။ ဆိုလိုချင်တာက ပထမ function ရဲ့ outputကို နောက် function တခုရဲ့ input အနေနဲ့ သုံးတာကိုခေါ်တယ်။ ဥပမာနေနဲ့

```
let toUpper = (str) => str.toUpperCase();
let prependHi = (str) => "Hi " + str;
function compose(funcA, funcB) {
  return function (str) {
    return funcA(funcB(str));
  };
}

let composition = compose(toUpper, prependHi);
console.log(composition("Merry how are you?"));
```

ဒီမှာဆို compose functionမှာ functionတွေကိုလက်ခံပီး ပေးလိုက်တဲ့ functionတွေကို 1ခုနဲ့တခု invoke and combine လုပ်ပီး နောက်ဆုံးမှာ result ကိုပြန်ပေးလိုက်တာဖြစ်တယ်။ Compositionလုပ်မယ်ဆို function တွေက single input single output ဖြစ်ရမယ်။ အဲ့တော့ multiple input လုပ်မယ်ဆိုရင် ဘယ်လိုဖြစ်လာမလဲ။ အဲ့လိုမျိုးလုပ်မယ်ဆိုရင် currying ကိုသုံးရမယ်။

Identity Function

Identity function ဆိုတာ valueတခုပေးလိုက်ရင် ပေးလိုက်တဲ့ တန်ဖိုးကိုပဲ ပြန်ပီး return

ပြန်ပေးတဲ့ function မျိုးကိုခေါ်တယ်။ အဲ့တော့ ဒီကောင်က ဘယ်လိုနေရာမျိုးမှာ အသုံးဝင်လဲဆိုရင် functional programmingတွေမှာ function onlyပဲ သုံးတာဖြစ်တဲ့တွက်ကြောင့် primitive value တွေကို compose လုပ်မယ်ဆိုရင် အဲ့လိုနေရာမျိုးတွေမှာ ဒီကောင်ကို သုံးနိုင်တယ်။ code နေနဲ့ဆိုရင်

let identity = (x) => x;

ဒီကောင့်ကို unary identity functionလို့လဲခေါ်တယ်။

Currying

Curry ဆိုတာက multiple input လက်ခံတဲ့ functionတွေကို single input ပဲ လက်ခံအောင် ပြောင်းပေးတဲ့ကောင်မျိုးကို ခေါ်တယ်။ အဲ့အချိန်မှာ input 2ခုလိုတယ်ဆိုပါစို့ ဒါပေမဲ့ တခုပဲ ပေးလိုက်သေးတယ်ဆို မလုပ်သေးပဲ input တွေအကုန်စုံမှ processလုပ်တဲ့ကောင်မျိုး ကိုခေါ်တာ။ code example အနေနဲ့

```
function add(x, y) {
  return x + y;
}

function addCurry(x) {
  return function (y) {
    return x + y;
  };
}

let currying = addCurry(1);
console.log("add curring 1 and 2 :", currying(2));
```

```
console.log("add curring 1 and 2 and 3 :", currying(3));
```

အပေါ်က add functionလို input 2ခုကို တိုက်ရိုက် processလုပ်တာမဟုတ်ပဲနဲ့ input 1ခုချင်းစီကို ခွဲလက်ခံပီး input တွေစုံမှ လုပ်တာမျိုးကိုခေါ်တယ်။

Currying binary function syntax code example အနေနဲ့

```
function add(x, y) {
  return x + y;
}

const curry = (binaryFunc) => {
  return function (firstArg) {
    return function (secondArg) {
      return binaryFunc(firstArg, secondArg);
    };
  };
};

const curryAdd = curry(add);
console.log("Add of 3 and 4 is ", curryAdd(3)(4));
```

အခဲ့လိုမျိုးရေးလို့ရတယ်။ invoke function ကို ပေးလိုက်တဲ့ param တွေအကုန်စုံမှ apply လုပ်လိုက်တဲ့ ပုံစံမျိုးပေါ့။

Partial Application

Partial applicationဆိုတာက currying မှာ လိုအပ်တဲ့ functionတွေအကုန်လုံးကို မပေးသေးပဲနဲ့ တပိုင်းတစနဲ့ သုံးထာတာမျိုးကိုခေါ်တယ်။

Js functional programming တွက်ဆို ramda.jsကို သုံးလို့ရတယ်။

Pipe

Pipeကလည်း composition နဲ့တူတူပဲ ကွာတာတခုက compose က function invokeကို right to left လုပ်ပီး pipeက ကျတော့ left to right လုပ်တာပါပဲ။

code example :

const pipe = (funcA, funcB) => (x) => funB(funA(x));

Tap Function

Tap function ဆိုတာက ဘယ်လိုလဲဆိုတော့ input လာတဲ့ ကောင်ကို ပြန်ပီး return ပြန်ပေးတဲ့ ကောင်မျိုးကိုခေါ်တယ်။ ဘယ်လိုနေရာမှာ သုံးလဲဆိုရင် data ပေးလိုက်တာတွေကို debug လုပ်ချင်တဲ့နေရာတွေမှာသုံးတယ်။ For Code Example

```
const tap = (fn, a) => (fn(a), a);
tap(console.log, "hello guy");
```

Functional programming က ကောင်တွေကို ကိုယ်တိုင်လိုက်မရေးပဲ Ramda.js လိုကောင်မျိုးကို သုံးပီးတော့လည်းရေးနိုင်တယ်။

Point Free Style

Point free style ဆိုတာက function တွေကို အရင်ခေါ်ပီး parameter ကိုနောက်ဆုံးမှာမှ ပေးလိုက်တဲ့ ရေးသားတဲ့ style မျိုးကိုခေါ်တယ်။ sample code အနေနဲ့

```
let data = [1, 1, 1, 2, 2, 3, 5, 5, 5, 5];
let topKValue = R.pipe(
  R.groupBy(R.identity),
```

```
  R.map((ele) => [ele[0], ele.length]),
  R.values,
  R.sort((a, b) => b[1] - a[1]),
  R.map((ele) => ele[0]),
  R.take(2)
);

console.log("Get Top 2 values", topKValue(data));
```

Functor

Functor ဆိုတာ map ဆိုတဲ့ functionပါတဲ့ကောင်တွေကိုပြောတာ။ ဥပမာဆိုရင် js object, array လိုကောင်မျိုးပေါ့ အဲ့နေရာမှာ ဘာလို့map က လိုတာလဲလို့ပြောရင် imperativeမှာဆို side effect သုံးပီး ရှိနေတဲ့ stateကိုပြင်နိုင်တယ်။ ဒါပေမယ့် FPမှာကျတော့ side effect မဖြစ်အောင် pure functionလုပ်ဖို့လိုလာပီ အဲ့တော့ dataတွေကို transform လုပ်ဖို့ map လိုလာတယ်။ အဲ့လိုမျိုး map လုပ်ဖို့ဆိုရင် container or box or lift ဖြစ်နေဖို့လိုတယ်။ အဲ့တော့ container ဆိုတာဘာလဲဆိုရင် valueတခုခုကိုယူပီး somethingတခုခု နဲ့ wrap လုပ်ထားတာမျိုးကိုပြောတာ။ Java မှာဆို wrapper လိုမျိုးကိုပြောတာ။ ဥပမာအနေနဲ့ဆိုရင် x x = 1; ဆိုပီး ရှိတယ်ဆိုပါဆို့ အဲ့တာကို imperative မှာ တန်ဖိုးတိုးချင်တယ်ဆိုရင် x++; ဆိုပီး ရေးလိုက်လို့ရတယ်။ ဒါပေမယ့် FPနဲ့နေရေးမယ်ဆို side effect မဖြစ်ဖို့လိုတယ် အဲ့တော့ functor ကို သုံးပီးရေးကြည့်မယ်ဆိုရင် အောက်ကလိုရေးလို့ရတယ်။

```
let Functor = (v) => ({
  value: v,
  map: (f) => Functor(f(v)),
  valueOf: () => v,
```

```
});

let x = 1;
let incFn = (val) => val + 1;

let functorInc = Functor(x);
let newX = functorInc.map(incFn);
console.log("Old X : ", x, " NewX : ", newX.valueOf());
```

အဲ့လိုပုံစံမျိုးနဲ့ မူလတန်ဖိုးကိုလည်း side effect မဖြစ်စေပဲ ရေးလို့ရတယ်။

of function in Functor

functor ရဲ့ of function ဆိုတာ တခြားမဟုတ်ပါဘူး။ သူက အဓိကက Functor object ကို ဒဲ့မဆောက်ပဲ Functor.of(value) ဆိုပီး ကြားခံအနေနဲ့ liftလုပ်ပီး container ဆောက်ချင်ရင် သုံးတဲ့ကောင်ပဲဖြစ်တယ်။

```
Functor.of = (v) => Functor(v);

let functorInc = Functor.of(x);
```

ဒီလိုမျိုးကြားခံအနေနဲ့ container liftလုပ်ပေးတဲ့ကောင်မျိုးဖြစ်တယ်။

MayBeFunctor

MaybeFunctorက ဘာလဲဆိုရင် FPမှာ error handlingတွက်သုံးတဲ့ a kind of functor ပဲဖြစ်တယ်။ ဥပမာနေနဲ့ဆိုရင်

```
let upper = (value) => value.toUpperCase();
let appendHi = (val) => "Hi" + val;
```

```
let result = R.pipe(upper, appendHi);
console.log(result(null));
```

ဒီကုဒ်မှာဆို ပေးလိုက်တဲ့ value က nullဖြစ်နေတဲ့ကြောင့် null Exceptionတက်သွားမယ် အဲ့တော့ အဲ့လိုပြဿနာမျိုးကို maybe Functor ကို သုံးပီး ဖြေရှင်းလို့ရတယ်။

```
let MayBe = (v) => ({
  value: v,
  map(f) {
    return this.isNothing() ? MayBe.of(null) : MayBe.of(f(v));
  },
  valueOf: () => v,
  isNothing: () => v === undefined || v === null,
});

MayBe.of = (v) => MayBe(v);

let emptyMaybe = MayBe.of(null);
let upperMaybe = emptyMaybe.map(upper).map(appendHi);
console.log(emptyMaybe.valueOf());
```

အပေါ်က maybe functorမှာလို isNothing()ဆိုတဲ့ error check functionကို functor မှာထည့်ပီး အဲ့ဒေတာကို map မှာပြန်check check ပီးမှ functor createလုပ်ပီး ဖြေရှင်းနိုင်တယ်။ Error ဖြစ်တဲ့ အခြေနေကိုထိမ်းလိုက်ပေမယ့် ဘယ်နားမှာ error တက်သွားတာလဲ errorဖြစ်ရင်ဘာလုပ်မလဲဆိုတဲ့ အခြေနေတွေကို handle မလုပ်နိုင်ဘူး အဲ့တော့ ဒီပြဿနာကို ဖြေရှင်းဖို့တွက် either ဆိုတဲ့ functor ကိုသုံးနိုင်တယ်။

Either Functor

Functor က ဘယ်ကလာလဲဆိုရင် railway oriented programmingက theoryကနေလာတယ် process တခုက passဖြစ်သွားရင် ဘာလုပ်မလဲ fail ဖြစ်သွားရင်ဘာလုပ်မလဲ အဲ့လိုမျိုး happy state မှာဘာလုပ်မယ် unhappy state ဘာလုပ်မယ် သတ်မှတ်တာမျိုးကိုခေါ်တယ်။ အဲ့မှာဆို success stateတွေဆို right function မှာလုပ်ပီး failure state တွေဆို left functionမှာလုပ်တယ်။ code example အနေနဲ့

```
//happy case
const right = (v) => ({
  map: (f) => right(f(v)),
  matchWith: (pattern) => {
    pattern.right(v);
    return right(v);
  },
});

//sad case
const left = (v) => ({
  map: () => left(v),
  matchWith: (pattern) => {
    pattern.left(v);
    return left(v);
  },
});

right(4)
  .map((x) => x * x)
  .matchWith({
    right: (v) => console.log(v),
    left: (v) => console.log("left" + v),
```

```
  });

left(4)
  .map((x) => x * x)
  .matchWith({
    right: (v) => console.log(v),
    left: (v) => console.log("left " + v),
  });

function div(a, b) {
  if (b == 0) {
    throw Error("Division by zero");
  } else {
    return a / b;
  }
}

let Try = (f) => {
  try {
    let result = f();
    return right(result);
  } catch (e) {
    return left(e);
  }
};

Try(div.bind(null, 3, 01)).matchWith({
  right(v) {
    console.log("Ok ", v);
  },
  left(e) {
    console.log("Error ", e);
  },
});
```

အပေါ်က ကုဒ်ကလို success ဖြစ်ရင်လုပ်မယ့် right case နဲ့ failure ဖြစ်ရင် လုပ်မယ့် left

case ဆိုပီး နှစ်ခုရှိတယ်။ အဲ့ကောင်၂ ခုလုံးကို pattern matching နဲ့ ချိတ်ဆက်ပီး ပီးတော့မှ functor ဆောက်တဲ့အချိန်မှ right caseဆိုဘာလုပ်မယ် left caseဆို ဘာလုပ်မယ်ဆိုပီး သတ်မှတ်ပေးတယ်။ imperative မှာဆို try catch မှာ catch ဖြစ်သွားရင် နောက်ထပ် အခြားတခု ဆက်လုပ်လို့မရတော့ဘူး either နဲ့ဆို failure ဖြစ်သွားပေမယ့်ဆက်လုပ်လို့ ရသေးတယ်။ အဲ့တာက either ရဲ့main power ပဲ

Monoid

Monoid မှာ binary operation and identity ရယ်ရှိရမယ်။ binary operationဆိုတာ input 2 ခုကနေ output value 1ခုထုတ်ပေးရမယ်။ ဘယ်ညာမှာ ရှိတဲ့ type 2ခုက တူရမယ်။ ပီးတော့ ဘယ်ဘက်က operand 2ခုကလည်း type တူရမယ်။ a +b = c အဲ့လိုပုံစံမျိုးပေါ့။ အဲ့မှာ a, b, c သုံးခုလုံးက type တူရမယ်။ ပြောရမယ်ဆို input value 2ခု type ကတူရမယ် ပီးတော့ operationလုပ်ပီး ရလာတဲ့ value type ကလည်းပဲ တူရမယ်။

နောက်တခုက associative ဖြစ်ရမယ်။ ဘယ်လိုမျိုးလဲဆိုရင် a + b+ c ကိုပေါင်းတဲ့တန်ဖိုးက (a + b) + c or a + (b + c) တန်ဖိုးနဲ့တူရမယ်။ ပြောချင်တာက +, * operation တွက်ဆို result က တူနေရမယ်။

Neutral element or identity element ဖြစ်ရမယ်ဆိုတာက သူ့ကို +, * operation လုပ်ရင် ပြောင်းလဲချင်းမရှိပဲ သူ့တန်ဖိုးပြန်ရရမယ်။ ဥပမာ 1 + 0 = 1, 2 + 0 = 2, 1 * 1 = 1, 2 * 1 = 2 အဲ့လိုမျိုး အပေါင်းတွက်ဆို 0 အမြှောက်တွက်ဆို 1 unique ဖြစ်နေတယ့် တန်ဖိုးရှိရမယ်။

monoid တွေက ဘာအသုံးဝင်လဲလို့ပြောရင် merge reduce framework တွေ အတွက် big data calculation တွေမှာဆို parallel processing ပုံစံမျိုးနဲ့ လုပ်ဆောင်လို့ရတယ်။ ဘာလို့လဲဆိုရင် identity ဖြစ်တဲ့တွက်ကြောင့် same result ဖြစ်မှာ ဖြစ်တဲ့တွက် ဒေတာတွေကို အပိုင်းတွေပိုင်းပီး လုပ်ဆောင်နိုင်တယ်။ အဲ့လို အားသာချက်တွေ ရှိတယ်။

Code Example အနေနဲ့ဆိုရင်

```
let adderMonoid = {
  binaryOperation: (x, y) => x + y,
  neutralEle: 0,
};

let multiplyMonoid = {
  binaryOperation: (x, y) => x * y,
  neutralEle: 1,
};

let arr = [1, 2, 3, 4, 5];
let total = arr.reduce(adderMonoid.binaryOperation,
adderMonoid.neutralEle);
console.log("Total : ", total);

let multiply = arr.reduce(
  multiplyMonoid.binaryOperation,
  multiplyMonoid.neutralEle
);
console.log("Multiply : ", multiply);
```

အဲ့လိုမျိုး monoid ကိုသုံးပီး reduceလုပ်ပီး data calculation တွေကို သုံးနိုင်တယ်။

Monad

Monad ဆိုတာက ပြောရမယ်ဆိုရင် FP design pattern ပဲဖြစ်တယ်။ FPမှာကျတော့ design pattern လို့မသုံးပဲ Functor, Monad လို့သုံးတယ်။ Functor မှာဆို map ဆိုပီး ရှိတယ်။ map ကကျတော့ Monad(Monad(a)) ဆိုရင် Monad(Monad(b))ဆိုပီး data ပြန်ရတယ်။ အဲ့ကျတော့ ဒေတာက two level ဖြစ်နေတ်။ အဲ့ကျတော့ monad မှာကျတော့ flatMapကိုသုံးလိုက်တယ်။ flatMap ကျတော့ Monad(Monad(a))ဆိုရင် Monad(b) ဆိုပီး ပြန်ရတယ်။ ဒေတာကို two level ကနေ one level ကို flatten လုပ်လိုက်တယ်။ ဥပမာ အနေနဲ့ဆိုရင် array ကိုပြောရမယ် array ကို map လုပ်မယ်ဆိုရင် array(array) ပြန်ရမယ် Monad(Monad(b))လိုပေါ့ ဒါပေမဲ့ flat map လုပ်လိုက်မယ်ဆိုရင်တော့ Monad(b) ရမယ်။ ကုဒ်နမူနာအနေနဲ့ဆိုရင် အောက်ကလို ပုံစံမျိုးဖြစ်မယ်။

```
let arr = [1, 2, 3, 4];
let result = arr.map((x) => [x, x * x]);
let result1 = arr.flatMap((x) => [x, x * x]);

console.log("Map ", result);
/* Output
0 : [1, 1]
1 : [2, 4]
2 : [3, 9]
3 : [4, 16]
*/
console.log("Flat Map ", result1);
// Output [1, 1, 2, 4, 3, 9, 4, 16]
```

Monad ကပြောရမယ်ဆိုရင် container wrap လုပ်ထားတာကို unwrap လုပ်ပီး dataပြန်

ပေးတဲ့ပုံစံမျိုးပေါ့။ မြင်သာအောင်ပြေရမယ်ဆိုရင် ငှက်ပျောသီး အခွံနွာသလိုမျိုးပေါ့ အခွံက container တကယ်ငါတို့လိုချင်တာက အခွံအောက်က အသား အဲ့တော့ အခွံကို နွာလိုက်တယ် unwrap လုပ်လိုက်တာပေါ့ monad မှာကျအဲ့တာကို flat map သုံးပီး flatten လုပ်ချလိုက်တယ် အဓိက concept ကတော့ container unwrap လုပ်လိုက်တာပါပဲ။

Type Lift

Type Lift ဆိုတာ ဘာလဲဆိုရင် value ကနေ container wrap ပီး Functor or Monad ပြောင်းလိုက်တာမျိုးကိုခေါ်တယ်။ v => Functor(v)

Flatten

Flatten ကကျတော့ type lift နဲ့ပြောင်းပြန်ပေါ့ သူကကျတော့ wrap container Monad or Functor ရှိတယ်။ အဲ့ကနေ unwrap or unboxing လုပ်ပီး value change ပစ်လိုက်တာမျိုးကို ခေါ်တယ်။ Monad(v) => v

Use Case of Monad

Composition မှာဆိုရင်ကျ compose လုပ်ဖို့ဆိုရင် single input, single output, same input/output type ဖြစ်ဖို့လိုတယ်။ အဲ့မှာ multiple input ဖြစ်လာပီဆို ဘယ်လိုလုပ်မလဲ အဲ့ကျတော့ currying ကို သုံးပီး ဖြေရှင်းလို့ရတယ်။

Currying မှာဆိုရင် multiple input function တွေကို single input ဖြစ်အောင်ပြောင်းပီး

input စုံပီဆိုမှ ပြန်ပီး input တွေကို bound လုပ်ပီး output processလုပ်ပေးတယ်။

Monad ရဲ့အလုပ်ကကျတော့ ဘာလဲဆိုရင် different data type ကို same type အဖြစ် wrap လုပ်ပစ်လိုက်တယ်။ အဲ့တော့ ပထမတခုက string value ပြန်ပေးတယ် အဲ့ကပြန်ပေးတဲ့ ဒေတာကို သုံးတဲ့ function မှာကျ monad လိုတယ် အဲ့လို different ဖြစ်နေတဲ့ type မျိုးတွေကို ဖြေရှင်းဖို့ဆိုရင်ကျတော့ monad ကို သုံးလို့ရတယ်။ ဥပမာအနေနဲ့ဆို

```
g:                  a => M(b)
f:                  b => M(c)
h = composeM(f, g): a => M(c)
```

အပေါ်က ဒေတာမှာဆို g functionက a ဆိုတဲ့ဒေတာကို လက်ခံပီး Monad b ကို return ပြန်တယ်။ function f က ကျတော့ value bကို လက်ခံပီး Monad c ကို return ပြန်တယ် အဲ့လိုမျိုး မတူညီတဲ့ ဒေတာ input typeတွေကို compose လုပ်မယ်ဆို monad ကိုသုံးရမယ်။ ပထမ function g ကနေ ရလာတဲ့ Monad(b)ကို function gကို မပို့ခင် Monad(b) ကို flatten လုပ်ပီး value bအဖြစ်ပြောင်းပစ်ပီးမှ လုပ်ရမယ်။ အဲ့လို အခြေနေမျိုးမှာဆိုရင် Monad ကို သုံးတယ်။

Monad လို့ပြောပီဆိုရင် functor, monoid ဖြစ်ရမယ်။ ပီးရင် သူမှာ lift or unit(a => M(a)), flatten or Join(M(a) => a) and map(M(a) => M(b)) ဆိုပီး ပါရမယ်။ ပီးရင် Flatten နဲ့ map ကို ပေါင်းထားတဲ့ Flat Map or Chain (M(M(a)) => M(b))ဆိုပီး type change ပေးတဲ့ကောင်ပါရမယ်။ Example Code အနေနဲ့ဆိုရင်

```
const Monad = {
  of: (value) => ({
    value: value,
```

```
    inspect: () => `Monad(${deepInspect(value)})`,
    map: (fn) => Monad.of(fn(value)),
    flatMap: (fn) => fn(value),
  }),
};

const myMonad = Monad.of(1); // Monad(1)
const add1Monad = (a) => Monad.of(a + 1);
console.log("Two level ", myMonad.map(add1Monad));

const myMonadPlus1 = myMonad.flatMap(add1Monad);
console.log("One layer ", myMonadPlus1.value);
```

ဒီမှာဆိုရင် map လုပ်ဖို့ပေးလိုက်တဲ့ကောင်က ရိုးရိုးfunction မဟုတ်တော့ပဲ container ပါတဲ့ကောင်ဖြစ်တဲ့တွက်ကြောင့် map လုပ်ရင် ရတဲ့ဒေတာ မှန်မှာမဟုတ်ဘူး အဲ့တော့ flat map နဲ့ chain လုပ်ပီးတော့မှ resultမှန်မှန်ကန်ကန်ထွက်လာမယ် အဲ့လိုနေရာမျိုးမှာ monad ကိုသုံးတယ်။

FP Explain Details Link By Sir Thet Khine

Gentle introduction to lambda calculus (λ calculus) Part 1

https://www.facebook.com/thet.khine.587/posts/pfbid031A6PHGGroqCqyayQdT7BykZAMyaHsdLM5UHGdHji2dNDgEd53sqtU7X8tqh5w57ml

Gentle introduction to lambda calculus (λ calculus) Part 2 Church Numeral

https://www.facebook.com/thet.khine.587/posts/pfbid0LctetV1jJHYmoWmNDBHcAZFuWsmWALFey2Vo3MeJXdhUaAHmSGDvZScgfkU7Xigcl

Gentle introduction to lambda calculus (λ calculus) Part 3 (Arithmetic, Boolean)

https://www.facebook.com/notes/1068691256919412/

Practical Functional Programming Series(Part 1)

https://www.facebook.com/thet.khine.587/posts/pfbid02mhgKCEaUX3itx3f5ndKz2TnLKpY7s3gJq9oC74Rpwfp5omdM8n3M83iBsm8AdG8gl

Practical Functional Programming Series(Part 2)

https://www.facebook.com/thet.khine.587/posts/pfbid02LK682E6JYUAtSS8bXhv

kudfYeGv9oNdsddbwq36rpu6ZAp6bNCYXuWkendYdM3p8l

Practical Functional Programming Series(Part 3)

https://www.facebook.com/groups/programmingchannel/permalink/2241744616120691/

Practical Functional Programming Series(Part 4)

https://www.facebook.com/groups/programmingchannel/permalink/2243123015982851/

Practical Functional Programming Series(Part 5)

https://www.facebook.com/thet.khine.587/posts/pfbid0GMRPTEnRcfAVgFxEVT3Npi5n1DL431Xw8k7qRX3oxy8mvecbNxwrqqFwYxup8MLRl

Practical Functional Programming Series(Part 6)

https://www.facebook.com/thet.khine.587/posts/pfbid0R2LHqHhUvD5dQ3vgGk6Lbcy3LWsydEFThEScYCatsLf2rE1bkyQmSBJMNcaYHYwml

Practical Functional Programming Series(Part 7)

https://www.facebook.com/groups/programmingchannel/permalink/2619787271649755/

Practical Functional Programming Series(Part 8)

https://www.facebook.com/thet.khine.587/posts/pfbid02pBQ6xwotFckBWgqG9fhteEGW2YG2qwEfUGNcdrVDw5Rsk4jWsU73L2z1JvEwqoodl

Practical Functional Programming Series(Part 9)

https://www.facebook.com/groups/programmingchannel/permalink/2645788765716272/

Practical Functional Programming Series(Part 10)

https://www.facebook.com/thet.khine.587/posts/pfbid034UxXswVkUCM3rtu2gGVqT9iYRwrmetwUmNjoD24srppTvFCm6vBdX9pbkhPcJPPCl