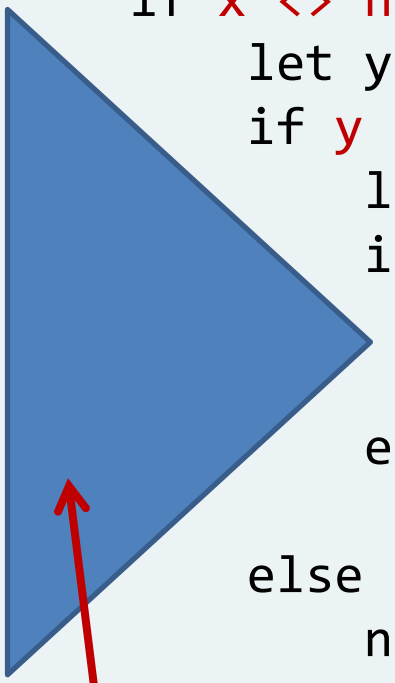
Monads and Async

Taming the "pyramid of doom"

```
let example input =
    let x = doSomething input
    if x <> null then
        let y = doSomethingElse x
        if y <> null then
            let z = doAThirdThing y
            if z <> null then
                let result = z
                result
            else
                null
        else
            null
    else
        null
```

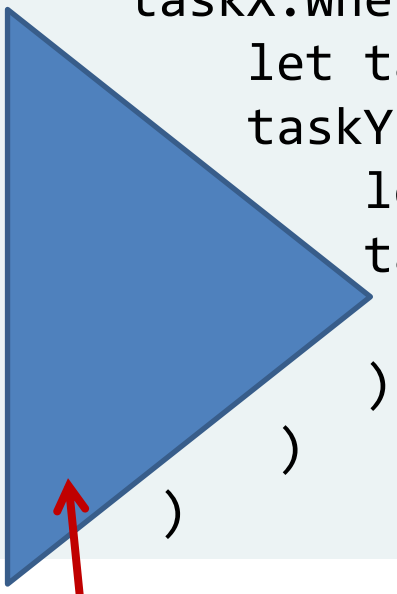


Nested null checks

The "pyramid of doom"

I know you could do early returns, but bear with me...

```
let taskExample input =
    let taskX = startTask input
    taskX.WhenFinished (fun x ->
        let taskY = startAnotherTask x
        taskY.WhenFinished (fun y ->
            let taskZ = startThirdTask y
            taskZ.WhenFinished (fun z ->
                z  // final result
            )
        )
    )
```



Nested callbacks

Another "pyramid of doom"

Let's fix this!

```
let example input =
    let x = doSomething input
    if x <> null then
        let y = doSomethingElse x
        if y <> null then
            let z = doAThirdThing y
            if z <> null then
                let result = z
                result
            else
                null
        else
            null
    else
        null
```

Nulls are a code smell:
replace with Option!

```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            let z = doAThirdThing (y.Value)
            if z.IsSome then
                let result = z.Value
                Some result
            else
                None
        else
            None
    else
        None
```

Much more elegant, yes?

No! This is fugly!

But there is a pattern we can exploit...

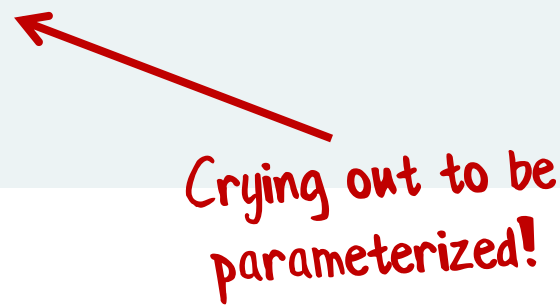
```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            let z = doAThirdThing (y.Value)
            if z.IsSome then
                // do something with z.Value
                // in this block
            else
                None
        else
            None
    else
        None
```

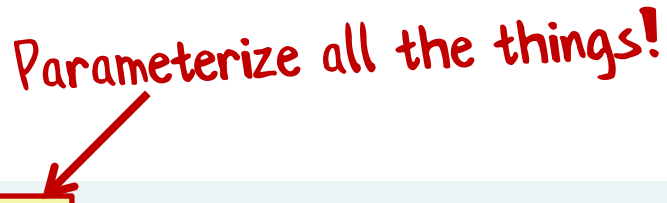
```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            // do something with y.Value
            // in this block



        else
            None
    else
        None
```

```
let example input =
    let x = doSomething input
    if x.IsSome then
        // do something with x.Value
        // in this block




    else
        None
```

Can you see the pattern?

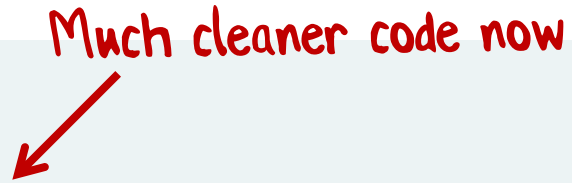
```
if opt.IsSome then
    //do something with opt.Value
else
    None
```





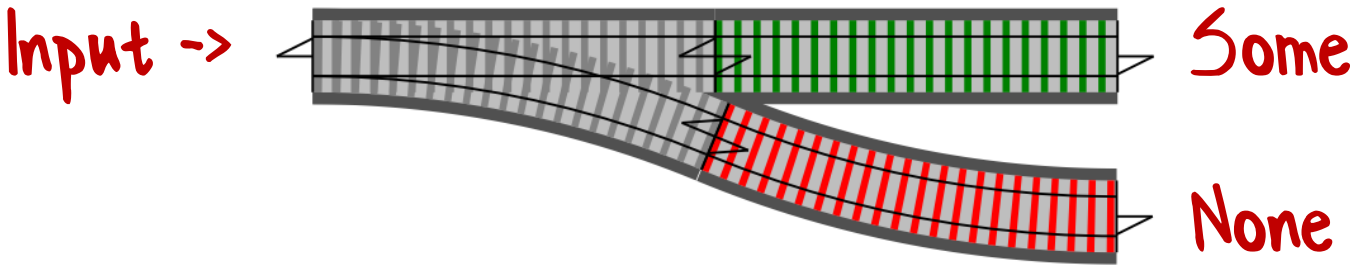

```
let ifSomeDo f opt =
    if opt.IsSome then
        f opt.Value
    else
        None
```

```
let ifSomeDo f opt =
    if opt.IsSome then
        f opt.Value
    else
        None
```




```
let example input =
    doSomething input
    |> ifSomeDo doSomethingElse
    |> ifSomeDo doAThirdThing
    |> ifSomeDo ...
```

This is an example of a more general problem



Pattern:
Use bind to chain options

Before

```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            let z = doAThirdThing (y.Value)
            if z.IsSome then
                let result = z.Value
                Some result
            else
                None
        else
            None
    else
        None
```

After

```
let bind f opt =
    match opt with
    | Some v -> f v
    | None -> None
```

Same as "ifSomeDo"

After

```
let bind f opt =
    match opt with
    | Some v -> f v
    | None -> None
```

```
let example input =
    doSomething input
    |> bind doSomethingElse
    |> bind doAThirdThing
    |> bind ...
```

No pyramids!

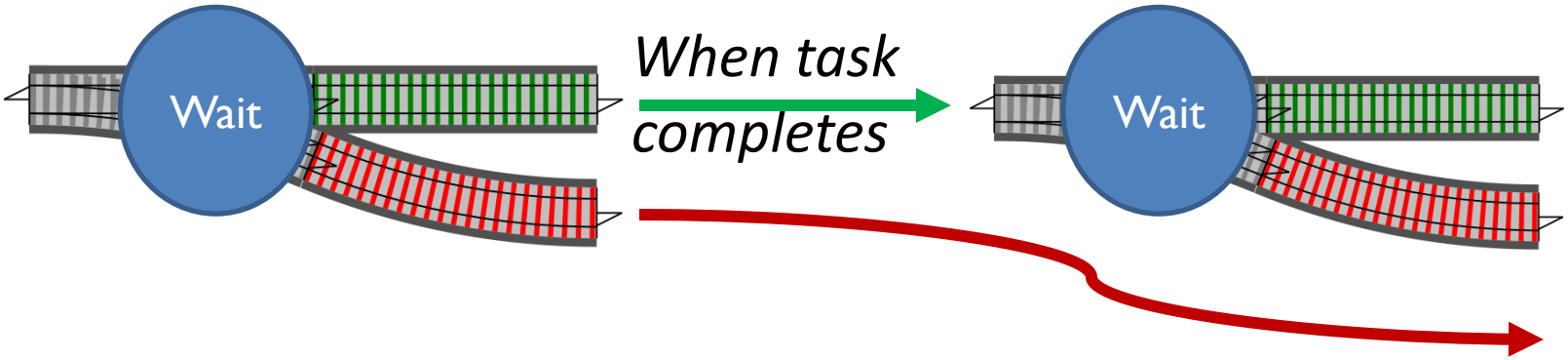
Code is linear and clear.

This pattern is called "monadic bind"

Pattern:

Use bind to chain tasks

a.k.a "promise" "future"


Wait
When task
completes
Wait

Before

```
let taskExample input =
    let taskX = startTask input
    taskX.WhenFinished (fun x ->
        let taskY = startAnotherTask x
        taskY.WhenFinished (fun y ->
            let taskZ = startThirdTask y
            taskZ.WhenFinished (fun z ->
                z  // final result
            )
        )
    )
```

After

```
let taskBind f task =
    task.WhenFinished (fun taskResult ->
        f taskResult)

let taskExample input =
    startTask input
    |> taskBind startAnotherTask
    |> taskBind startThirdTask
    |> taskBind ...
```

This pattern is also a "monadic bind"

"Monad" is an FP pattern

A **monad** is

i. An effect type
   - e.g. `Option<>, List<>, Async<>`

ii. Plus a `return` function
   - a.k.a. `pure unit`

iii. Plus a `bind` function that converts a "diagonal" (world-crossing) function into a "horizontal" (E-world-only) function
   - a.k.a. `>>= flatMap SelectMany`

iv. And bind/return must have sensible implementations
   - the Monad laws

The F# async type

Async in F# is a *type*

- `"fetchUrl"`

  `string -> Async<string>`
- `"loadCustomer"`

  `customerId -> Async<Customer>`

Async in F# is a *type*

- Combine them using bind
- Or use in an "async" computation expression

Async combined with Result

- "fetchUrl"

```
string ->
  AsyncResult<string,HttpError>
```

- "loadCustomer"

```
customerId ->
  AsyncResult<Customer,DbError>
```

Demo:

`01a-Async.fsx`
`02a-AsyncRopWithTicTacToe.fsx`
`02b-AsyncRopWithCoffeeMaker.fsx`