Introduction to Monads

Functional Programming Patterns in F#

Disclaimer

- May contain error
- If you are Haskell/FP-expert, please let me know any errors

Speaker

- Chang Hai Bin
- Data Engineer at Hedge Fund
- M.Sc. Uni. of Michigan (Math)

- Financial Engineer (Numerical Technologies)
- Business Analyst (U.S. Mattress)

Simple Code

- `let x = 2 + 3`

- `// x = 5`

Data Extraction

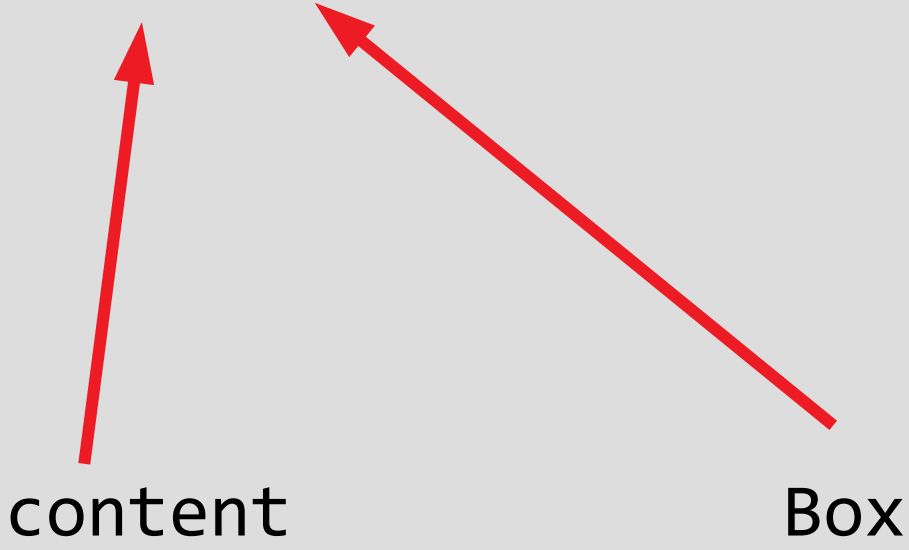
- `let! a = x`

Data Extraction

- `let! a = x`

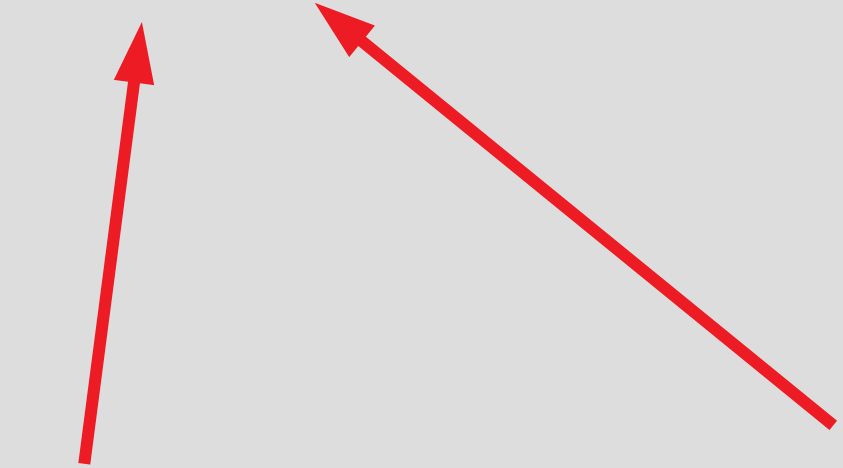
Data Extraction

- `let! a = x`



Data Extraction

- `let! a = x`



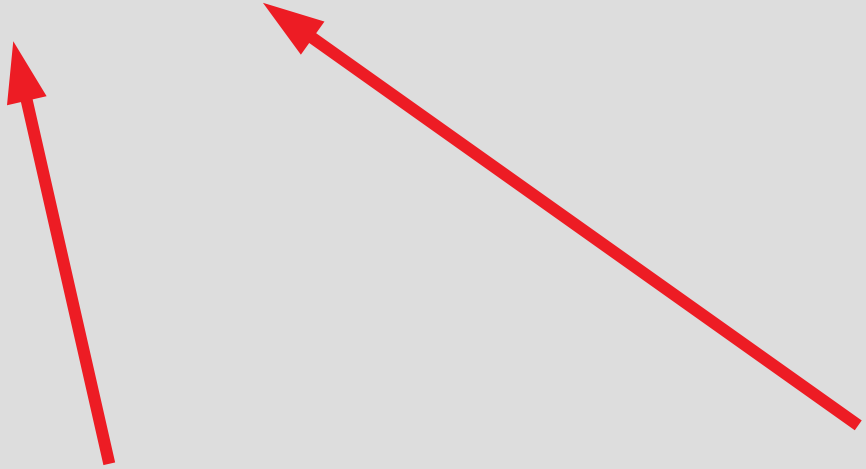
content

Box

(Do extra stuff in background)

Data Extraction

- `let! a = x`



Extract a Out of x

(Do extra stuff in background)

Repeated Pattern

- ```
  let w =
      {
          let a = 1
          let b = 2
          let c = 3
          return a + b + c
      }
  ```
  ```

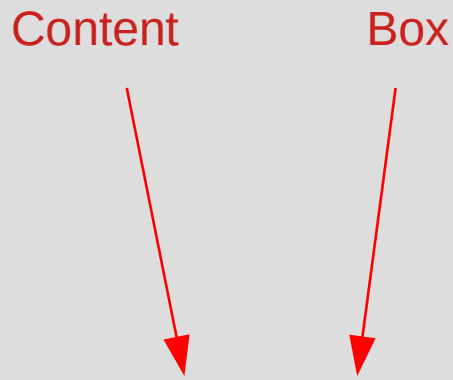
Repeated Pattern

```
let w =
    opt {
        let! a = x
        let! b = y
        let! c = z
        return a + b + c
    }
```

Repeated Pattern

```
let w =
    opt {                           (Environment)
        let! a = x
        let! b = y                  (Extract)
        let! c = z
        return a + b + c
    }
```

Repeated Pattern



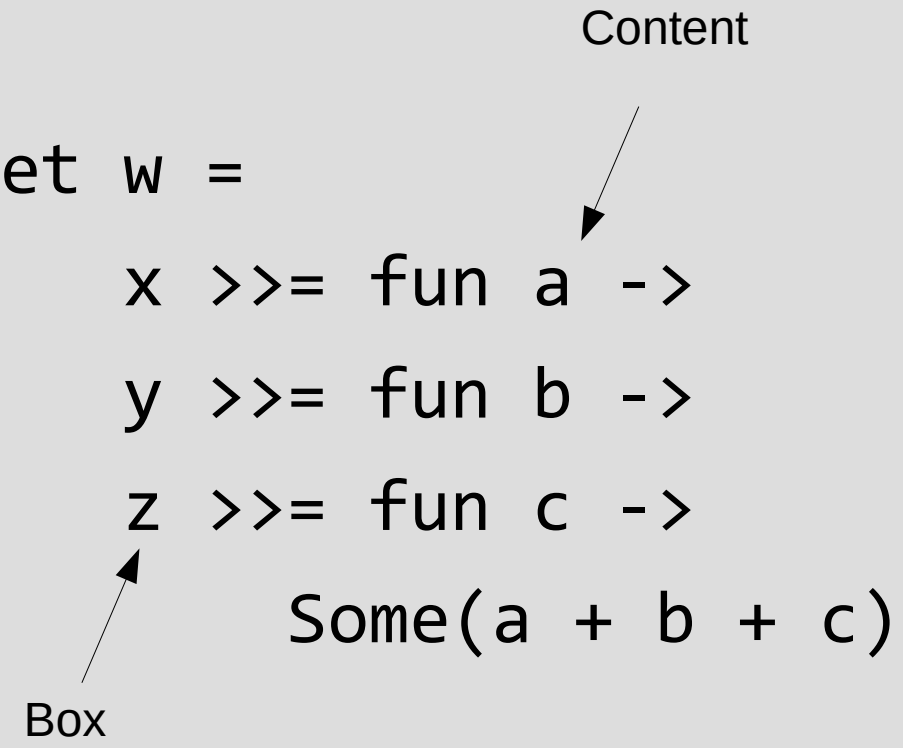
- `let w =`

```
    opt {                          (Environment)
        let! a = x
        let! b = y                 (Extract)
        let! c = z
        return a + b + c
    }
```

Haskell-Style Syntax

- ```
  let w =
      x >>= fun a ->
      y >>= fun b ->
      z >>= fun c ->
          Some(a + b + c)
  ```
  ```

Haskell-Style Syntax



- `let w =`

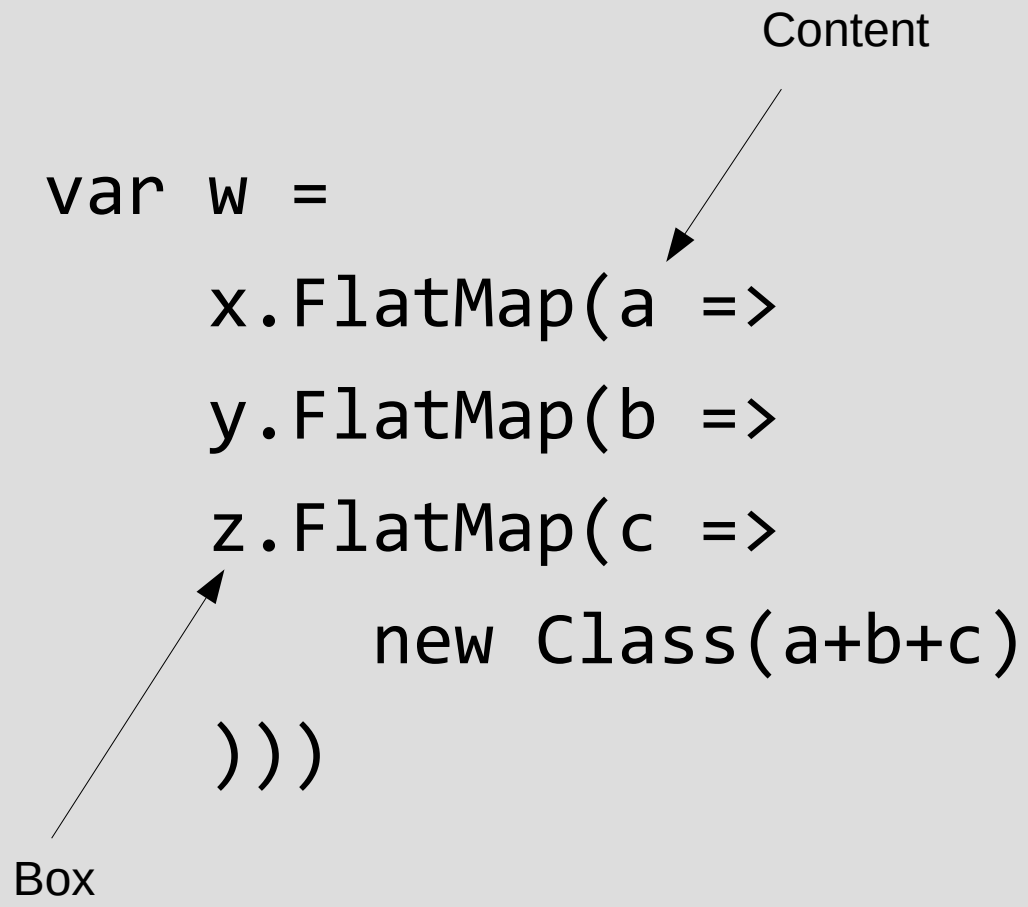
```
    x >>= fun a ->
    y >>= fun b ->
    z >>= fun c ->
        Some(a + b + c)
```

- `>>=`
  - Extract
  - Handle special case (background)

C# Style syntax

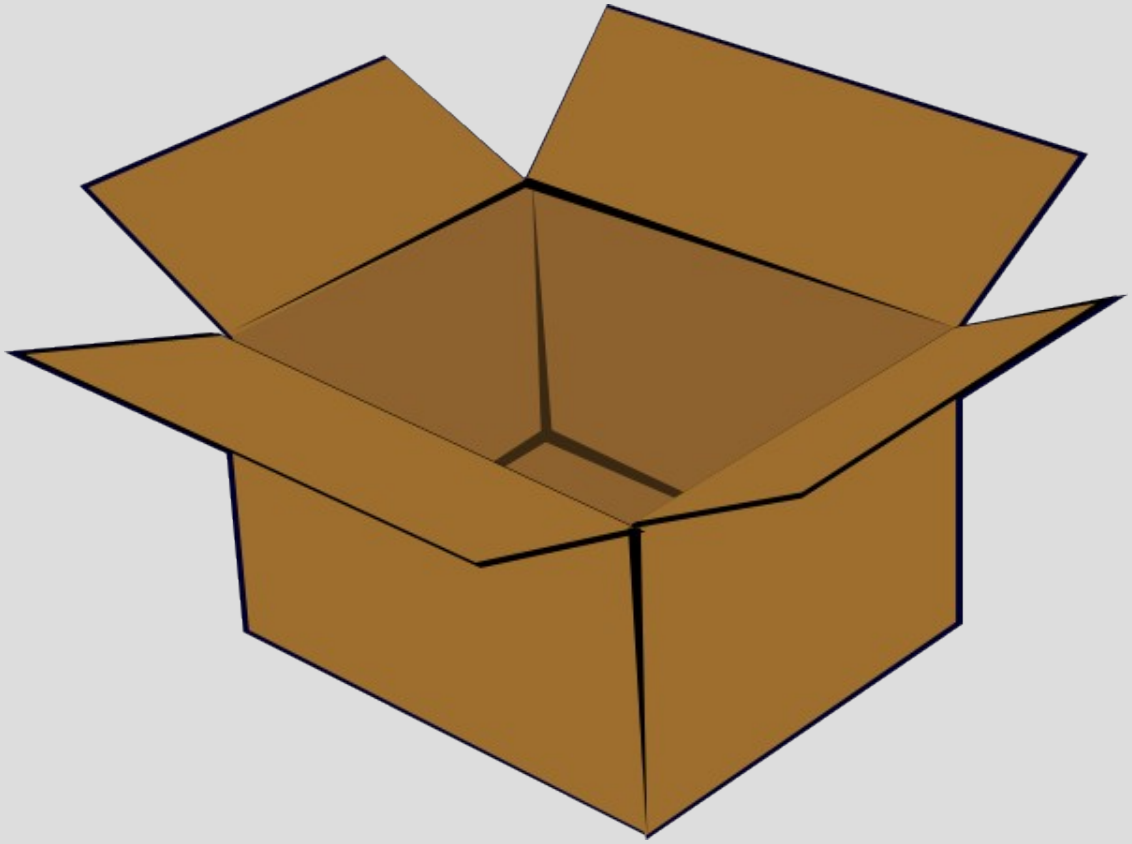
- ```
  var w =
      x.FlatMap(a =>
      y.FlatMap(b =>
      z.FlatMap(c =>
          new Class(a+b+c)
      )))
  ```
  ```

C# Style syntax




```
var w =
    x.FlatMap(a =>
    y.FlatMap(b =>
    z.FlatMap(c =>
        new Class(a+b+c)
    )))
```

1. Option Type (Nullable)



1. Option-Type (Nullable)

```
var x = formula1()
var y = formula2()
var z = formula3()

var result = x.Value + y.Value + z.Value
```

1. Option-Type (Nullable)

```
var x = formula1()
var y = formula2()
var z = formula3()

var result = x.Value + y.Value + z.Value
```

What happens is x is null?

- In C#, use `null` to represent missing values.
- Tony Hoare: The Billion Dollar Mistake
- https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

1. Option-Type (Nullable)

- In F#, use Some( ) and None

- Called Option-type

```
let x = Some(1)
let y = Some(2)
let z = Some(3)

What is   x + y + z   ?
```

```
let x = Some(1)
let y = Some(2)
let z = Some(3)
```

What is `x + y + z` ?

```
x + y + z = Some(6)
```

```
let x = Some(1)
let y = Null
let z = Some(3)

What is   x + y + z   ?
```

```
let x = Some(1)
let y = Null
let z = Some(3)
```

What is `x + y + z` ?

`x + y + z = Null`

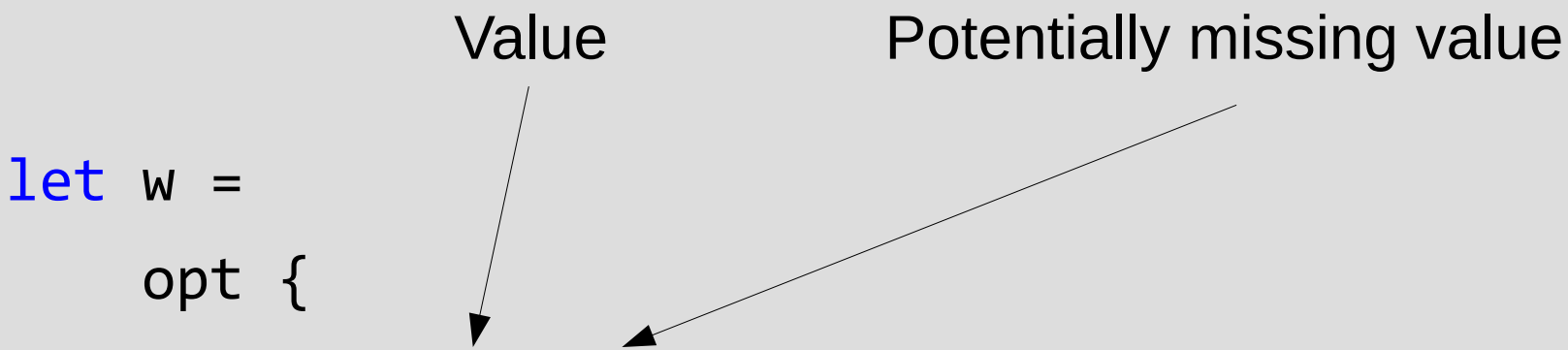
C#

```
var w =
    if(x != null){
        if(y != null){
            if(z != null){
               return x + y + z;
            }
        }
    }
    else {
        return null;
    }
```

F# Code

```
let w =
    opt {
        let! a = x
        let! b = y
        let! c = z
        return a + b + c
    }
```

F# Code




```
let w =
    opt {
        let! a = x
        let! b = y
        let! c = z
        return a + b + c
    }
```

2. Result-Type (Exception)

2. Result-Type (Exception)

- In C#, use Exception to represent errors.

- Problem:
  - Exception can be thrown anywhere
  - Lots of try-catch statements

C#

```
var w =
    var a = x.function1();
    var b = y.function2();
    var c = z.function3();
    return a + b + c;
```

Where’s the error?

C#

```
var w =
    try {
        var a = x.function1();
        var b = y.function2();
        var c = z.function3();
    }
    catch (Exception e){
        if      (e == Exception1){ ......}
        else if (e == Exception2){ ......}
        ........
    }
```

C#

```
var w =
    var (a, err1) = x.function1();
    var (b, err2) = y.function2();
    var (c, err3) = z.function3();
    if (err1 == ‘’){
        if(err2 == ‘’){
            if(err3 == ‘’){
                ......
            }
        }
    } else ......
```

```
let x = Ok 1
let y = Ok 2
let z = Ok 3

What is   x + y + z  ?
```

```
let x = Ok 1
let y = Ok 2
let z = Ok 3

What is   x + y + z  ?


x + y + z = Ok 6
```

```
let x = Ok 1
let y = Ok 2
let z = Error “Msg3”

What is   x + y + z  ?
```

```
let x = Ok 1
let y = Ok 2
let z = Error “Msg3”
```

What is x + y + z ?

```
x + y + z = Error “Msg3”
```

(Keep Exception as string)

```
let x = Ok 1
let y = Error “Msg2”
let z = Error “Msg3”

What is   x + y + z  ?
```

```
let x = Ok 1
let y = Error “Msg2”
let z = Error “Msg3”

What is   x + y + z  ?
```

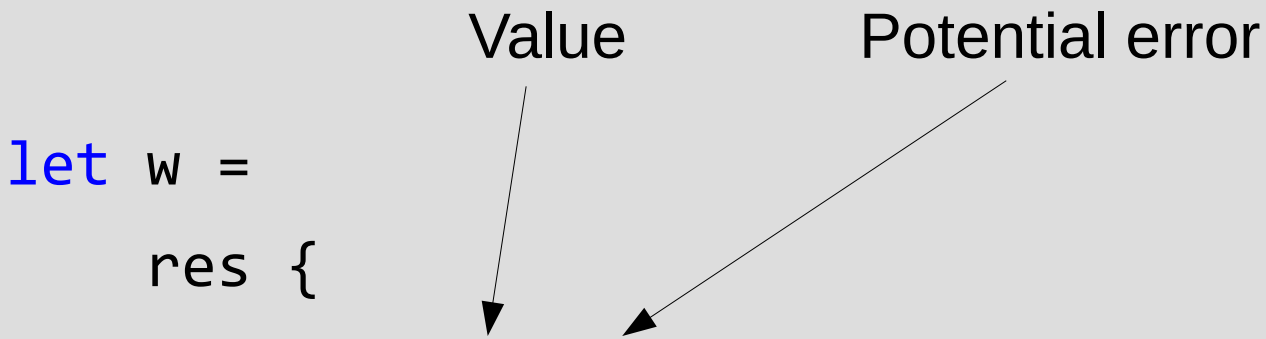
```
x + y + z = Error “Msg2”
```

(First exception)

F# Code

```
let w =
    res {
        let! a = x
        let! b = y
        let! c = z
        return a + b + c
    }
```

F# Code




```
let w =
    res {
        let! a = x
        let! b = y
        let! c = z
        return a + b + c
    }
```

3. List

3. List

```
let x = [1;2;3]
let y = [10;100]

What is    x * y    ?
```

3. List

```
let x = [1;2;3]
let y = [10;100]

What is    x * y    ?


x * y = [10;100;  20;200;  30;300]
```

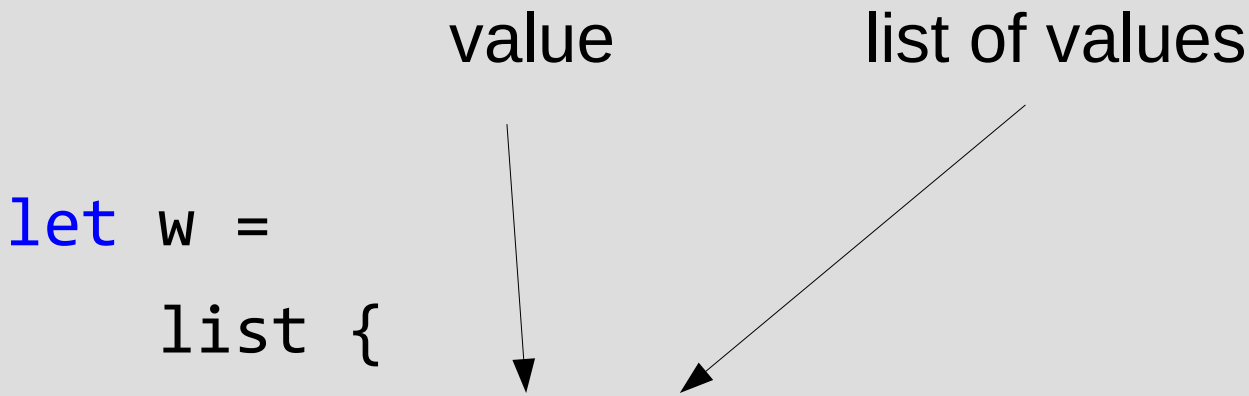
C#

```
var w =
    var w = new List();
    for(var a in x){
        for(var b in y){
            w.Add(a * b);
        }
    }
    return w;
```

F#

```
let w =
    list {
        let! a = x
        let! b = y
        return a * b
    }
```

F#




```
let w =
    list {
        let! a = x
        let! b = y
        return a * b
    }
```

4. Logging

4. Logging

```
let w =
    let a = formula1()

    let b = formula2()

    let c = formula3()

    return a + b + c
```

4. Logging

```
let w =
    let a = formula1()
    Console.Write(“Value is ”+ a)
    let b = formula2()
    Console.Write(“Value is ”+ b)
    let c = formula3()
    Console.Write(“Value is ”+ c)
    return a + b + c
```

Original

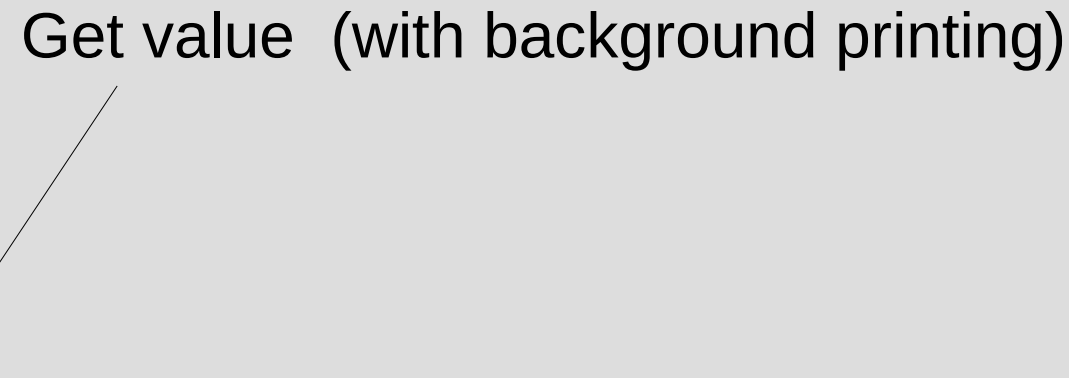
```
let w =

    let a = formula1()
    let b = formula2()
    let c = formula3()
    return a + b + c
```

F#

```
let w =
    logging {
        let! a = formula1()
        let! b = formula2()
        let! c = formula3()
        return a + b + c
    }
```

F#




```
let w =
    logging {
        let! a = formula1()
        let! b = formula2()
        let! c = formula3()
        return a + b + c
    }
```

5. Delayed Computation

5. Delayed Computation

- Analogy: Cooking recipe

- Recipe X takes 2 hours to cook,
- Recipe Y takes 3 hours to cook.

- W = X + Y
- W should take 5 hours to cook.

- Recipe X takes 2 hours to cook,
- Recipe Y takes 3 hours to cook.

- W = X + Y
- Writing down recipe W on a piece of paper should not take 5 hours.

5. Delayed Computation

- Algorithm X: 2 min to run,
- Algorithm Y: 3 min to run.

- W = X + Y
- W should take 5 minutes to run.

- But creating the instructions for W should not take that long.

C#

```
var c =
    {
        var a = x();
        var b = y();
        return a + b;
    };
```

This takes 5 minutes!

C#

```
Func<A> w =
    () => {
        var a = x();
        var b = y();
        return a + b;
    };                          "w" is the recipe


var c = w();                    Run the recipe here
```

F#

```
let w =
    delay {
        let! a = x
        let! b = y
        return a + b
    }

let c = w()
```

F#


```
let w =
    delay {
        let! a = x
        let! b = y
        return a + b
    }

let c = w()
```

Async (Delayed Comp.)

- Available in F#.
- Library is nicely designed.
- Can also do multi-threading.

6. State Monad

6. State Monad 1 (Random)

- Pure function:

- Gives the same output everytime it was given the same input.


PUSH


Enjoy
Coke


$1.00 DEPOSIT NICKELS, DIMES OR QUARTERS
(Receive Change Below)

C#

```
var result =
    var random = new Random(97);
    var a = random.GetDouble();          // 0.97
    var b = random.GetDouble();          // 0.09
    var c = random.GetDouble();          // 0.73
    return a + b + c;
```

Different result for each “GetDouble”!

How are numbers generated

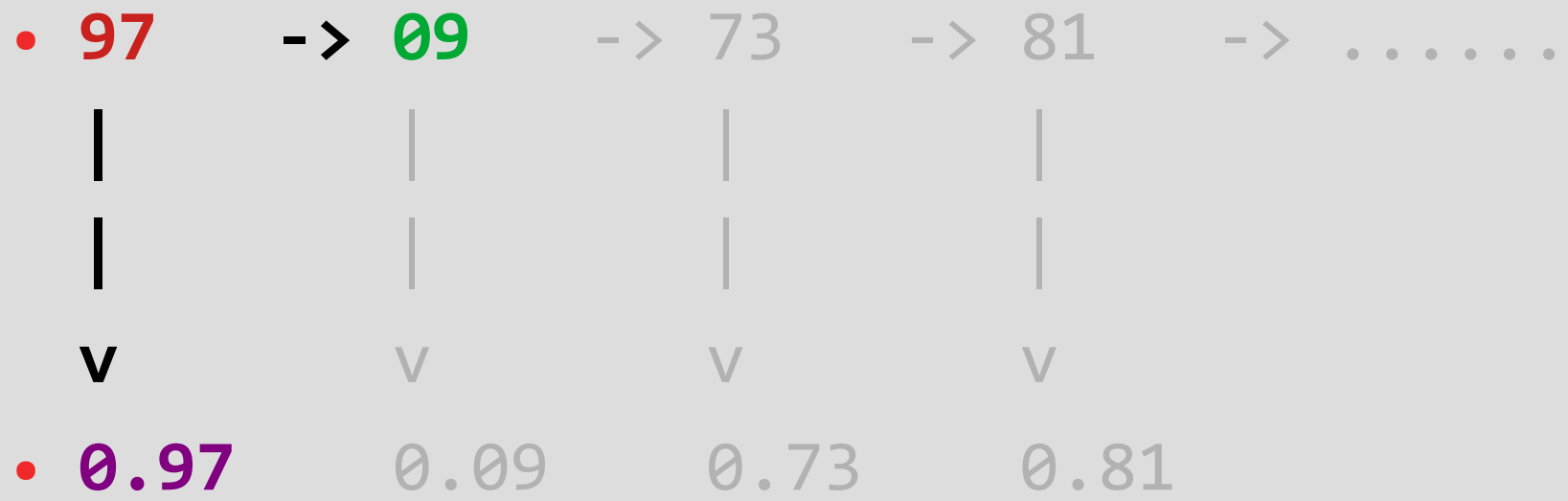
- Take original number
- Multiply by 97
- Take last 2 digits (or mod 100)

- 97 -> 09 -> 73 -> 81 -> 57 -> ......

- Not cryptographically safe.

Div by 100

- 97 -> 09 -> 73 -> 81 -> ......
  | | | |
  | | | |
  v v v v
- 0.97 0.09 0.73 0.81


97 -> 09 -> 73 -> 81 -> ......
0.97 0.09 0.73 0.81


97 -> 09 -> 73 -> 81 -> ......
0.97 0.09 0.73 0.81

GetDouble()

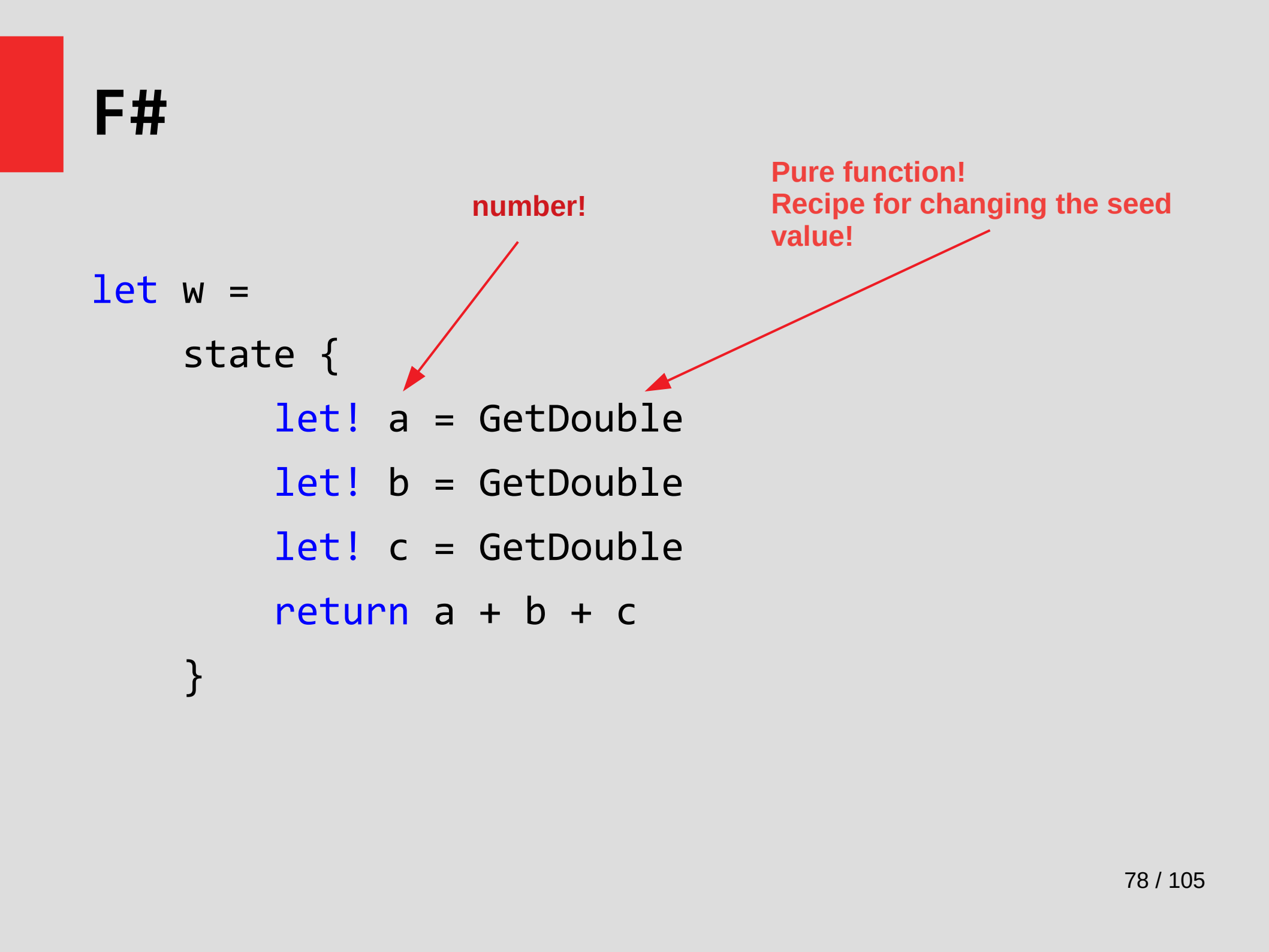
- GetDouble:

```
old_start -> (randomNumber, new_start)
```

F#

```
let w =
    state {
        let! a = GetDouble
        let! b = GetDouble
        let! c = GetDouble
        return a + b + c
    }
```

F#




```
let w =
    state {
        let! a = GetDouble
        let! b = GetDouble
        let! c = GetDouble
        return a + b + c
    }
```

6. State Monad 2 (Mut. List)

```
var result =
    var s = new List();
    s.Append(100);
    s.Append(20);
    s.Append(3);
    var a = s.Remove();
    var b = s.Remove();
    var c = s.Remove();
    return a + b + c;
```

6. State Monad 2 (Mut. List)

```
var result =
    var s = new List();
    s.Append(100);
    s.Append(20);
    s.Append(3);
    var a = s.Remove();
    var b = s.Remove();
    var c = s.Remove();
    return a + b + c;
```

Different result each time!

Remove()

- Given a List.
- Returns a value (if possible)
- Modify the list.

Pop

- Remove: List -> (first, remain)

- (If possible)

Original

```
var result =
    var s = new List();
    s.Append(100);
    s.Append(20);
    s.Append(3);
    var a = s.Remove();
    var b = s.Remove();
    var c = s.Remove();
    return a + b + c;
```

F#

```
let w =
    state {
        do! Append 100
        do! Append 20
        do! Append 3
        let! a = Remove
        let! b = Remove
        let! c = Remove
        return a + b + c
    }
```

F#

```
let w =
    state {
        do! Append 100
        do! Append 20
        do! Append 3
        let! a = Remove
        let! b = Remove
        let! c = Remove
        return a + b + c
    }
```

Value!

Recipe to cut a list of numbers

7. Reader Monad

7. Reader Monad

```
Func Example(logger) =

    var a = f(logger);
    var b = g(logger);
    var c = h(logger);
    return  a + b + c;
```

F#

```
let Example =
    reader {
        let! a = f
        let! b = g
        let! c = h
        return a + b + c
    }
```

```
Example : logger -> result
```

F#


```
let Example =
    reader {
        let! a = f
        let! b = g
        let! c = h
        return a + b + c
    }
```

```
Example : logger -> result
```

With more inputs

```
Func Example(logger) =

    var a = f(a1,logger);
    var b = g(b1,b2,logger);
    var c = h(c1,c2,c3,logger);
    return  a + b + c;
```

F#

```
let Example* =
    reader {
        let! a = f a1 __
        let! b = g b1 b2 __
        let! c = h c1 c2 c3 __
        return a + b + c
    }
```

*Underline for visual aid

```
Example : logger -> result
```

8. IO Monad


He is afraid.
He is totally alone.
He is 3 million light years from home.
A STEVEN SPIELBERG FILM
E.T.
THE EXTRA-TERRESTRIAL
A STEVEN SPIELBERG FILM E.T. THE EXTRA-TERRESTRIAL
DEE WALLACE PETER COYOTE HENRY THOMAS AS ELLIOTT
MUSIC BY JOHN WILLIAMS WRITTEN BY MELISSA MATHISON
PRODUCED BY STEVEN SPIELBERG & KATHLEEN KENNEDY
DIRECTED BY STEVEN SPIELBERG A UNIVERSAL PICTURE
DISTRIBUTED BY UNITED INTERNATIONAL PICTURES


WARNING
BIOHAZARD

8. IO Monad

```
var result =
    x = System.IO.ReadFile(file1)
    y = System.IO.ReadFile(file2)
    total = x.wordCount + y.wordCount
    return total;


result: int
```

```
var result = {...........}
result: int
```

```
var result = {............}
result: int
```

Don’t know if this code interacts with outside world, e.g.

1. Read/Write a file
2. Contacted a Database
3. Sent Http Requests

etc.

```
x = System.IO.ReadFile(file1)
```

What if someone modifies the file?

Each time may have different results!

Impure!

```
x = System.IO.ReadFile(file1)
```

What if the file doesn’t exist?

Error!

C#

```
var result =

    x = System.IO.ReadFile(file1)
    y = System.IO.ReadFile(file2)
    total = x.wordCount + y.wordCount
    return total;
```

F#

```
let result =
    IO {
        let! x = ReadFile file1
        let! y = ReadFile file2
        total = x.wordCount + y.wordCount
        return total;
    }
```

Handles special case nicely.

F#


```
let result =
    IO {
        let! x = ReadFile file1
        let! y = ReadFile file2
        total = x.wordCount + y.wordCount
        return total;
    }

Handles special case nicely.
```

F#

```
let result = IO {......}
result: IO<int>
```

F#

```
let result = IO {......}
result: IO<int>
```

Result is a special class/object wrapping an integer.

Very likely interacting with “outside world”.

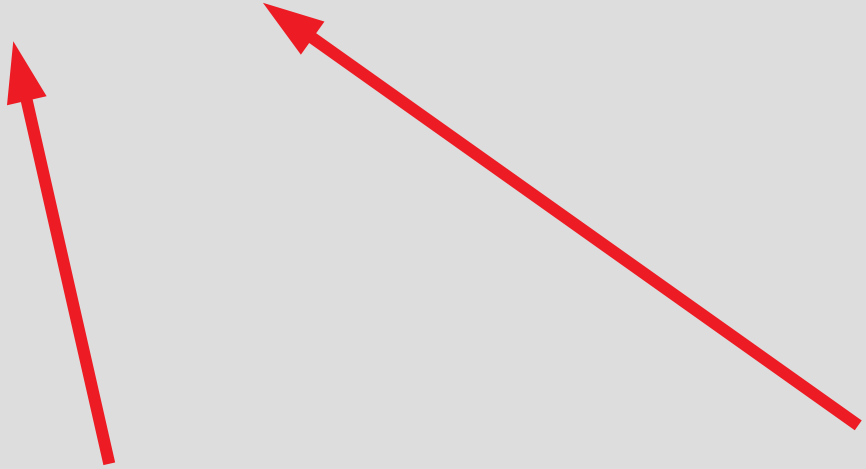
Know it just from type signature, without reading the code.

Summary

- Null Values
- Exceptions
- Nested for-loop
- Logging
- Delayed Comp./Async
- Changing Values
- Dependencies
- IO

Pattern

- `let! a = x`



Extract a Out of x

(Do extra stuff in background)