Functional Programming (FP) By Sir Thet Khine

Gentle introduction to lambda calculus (λ calculus) Part 1

lambda calculus က theoretical computer science မှာ Computability theory မှာသုံးဖို့.အတွက်ထုတ်ထားတဲ့ mathematical model ပါပဲ။ Computability theory ဆိုတာ ဘယ်လို problem တွေကို တော့ program ရေးပြီးတွက်လို့.ရတယ် ဘယ်ကောင်တွေကိုတော့ တွက်လို့.မရဘူး အဲ့တာကို ဆုံးဖြတ်ပေးတဲ့ theory။ ဥပမာ programming language တော်တော်များများမှာ အောက်က code သည် compile လုပ်မယ်။

```
int getNo()
{
    while(true);
}
```

ဘာစိတ်ဝင်စားစရာပါလဲဆိုတော့ return statement မပါပဲနဲ့.ကို compile လုပ်လို့.ရတာ။ မေးစရာမေးခွန်းက compiler ရေးတဲ့ကောင်တွေက အဲ့ဒီ code သည် return မပါဘူး ဘယ်တော့မှ ဆုံးမှာမဟုတ်ဘူးဆိုတာကို မသိဘူးလားဆိုတာ ။ ရေးတဲ့ကောင်တွေကတော့ သိလိမ့်မယ်၊ ဒါပေမဲ့သူကို algorithm နဲ့.ဆုံးဖြတ်ဖို့.ကျတော့ program တပုဒ်က ဘယ်အချိန်မှာ ရပ်မယ် ဆိုတာ မသိနိုင်ဘူး ဒါကို halting problem လို့.ခေါ်တယ်။ Computability theory ထဲကတခုပေါ့။ ဆက်ပြောရင် ခုနက lambda calculus ဆိုတာ

Turing machine လို abstract model ပဲ။ Turing machine ထက်ပိုအားသာတာက သူက mathematical proof ပါပြလို.ရတယ် ။ Turing machine model ကိုအခြေခံထားတဲ့ programming language တွေက imperative programming language တွေဖြစ်လာပြီးတော့ lambda calculus ကိုအခြေခံထားတဲ့ကောင်တွေက functional programming language ြဖစ်လာတယ်။

Lambda calculus ကိုရှင်းရင် lambda ဆိုတာ Greek အက္ခရာ ဒီ symbol λ လေးကိုပြောတာ အဲ့တာကလဲ ပထမ ^X လိုကောင်လေးကနေ စာရိုက်တဲ့ကောင်က မှားပြီး λ ဖြစ်သွားတယ်ပြောတာပဲ။ Calculus ဆိုတာက ဒီနေရာမှာ differentiate , integrate တွေကို ဆိုလိုတာမဟုတ်ပဲနဲ. Mathematical branch တခုကိုပဲဆိုလိုတာဖြစ်တယ်။

Lambda calculus သည် theory အရ turing machine လို အသေးငယ်ဆုံး programming language တခုပဲ။ သူ.မှာ lambda term ၃ ခုပဲပါတယ်။ နောက်သူတို.ကို transformation reduction လုပ်ဖို. Rule လေးနည်းနည်းပဲပါတယ်။ အဲ့ကနေ မှတဆင့် အားလုံးကို function တွေ သုံးပြီး တည်ဆောက်ယူလို.ရတယ် ဥပမာ number လို primitive တွေ lambda calculus မှာမရှိဘူး အဲ့အစား number တွေကို function တွေနဲ.ပဲတည်ဆောက်တယ် Boolean ဆိုတာလဲမရှိဘူး ခုနကလိုပဲ function တွေနဲ.ပဲ lambda term ၃ ခုကိုသုံးပြီးတည်ဆောက်တယ်။ နောက် arithmetic operator တွေ control structure တည်ဆောက်တာတွေ အဲ့ကောင်တွေ အားလုံးကို လဲ function တွေသုံးပြီးတော့ပဲတည်ဆောက်တယ်။

Lambda term

Variable

x –variable သူကတော့ရှင်းတယ် variable ကိုပြောတာ lambda calculus မှာ တကယ်တော့ variable ဆိုတာထက် innumtable ဖြစ်တဲ့ constant လို. Programming concept အရ ကြည့်မှ မှန်မယ်။ Math ကျတော့ variable လို.ပဲသုံးတယ်

Function Abstraction

λ x.x – λ လေးနောက်မှာ x လေးသို.မဟုတ် တခြား variable လေးပါမယ် နောက် dot(.) ကလေးပါမယ်ဆိုရင် ဒါကို function abstraction လို.ခေါ်တယ်။ Programming အရတော့ function definition ပါပဲ။ ခုနက λ x.x ကို JS နဲ့.ရေးကြည့်ရင် ဒီလိုရမယ်

```
f = x => x;
```

f သည် function တခုဖြစ်တယ် သူ.ကို parameter တခုလက်ခံတယ် အဲ့တာကို x လို.ခေါ်မယ် သူ.အလုပ်က x ကလေးကိုပဲ return ပြန်တယ်ပေါ့။ တခုသတိထားရမှာက lambda calculus မှာ function တွေသည် parameter တခုကိုပဲ လက်ခံလို.ရတယ်။ unary function ပေါ့။ တကယ်လုိ. Parameter တခုထက်ပိုလိုတဲ့ကောင်တွေဆိုရင် currying လို parameter ကိုအဆင့်ဆင့်လက်ခံတဲ့ ပုံစံမျိုးရေးရတယ်။ Parameter တခုပဲလက်ခံတာက ဘာကောင်းလဲဆိုတော့ function composition မှာ function တွေကို အဆင့်ဆင့် pipeline

လုပ်လို.ရတာပဲ။ ဥပမာ function ၁ခုက output သည် နောက် function ၁ ခုရဲ. Input အနေနဲ. ဝင်သွားမယ်။ ဆိုပါစို. X နဲ. Y parameter ၂ခုလက်ခံရမဲ့ function ကို curry နဲ.ရေးမယ်ဆိုဒီလိုရမယ်။

f = x=>y=> x+y;

f =(x=>(y=>x+y));

ကွင်းတွေ ခက်ထားတာကိုကြည့်လိုက်ရင်ပိုနားလည်မယ်။

Function Application

တကယ်လို.များ f နဲ. x က lambda term ဆိုပါစို. ဒါဆို fx လို.ရေးထားရင် ဒါသည် function application ပဲ။ ဆိုချင်တာက function invoking ကိုပြောတာ။ Lambda မှာ fx ဆိုရင် JS နဲ.ရေးရင် ဒီလိုရမယ်.

f(x)

Lambda calculus ဖို.ကျတော့ သီးသန်. Parenthesis တွေမလိုဘူး။

Free and Bound Variables

ဆိုပါစို. λ x.xy ဆိုတဲ့ lambda expression မှာ λ x. သည် function header ပေါ့ အဲ့မှာမှ x သည် function parameter ပေါ့ အဲ့တာကို xy ဆိုတဲ့ကောင်ကကျတော့ function body ပေါ့ header က x သည် body က x နဲ့တူတူပဲ အဲ့တာကို bound တယ်လို.ခေါ်တယ် ဆိုတော့ကာ

λ x.xy ဆိုတဲ့ expression မှာ x သည် bound variable ဖြစ်တယ်၊ function header or parameter မဟုတ်တဲ့ function body က variable ကို free variable လို့ခေါ်တယ်။

Beta Reduction

Lambda expression တွေကို evaluate လုပ်တဲ့အခါ transformation rule တွေ ၃ ခုလောက်နဲ့ပြီးတယ် အဲ့ထဲကမှ beta reduction က အရေးပါတယ်။

(λ x.x)(2) ဆိုပါစို့. ဒါဆို JS အမြင်နဲ့ကြည့်ရင် ဒီလိုဖြစ်မယ်။

(f = x=>x)(2)

သူ့ရဲ့. တန်ဖိုးသည် 2 ရမယ် ဒီသဘောတရားကို 2 သည် lambda bound variable x နဲ့အစားထို့လို့ရတယ် lambda body က x ရှိတဲ့နေရာတွေကို 2 နဲ့အစားထိုးမယ် ဒါကို beta reduction လို့ခေါ်တယ်။

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

Lambda calculus မှာ number တွေ arithmetic operator တွေ conditional statement တွေ တခုမှမပါဘူး။ ခုနက number တွေ arithmetic တွေ Boolean တွေ control structure တွေ အားလုံးကို lambad abstraction ဖြစ်တဲ့ variable, lambda abstraction, lambad application အဲ့သုံးခုတည်းနဲ့ တည်ဆောက်ကြတယ်။ ဆိုချင်တာက ရှိသမျှ အားလုံးကို

function တွေနဲ့.ပဲတည်ဆောက်ကြတယ်။

Church Encoding

Church ဆိုတာ lambda calculus ကိုထွင်ခဲ့တဲ့ Alonzo Church ကိုပြောတာ။ Church encoding ဆိုတာ ခုနကပြောတဲ့ number တွေ Boolean တွေ function တွေနဲ့. Encode လုပ်တဲ့နည်း တည်ဆောက်သော နည်းလို.ပြောရမယ်။ Encoding ဆိုတာ ပုံစံတမျိုးမျိုးနဲ့.သိမ်းတယ် လို.ဆိုရမယ်။ ဥပမာ မြန်မာလို တစ်လို.ရေးပေမဲ့ English မှာကျတော့ one မဟုတ်လား။ တစ်လို.ဆိုဆို one လို.ရေးရေး အဓိပ္ပာယ်ကတော့ ဂဏာန်း နံပါတ် ၁ ကို ကိုယ်စားပြုတယ်ပေါ့ ။ ဒီသဘောကို encoding လို.ဆိုရမယ်။ ဒါကအရေးကြီးတယ်။

Church Numeral

Church numeral ဆိုတာ number တွေကို function တွေနဲ့. Encode လုပ်တယ် ဖော်ပြတဲ့နည်းကိုပြောတာ။ Number တခုမှာ input 2 ခုပါတယ် တခုက f ဆိုပါစို့. Function ပေါ့ ဘာလုပ်မယ်ဆိုတဲ့ function။ နောက်တခုက x ပေါ့ သူက ခုနက ဘာလုပ်မယ်ဆိုတဲ့ f function ဆီကိုပေးရမဲ့ input ပေါ့။ number တွေရဲ့. Concept က ဒီလို zero ဆိုတာ သုည အခါလုပ်တယ်ပေါ့ ဆိုချင်တာက function f ကို တခါမှ မခေါ်ဘူး။

ဥပမာ zero ကို ဒီလို နည်းနဲ့. Encode လုပ်တယ်။

λ f. λ x.x

ဆိုချင်တာက zero သည် input 2 ခုပါသော function

λ f က တခု λ x က တခု

zero ဖြစ်တဲ့အတွက် function ကို တခါမှ မလုပ်ခိုင်းဘူး အဲ့တော့ ဝင်လာတဲ့ input ကို ဒီအတိုင်းလေးပဲ ပြန်ပေးလိုက်မယ်။ ၁ ဆိုရင်တော့ တကြိမ်တော့ အလုပ်ခိုင်းမယ် ဒီသဘောပေါ့။ zero ကို JS နဲ့.ရေးရင် ဒီလိုရမယ်

ZERO = f => x => x

ZERO ကိုခေါ်သုံးချင်တဲ့အခါ ဒီလိုသုံးရလိမ့်မယ်။

ZERO(f)(x)

f ကတော့ လုပ်စေချင်တဲ့ function အဲ့တာက ဘာသဘောလဲဆိုတော့ zero အကြိမ် လမ်းလျောက်မယ်ဆိုပါစို့. F ဆိုတာ လမ်းလျောက်မယ်ဆိုတဲ့ function ပဲ။ လမ်းလျောက်မယ်ကို zero numeral ထဲပေးလိုက်ရင် တခါမှလုပ်မှာမဟုတ်ဘူး။ ဘာလို့.ဆိုတော့ ZERO = f => x => x အဲ့မှာ function f ကို တခါမှ ခေါ်ထားတာမဟုတ်လို့.။ ဒါဆို One အကြောင်းပြောရအောင် one ဆိုတော့ တခါတော့လုပ်ရမှာပေါ့။ ခုနက zero လိုပဲ လုပ်မဲ့ function f နဲ့. Input ပါရမယ်။

One = λ f. λ x.fx ဒီမှာ fx ဆိုတာ f(x) ဆိုတာကိုပြောတာ one ဆိုတော့ function ကုိ တခါတော့ခေါ်လုပ်မယ်ပေါ့။ ၂ဆိုရင် lambda calculus အရဆိုဒီလိုရမယ်။

λ f. λ x.f(fx)

Two မှာ f(fx) ဆိုတာ f(f(x)) = function ကို ၂ခါ ခေါ်တယ် apply လုပ်တယ်ပေါ့။ ဒါဆို ၃

ဆိုရင် ၃ ခါ apply လုပ်တယ် f(f(f(x))) ဒီလိုသွားတယ် သဘောက။ အောက်မှာ lambda နဲ့ JS code တွေတွဲပြီးပြထားတယ်။

ZERO = f => x => x // λ f. λ x.x

ONE = f => x => f(x) // λ f. λ x.fx

TWO = f => x => f(f(x)) // λ f. λ x.f(fx)

THREE = f =>x = f(f(f(x))) // λ f. λ x.f(f(fx))

ကောင်းပြီ number တွေတော့ရပြီ သူတို့ကို ဘယ်လို ခေါ်သုံးရမှာတုန်းဆိုရင် ဒီလိုသုံးလို့ရတယ်။

```
let ZERO = f => x => x  ;

let ONE = f => x => f(x);

let TWO =  f => x => f (f(x));

let THREE = f => x => f (f(f(x)));

let TO_INTEGER = n=> n(x=>x+1)(0)

let f = x  =>{

console.log('Jump ');

return x+1;

}

//console.log(TO_INTEGER(ZERO));
```

```
console.log(ONE(f)(0));

console.log('Jump 3 times');

console.log( THREE(f)(0));
```

ဒီနေရာမှာ Church numeral တွေက လူတွေ မြင်ရတဲ့ 1, 2, 3 နံပါတ်တွေကို ကိုယ်စားပြုတာမဟုတ်ဘူး ဘယ်နှခါလုပ်မယ်ဆိုတဲ့ Math concept ကိုပဲ ကိုယ်စားပြုတာ အဲ့တော့ လူတွေလို နံပါတ်တွေ 1 ,2,3 မြင်ချင်တဲ့အခါ TO_INTEGER ဆိုတာလေးကိုရေးပြီးသုံးလိုက်တယ် ။ အစ intial value မှာ 0 ကိုပေးတယ် function f ကို တခါ apply လုပ်တိုင်း ခေါ်တိုင်း 1 တိုးပေးသွားရင် လူတွေ သုံးနေကျ နံပါတ်ကို မြင်ရလိမ့်မယ်။ အပေါ်က code ကို run ကြည့်ရင် ဒါမျိုးရမယ်။

```
Jump // ဒါက ONE(f)(0) အတွက် output

အောက် ကကောင်ကတော့ THREE(f)(0) အတွက် output ရတာ။

Jump

Jump

Jump

3
```

The Successor function

ခုနက နံပါတ်တေွကို hard code လုပ်နေရတာ အဆင်မပြေဘူး အဲ့တော့ successor

function ကိုသုံးမယ် successor function ဆိုတာ church numeral တခုကို လက်ခံမယ် ဥပမာ numeral one ကိုလက်ခံရင် သူက church numeral two ကို return ပြန်ပေးရမယ်။

Lambada calculus notation နဲ့ဆို ဒီလိုရမယ်

λ n. λ f. λ x.f(nfx)

အဲ့ဒီမှာ n ဆိုတာက curch numeral ဖြစ်တယ် ဥပမာ ZERO,ONE သူတို့ပေါ့။

f (n f x) ဆိုတာကိုရှင်းရရင် n(f)(x) ဆိုတာ church numeral n ကို run တာပေါ့ ဥပမာ ONE ကို run ချင်ရင် ONE(f)(x) မဟုတ်လား။ အဲ့တော့ခုနက n(f)(x) ကို f ထဲထဲ့ပြီး ထပ် run လိုက်ရင် အပိုတခေါက် run မယ် ဒါဆို n+1 ခါ run ပြီပေါ့။

ဒါကြောင့် f(n f x) သည် n+1 ဒါမှမဟုတ် successor function ရတယ်ပေါ့။ JS နဲ့ဆို ဒီလို define လုပ်လို့.ရမယ်။

SUCC = n => f => x => f(n(f)(x))

ခုနက code တွေ အောက်ကို ဒါလေး ထဲ့လိုက်ရအောင်

let SUCC = n => f => x => f(n(f)(x))

let FOUR = SUCC(THREE);

console.log('Run four ');

console.log(FOUR(f)(0));

ဒါကို run လိုက်ရင်

Jump

Jump

Jump

Jump

4

ဆိုပြီး FOUR ဖို့.ရလိမ့်မယ်။

ဒါဆို successor ရပြီဆိုရင် +, * စတာတွေ လုပ်လို့.ရပြီ အဲ့တာကတော့ နောက်ဆက်ရန်ပေါ့။

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

အရင်အပိုင်းတုန်းက Successor အကြောင်းရှင်းသွားတယ် Successor ဆိုတာ church numeral n ပေးလိုက်ရင် n+1 ပြန်ပေးတဲ့ကောင်ပေါ့။

Addition

Successor ကိုသုံးပြီးတော့ church number ၂ခုကိုပေါင်းလို့.ရတယ်။ ဥပမာ m+n ဆိုပါစို့. တကယ်က m ကို +1 ပေါင်း n ကြိမ်တိုးတာလို့.ယူဆတာ။ ဥပမာ 3+2 ဆိုပါစို့.

ဒါဆို ဒီလို စဉ်းစားတာ 3+1+1 ပေါ့ (+1) နဂို 3 ကို +1 ပေါင်း 2 ကြိမ်လုပ်လိုက်ရင် 3+2 ရမယ်ပေါ့။ lamdbda calculus နဲ့.ဆုိဒီလိုရမယ်

PLUS = λ mn. λ fx.(m f) ((n f) x))

M ေရာ n ရောက church number တွေ reduction step တွေကိုတော့ ထဲမရေးတော့ဘူး ပိုရှုပ်သွားမှာစိုးလို. ((n f) x) ဆိုတာက church numeral n ကိုခေါ်သုံးတာ။ အဲ့ကောင်ကို ရှေ့က f ((n f)x) ဆိုတာက successor function အဲ့တာကို m function ထဲ ထဲ့လိုက်တော့ m ကို +1 n ခါတိုးတာပဲ အဲ့တော့ m+n ရတယ်ပေါ့

```
let ONE = f => x => f(x)

let TWO = f => x => f(f(x))

let SUCCESSOR = n => f => x => f( n(f)(x));

var hi = (x)=>{

console.log(x);

return x+1;

}

let SUM =    m => n => n(SUCCESSOR)(m);

console.log(SUM(TWO)(TWO)(hi)(0));
```

အပေါ်က code ကို run လိုက်ရင်

0

1

2

3

4 ဆိုပြီး ရလိမ့်မယ် နောက်ဆုံး 4 သည် SUM(TWO)(TWO)(hi)(0) အတွက်အဖြေ။

အပေါင်းကိုရပြီဆိုရင် Multiplication ကိုလဲလုပ်ယူလို့.ရပြီ lambda calculus နဲ့.ဆိုဒီလိုရမယ်။

mult = λ m. λ n. λ f .m (n f)

let ONE = f => x => f(x)

let TWO = f => x => f(f(x))

let SUCCESSOR = n => f => x => f(n(f)(x));

let THREE = SUCCESSOR(TWO);

var hi = (x)=>{

console.log(x);

return x+1;

}

let MULT = m => n => f => m(n(f))

console.log(MULT(THREE)(TWO)(hi)(0));

အမှန်က church numeral m ကို ခေါ်ရင် numeral_function(function_to_do)(seed) အဲ့လိုခေါ်ရတယ်မို့.လား ဥပမာ TWO ကို သုံးချင်ရင် TWO(hi)(0)ပေါ့

အဲ့နေရာမှာ multiplication ဆိုတော့ hi နေရာမှာ n ကိုအစားပေးပြီးလုပ်ခိုင်းလိုက်တာ အဲ့တော့ အမြှောက်ရမယ်။

Predecessor Function

အနှုတ်ဖို့.ကျတော့ n-1 function လိုတယ် သူကို predecessor function လိုခေါ်တယ်သူက တော်တော်ရှုပ်တယ် concept ကတော့ pair တွေဆောက်မယ် ဥပမာ 1အတွက်ဆိုရင် (0,1) ပေါ့

2 ဆိုရင် (1,2) အဲ့ထဲ့ကမှ first ဖြစ်တဲ့1 ကိုယူမယ် အဲ့လိုစဉ်းစားတာ။ ဒီလိုရမယ် zero အတွက်ဆိုရင်တော့ predecessor က zero ပဲပြန်မယ်။

```
let PREDECESSOR    = n => f => x => n(g => h => h(g(f)))(_ => x)(u => u);
```

ဒါဆို အနှုတ်ကို ဒီလိုရေးလို့.ရပြီ ခုနက အပေါင်းတုန်းက စဉ်းစားသလိုပဲပေါ့

```
Let SUBTRACTION      = m => n => n(PREDECESSOR)(m)

let ONE = f => x => f(x)

let TWO = f => x => f(f(x))

let SUCCESSOR = n => f => x => f( n(f)(x));

let THREE    = SUCCESSOR(TWO);

var hi = (x)=>{

console.log(x);

return x+1;

}

let PREDECESSOR    = n => f => x => n(g => h => h(g(f)))(_ => x)(u => u);
```

let SUBTRACTION = m => n => n(PREDECESSOR)(m)

console.log(SUBTRACTION(THREE)(TWO)(hi)(0));

အပေါ်က code ကို run ရင်

0 1 ဆိုပြီးရမယ် ၃ ထဲက ၂ နှုတ်တာ ၁ ကျန်တယ်ပေါ့။

Division ကတော့ recursion သုံးပြီး နှုတ်နှုတ်သွားတာ

Church Boolean

Lambda calculus မှာ function ကလွဲရင်ကျန်တာဘာမှမရှိဘူးအဲ့တော့ Boolean number တွေကိုလဲ Function တွေနဲ့ပဲ define လုပ်တယ်။ Boolean တခုသည် တန်ဖိုး 2 ခုကိုလက်ခံတယ် ဥပမာ frist, second ဆိုပါစို့., Chruch Boolean True သည် input 2 ခုက ပထမတခုကို return ပြန်တယ် ပေးထားတဲ့ဥပမာဆို first ကို return ပြန်မယ် False ကျတော့ ဒုတိယတခု second ကို return ပြန်တယ်။

Lambda calculus notation နဲ့ဆို ဒီအတိုင်းရမယ်

TRUE:= λ tf.t

FALSE := λ tf.f

JS နဲ့ဆိုဒီလိုရမယ်

TRUE = t => f => t

FALSE = t => f => f

True, False ကို ဒီလိုသုံးလို.ရမယ်

```
let TRUE  = t => f => t

let FALSE = t => f => f

console.log(TRUE(true)(false));

console.log(FALSE(true)(false));
```

ဒီနေရာမှာ TRUE, FALSE ကို parameter ပေးလိုက်တဲ့ input JS က တကယ့် Boolean တွေသည် အရေးမပါဘူး မြင်သာအောင်သာပေးလိုက်တာ။ Church numeral တွေတုန်းက concept ပဲပေါ့။

AND, OR

And ဆိုရင် condition ၂ခုလုံးကမှန်မှ လုပ်ရမှာ

```
AND  = p => q => p(q)(p)
```

ဒီကောင့်ကိုရှင်းရမယ်ဆိုရင် p ရယ် q ရယ်သည် church Boolean တွေဖြစ်မယ်

p(_)(_) လို.စဉ်းစားကြည့်ရအောင် တကယ်လို. P ကသာ false ဖြစ်မယ်ဆိုရင် ဒုတိယ argument ကိုရွေးရမယ်(False definition အရ) ဒါဆို p က false ဖြစ်တယ်ဆိုတဲ့ယူဆချက်အတွက် p(_)(p) ဖြစ်မယ် တကယ်လို. P က true ဖြစ်ရင်တော့ q အပေါ်မူတည်သွားပြီ အဲ့တော့ True သည် ပထမတခုကို ယူမှာ အဲ့တော့ p(q)(p) ဖြစ်မယ်။

Or ကိုအဲ အဲ့သဘောသုံးပြီးတည်ဆောက်နိုင်တယ်

OR = p => q => p(p)(q)

ဒီလောက်ဆို အတော်ရှုပ်ပြီမို့. တော်လောက်ပါပြီ code တွေက သူများတွေရေးပြီးသားရှိတော့ မရေးတော့ပါဘူး ဒီမှာသွားယူလို့.ရပါတယ်

https://github.com/gtramontina/lambda/blob/master/lambda.js

Practical Functional Programming Series(Part 1)

Functional Programming က mainstream programming language တွေ နောက် frontend ဘက်တွေ မှာလဲသုံးလာတော့ သူ့.အကြောင်းရေးဖို့.လုပ်လိုက်တာ။

ဟိုးအရင်က Gentle introduction ရေးဖူးတယ် ဒီမှာ

https://web.facebook.com/thet.khine.587/posts/10206463929836573

Functional programming က ဘယ်ကစလဲလို့.ပြောရင် Fortan ပြီးပေါ်တဲ့ language LISP က စတယ်လို့.ပြောရမယ်။ Fortan လို language မျိုးကို imperative paradigm အနွယ်ဝင်လို့.ပြောရမယ် ဘာကိုဆိုလိုတာလဲဆိုတော့ Programming သည် လက်ရှိ von neumann architecture နဲ့.နီးစပ်တယ် အဲ့အတိုင်းရေးတာကို Imperative programming လို့.ခေါ်တာ။ မြင်သာအောင်ပြောရရင် Computer memory ထဲကနေ variable တွေကိုယူ +,-,*,/လုပ်တယ် conditional statement, loop တွေသုံးတယ် အဲ့လို programming style မျိုးကို imperative လို့.သုံးတာ။ လက်ရှိသုံးနေတဲ့ hardware နဲ့.နီးစပ်တယ်။ Conditional စစ်တာ loop ဆိုတာမျိုးက assembly, machine code မှာ တိုက်ရိုက် support ပေးထားတာ။ Functional programming paradigm ကို ခုနက စဉ်းစားတဲ့ von neumann architecture ကနေ ခွဲထွက်စဉ်းစားပြီး computation လုပ်လို့.မရဘူးလား ဆိုတဲ့ ယူဆချက်တွေကနေ ပေါ်လာတာ။ Imperative paradigm တွေက Turing machine model ပေါ် အခြေခံထားတာ။ Turing machine ဆိုတာ infinite tape တခုရှိမယ် အဲ့ tape ပေါ်မှာ

လိုချင်တာတွေရိုက်လို.ရမယ် ရှေ.ရွေ.တာ နောက်ရွေ.တာ သွားလို.ရမယ် ဒါဆိုရင် computable function(program ရေးပြီးတွက်ထုတ်နိုင်တာမှန်သမျှ) တွေအကုန် ရေးလို.ရမယ်လို. model အရ သတ်မှတ်ကြတာ။

Functional programming paradigm ကိုကျတော့ ဘယ်ကနေ model ယူစဉ်းစားလဲဆိုတော့ Math က Lambda Calculus ဆီကနေယူထားတာ , Lambda calculus မှာ computation ကို function တွေနဲ.ပဲ အားလုံးစဉ်းစားမယ် ဥပမာ အားလုံး ဘာမှမရှိဘူး function ရှိမယ် function ဆောက်လို.ရမယ် function ကို first class ပေးမယ် အဲ့တာက လွဲရင် variable, loop, control structure,number, boolean အစရှိတာတွေ ဘာဆိုဘာမှမရှိဘူး အားလုံးကို function ဆိုတဲ့ abstraction ရယ် နောက် reduction rule လေး ၃ခုလောက်နဲ. computation အကုန်လုပ်လို.ရတယ် ၊ဥပမာ Java နဲ.ရေးထားတဲ့ algorithm တခုကို lambda calculus နဲ.ရေးလို.ရှိရင်လဲ theory အရ ရနိုင်တယ် ဒါကိုပြောတာ။

၃ ပိုင်းရေးထားတာရှိတယ် နားလည်ရခက်မယ် ဒါပေမဲ့ lambda calculus နားမလည်ပဲ functional programming ကိုသေချာနားလည်နိုင်ဖို.မလွယ်ဘူး။

Part 1

Gentle introduction to lambda calculus (λ calculus) Part 1

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

Part 2

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

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

Part 3

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

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

Functional programming paradigm ဆိုရင် အောက်က အချက်တွေ ပါလေ့ရှိတယ်

First Class Function

Higher Order Function

Pure Function

Recursion (Proper tail call recursion)

Strict vs Lazy Evaluation.

Immutable Data Structure

Pattern Matching.

Algebraic Data Type

နောက် functional design pattern တွေဖြစ်တဲ့ Functor, Applicative Functor, Monad,

Monoid, စတဲ့ Math ဘက်က Category Theory ဘက်က ဆင်းလာတဲ့ကောင်တွေကို သုံးတာလဲရှိတယ်။ အဲ့တာတွေက language အပေါ်မူတည်တာလဲရှိတယ်။ ဥပမာ monad ကို Haskell မှာ စပြီးသုံးတာ ဘာလို.လဲဆိုတော့ Haskell က pure functional programming language, ဆိုချင်တာက သူ.မှာ assignment ဆိုတာမရှိဘူး mutable data structure ဆိုတာမရှိဘူး ပြောရရင် သာမာန် imperative programming language တွေမှာသုံးတဲ့ variable ဆိုတဲ့ concept မှာ အချိန်ပေါ်မူတည်ပြီး value ပြောင်းနိုင်တယ် ။ ဒါပေမဲ့ haskell မှာ constant ပဲ နောက် သူ.မှာရှိတဲ့ data structure တွေသည် အားလုံးသည် immutable ဖြစ်တယ် ဆိုချင်တာက List, Set အစရှိတာတွေမှာ element တခုထဲ့လိုက်ရင် နဂိုရှိပြီးသား List ထဲကို အသစ်ဝင်သွားတာမဟုတ်ပဲ list အသစ်ရမယ်။ ခုနကလို Haskell မှာ Pure function တွေပဲရေးလို.ရတယ် ဒါကသက်သက်ရှင်းရမှာ Pure Function ဆိုတာ side effect မပါတာကိုပြောတာ side effect ဆိုတာ တိုတိုတုတုတ်ရေးရရင် function ရဲ. result သည် funtion prameter ရယ် constant တွေကလွဲရင် ကျန်တာမပါတာပြောတာ IO ထုတ်တာတွေ file ဖတ်တာတွေသည်လဲ side effect ပဲ variabl ပြောင်းမှမဟုတ်ဘူး၊ အဲ့တော့ IO လိုကောင်မျိုးကို Haskell မှာလုပ်ချင်တော့ IO သည် impure function ဖြစ်တယ် ဒါကို Monad နဲ. wrap လုပ် ပြီးသုံးမယ်ပေါ့။ ကျန်တဲ့ Impure ဖြစ်တဲ့ Scala, F# လိုကောင်မျိုးမှာ အဲ့လိုမလိုဘူ သုံးလဲရတယ် ဒါပေမဲ့ efficient မဖြစ်ဘူးပေါ့။

Classification of functional programming language

Pure functional programming language

ခုနက Haskell လိုကောင်မျိုးကို pure functional programming language လို့သုံးတယ် ဘာလို့ဆိုတော့ mutable variable မရှိဘူး pure function တွေပဲလက်ခံတယ်၊ နောက် imperative style ဘာမှရေးမရဘူး အဲ့ကောင်တွေသည် pure functional programming language.

Impure Functional Programming Language

Lisp, Scheme, Clojure, Standard ML, OCaml, F# , Erlang အဲ့ကောင်တွေကို impure Functional Programing language လိုခေါ်တယ် ဘာလို့ဆိုတော့ side effect ပေးသုံးလို့။

Programming language that support functional Programming

ဒါကတော့ Java,C#, Kotlin, JavaScript တို့လို functional program တွေ တပိုင်းတစ အနေနဲ့ရေးလို့ရတဲ့ကောင်တွေကိုပြောတာ။ Imperative based programming language တော်တော်များများသည် functional programming မှာတွေ့ရတဲ့ tail call optimization မပေးဘူး (Kotlin မှာ tail call ပါတယ် ဒါပေမဲ့ recursive call ကို loop နဲ့. JVM byte code မှာပြောင်းပေးတာမျိုးပဲ JVM မှာ tail call optimization မပါဘူး) tail call optimization ဆိုတာ imperative language တွေမှာ recursive function တွေရေးရင် function call ခေါ်တဲ့ depth များသွားရင် method call stack မှာ ရှည်လာပြီးတော့ out of memory, stack

overflow error တက်သွားတာ ဒါမျိုးကို functional programming language တွေ မယ်မဖြစ်အောင် tail call optimization နဲ့ထိမ်းပေးထားတယ်(သက်သက်ရေးရမှာ)

Practical Functional Programming Series(Part 2)

How Functional Paradigm Differ with Imperative(State vs Immutable Data Structure)

Functional Programming paradigm က imperative programing paradigm နဲ့. အဓိက ဘယ်လို thinking တွေ problem solving တွေ ဘယ်လိုကွာသလဲဆိုတာ ဒီအပိုင်းမှာရှင်းမှာပါ။

State vs Immutable Data Structure.

Imperative Paradigm ရဲ့. အဓိက concept သည် state change အပေါ်မှာမူတည်တယ် variable တွေရဲ့. တန်ဖိုးကို assignment statement သုံးပြီးတော့ ပြောင်းလဲသွားခြင်းနဲ့. အလုပ်လုပ်တယ်။ Functional Programming ကျတော့ pure function တွေကိုသုံးပြီးတော့ side effect မပါအောင် ထိန်းပြီးအလုပ်လုပ်တယ်။ Immutable Data Structure တွေကိုသုံးတယ်။ State ဆိုတာ impeative paradigm မှာသုံးတဲ့ variable တွေလို့.ဆိုလိုရမယ်။ Mutable data structure ပေါ့ အချိန်ပေါ်မူတည်ပြီး

တန်ဖိုးပြောင်းလဲလို.ရတဲ့ကောင်တွေပေါ့။ Impeative paradigm မှာသုံးတဲ့ variable တွေသည် value ကို time အပေါ်မူတည်ပြီးတော့ပြောင်းလို.ရတယ် ဥပမာ

int num = 1;

num += 4;

ပထမ line မှာ num သည် 1 ဖြစ်မယ် ဒါပေမဲ့ ဒုတိယ line ကျတော့ num သည် 5 ဖြစ်မယ်။ ဆိုချင်တာက num ဆိုတဲ့ varaible သည် အချိန်ပေါ်မူတည်ပြီးတော့ တန်ဖိုးအမျိုးမျိုးပြောင်းနေမယ်။ FP မှာကျတော့အဲ့လိုမဟုတ်ဘူး တကယ့် real functional programming langauge တွေ pure functional programming language တွေမှာဆိုရင် variable သည် contant ပဲဖြစ်တယ်(ဥပမာ Haskell) ။ သူ.ရဲ. constant ဆိုတဲ့သဘောကလဲ impeative က constant နဲ့.မတူဘူး immutable လို.ပြောရမယ်။ ဥပမာ Java, JavaScript က String object တွေလိုပေါ့။

ဥပမာ Java မှာ

final String str = “Hello”;

final String str2 = str.toUppserCase();

အဲ့ဒီမှာ str.toUppserCase() လို.ဆိုလိုက်ရင် str ဆိုတဲ့ string ထဲက element တွေသည် upper case character ကိုပြောင်းမသွားဘူး နဂိုမူလအတိုင်းကျန်တယ် str.toUppserCase လို.ခေါ်လိုက်ရင် String အသစ်ရမယ် အဲ့ String အသစ်က upper case တွေချည်းပါတာဖြစ်မယ် မူလရှိရင်းစွဲ str ဆိုတဲ့ String ကိုဘာမှ မထိပဲနဲ. String

အသစ်ပြန်ပေးတယ် Data structure တွေမှာ သူ့ထဲကိုမထိပဲနဲ့. operation လုပ်တဲ့အခါ အသစ်ပြန်ပေးတဲ့ data structure တွေကို immutable data structure လို့ခေါ်တယ် ဥပမာ Java က Stack, Array စတာတွေသည် mutaable data structure, String သည် immutable data structure.

Immutable data structure တွေကို purely functional data structure လို့လဲခေါ်ကြသေးတယ်။ FP ရဲ့ philosophy အရ ဘာကောင်းလို့. immutable data structure တွေကိုသုံးတာလဲပေါ့။ Immutable data structure ဆိုတော့ ဘယ်သူက မှပြင်လို့မရဘူး ဆိုချင်တာက Object တခုတည်းကို thread ၂ ခုက တပြိုင်းတည်း ပြင်တာ modify လုပ်တာ မရှိတော့ဘူး ။ Modify , mutate လုပ်လို့ရတဲ့ data structure မှ မဟုတ်တာကိုး။ တကယ်လို့ modify လုပ်လို့ရတဲ့ data structure ဆိုရင် Thread တခုက ပြင်နေရင် နောက် Thread တခုက ဝင်ပြင်တဲ့အခါ synchronization issue တွေ ဖြစ်လာမယ်။ အဲ့တော့ အဲ့လိုပြဿနာမပေါ်ဖို့. concurrency ကို control လုပ်ရမယ် ။ lock ချရမယ်။ အိမ်သာ တလုံးကို လူ ၂ယောက် တပြိုင်တည်း တက်လို့မရသလိုပေါ့။ တက်မယ်ဆိုရင် sychronization lock လိုမယ် ။ တယောက်ပြီး မှ နောက်တယောက်ရမယ် အဲ့တော့ ခုနက lock သည် waiting time ကိုဖြစ်စေတယ် ကြာစေနိုင်တယ် အဲ့လိုဖြစ်ရတာသည် mutable datastructure ကိုသုံးလို့. immutabl ကိုသုံးတော့ ဘယ်သူမှ ပြင်လို့မရဘူ အသစ်တွေပဲ ရမယ် အဲတော့ synchronization ထိန်းစရာမလိုဘူး massive concurrency မှာအားသာတယ် ဥပမာ ခုသုံးနေတဲ့ messenger တွေနောက်က server

တော်တော်များများသည် Erlang ဆိုတဲ့ Fucntional Programming langauge သုံးရေးထားတာ။ ဘာလို. Erlang သုံးလဲဆိုရင် concurrency ဆိုရင် FP က synchronization ထိန်းစရာမလိုတဲ့အတွက်ပိုပြီး ကောင်းလို.မြန်လို.။

အဲ့လို philosophy ကွာတော့ FP သည် concurrency မှာ အဆင်ပြေတယ် ဒါပေမဲ့ state ကို ထိန်းရတဲ့ real time နဲ.ဆိုင်တာတွေ state change ရတာတွေနဲ.ဆိုင်တဲ့ embedded system လိုကောင်တွေမှာတော့ FP သည် အဆင်မပြေနိုင်ဘူး။

နောက်တခုက easy to reason လုပ်လို.ရတယ် ဥပမာ အောက်က function ဆိုပါစို.

```
int a = 10;

int getSomething(int b)

{

        return b+a;

}
```

getSomething function သည် ပြင်ပ state ဖြစ်တဲ့ variable ဖြစ်တဲ့ aအပေါ်ကိုမူတည်တဲ့ b+a ကို return ပြန်တယ် ဆိုချင်တာက getSomething(2)ကို ၂ နေရာမှာ ခေါ်မယ်ဆိုပါစို. parameter b ကို ၂ ပဲပေးတယ် ဒါပေမဲ့ return result သည် အတူတူပဲရမယ်ပြောလို.မရဘူး ဘာလို.လဲဆိုတော့ function result သည် variable a အပေါ်မူတည်နေလို. တယောက်ယောက်က သာ a ကိုပြောင်းလိုက်ရင် အဖြေပြောင်းနိုင်တယ် အဲ့လို function မျိုးကို impure function လို.ခေါ်တယ် FP က function တွေသည် Math က function

တွေလိုမျိုး pure ဖြစ်တယ် state ကို မမူတည်တဲ့အတွက် ဘယ်အချိန်ခေါ်ခေါ် input တူရင် output တူတယ် အဲ့တော့ reason လုပ်ရတာလွယ်တယ်

အဓိက state ကိုသုံးတာ မကောင်းတဲ့အချက်က ဥပမာ ဒီ state ကို change တာများရင် ဘယ်သူက ပြောင်းလို.ပြောင်းမှန်းမသိတာ ဘာထွက်မယ်မှန်းရခက်တာမျိုးဖြစ်လာတယ် ဒါကြောင့် OOP မှာ state ကို Object ထဲငုံပြီး ထိန်းတယ် ဒါလဲပြောင်းလို.ရနေသေးတဲ့အတွက် မှန်းရခက်တာတွေ ဖြစ်လာတယ် FP မှာကျတော့ အဲ့လိုမသုံးဘူး အဲ့တော့ reason လုပ်ရတာ ပိုလွယ်တယ်။

တကြောင်းတည်း ချုပ်ပြောရရင် imperative သည် state change တယ် FP သည် transformation ကိုလုပ်တယ်။

Practical Functional Programming Series(Part 3)

How Functional Paradigm Differ with Imperative(Control Flow Structure)

Control flow ဆိုတာ program တခုမှာ selection, repetition, exception handling, method call return အစရှိတဲ့ program ရဲ. execution of flow ကိုပြောင်းစေတဲ့ statement တွေ ကိုပြောတာပါ။ If, If else statement, loop, method call, exception handling, break, return စတဲ့ statement တွေပါပါတယ်။ များသောအားဖြင့် basic control flow ကို

ခွဲရင် branching နဲ့. looping ရပါမယ်။ If/Switch conditional statement တွေရယ် loop တွေရယ်ပေါ့။

Conditional Control Flow

ဒီနေရာမှာ Imperative paradigm မှာ if/switch statement တွေကို သုံးပါတယ်။ ဒီနေရာမှာ imperative programming paradigm မှာ if statement သည် statement အဖြစ်များပါတယ် Functional programming language တွေမှာကျတော့ expression အနေနဲ့.တွေ့.ရတာများပါတယ်။

ဥပမာ F# မှာ အောက်က code ဆိုပါစို့.

let isPositive x = if x>0 then “Positive”

else “Negative”

isPositive ဆိုတာ function ပါ x က parameter ပါ အခဲ့မှာ if နဲ့.စစ်ထားပါတယ် ဒီမှာ x>0 ဖြစ်ရင် true ပြန်ပါမယ် မဟုတ်ရင် false ပါ (x>0လို့.ရေးလို့.ရတာကိုခနမေ့ထားပါ) ဆိုချင်တာက if သည် statment မဟုတ်ပဲနဲ့. value တခုခုပြန်ပေးရတဲ့ expression ဖြစ်တာကိုပြောတာပါ။

နောက်တခုက if statement ကိုသုံးတာသည် တကယ်တမ်း functional မဆန်ပါဘူး ဘာလို့.လဲဆိုတော့ functional programming language တွေမှာ သူ့.ထက်ကောင်းတဲ့ Pattern matching ဆိုတဲ့ feature တွေရှိလို့.ပါပဲ ဥပမာ

```
let dayOfMonth = function
    | 2 -> 28
    | 4|6|9|11-> 30
    |_ -> 31
```

အပေါ်က dayOfMonth ဆိုတာသည် လ တလမှာဘယ် နှရက်ရှိသလဲဆိုတာကို return ပြန်ပါတယ် Feb - 28 ပေါ့ (ဒီနှစ်ကိုတော့ခနမေ့ထား) ကျန်တဲ့ 4,6,9,11 တွေက ၃၀ အဲ့တာတွေမဟုတ်တဲ့ကောင် (_) သူက 31 ပေါ့။ ဒါကို Pattern Matching လို့.ခေါ်ပါတယ်။ Powerful ဖြစ်လို့.များသောအားဖြင့် Functional programming မှာ if/switch အစားသူ.ကိုသုံးပါတယ်။

```
let rec pow = function
    | (x,0) -> 1
    | (x,n) -> x * pow(x, n-1)
```

အပေါ်က code က tuple ကို pattern match လုပ်ထားတာပါ pow ကိုတွက်တဲ့ function ပါ , zero ကို

| (x,0) -> 1 ဆိုတာ pow(2,0) , pow(3,0) အစရှိတဲ့ ဘယ် number ရဲ့. 0 ကိုမဆို match ဖြစ်ပါလိမ့်မယ်။ သူ.ဆိုရင် 1 ပြန်ပါလိမ့်မယ် ။ တခြားကောင်တွေဆိုရင် (x,n) -> x * pow(x, n-1) နဲ့.သုံးပါလိမ့်မယ်။

Repetition

Imperative programming language တွေမှာတော့ for, while, do, util အစရှိတာတွေကိုသုံးပါတယ် ဒါပေမဲ့ Functional programming အရတော့ အားလုံးကို function သုံးပြီးရေးပါတယ်။ Recursive function တွေသုံးပြီးရေးသလို Utility function တွေဖြစ်တဲ့ each, map, reduce, fold, unfold, zip အစရှိတာတွေသုံးပြီးတော့လဲ ရေးပါတယ်။

ဥပမာ အောက်က ကောင်သည် factorial function ကို F# နဲ့ရေးထားတာပါ Pattern Matching , Recursive function သုံးပြီးရေးပါတယ်။

```
let rec fac = function
    | 0 ->1
    | n -> n * fac (n-1)
```

Recursive function တွေသည် အဆင့်ဆင့်ခေါ်တဲ့ depth များလာရင် StackOverflow error ဖြစ်လေ့ရှိပါတယ် ဒါကို ထိန်းဖို့ FP language တွေမှာ tail call optimization ဆိုတာထဲ့ပေးထားပါတယ်။

```
printfn "All product %A" ([1;2;3;] |> List.reduce (*))
```

အပေါ်က code သည် F# မှာ 1,2,3 ဆိုတဲ့ list ကို အားလုံးမြှောက်တာကိုလုပ်ပါတယ် ပြီးရင် အဖြေကို printfn နဲ့ထုတ်ပါတယ် သူက List.reduce ဆိုတဲ့ utility ကိုသုံးသွားတာပါ * သည် F# မှာ function ဖြစ်ပါတယ်။ reduce ဆိုတဲ့ function ကိုပေးလိုက်တဲ့ function ပါပဲ။ |>

operator ကတော့ list [1;2;3] ကို List.reduce ဘယ်ဘက်မှာရေးလို.ရအောင်လုပ်ပေးတာပါ [1;2;3] ကို List.reduce ညာဘက်ကို ပို.ပေးဆိုတဲ့သဘောပါပဲ။

Null value checking

Haskell လို pure functional programming language တွေမှာ null value မရှိပါဘူး အဲ့တော့ စစ်စရာမလိုပါဘူး။ အဲ့အစား ဘယ်လုသုံးလဲဆိုရင် Maybe ဆိုတဲ့ Monad ကိုသုံးပါတယ် ဘာနဲ့.တူသလဲဆိုရင် Java က Optional နဲ့.သဘောတူပါတယ်။ TypeSafe ပိုဖြစ်တာပေါ့။F# မှာတော့ Option type ကိုသုံးပါတယ်။ null ပေးစစ်ပေမဲ့ bad practice ဖြစ်ပါတယ်။

Exception Handing

F# မှာ failwith ဆိုတာနဲ့. exception ကိုလွင့်လို.ရပါတယ်။ ဒါပေမဲ့ Functional programming ဆန်ဆန်ရေးချင်ရင်တော့ continual passing style သုံးပြီး exception handling လုပ်တာကောင်းပါတယ်။ Node.js က error callback တွေပုံစံပေါ့ ဒါပေမဲ့ အဲ့ထက်အဆင့်မြင့်ပြီး ပိုသေသပ်တဲ့ FP style တွေရှိပါတယ် railway oriented programming style လို.ခေါ်ပါတယ် သက်ဆိုင်ရာအခန်းလိုက်သက်သက်ရေးမှ အဆင်ပြေပါမယ်။

Practical Functional Programming Series(Part 4)

How Functional Paradigm Differ with Imperative,OO (Abstraction)

ဒီတခေါက်ကတော့ Abstraction မှာ Imperative, OO language တွေနဲ့.ဘယ်လိုကွာသွားလဲဆိုတာကိုပြောမှာပါ။

Abstraction ဆိုတာ မလိုအပ်တဲ့ complex detail (implementation) တွေကို hide လုပ်ပြီး ရိုးရှင်းတဲ့ ကောင်တွေနဲ့.အစားထိုး represent လုပ်တာကို abstraction လို့.ခေါ်ပါတယ်။ ဥပမာ aircon တခုကို အသုံးပြုဖို့. သူ့.ကိုယ်ဘယ်လိုတည်ဆောက်ထားတယ်ဆိုတဲ့ (implementation) ကိုသိစရာမလိုပါဘူး remote control (interface) ကိုသိရင် ရပါပြီ။ ဒါသည် abstraction ပါပဲ။ ပြောရရင် programming language တွေမှာ more complex thing တွေကို သေးသေးလေးတွေကနေ ဘယ်လိုဆောက်ယူလို့.ရသလဲဆိုတာကို ဆိုချင်တာပါ။

Abstraction မှာ ၂မျိုးခွဲလို့.ရပါတယ် Data Abstraction နဲ့. Control Abstraction ပါ။ Data Abstraction ဆိုတာ custom data type, abstract data type တွေ ဘယ်လိုတည်ဆောက်လို့.ရသလဲ ဘယ်လိုတွေ ပေးထားသလဲကိုဆိုတာပါ။ ဥပမာ structure, enum, class, record , dictionary, array အစရှိတာတွေသည် data abstraction ပါ။ OOP language တွေမှာ data abstraction အတွက် Inheritance လိုကောင်မျိုးကို သုံးလို့.ရပါသေးတယ်။ Control Abstraction အတွက်ကျတော့ OOP language တွေမှာ

method call, dynamic dispatch(polymorphic call via virtual method) တွေ ပေးထားပါတယ်။

Control Abstraction ဆိုတာ control structure တွေကို ကိုယ်တိုင် တည်ဆောက်နိုင်တာကိုပြောတာပါ။ Language အနည်းစုမှာပဲတွေ့ရပါတယ်။ ဥပမာ C,C# မှာ macro တွေသုံးပြီး custom loop တွေလိုမျိုးရေးလို့ရပါတယ်။ Scala မှာလဲ custom loop တွေလိုမျိုးရေးလို့ရအောင် ပေးထားပါတယ်။ Ruby,Swift မှာလဲ လုပ်လို့ရပါတယ်။

```
def my_while(condition)
loop do
break unless condition.()
yield
end
end
i = 0
my_while -> { i < 5 } do
puts "Hello #{i}"
i += 1
end
```

အပေါ်က ruby code မှာ my_while ဆိုတာ custom control structure ပါ။ while လိုပဲ

အလုပ်လုပ်ပါတယ်။ Ruby မှာ method တွေကို code block တွေပေးလိုက်လို.ရပါတယ်။ Scala မှာလဲ အဲ့လိုပေးလိုက်လို.ရပါတယ်။ Swift မှာကျတော့ closures ဆိုပြီးပေးလိုက်လို.ပါတယ်။

Data Abstraction Function

Functional programming language တွေမှာ အဓိက ပေးထားတဲ့ data abstraction တွေက Imperative, OOP အစရှိတာတွေထက် powerful ဖြစ်ပါတယ်။ Type Inferencing ကိုပေးထားတဲ့ကောင်တွေလဲရှိပါတယ် (statically typed functional programming language တွေမှာပဲရပါတယ်) , Lis, Tuple, Record,အစရှိတဲ့ကောင်တွေကို ပေးလေ့ရှိပါတယ် တခြား Imperative, OO Language တွေမှာ မတွေ့.ရတဲ့ Algebraic Data Type , Recursive Data Type ဆိုတာတွေပေးပါတယ်။

Algebraic Data Type ဆိုတာ ဥပမာ

data Tree = Empty

| Leaf Int

| Node Tree Tree

Tree ဆိုတာ empty လဲ ဖြစ်နိုင်ပါတယ် Leaf ဒါမှမဟုတ် Node ၂ ခုပါတဲ့ Tree တွေလဲဖြစ်နိုင်ပါတယ်။ | သူက Type is one or another တခုမဟုတ်တခု ဆိုလိုတာပါ TypeScript မှာတော့ union type လို.ခေါ်ကြတယ်။ ဒီ Algebraic data type တွေက

pattern matching နဲ.ပေါင်းလိုက်တဲ့အခါ တော်တော် အသုံးဝင်ပါတယ်။ Programming Language parser တွေမှာသုံးတဲ့ AST(Abstract Synatx Tree) လိုကောင်မျိုးကို Algebric data type နဲ. represent လုပ်ရင် တော်တော် အသုံးတည့်ပါတယ်။အပေါ်က Tree သည် recursive data type လို.ပြောရင်လဲရပါတယ် ဘာလို.ဆိုတော့နောက်ဆုံး Type, Node Tree Tree မှာ Tree data constructure ကိုပြန်သုံးလို.ပါပဲ။

F# မှာလဲ Algebraic data type တွေပေးသုံးပါတယ်။ Algebraic Data Type လို.တော့မခေါ်ပါဘူး discriminated union လို.ခေါ်ပါတယ်။ TypeScript မှာလဲ discriminated union လို.ပဲသုံးပါတယ်။ Haskell မှာ Type Class ဆိုတာမျိုးရှိပါတယ် Java, C# က Generic လိုမျိုး Adhoc polymorphism ပေးနိုင်ပါတယ်။

Control Abstraction in Functional Programming Language

Functional programming language တွေရဲ. အဓိက control abstraction ကတော့ function တွေပါပဲ။ function တွေသည် first class value တွေဖြစ်ကြပါတယ်။ နောက် function တွေကို တခုနဲ.တခု ပေါင်းပြီး (compose) လုပ်ပြီး (တကယ်က function တခုကို ခေါ် သူ.ရဲ. output ကိုနောက် function input အနေနဲ.ပေး အဲ့လိုနည်းနဲ. function တွေကိုပေါင်းပြီး) သုံးကြပါတယ်။ Compose လုပ်ရင်လုပ် မလုပ်ရင် pipe လုပ်ပြီးသုံးကြပါတယ်။ Function တွေကို callback လိုပုံစံထားပြီး continuation-passing style ကိုသုံးပြီးတော့လဲ control ကိုထိန်းကြပါတယ်။

Practical Functional Programming Series(Part 5)

Function & Pure Function in Functional Programming

Functional programming language တွေက function concept က imperative အနွယ်ဝင် language တွေက function တွေနဲ့. ကွဲပါတယ်။ ဘယ်က ဟာနဲ့. တူလဲဆိုရင် Mathematics က function တွေကိုယူတာပါ။ Math က function ဆိုပါစို့. + ဆိုတာ function တခုဆိုပါစို့. သူ့မှာ parameter ၂ခုပါမယ် 2+3 သည် အမြဲတမ်း 5 ဖြစ်ပါတယ်။ ဆိုချင်တာက math မှာ function တွေသည် input တူတူပဲပေးလိုက်ရင် ဘယ်အချိန် run ပါစေ output တူတူထွက်ပါတယ်။ ဥပမာဆိုပါစို့.

```
function add( a, b )
{
    return a+b;
}
```

အပေါ်က ကောင်သည် add မှာ input သာ တူရင် အဲ့ဒီ function ကို ဘယ်နှကြိမ် run run ဘယ်လိုအခြေအနေမှာပဲ run ပါစေ output တူပါမယ်၊ ဥပမာ add(2,3) ကို ဘယ်လို ခေါ်ခေါ် ဘယ်အချိန် ခေါ်ခေါ် 5 ပဲရမယ်။ တခြားတန်ဖိုးမရဘူး အဲ့တော add(2,3) ကို time t1, မှာ ခေါ်တာနဲ့. နောက်ထပ် time t2 မှာခေါ်တာသည် မခြားနားဘူး အဖြေတူတူပဲရရမယ်။ ဒါပေမဲ့အောက်က imperative က function မျိုးကိုကြည့်ပါ။

```
var b = 10;

function plusB(a)

{

    return a +b;

}
```

plusB ဆိုတဲ့ function ကို plusB(2) လို. ခေါ်တာချင်းအတူတူ ဘယ်အချိန်ခေါ်သလဲ အပေါ်မူတည်ပြီး output ပြောင်းနိုင်ပါတယ်။

ဥပမာ

```
plusB(3)// 13;

b = 2;

plusB(3)// now 5;
```

ပထမ plusB(3) သည် 13 ရမယ် ဒုတိယ plusB(3) သည် 5 ရမယ်။ ဆိုချင်တာက အရင် example add လို ဘယ်အချိန်ခေါ်ခေါ် input တူရင် output တူမယ်ဆိုတာမရှိတော့ဘူး ဘာလို.လဲဆိုတော့ plusB သည် external state ဖြစ်တဲ့ b ဆိုတဲ့ variable အပေါ်မူတည်နေသေးတယ်။ b ကို ကြိုက်တဲ့သူက ပြောင်းလို.ရတယ် ၊ b အပေါ်မူတည်တဲ့အတွက် သူ.ရဲ. output သည် ပထမတခါ plusB(3) နဲ. နောက်တခါ plusB(3) မတူတော့ဘူး

Functional programming မှာရှိတဲ့ function တွေသည် input တူရင် output တူရမယ်။

အဲ့လိုဖြစ်ဖို. pure function ဖြစ်ဖို.လိုတယ်။ Functional programming က function တွေသည် mapping ကိုလုပ်တာပဲ transformation ကိုလုပ်တာပဲဖြစ်တယ်။ ဆိုချင်တာက ဝင်လာတဲ့ input value အပေါ်မူတည်ပြီး အဖြေတွက်တယ် အဖြေထုတ်တယ်။ Input value ကိုပြောင်းတာမလုပ်ဘူး နောက် တခြား environment က (output သည်i nput value နဲ. constant မဟုတ်သော တခြား variable တွေအပေါ်မမူတည်ဘူး) variable တွေ အပေါ်မမူတည်ဘူး။ အဲ့လို input တူရင် output တူတဲ့ function တွေကို pure function လိုခေါ်တယ်၊ ဘာကောင်းလဲဆိုတော့ referential transparency ရှိတယ်လို.ဆိုတယ်။

Pure Function

အပေါ်က add လို function မျိုးကို pure function လို.ခေါ်တယ်။ Pure Function ဆိုတာဘာလဲဆိုရင် function ရဲ. output(return value) သည် input parameter ရယ် နောက် constant တွေရယ်အပေါ်မှာပဲမူတည်တယ် တခြား variable တွေ environment state တွေအပေါ်မမူတည်ဘူး။ နောက် function သည် တခြား environment (memory, IO) အစရှိတာတွေကို မပြောင်းလဲရဘူး ဒါကို pure function လို.ခေါ်တယ်။ တနည်းအားဖြင့် side effect မရှိသော function ပေါ့။

Side effect ဆိုတာဘာလဲဆိုတော့

1.Input parameter ဒါမှမဟုတ် constant မဟုတ်ပဲ တခြား သော variable တွေကိုယူသုံးတာ သွားပြီးတော့ ပြောင်းလဲပစ်တာ(write လုပ်တာပေါ့)

2. File, read, write, IO (console.log ပါလဲ side effect လို.ပြောလို.ရတယ် ဒါသည် environment ကိုပြောင်းစေလို.)

3. ခြုံပြောရရင် ဒီ function သည် input parameter, constant နဲ. တခြား function တွေကိုပဲသုံးမယ် IO မထိရဘူး

Side effect မရှိတဲ့ function တွေကို pure function လိုခေါ်တယ်။ Haskell လို Pure functional programming တွေမှာ IO တွေကိုသုံးမယ်ဆိုရင် (IO သည် side effect) monad လိုကောင်တွေကိုသုံးရတယ်? impure ဖြစ်တဲ့ side effect ပါတဲ့ ကောင်တွေကို wrap လုပ်ပြီးသုံးတယ်ပြောရမယ်။

1 functional programming မှာ pure function နဲ.ရေးထားရင် add(2,3) ဘယ်နေရာတွေ.တွေ. 5 နဲ.အစားထိုးလိုက်လို.ရတယ် ။ ပြန်တွက်စရာမလိုဘူ းအဲ့တော့ cache လုပ်ထားလို.ရတယ်။referential transparency ဆိုတာ pure function တခုကို input တူရင် ဘယ်နေရာမှာသုံးသုံး output ကိုအစားထိုးပလိုက်လို.ရတယ်။

2 Logical reasoning မှာကောင်းတယ် ဆိုချင်တာက imperative မှာ trace, debug လုပ်ဖူးရင်သိပါလိမ့်မယ် function တွေသည် တခြား variable ရဲ. state ကိုမှီခိုတဲ့အခါ အဲ့ဒီ variable တွေကို ဘယ်သူတွေက သုံးထားလဲ ဘယ်နေရာပြောင်းထားလဲပါ လိုက်စစ်ရတယ်။ အဲ့တာမျိုး FP မှာမလိုဘူး ပိုကောင်းတယ်ပေါ့။

3 Composition လုပ်လို.လွယ်တယ် ဆိုချင်တာက function တခုနဲ.တခု အဆင့်ဆင့်ပေါင်းပြီးလုပ်သွားတာ (နောက်မှ တပိုင်းသက်သက်ထပ်ရေးရမယ်)

4 Parallelization လုပ်လို့.လွယ်တယ်။ ဆိုပါစို့. 2+3+4+5 ဆိုပါစို့. သူ.ကို 2+3 ကိုခွဲလုပ်မယ် 4+5 ကိုသက်သက်လုပ်မယ် အဲ့တာတွေလုပ်လဲကိစ္စမရှိဘူးဘာလို့.လဲဆိုတော့ အပြင် state အပေါ်မမူတည်တဲ့အတွက် time order ကိုကြည့်စရာမလိုဘူး၊ (imperative မှာ ဒါမျိုးမရဘူး)

5 Test ရေးလို့.ကောင်းတယ် input က တူရင် output တူတော့ဘယ်အခြေအနေရောက်မှ test က မှန်မယ်ဆိုတာမျိုးမရှိဘူး (precondition မလိုဘူး)

Imperative က function တွေနဲ့. FP က function တွေ မတူတဲ့နောက်တခုက FP က function တွေသည် return value ကိုအမြဲပြန်တယ်။ function သည် input ကနေ output ကို mapping လုပ်တာသာဖြစ်တယ် ။ အဲ့ဒီတော့ state ကိုသိမ်းထားခွင့်မရှိဘူး အဲ့တော့ return မပြန်ရင် state လဲသိမ်းမရတဲ့အတွက် ဘာမှအသုံးမဝင်ဘူးဖြစ်လိမ့်မယ်။ အဲ့တော့ input ရှိမယ် (input ကလဲ imperative paradigm မှာမရှိလဲရတယ် FP မှာတော့မရဘူး ဘာလို့.ဆိုတော့ side effect မရှိလို့.) output ပြန်ရမယ်။ parameter ကတော့ ကြိုက်သလောက် ထားလို့.ရတယ် return ကတော့ တခုပဲပြန်ကြတယ်။ များသောအားဖြင့် parameter ကို တခုတည်းထားရေးတာမျိုးရှိတယ်(ဘာကြောင့်လဲဆိုတာ နောက်အပိုင်းတွေကျမှ သက်သက်အခန်းခွဲရေးရမယ်)

အချုပ်ပြောရရင် FP က function တွေသည် math ကလိုပဲ ဖြစ်တယ်. input တူရင် output တူတယ်။ input , output အမြဲရှိရတယ်။ pure ဖြစ်တယ်။

Practical Functional Programming Series(Part 6)

Currying & Partial Application

Currying

Currying ဆိုတာ FP မှာ ဟင်းချက်တာနဲ့.ဆိုင်တာကိုပြောတာမဟုတ်ဘူး။ Mathematican Haskell Curry ကို ဂုဏ်ပြုပြီးပေးထားလို. curry လို.ခေါ်တာ။

Currying ကဘာလုပ်တာလဲဆိုရင် သူက function တခုမှာ argument တွေ အများကြီးရှိတယ်ဆိုပါစို့။ Argument အားလုံးကို တပြိုင်တည်းပေးလိုက်တာမဟုတ်ပဲနဲ့. တကြိမ်မှာ တခုချင်းဆီပေးမယ်။ ခုနကပေးလိုက်တဲ့ argument တွေကို wrap လုပ်ပြီး function အသစ်ပြန်ရမယ်။ နောက် argument လိုသလောက်အထိ တခုချင်းပေးနေတာကို ခေါ်တာ။ ဥပမာ f(a,b,c) ဒီလို function ရှိတယ်ဆိုပါစို့. ဒါဆိုရင် curry လုပ်ရင် f(a)(b)(c) အဲ့လိုပုံစံရအောင်လုပ်တာကို currying လို.ခေါ်တယ်။ Haskell, F# အစရှိတဲ့ Fp language တွေမှာ currying က library သုံးစရာမလိုဘူး langauge feature အနေနဲ့. default ပါလာတယ်။

လက်တွေ့. code example ကိုကြည့်ရအောင် JS နဲ့.ပေါ့။

```
function multiply(x, y) {
    return x * y;
}
```

အပေါ်က code က argument 2 ခုလက်ခံတယ် non-curry form ပေါ့ high-arity function လို.ခေါ်တယ် argument တခုထက်ပိုသော function ပေါ့ အဓိပ္ပာယ်က currying သည် multiple argument function တွေကို argument တခုချင်းလက်ခံသော function ဖြစ်အောင်ပြောင်းပစ်တာကိုဆိုတာပါ။ Curried ပုံစံနဲ့.ရေးမယ်ဆိုရင် ဒီလိုရပါမယ်။

```
function curriedMultiply(x) {
   return function(y) { return x * y; }
}
```

ဒါသည် curried form နဲ့.ရေးထားတာပေါ့။ Mainstream FP မှာအဲ့လို လိုက်ရေးစရာမလိုဘူး နောက် JS မှာလဲ FP library တွေမှာ curry method ဆိုတာရှိတယ်။ ဥပမာ Ramda လိုမျိုးပေါ့။ ခုနက curried form function ကိုခေါ်မယ်ဆို ဒီလိုခေါ်ရမယ်။

```
curriedMultiply(2)(3)
```

ဆိုပြီးတော့။ ပထမအဆင့်အနေနဲ့. curriedMultiply(2) ဆိုရင် ပေးလိုက်တဲ့ parameter 2 ကို closure အနေနဲ့.သိမ်းပြီး function အသစ်ပြန်လာမယ် ဆိုပါစို့.ဒါမျိုးလဲရေးလို.ရမယ်ပေါ့။

```
let multByTwo = curriedMultiply(2)
```

ဒါဆို multByTwo သည် function တခု သူသည် argument တခုလက်ခံဦးမယ် လိုနေတဲ့ y ပေါ့၊ဒါမျိုးရေးလို.ရမယ်

```
multByTwo(3)//6
multByTwo(5)//10
```

ဒါဆိုရင် currying ဆိုတာကို သဘောပေါက်ပါပြီ။ မေးစရာရှိတာက ဘာလို. အလုပ်ရှုပ်ခံပြီး parameter တွေ အကုန်ပေးလို.ရသားနဲ. တခုချင်း အရစ်ရှည်ပေးနေတာလဲဆိုတာပေါ့။ Parameter အကုန်ပေးလို.မရတဲ့နေရာတွေမှာသုံးလို.ရတယ်။ Code ပို clean ဖြစ်တယ်ပေါ့ အပေါ်က multByTwo သည် ၂ နဲ.မြောက်မယ်ဆိုတဲ့ function အသစ်ဖြစ်သွားတယ် ပြန်ပြန်ပြီး သုံးလို.ရတယ် ဒါမျိုးအတွက်လဲ အသုံးဝင်တယ်။

နောက်တခု အရေးကြီး အသုံးဝင်တဲ့နေရာက FP paradigm နဲ.ဆက်စပ်နေတယ်။ Software တွေက Complex ဖြစ်လာရင် သူတို.ကို အစိတ်အပိုင်းသေးသေးလေးတွေ ခွဲရတယ် ပြီးရင် အဲ့တာတွေကို ပြန်ပြီးတော့ စုပေါင်းရတယ်၊ ချိတ်ဆက်ရတယ်ပေါ့။ OOP မှာဆိုရင် class လေးတွေခွဲတယ် inheritance, composition, association စတာတွေနဲ.ပြန်ချိတ်ကြတယ်။ FP မှာကျတော့ function လေးတွေ ခွဲကြတယ် function တွေကို ဘယ်လိုချိတ်ဆက်အသုံးပြုသလဲဆိုတော့ composition ကိုသုံးတယ်။

Composition

Composition ဆိုတာ FP paradigm မှာတော့ function တွေသည် series အလိုက်ရှိမယ် ဥပမာ f1, f2, f3, f4 ပေါ့ အဲ့ကောင်တွေကို တခုချင်းဆီခေါ်မယ် ဥပမာ f1 ကိုခေါ်မယ် f1 အလုပ်လုပ်မယ် နောက် result ကို f2 ရဲ. parameter အနေနဲ.ပြန်ပေးမယ် f2 က return ပြန်လာတာကို f3 ကို parameter အနေနဲ.ပြန်ပေးမယ် ဒါကို function composition လို.ခေါ်တယ်။Composition ကလဲ သက်သက် note တခုရေးမှပြည့်စုံမယ်။

f4(f3(f2(f1(x)))

အိုကေ ဒါဆို composition ကိုမြင်ပြီဆိုပါစို့. အလွယ် JS မှာမြင်နိုင်တာက map, filter အဲ့ကောင်တွေသည် compose လုပ်ပြီး သုံးလို့.ရတယ်ပေါ့။ အဲ့မှာ ပြဿနာက ဘာလဲဆိုတော့ function တခုရဲ့. output ကို နောက် function တခုက သုံးတာဗျ။ အဲတော့ စဉ်းစားကြည့် f1 က argument ၁ ခုယူမယ် f2 က ၂ ခုယူမယ် ဆိုပါစို့. ဒါဆို ခုနကလို လွယ်လွယ်ကူကူ compose လုပ်လို့.ရပါ့မလား။ အဆင်မပြေတော့ဘူး ။ အဲ့တော့ ဘာလိုလဲဆိုတော့ function တခုသည် single input, single output လိုလာတယ်၊ ဒါမှ composition က အဆင်ပြေမှာ။ အဲ့တော့ multiple input ရှိနေတဲ့ function တွေကို single input လက်ခံတဲ့ function ဖြစ်အောင် ပြောင်းဖို့.လိုတယ် ၊ အဲ့တော့ currying ကိုသုံးတယ်ပေါ့။

Composition ကိုမြင်သာအောင်ပြောရရင် Unix က pipe တွေလိုပဲ တခု input ထဲ့မယ် process လုပ်မယ် နောက် သူ. result သည် တခြားတခုရဲ့. input ဖြစ်မယ်ပေါ့။

အဲ့လိုနေရာမသုံးချင်ပါဘူး တခြားနေရာသုံးမယ်ဆိုပါစို့. ဥပမာ field တွေကို validation စစ်မယ်ပေါ့ ဒါဆို

function validationCheck(validationCheck,field) ဒီလိုလိုမယ် အဲ့မှာ

nullChecker = validationCheck(nullCheck)

ဒါဆိုရင် nullCheck ဆိုတာ null လားစစ်ပေးတာ အဲ့မှာ filed အားလုံးဖို့. အကုန်လိုက်ရေးစရာမလိုပဲ curry သုံးလိုက်ရင် ဒီလိုသုံးလို့.ရတာပေါ့ OOP မယ် Builder design pattern နဲ့. concept တူသွားမှာပေါ့ ဒီလိုသုံးရင်တော့။

nullChecker(field1)

nulChecker(field2)

Partial Application

Application ဆိုတာ function ကိုခေါ်တာ execute လုပ်တာကိုဆိုလိုတာပေါ့ partial ဆိုတာ function တခုကလိုတဲ့ argument ကို အကုန်မပေးပဲ တခုဒါမှမဟုတ် တခုထက်ပဲပိုပေးတယ် အကုန်တော့မဟုတ်ဘူး အဲ့သဘောကို partial application လို.ဆိုတာ။ JS example ဆိုပါစို့.

```
function add(a, b){
    return a + b;
}
var partial = add.bind(null, 1);
```

ဒီမှာ parameter 2 ခုလိုတာကို ၁ခုပေးလိုက်တယ် အဲ့ကောင်သည် partial application ပဲ။ partial(2) ဆိုရင် ၃ ရမယ်။ ဘာကွာလဲမေးရင် currying က function သည် exactly 1 argument ပဲလက်ခံတဲ့ function ကို return ပြန်ရမယ် partial application ကတော့ လိုတဲ့ ကျန်နေတဲ့ argument အရေအတွက် ကြိုက်သလောက် လက်ခံတဲ့ function return ပြန်နိုင်တယ်။

Practical Functional Programming Series(Part 7)

Compose & Pipe

Functional programming ရဲ့အဓိက major theme က pure function လေးတွေဆောက်ပြီးတော့မှ အဲ့တခုချင်းစီကို composition, pipeline စတာတွေနဲ့သုံးတာပေါ့။

Composition vs Pipe

သူတို့ နှစ်ခုကတော့ ခပ်ဆင်ဆင်တူတယ်။ main theme ကတော့ function တွေကို အဆင့်ဆင့်ဆောက်သွားတာ။ ဥပမာ function a, b ရှိတယ်ဆိုပါစို့. ဒါဆို a နဲ့ b ပေါင်းပြီး function အသစ်တည်ဆောက်တာ။ အဲ့လိုဆောက်တဲ့နေရာမှာ a ရဲ့ output သည် b အတွက် input ဖြစ်နေတာမျိုး။

JS example နဲ့ဆို အောက်က function ၂ခုရှိတယ်ဆိုပါစို့

```
let upper =(x)=>x.toUpperCase();

let appendHi = (x) => x +" hi";
```

upper က ပေးလိုက်တဲ့ string တခုကို upper case အနေနဲ့ပြောင်းမယ်။ appendHi

ကကျတော့ ပေးလိုက်တဲ့ string တခုကို hi ဆိုတာ ထပ်ထဲ့မယ်။ ဒီနှစ်ခုမှာ တူတာတခုက ဘာလဲဆိုတော့ input parameter တခုစီပဲလက်ခံတယ်။ နောက် return ပြန်တယ်ပေါ့။ သူလဲ တခုပဲ။ JS မှာ function တွေသည် return value တခုပဲပြန်လို့ရတယ်။ Go မှာဆိုရင်တော့အများကြီးရတယ်။

ဒါဆို compose ဆိုတာရေးကြည့်ရအောင်

let compose= (funA,funB)=> x=> funA(funB(x));

ဒီလိုရမယ်။ အဓိက compose သည် funcA,funcB ဆိုတဲ့ function ၂ခုကို parameter အနေနဲ့လက်ခံမယ်။ (ဒီနေရာမှာ compose သည်ဴ theory အရ function အများကြီးလက်ခံလို့ရတယ် code ရှင်းအောင် ၂ခုပဲလက်ခံပြထားတာ။ ပထမဆုံး funB(x) ဆိုပြီးခေါ်တယ် ။ သူကရလာတဲ့ output or return value ကို funA ဆိုပို့တယ်။ ဆိုချင်တာက compose , pipe ဆိုတာတွေရဲ့အဓိက သဘောသည် function လေးေတွေချင်း ပိုက်တွေဆက်သလိုဆက်သွားမယ်။ အဓိက တခုရဲ့ output ကိုနောက်တခုက input အနေနဲ့ အဆင့်အဆင့်သုံးသွားမယ်။ ဒါမို့ currying လိုတာ။ မဟုတ်ရင် function တခုက input ၂ခုလက်ခံတယ်ဆိုရင် ခုနကလို ချိတ်ဖို့အဆင်မပြေတော့ဘူး။ ဒါဆို compose သုံးကြည့်ရအောင်။

```
let appendHiAndUpper = compose(upper,appendHi);

console.log( appendHiAndUpper("how are you "));
```

ဒါဆို log မှာ HOW ARE YOU HI ဆိုပီးထွက်လာမယ်။ ဒီနေရာမှာ appendHiAndUpper သည် upper နဲ့ appendHi ပေါင်းထားတဲ့ function ပဲ။ compose လုပ်ထားတာလို့ပြောရမယ်။တဆင့်ခြင်းအနေနဲ့ဒီလိုလုပ်သွားတာ။

ပထမဆုံး “how are you” ဆိုတဲ့ input ကို appendHiAndUpper ကိုပေးတယ်။ appendHiAndUpper သည် upper နဲ့ appendHi compose ထားတာ။

compose မှာ ညာဘက်အဆုံးက appendHi ကို ခေါ်တယ်။ () တွေအရ ။ appendHi(“how are you”) ဆိုတော့ “how are you hi” ဆိုတာရမယ်။ အဲ့ဒီ ကောင်ကို upper(“how are you hi”) ဆိုပြီးပေးတော့ “HOW ARE YOU HI” ဆိုပြီး ရမယ်။

ဒီနေရာမှာ compose ရဲ့အသုံးဝင်ပုံက function တွေသည် imperative, OO လိုအသေချိတ်ထားတာမဟုတ်ပဲ။ ပြန် combine လုပ်ပစ်လိုက်တာ။ ဘာကောင်းလဲမေးရင် ခုနကလို အဆင့်အဆင့်တည်ဆောက်ပြီး abstraction အနေနဲ့ ခွဲထုတ်လို့ရတယ် ပြန်ပေါင်းလို့ရတယ်။

Compose ကိုသက်သက်ရေးစရာမလိုဘူး အဲ့လိုပဲ currying ကိုလဲလိုက်ရေးစရာမလိုတဲနည်းရှိတယ်။ Functional language တွေမှာဆို builtin ပေးထားတာ။ JS ဖို့ကျတော့ Ramda လို library တွေလိုတယ်။ Ramda ကိုသုံးရင် အောက်ကကောင်လိုရေးလို့ရတယ်။

```
let newFunc = R.compose(upper,appendHi);
console.log( newFunc("how are you "));
```

Compose ရဲ့ပြဿနာက နောက်ဆုံးက အရင်လုပ်မဲ့ကောင်ဖြစ်နေတယ်။ ဖတ်ရခက်တယ် အဆင်မပြေဘူးပေါ့။ တချို့နေရာတွေမှာတော့ သုံးသင့်တယ်။ ဒါပေသိ လုပ်မဲ့ကောင်ကို ရှေ့ဆုံးကနေ ထားချင်တယ် left to right compose လိုလုပ်ချင်တယ်ဆိုရင် pipe ကိုသုံးလို့ရတယ်။ Ramda နဲ့ဆိုဒီလိုရေးလို့ရတယ်။

```
let pipeFunc = R.pipe(appendHi, upper);
console.log("use pipe ", pipeFunc("how are you "));
```

pipe က compose နဲ့သဘောတရားချင်းဆင်တယ် ဒါပေသိ parameter ထားတဲ့အခါ

အရင်ခေါ်မဲ့ function ကိုအရင်ရေးတယ် အဲ့တော့ဖတ်ရတာ လွယ်တယ်။ သူတို့ခုသက်သက်ဆီ သုံးတဲ့ usecase တွေတော့ရှိတယ်။

ဒါတွေက JS အတန်းမှာသင်တာ code က ဒီမယ်။

https://github.com/.../blob/master/chapter8/fp/compose.html

Practical Functional Programming Series(Part 8)

Currying in F# and Ramda

Functional programming language တွေမှာ currying, compose, pipe စတာတွေက built in ပေးထားတယ်။ဥပမာ F# မှာဆို function တိုင်းက curry ဖြစ်ပြီးသားပဲ။ အောက်က code ကိုကြည့်ရအောင်။

let add x y = x + y

let addTwo = add 2

F# မှာ function တွေကို keyword ဘာညာမလိုဘူး let နဲ့တန်းဆောက်လို့ရတယ် let add x y ဆိုတဲ့သဘောက add သည် function ဖြစ်တယ် x y သည် parameter လို့ပြောတာ။ = နောက်က x+y ဆိုတာက function က return ပြန်မဲ့ value တွေ။ အဲ့အောက်မှာ နောက် addTwo မှာ ဘာလုပ်လဲဆိုတော့ add 2 ဆိုပြီး add ဆိုတဲ့အပေါ်က function ကို parameter 1 ခုပေးလိုက်တယ် (ဒီနေရာမှာ 2 ပေါ့) အဲ့တော့ addTwo သည် partial function ဖြစ်သွားပြီ။

နောက် သူ့ကိုဒီလိုခေါ်လို့ရမယ်။

```
printfn "AddTwo to    5 => %A" (addTwo 5)

printfn "AddTwo to    2 =>%A =>" (addTwo 2)
```

addTwo 5 ဆိုတာ ခုနက partial function addTwo ကို နောက်လိုတဲ့ parameter တခုပေးလိုက်တာအဲ့တော့ 7 ရမယ် နောက်တကြောင်းကတော့ addTwo 2 ဆိုတော့ 4 ရမယ်။ JS လို language မှာကျတော့ currying မပါသေးဘူး အဲ့တောဘာလုပ်လို့ရသလဲဆိုရင် Ramda လို library တွေသုံးလို့ရတယ်။ ဒီလိုပေါ့။

```
let add= (x,y) => x+y;

console.log("Add ",add(1,2));

let addCur = R.curry(add);

console.log("Add Curry ",addCur);

let add4 = addCur(4);

console.log("Add 4+1 ",add4(1));

console.log("Add 4+2 ",add4(2));
```

add က parameter ၃ ခုလက်ခံတဲ့ function အဲ့တာကို curry လုပ်ချင်ရင် R.curry method ကိုသုံးလိုက်တယ် ဒါဆိုသူက curry version return ပြန်လိမ့်မယ်။ Ramda ကတခုကောင်းတာက parameter တွေကို တခုချင်းပေးယုံတင်မကဘူး အများကြီးလဲတန်းပေးလို့ရတယ်။ ဥပမာ ဒါမျိုးတွေရမယ်။

addCur(2,4)

addCur(2)(4)

အပေါ်က example မှာ let add4 = addCur(4); ဆိုပြီး parameter တခုပဲပေးပြီး add4 ဆိုတဲ့ partial function တခုဆောက်တယ်။ နောက်ကျတော့ add4(1) ဆိုပြီး နောက်လိုတဲ့ parameter တခုပေးလိုက်ရင် 5 ဆိုပြီးရမယ်။

Realworld မှာ ဥပမာ api တခုရှိတယ်ဆိုပါစို့.

function api(url,path)

{

}

အဲ့ကောင်ကို curry ထဲထဲ့ပြီး ပထမ url တခုပဲပေးမယ်။ နောက် တော့်မှလိုအပ်တဲ့ path ကိုပေးမယ်ဆိုရင် ပိုရှင်းပြီး specific version ရသွားတာပေါ့။ နောက် React မှာ action creator တွေမှာ action type နဲ့ item ပေးရတာမျိုးမှာ ခုနက currying ကိုသုံးပြီး specific version တွေဆောက်လို့ရမယ်။

Code ကတော့ JS FP သင်ခန်းစာက ဒီမှာ။

https://github.com/.../chapter8/fp/curry_with_ramda.html

Practical Functional Programming Series(Part 9)

What is category theory

Object Oriented Programming ကျတော့ ခနခနဖြစ်လေ့ရှိတဲ့ solve ရလေ့ရှိတဲ့ design problemတွေအပေါ်မူတည်ပြီး GoF design pattern လိုကောင်မျိုးနဲ့ solve လုပ်ကြတယ်။

Functional programming မှာကျတော့ category theory ကကောင်တွေကိုသုံးတယ်။ FP design pattern ဆိုတဲ့သဘောပေါ့။

Category theory ဆိုတာ Math ကလာတာ။ Category theory ဆိုတာ mathematical structure တွေကိုလေ့လာတာကိုဆိုတာ။

Mathematical structure ဆိုတာက set ကိုပြောတာ set ကိုမှ set အပေါ်မှာလုပ်လို့ရတဲ့ operation တွေ (ဥပမာ set က real number ဆိုပါစို့ operation ဆိုတာ +,-,*/ ဒါမျိုး),

relation တွေ (Caterisan Product လိုကောင်တွေ) metric, topology စတာတွေ လုပ်လို့ရတဲ့ set ကိုဆိုချင်တာ။

အဓိက functional programming က function တွေသည် math က function လိုအလုပ်လုပ်တယ်။

mapping လုပ်တယ်ပြောရမှာပေါ့။ 8 တန်းသခ်ၤာသင်ဖူးရင်သဘောပေါက်လိမ့်မယ်။

f:A->B ဆိုပါစို့. f က function ကိုဆိုချင်တာ။ A သည် domain လို့ပြောတယ်။ B သည် codomain ဆိုချင်တာက programming အရဆို A သည် function f ကိုပေးရတဲ့ input

အစုဝေး(set), B သည် f က ရတဲ့ output(return value) အစုအဝေး(set)

ဥပမာ

let square =(x) = x * x ဆိုပါစို့

ဒါဆို square သည် input x (number domain) ကနေ output x*x (coddomain ကို) map ပေးတာ။

function ဆိုတာ input(domain set) ကနေ output(codomain set) ကို mapping လုပ်ပေးတာပဲ။

Category theory ကဘာတွေအရေးပါလဲဆိုတော့ Functional programming မှာအဓိက main theme က composition ပေါ့။

composition အကြောင်းအရင်ရရေးထားပြီးသားပေါ့။ အဓိက multiple function တွေကို single function ဖြစ်အောင် တည်ဆောက်ယူတာမျိုးပေါ့။

function သေးသေးလေးတွေကို စုပြီး functionality တူတဲ့ new function တည်ဆောက်ယူတာမျိုးပေါ့။

အဲ့လို compose လုပ်တဲ့အခါ law တွေလိုက်နာမှ အဆင်ပြေမယ်။ ဥပမာ single input, single output ရှိမှ compose လုပ်လို့အဆင်ပြေတာမျိုး။

ဒါတွေကို Category theory က ကောင်တွေကနေ အထောက်ပံ့ပေးနိုင်တယ်။

ဒါက စာဆန်ဆန်ပြေတာ လက်တွေ့မှာသုံးနေတဲ့ Cateogry theory ကကောင်တွေရှိတယ်။

ဥပမာ Java ဘက်က Optional ဆိုတာမျိုးက MayBe functor နဲ့တူတယ်(Functor ဆိုတာ

Category theory အောက်ကဟာတခု အဓိက သူမှာ Map ဆိုတဲ့ function ပါရမယ် ဒါဆို Functor လို့ပြောလို့ရပီ)

ဘာလို့ Optional ကိုသုံးလဲဆိုရင် imperative က null value စစ်တာမျိုး မစစ်ပဲနဲ့ chain of operation (map တွေ အဆင့်အဆင့်လုပ်လို့ရအောင်သုံးချင်လို့) ဒီလိုနေရာမျိုးဆို Functor ကအသုံးဝင်တယ်ပေါ့။

နောက်တခု Monad ဆိုတာမျိုး။ ဥပမာ JS က Promise သည် Monad နီးပါးပေါ့(Theory အရ Monad law အကုန်ကိုတော့ မလိုက်နာဘူး ဒါမို့နီးပါးလို့ပြောတာ)။

Monad သည် Functor ဖြစ်ရမယ်သူမှာ chain ဆိုတဲ့ function ပါရမယ်။ အဲ့တော့ Promise ကိုသုံးရင် map တွေဆင့်ခေါ်လို့ရတယ်(functor မို့လို့) နောက် error handling ဘာညာ delayed computation စတာတွေ ပိုအဆင်ပြေသွားတယ်။

ဒါသည် realworld က Category Theory ကိုသုံးပုံပေါ့။

Category ရဲ့ definition ကို Benjamin C.Pierce - Basic Category Theory for Computer Scientist ရဲ့စာအုပ်ကနေ ယူထားတာ။

Rule 5 ခုရှိတယ်

1. a collection of objects;

ဒီနေရာမှာ Cateogry သည် object ဖြစ်ရမယ်။ Object ဆိုတာ OOP က object တွေဆိုလိုတာမဟုတ်ဘူး။

အားလုံးဖြစ်နိုင်တယ် ဥပမာ number တွေ char တွေ set တွေ human စတာတွေ အကုန်ပေါ့။

Set theory အရကြည့်ရင် set ထဲပါနိုင်တဲ့ element တွေပေါ့။

2. a collection of arrows (often called morphisms);

Category theory အရ function ကို arrow လို့သုံးတယ်။ Morphism လို့လဲခေါ်တယ်။

3. operations assigning to each arrow f an object dom f, its domain,

and an object cod f,its codomain(we write f:A-> B or to show that dom f = A and cod f = B; the collection of all arrows with domain A and codomain B is written C(A, B));

ဒါက mapping ကိုပြောတာ function ပေါ့။ input ကို domain ခေါ် ပြီး output ကို codomain လို့ခေါ်တာ။

4. a composition operator assigning to each pair of arrows f and g, with cod f =dom g,a composite arrow g of :dom f -> cod g, satisfying the following associative law:

for any arrows f : A -> B, g : B -> C, and h : C -> D (with A, B, C, and D not necessarily distinct),

h . (g . f)=(h . g) . f;

Composition operator ရှိရမယ် composition ကိုမြင်အောင်ပြောရရင်

cow => milk (ဒါက နွားကနေ နွားနို့ထုတ်ပေးတဲ့ function f ပေါ့ f: A->B လို့မှတ်မယ် A သည် cow, B သည် နွားနို့)

milk => Yoghurt (ဒါက နွားနို့ကနေ ဒိန်ချဉ်ထုတ်ပေးတဲ့ function g ပေါ့ g: B->C B

သည်နွားနို့ C သည် Yoghurt)

Yoghurt => Yoghurt Ice Cream (ဒါက ဒိန်ချဉ်ကနေ ဒိန်ချဉ် ice cream ထုတ်ပေးတာပေါ့

h: C->D C သည် Yoghurt D သည် Yoghurt ice cream)

ဒါဆို ခုနက သုံးခုကို ဒီလို compose လုပ်လို့ရမယ် တခုသတိထားရမှာက f output B(milk)သည် g ရဲ့ input ဖြစ်တယ်။ g ရဲ့ output သည် h ရဲ့ input ဖြစ်တယ်။

Composition ကဘာနဲ့အလားတူမလဲဆိုတော့ ခုနက နွားကနေ Yoghurt ice cream ထုတ်ပေးတဲ့ စက်ရုံနဲ့တူမယ်ပေါ့။

Function ၃ ခုကိုပေါင်းပြီး တခုတည်းလုပ်လို့ရမယ်ပေါ့။

နောက် associative law ဆိုတာ သခ်ၤာမှာ a+b+c ဆိုပါစို့ (a+b)+c က a+(b+c) နဲ့တူတူပဲ ဆိုချင်တာက a+b အရင်ပေါင်းပေါင်းဒါမှမဟုတ် b+c ကိုအရင်လုပ်လုပ်အဖြေတူတယ်။

ဒါမျိုးကို associative ဖြစ်တယ်ပေါ့။ number တွေကို category လို့ယူဆမယ် + သည် mapping (function ဖြစ်မယ်ဆိုရင်) number တွေ သည် associative law ကိုကိုက်တဲ့အတွက် Category လို့ပြောလို့ရတယ်ပေါ့။

(နွားကနေ Yoghurt Ice Cream ထုတ်ဖို့ associative ကိုစဉ်းစားရင်နည်းနည်းတော့လွဲနေမယ်)

5. for each object A, an identity arrow id A : A -> A satisfying the following identity law:

for any arrow f: A-> B,

id . f=f and f . idA=f.

နောက်ဆုံး law က Category တိုင်းမှာ identity function ရှိမယ်။ ဥပမာ

let identity= (x) => x;

ဘာပေးပေး ပေးတဲ့ input ပြန်ထုတ်ပေးတဲ့ function ကို identity function လို့လဲခေါ်တယ်။

ဥပမာ သန်ဂျာမှာဆို အပေါင်းဖို့ဆိုရင် 0 သည် identity function သဘောဆောင်တာပေါ့။

သူ့ဘာနဲ့ပေါင်းပေါင်း နဂို တန်ဖိုးရတာမို့လား။

အပေါ်ကအချက်တွေပြည့်စုံရင် Category လို့ပြောလို့ရပြီ.

Practical Functional Programming Series(Part 10)

Monoid & Semigroup

Monoid ဆိုတာ Category theory ကလာတာ။

Functional programming မှာ GoF လို OO pattern တွေမရှိဘူး။

OO မဟုတ်တော့ဘယ်လာ OO pattern တွေရှိမလဲ။ အဲ့အစား category theory ကကောင်တွေကိုသုံးတယ်။

အောက်က Monoid definition ပဲ။

A monoid is a set, S, together with a binary operation "•" (pronounced "dot" or "times") that satisfies the following three axioms:

သူမှာအောက်က ၃ ချက်ကိုလိုက်နာရတယ် ဒါဆို Monoid ဖြစ်ပြီ။

Closure

For all a, b in S, the result of the operation a • b is also in S.

Associativity

For all a, b and c in S, the equation (a • b) • c = a • (b • c) holds.

Identity element

There exists an element e in S, such that for all elements a in S, the equation e • a = a • e = a holds.

Closure

တကယ်တမ်းက monoid ဆိုတာ element ၂ခုအပေါ်မှာ binary operation လုပ်တာ။ result type ကလဲ element တွေရဲ့ type ပဲဖြစ်ရမယ်။

Binary operation ဆိုတာ + တို့ * တို့လို operand 2 ခုလိုတဲ့ကောင်တွေကိုပြောတာ။

+ တို့ * တို့သည် operand 2 ခုလက်ခံတယ် ပြီးရင် တခုကို result အနေနဲ့ထုတ်ပေးတယ်။

Monoid လို့ပြောလို့ရတယ်။

Closure property(ဒီနေရာမှာ JS က closure ကိုပြောတာမဟုတ်ဘူး)က ဘာတွေအသုံးဝင်လဲဆိုတော့ element ၂ခုကနေ ၁ခုဖြစ်အောင် reduce, fold လုပ်လို့ရတယ်။ အဲ့တော့ multiple element ဥပမာ list လို array လိုကောင်တွေကို operation တခုတည်းနဲ့

လုပ်လို့ရတယ်။

ဥပမာ functional programming က compose ဆိုတဲ့ function လိုပေါ့ function 2 ခုကို တခုဖြစ်အောင်လုပ်လို့ရတယ်။

ဒါကိုဆွဲဆန့်ရင် compose(compose(f1,f2),f3) ဒီလိုလုပ်လို့ရတယ်။

အဲ့တာဆို function တွေအများကြီး array လိုက်ကို compose လုပ်လို့ရတယ်ဆိုတဲ့သဘောပဲ။

ဆိုချင်တာက element အများကြီး ဥပမာ list, array လိုကောင်တွေကို operation တခုသုံးပြီး လုပ်လို့ရတယ်။

Function composition မှာအသုံးဝင်တယ်ပေါ့။

Associativity

ဒါကတော့ a+b+c ဆိုပါစို့ ။ သူ့ကို (a+b)+c လို့ပဲလုပ်လုပ်။ a+(b+c) လို့ပဲလုပ်လုပ် ပြောင်းလဲမှုမရှိရဘူးအဖြေတူရမယ်။

ဒါကိုပြောတာ ဥပမာ + နဲ့ * သည် ဒီ property ကိုပြေလည်တယ်။

နောက် string concatenation , list concatenation တွေသည်လဲ ဒါကိုပြေလည်တယ်။

ဘာအသုံးဝင်တာလဲဆိုတော့ functional programming က မြန်တယ်ဆိုတာမျိုးကြားဖူးပါလိမ့်မယ်။

ဘာလို့မြန်သလဲဆိုတော့ ခုနက list တွေမှာ ဒီလို property တွေရှိလို့၊

ဥပမာ list က element 100 ရှိတယ်ဆိုပါစို့. ဒါကို element 50 စီ core

တခုချင်းစီမှာခွဲလုပ်မယ်။

ပြီးမှ ပြန်ပေါင်းမယ် ဒါဆို အဲ့လိုလုပ်လို့ရတယ်။ အဲ့တော့ parallelization လုပ်လို့ရတယ်ပေါ့။

နောက် divide & conquer algorithm တွေမှာသုံးလို့ရတယ်။ အဲ့တော့မြန်တယ်ပေါ့။

လက်တွေဆို Map Reduce framework မှာ ဒါမျိုးကြောင့် Parallelization လုပ်နိုင်လို့ large scale ကိုအဆင်ပြေတာ။

Identity element

Identity element ဆိုတာ သူ့ကိုဘာနဲ့ apply လုပ်လုပ် လုပ်တဲ့ကောင်ပဲပြန်ရတာမျိုး၊

ဥပမာ အပေါင်းဆို 0 ကိုဘာနဲ့ပေါင်းပေါင်း 3+0 = 3 ရသလို နဂိုကောင်ပဲပြန်ရတဲ့ကောင်မျိုး။

အမြောက်ဆို 1 ပေါ့။

ဘာအတွက်သုံးလဲဆိုတော့ ခုနက list, array စတာတွေမှာ empty ဆိုရင်ဘယ်လိုလုပ်မလဲဆိုတာမျိုး အတွက်သုံးတာ။

အောက်က code က + operation ကို monoid အနေနဲ့လုပ်ထားတာ။

binary operation ကို FP category မှာ concat ဆိုပြီးသုံးတယ်။

```
var Sum = { empty: () => 0, concat: (x, y) => x + y }

var data = [1,2,3,4,5,6];

var result = data.reduce(Sum.concat,Sum.empty());

console.log(result);
```

Real world ရှိတဲ့ monoid တွေက string, list, array စတာတွေပေါ့ ခုနက အပေါ်က law တွေကိုလိုက်နာတယ်ဆိုရင်ပေါ့။

အောက်ဆုံးက Identity element မပါပဲ အပေါ်၂ခုပဲကိုက်ရင်တော့ Monoid မဟုတ်တော့ဘူး Semigroup လို့ခေါ်တယ်။

Reference Links

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/pfbid02LK682E6JYUAtSS8bX

hvkudfYeGv9oNdsddbwq36rpu6ZAp6bNCYXuWkendYdM3p8l

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