



Introduction to F#

Basics of Functional Programming



Remark

- For beginners only.

About the Speaker

- Chang Hai Bin
- B.Sc. (**Math**), NUS
- M.Sc. (**Math**), University of Michigan
- (Current) Financial Engineer, Numerical Technologies
- (Previous) Business Analyst, US-Mattress

What is Functional Programming (FP)?

- Based on combinatory logic
- Uses functions to solve problem
 - Functions as objects
 - Higher Order Functions
- Other Good Properties (depend on language)
 - Purity/Type-check/Recursive/Lazy-evaluation/homoiconicity

FP Languages

- Ancestor:
 - ML (1973)
 - Haskell (1990)
- Cousins:
 - OCaml(1996)
 - Scala (2004)
 - F# (2005)
 - Elm (2012)
 - ReasonML (2018)
- Remark: Some also considers LISP (1958) and their dialects (e.g. Clojure 2007) functional languages.

Who uses FP?

Haskell



Clojure / F#



OCaml



Scala



Why Learn FP?

- Concise code
- Ability to reason
- Unlock problems
- Better Salary



This Talk: F#

(Statically Typed)

There are also Scala, Haskell, Clojure, Elm Meetups in Singapore.

Should I switch from C# to F#?

- Early-adopter: Yes!
 - Fun! Easy to Learn! Access other FP languages!
- Normal-adopter: Yes.
 - Use functional-technique in C# code.
- Late-adopter/Skeptics: Maybe.
 - A lot of new features in C# comes from F#.
 - F# has better syntax to learn these concepts.

Core Concept

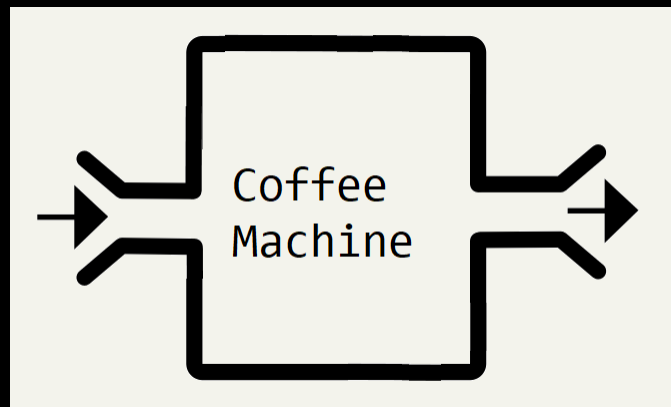
- Functions are things
- Compose functions

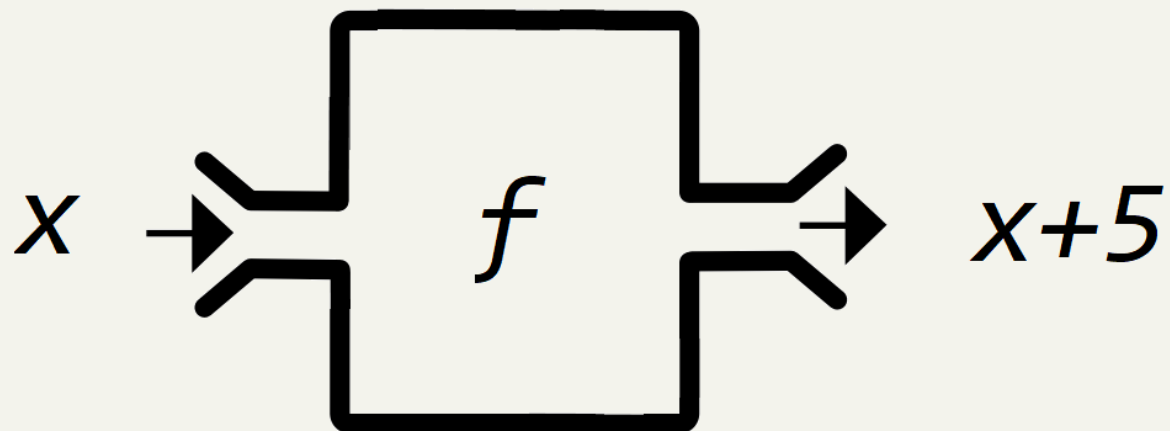


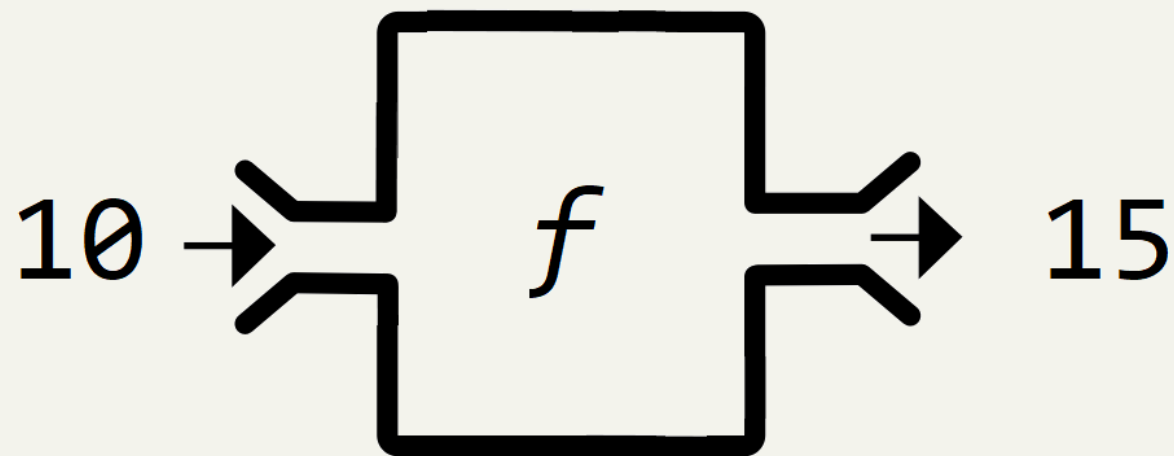


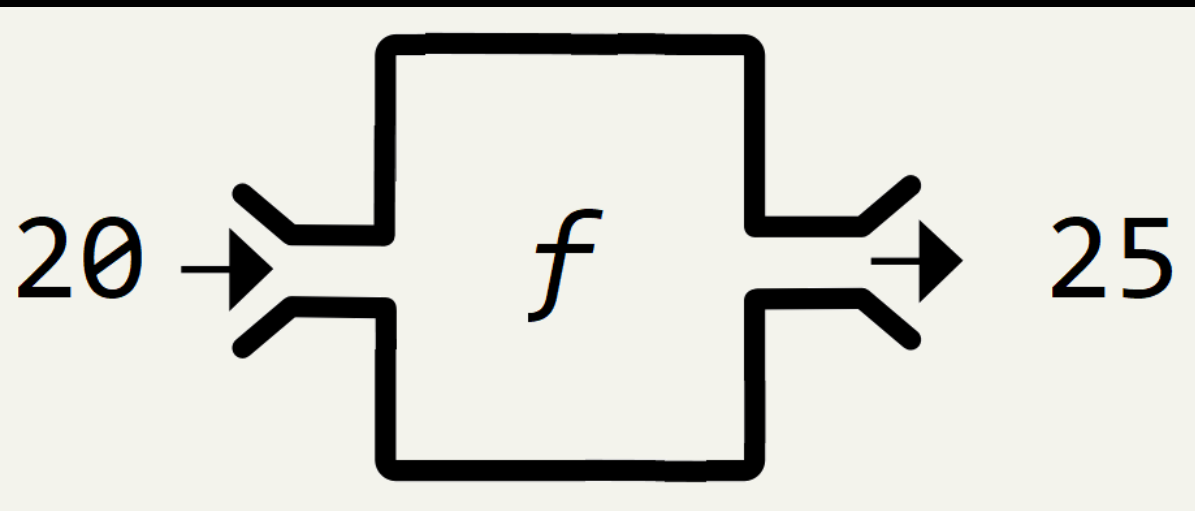
What is a function?

- Function is a machine that **take an input**, and **returns an output**









F# Example

- `let f x = x + 5`
- `f 30` `// 35`
- `f 100` `// 105`

Notation

	Math	F#
Define	$\text{let } f(x) = x + 5$	<code>let f x = x + 5</code>
Use	$f(100)$	<code>f 100</code>



- `let f x = "Hello" + x`

- `f "John" // "Hello John"`

- `f "Jane" // "Hello Jane"`




- `let f x = String.length x`

- `f "Hello" // 5`

- `f "Computer" // 8`

- `f : string -> int`



- `let f xs = List.sum xs`

- `f [1..10] // 55`

- `f [2;3;5;7;11] // 28`

- `f : List<int> -> int`

Multiple inputs

- `let f x y = x + y`
- `f 2 3 // 5`
- `f 30 70 // 100`
- `f : int -> int -> int`

Notation

	Math	F#
Define	$\text{let } f(x,y) = x + y$	<code>let f x y = x + y</code>
Use	$f(30, 70)$	<code>f 30 70</code>

Multiple inputs

- `let f x y z = x + y + z`
- `f 2 3 7 // 12`
- `f 30 70 200 // 300`
- `f : int -> int -> int -> int`

Multiple inputs

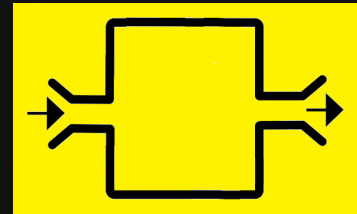
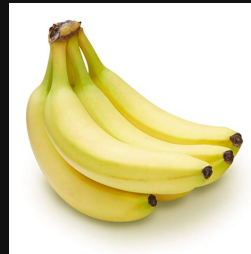
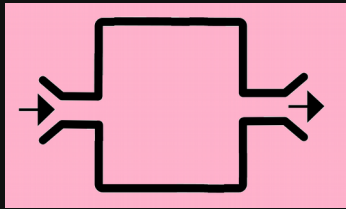
- `let f a b c d =`

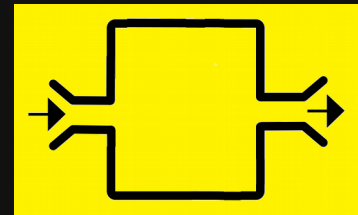
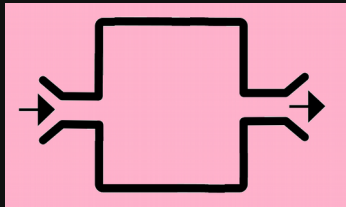
- `f : A -> B -> C -> D -> output`

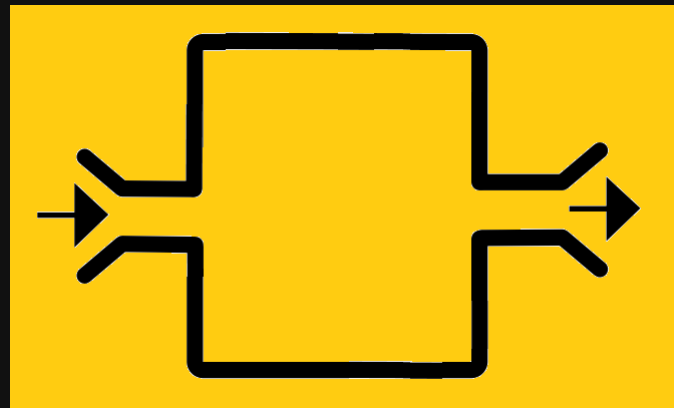
Function Composition

- Functions can be “connected” if the first output is the input of the second function.





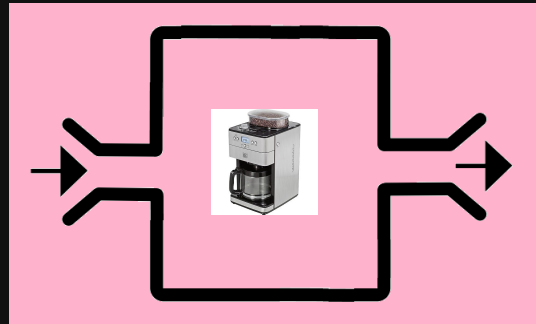




A bigger machine/function!

Banana is “hidden”

Coffee Machine



Programmer



```
for i in people.data.users:
    response = client.api.statuses.user_timeline.get(screen_name=i.scre
    print 'Got', len(response.data), 'tweets from', i.screen_name
    if len(response.data) != 0:
        ldate = response.data[0]['created_at']
        ldate2 = datetime.strptime(ldate, '%a %b %d %H:%M:%S +0000 %Y')
        today = datetime.now()
        howlong = (today - ldate2).days
        if howlong < daywindow:
            print i.screen_name, 'has tweeted in the past', daywindow,
            totaltweets += len(response.data)
            for j in response.data:
                if j.entities.urls:
                    for k in j.entities.urls:
                        newurl = k['expanded_url']
                        urlset.add((newurl, j.user.screen_name))
        else:
            print i.screen_name, 'has not tweeted in the past', daywind
```

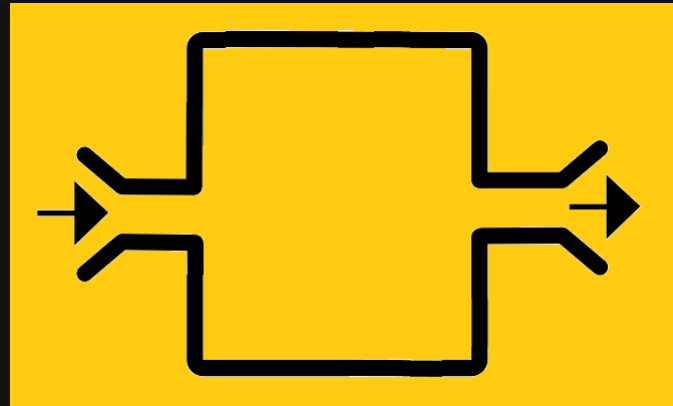
Coffee Machine



Programmer



```
for i in people.data.users:
    response = client.api.statuses.user_timeline.get(screen_name=i.scre
    print 'Got', len(response.data), 'tweets from', i.screen_name
    if len(response.data) != 0:
        ldate = response.data[0]['created_at']
        ldate2 = datetime.strptime(ldate, '%a %b %d %H:%M:%S +0000 %Y')
        today = datetime.now()
        howlong = (today-ldate2).days
        if howlong < daywindow:
            print i.screen_name, 'has tweeted in the past', daywindow,
            totaltweets += len(response.data)
            for j in response.data:
                if j.entities.urls:
                    for k in j.entities.urls:
                        newurl = k['expanded_url']
                        urlset.add((newurl, j.user.screen_name))
            else:
                print i.screen_name, 'has not tweeted in the past', daywind
```



```
for i in people.data.users:
    response = client.api.statuses.user_timeline.get(screen_name=i.scre
    print 'Got', len(response.data), 'tweets from', i.screen_name
    if len(response.data) != 0:
        ltdate = response.data[0]['created_at']
        ltdate2 = datetime.strptime(ltdate, '%a %b %d %H:%M:%S +0000 %Y'
        today = datetime.now()
        howLong = (today-ltdate2).days
        if howLong < daywindow:
            print i.screen_name, 'has tweeted in the past', daywindow,
            totaltweets += len(response.data)
            for j in response.data:
                if j.entities.urls:
                    for k in j.entities.urls:
                        newurl = k['expanded_url']
                        urlset.add((newurl, j.user.screen_name))
        else:
            print i.screen_name, 'has not tweeted in the past', daywind
```

- $f : A \rightarrow B$
- $g : B \rightarrow C$

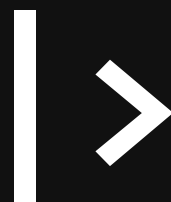
- $g (f a)$
 $// = g (b)$
 $// = c$

- `let f xs = List.sum xs`
- `let g x =
 if x > 80 then "A"
 else if x > 60 then "B"
 else "C"`
- `f : List<int> -> int`
- `g : int -> string`

- `let f xs = List.sum xs`
- `let g x =
 if x > 80 then "A"
 else if x > 60 then "B"
 else "C"`
- `g (f [20; 20; 50]) // g (90)
// "A"`

Types must match

- $f : A \rightarrow B$
- $g : C \rightarrow D$
- $g (f a)$ **ERROR!**
- Output of f not accepted by g



Key idea in F#

Pipe-forward operator

What does pipe-forward do?

- Change the order of the function and input



- `let f x = x + 5`

- `f 100 // 105`

- `100 |> f // 105`

- $f : A \rightarrow B$
- $g : B \rightarrow C$

- $g(f\ a)$

- $a \mapsto f \mapsto g$

- $f : A \rightarrow B$
- $g : B \rightarrow C$

- $g(f\ a)$

- a

$|> f$

$|> g$

- $f : A \rightarrow B$
- $g : B \rightarrow C$

- $g (f \ a)$

- a

(then do) f

(then do) g

- x
|> f
|> g

- Start with input x,
Apply input to f,
Apply previous result to g.



- x

|> f

|> g

|> h

- Start with input x,
Apply input to f,
Apply previous result to g,
Apply previous result to h.



- x

|> f

|> g

|> h

|> k

- Start with input x,

Apply input to f,

Apply previous result to g,

Apply previous result to h,

Apply previous result to k.

- x

|> f

|> g

|> h

|> k

*1st output = 2nd input
2nd output = 3rd input
etc.*

- Start with input x,

Apply input to f,

Apply previous result to g,

Apply previous result to h,

Apply previous result to k.

- x

|> f

|> g

|> h

|> k

- In C#:
- `x.Pipe(f).Pipe(g).Pipe(h).Pipe(k);`

- You can do it in C#, but not as natural.




Benefit

- Express Logic Step-by-Step
- Easier to read

Example

- Questions from Project Euler
- <https://projecteuler.net/>



- 
- (Question 1) Find the sum of all the multiples of 3 or 5 below from 1 to 999.

- (Question 1) Find the sum of all the multiples of 3 or 5 below from 1 to 999.

- ```
[1 .. 999]
|> List.filter (fun x -> x % 3 = 0 || x % 5 = 0)
|> List.sum
```

- (Question 1) Find the sum of all the multiples of 3 or 5 below from 1 to 999.

- ```
[1 .. 999]  
|> List.filter (fun x -> x % 3 = 0 || x % 5 = 0)  
|> List.sum
```

- Start with a list from 1 to 999
- (then do) filter to keep the numbers you want
- (then do) sum those remaining numbers.

- (Question 1) Find the sum of all the multiples of 3 or 5 below from 1 to 999.

- ```
[1 .. 999]
|> List.filter (fun x -> x % 3 = 0 || x % 5 = 0)
|> List.sum
```

- C# LINQ
- ```
Enumerable.Range(1, 999)  
.Where(x => x % 3 == 0 || x % 5 == 0)  
.Sum();
```

- (Question 6 Modified)

Calculate $1^2 + 2^2 + \dots + 100^2$

- (Question 6 Modified)

Calculate $1^2 + 2^2 + \dots + 100^2$

- ```
[1 .. 100]
|> List.map (fun x -> x * x)
|> List.sum
```

- (Question 6 Modified)

Calculate  $1^2 + 2^2 + \dots + 100^2$

- `[1 .. 100]`  
|> `List.map (fun x -> x * x)`  
|> `List.sum`
- Start with a list from 1 to 100  
(then do) convert each element to its square  
(then do) sum up the previous list.

- (Question 6 Modified)

Calculate  $1^2 + 2^2 + \dots + 100^2$

- `[1 .. 100]`  
|> `List.map (fun x -> x * x)`  
|> `List.sum`
- C# LINQ:
- `Enumerable.Range(1,100)`  
  `.Select(x => x * x)`  
  `.Sum();`



- (Additional Example)
- Calculate Squares of Prime Numbers
- Calculate  $2^2 + 3^2 + 5^2 + 7^2 + 11^2 + 13^2 + 17^2 + \dots + 97^2$

- (Additional Example)
- Calculate Squares of Prime Numbers
- Calculate  $2^2 + 3^2 + 5^2 + 7^2 + 11^2 + 13^2 + 17^2 + \dots + 97^2$

- `[1 .. 100]`  
|> `List.filter (fun x -> isPrime x)`  
|> `List.map (fun x -> x * x)`  
|> `List.sum`

*Need “isPrime”  
Helper Function*



# Partial Application

*Useful Language Design*



- `let AddAll w x y z = w + x + y + z`

- `let result = AddAll 1 2 3 4`


`// result = 10`



- `let AddAll w x y z = w + x + y + z`

- `let result = AddAll 1 2 3`

`// Missing one variable?`

- 
- `let AddAll w x y z = w + x + y + z`
  - `let result = AddAll 1 2 3`

`// No compilation error.`

`// result : int -> int`



If a function/machine:

- Needs 5 inputs
- But only 2 inputs provided,
- Still needs 3 additional inputs.



If a function/machine:

- Needs 5 inputs
- But only 2 inputs provided,
- Becomes a brand new function/machine that needs 3 inputs.





- `let f u v w x y = .....`

- `let result = f u v`

`// result : W -> X -> Y -> output`

# in C#

- `public static int Add (int x, int y){  
 return x + y;  
}`

- `Add(1);`

**Compile ERROR!**

# Special Case (n - 1)

- $f \ u \ v \ w \ a = b$
- $g \ x \ y \ z \ b = c$
- $f \ u \ v \ w : a \rightarrow b$
- $g \ x \ y \ z : b \rightarrow c$
- $a$ 
  - $|> f \ u \ v \ w$
  - $|> g \ x \ y \ z$

# Special Case (n - 1)

- $f \ u \ v \ w \ a = b$
- $g \ x \ y \ z \ b = c$
- $f \ u \ v \ w : a \rightarrow b$
- $g \ x \ y \ z : b \rightarrow c$
- $a$ 
  - $|> \boxed{f \ u \ v \ w}$
  - $|> \boxed{g \ x \ y \ z}$

*Assemble almost everything  
except the final component*

# Special Case (n - 1)

- [1 .. 100]  
|> List.filter (fun x -> x % 3 = 0 || x % 5 = 0)  
|> List.map (fun x -> x \* x)  
|> List.sum

# Special Case (n - 1)

- [1 .. 100]

```
|> List.filter (fun x -> x % 3 = 0 || x % 5 = 0)
```

```
|> List.map (fun x -> x * x)
```

```
|> List.sum
```

*Assemble almost everything  
except the final component*

# Special Case (n - 1) "this"

- `public static B1 f(this A1 a1, A2 a2, A3 a3){.....}`
- `public static C1 g(this B1 b1, B2 b2, B3 b3){.....}`
- `public static D1 h(this C1 c1, C2 c2, C3 c3){.....}`

- `f: A1,A2,A3 -> B1`
- `g: B1,B2,B3 -> C1`
- `h: C1,C2,C3 -> D1`

# Special Case (n - 1) “this”

- a1
  - .f(a2, a3)
  - .g(b2, b3)
  - .h(c2, c3);
- f: A1, A2, A3 -> B1
- g: B1, B2, B3 -> C1
- h: C1, C2, C3 -> D1



- a1

- .f(a2, a3)

- .g(b2, b3)

- .h(c2, c3);



- f(a1, a2, a3)

- .g(b2, b3)

- .h(c2, c3);



- h(g(f(a1, a2, a3), b2, b3), c2, c3);



- g(f(a1, a2, a3), b2, b3)

- .h(c2, c3);

# “Currying”

- `Func<A, Func<B, Func<C, Func<D, Z>>>>`
  - Flexible
- 
- `Func<A, B, C, D, Z>`
  - Not flexible (need to assemble everything)

- Func<A, Func<B, Func<C, Func<D, Z>>>>
- Func<A, Func B, Func<C, Func<D, Z>>> >
- Func<A, Func<B, Func<C, Func<D, Z>>>>>
- Func<A, Func<B, Func<C, Func >>>>

# “Currying”

- `Func<A, Func<B, Func<C, Func<D, Z>>>> Curry  
(Func<A, B, C, D, Z> f) {  
 return a => b => c => d => f(a, b, c, d);  
}`
- `var g = Curry(f);`
- `g(a)` `Compiles!`
- `g(a)(b)` `Compiles!`
- `g(a)(b)(c)` `Compiles!`



# Higher Order Functions

*Function as inputs*

# Primitive Types

- `public double f(double a, int b, string c)`  
`{.....}`
- Basic data types as inputs/outputs

# Functions as input

- `public double f(Func<double,int> g, string c)`  
`{.....}`

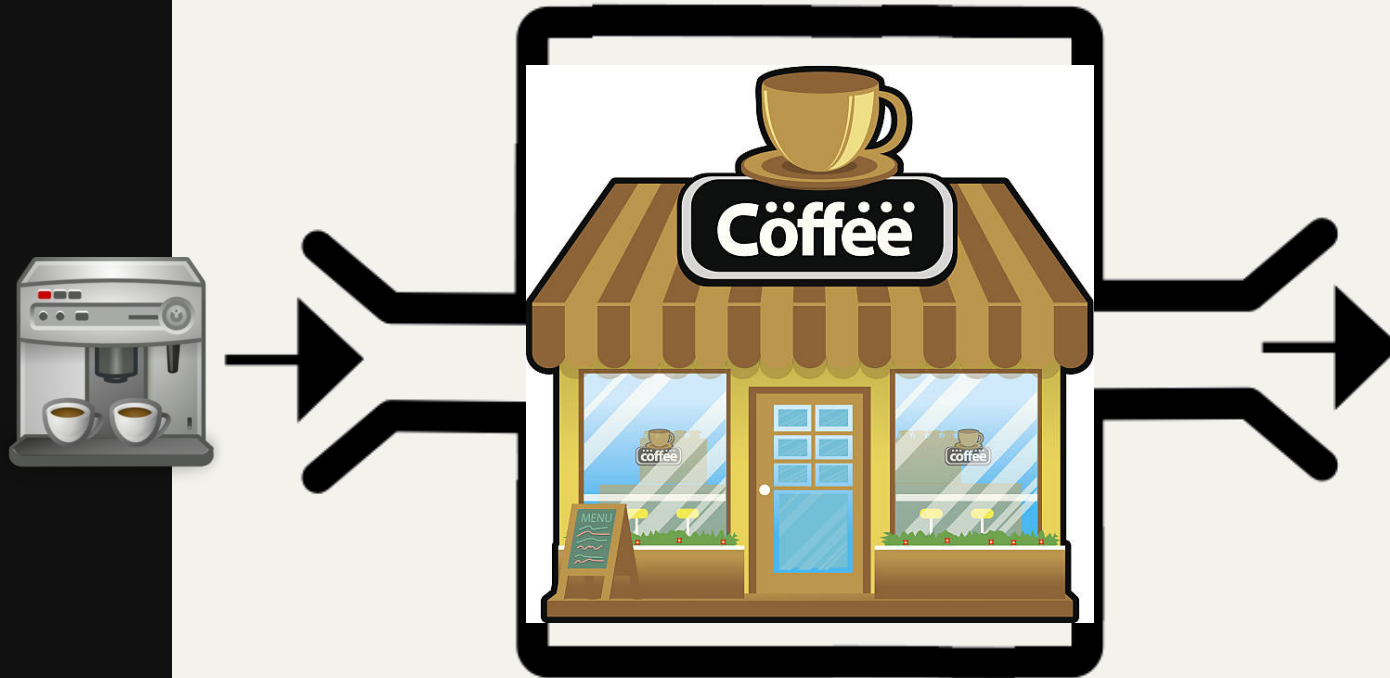
- Function “f” accepts another function “g” as input.

# Functions as output

- `public Func<int, double> f(double a, int b, string c){.....}`

- Function “f” returns another function as output.



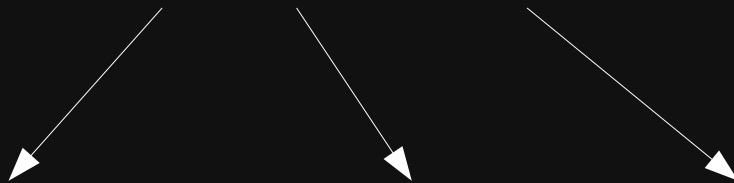


# filter, map

- [1 .. 999]  
|> List.filter (fun x -> x % 3 = 0 || x % 5 = 0)  
|> List.sum
- [1 .. 100]  
|> List.map (fun x -> x \* x)  
|> List.sum

# Filter

- `let filter f xs = .....`



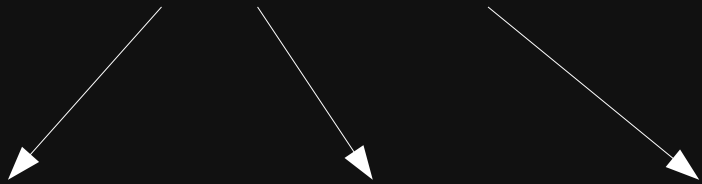
- `(X -> bool) -> List<X> -> List<X>`

- `List<X> filter(Func<X,bool> f, List<X> xs)`

- `LINQ.Where`

# Map

- `let map f xs = .....`



- `(X -> Y) -> List<X> -> List<Y>`

- `List<Y> map(Func<X,Y> f, List<X> xs)`

- `LINQ.Select`

# Insurance Pricing Example

- How much to charge a customer for an insurance product?
- `let Price = .....`



- `let Price =`  
    `.....`

- 
- e.g. Depends on Age.

- `let Price age =`

`.....`

- `age : int`

- `int -> $$$`

- e.g. Depends on probability of injury.

- `let Price age prob =`

`.....`

- `age : int`

- `prob: double`

- `int -> double -> $$$`



- What if the probability depends on **time**?

- `let Price age prob =`

`.....`

- `age : int`

- `prob: ???????`

- `int -> ??????? -> $$$`

- Pass in a function

- `let Price age probFunc =`

`.....`

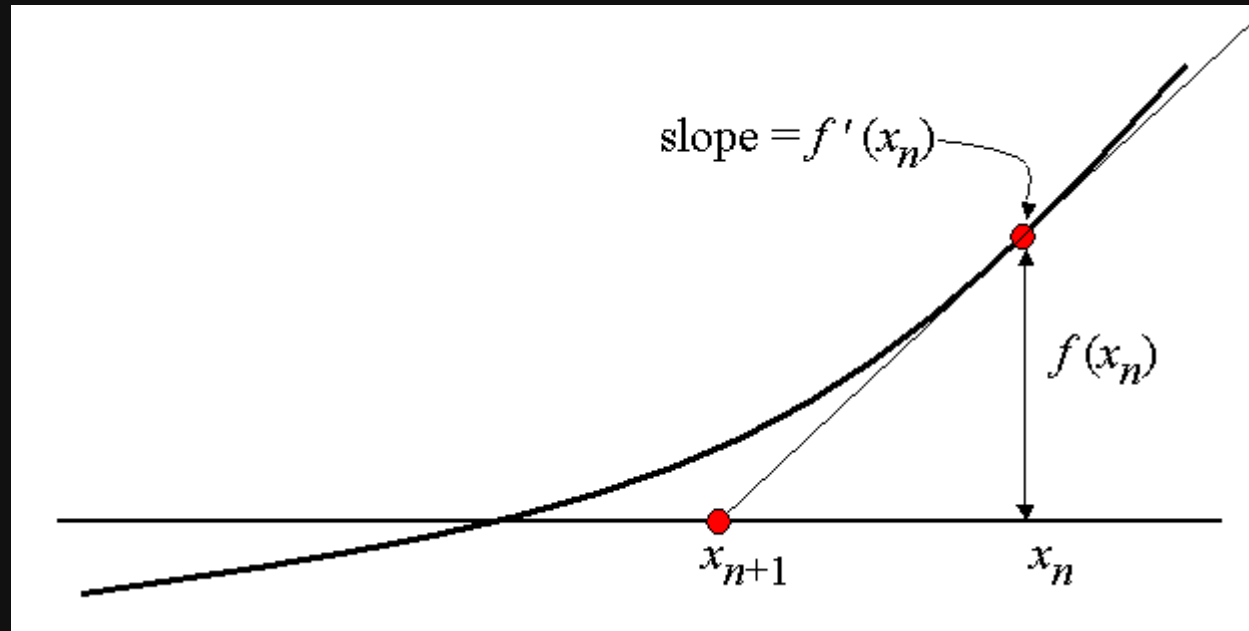
- `age : int`

- `probFunc: DateTime -> double`

- `int -> (DateTime -> double) -> $$$`

# Newton's Method Example

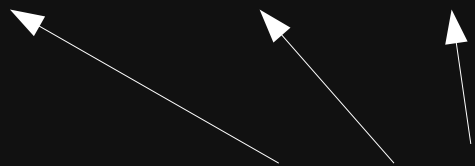
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



# Newton's Method Example

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- float -> Func -> Func -> float



- ```
let Newton start f df =  
  let mutable counter = start  
  while (.....) do  
    counter <- counter - (f counter) / (df counter)
```

Newton's Method Example

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$


- `float -> (float -> float) -> (float -> float) -> float`



- ```
let Newton start f df =
 let mutable counter = start
 while (.....) do
 counter <- counter - (f counter) / (df counter)
```


# Strategy Pattern

- ```
public class Package{  
    private IShippingStrategy iShippingMethod;  
    public double postalCost(Order order){  
        return iShippingMethod.Calculate(order);  
    }  
}
```
- ```
public class FedexStrategy : IShippingStrategy{
 public double iShippingMethod.Calculate(...)
 {.....}
}
```




- ```
public class Package{  
    private IShippingStrategy iShippingMethod;  
    public double postalCost(Order order){  
        return iShippingMethod.Calculate(order);  
    }  
}
```

- ```
iShippingMethod.Calculate:
 Order -> double
```




- ```
public class Package{  
    private Func<Order,double> iShippingMethod;  
    public double postalCost(Order order){  
        return iShippingMethod(order);  
    }  
}
```





- ```
public class Package{


 public double postalCost(Order order,
 Func<Order, double> iShippingMethod)
 {
 return iShippingMethod(order);
 }
}
```




```
public double postalCost(Order order,
 Func<Order, double> iShippingMethod)
{
 return iShippingMethod(order);
}
```




```
public double postalCost(Order order,
 Func<Order, double> f)
{
 return f(order);
}
```




```
public double postalCost(A a,
 Func<A, double> f)
{
 return f(a);
}
```



```
public B postalCost(A a,
 Func<A, B> f)
{
 return f(a);
}
```



```
public B postalCost(A a, Func<A, B> f)
{ return f(a); }
```




```
public B postalCost(A a, Func<A, B> f)
{ return f(a); }
```

```
postalCost a f = f a
```



`postalCost a f = f a`





postalCost a f = f a  
(|>) a f = f a



# Summary

# Summary of Tricks

- Chain/pipe functions as much as possible.
- Use partial application for get a new function.
  - “this” keyword for special case (n-1)
- Higher order functions.
  - Use Functions as inputs and outputs.

# What to learn next in F#?


- Sets, Lists, Dictionary
- Pattern Matching
- Union Type, Tuples, Records
- Option Type (Missing/null Values)
- Async
- Impure Operations



# Where to learn?

- FSharpforfunandprofit blog
  - <https://fsharpforfunandprofit.com/>
- Real-World Functional Programming
  - <https://www.manning.com/books/real-world-functional-programming>

# Conference videos?



2014 NDC  
new DevelopersConference();  
1-5 December - London, UK

Inspiring SINCE 2008  
Developers

Photo of a speaker at a podium with the NDC logo.

| OO pattern/principle              | FP equivalent              |
|-----------------------------------|----------------------------|
| • Single Responsibility Principle | • Functions                |
| • Open/Closed principle           | • Functions                |
| • Dependency Inversion Principle  | • Functions, also          |
| • Interface Segregation Principle | • Functions                |
| • Factory pattern                 | • You will be assimilated! |
| • Strategy pattern                | • Functions again          |
| • Decorator pattern               | • Functions                |
| • Visitor pattern                 | • Resistance is futile!    |

*Seriously, FP patterns are different*

- Scott Wlaschin (author for F#forfunandprofit)
  - Great tech educator.
  - Given many good talks during NDC Conference. (Available on Youtube)



# How to learn?

- Try out Project Euler Questions.
- Tips and tricks here:
  - <https://fsharpforfunandprofit.com/learning-fsharp/>
- I have some training materials for interns.

# Sources (Who uses FP)

- <https://www.janestreet.com/technology/>
- <https://reasonml.github.io/>
- <https://fsharp.org/testimonials/>
- <https://devblogs.nvidia.com/jet-gpu-powered-fulfillment/>
- <https://www.scala-lang.org/old/node/1658>
- <https://clojure.org/community/companies>
- [https://www.slideshare.net/naughty\\_dog/statebased-scripting-in-uncharted-2-among-thieves](https://www.slideshare.net/naughty_dog/statebased-scripting-in-uncharted-2-among-thieves)