# Using Free Monads to Simplify Workflows

Brett Hall
Wyatt Technology

If you haven't already, please clone
https://github.com/bretthall/workflow-demo

# Wyatt Technology OBSERVER$_{tm}$

Process Analytical Technology (PAT) for pharmaceutical process development, QC and production.

# Workflows

- A program that controls a process

# Workflows

- A program that controls a process
- Capture results from earlier actions for use in later actions

# Workflows

- A program that controls a process
- Capture results from earlier actions for use in later actions
- Branching and looping/recursion

# Workflows

- A program that controls a process
- Capture results from earlier actions for use in later actions
- Branching and looping/recursion
- Pause and resume workflow

# Workflows

- A program that controls a process
- Capture results from earlier actions for use in later actions
- Branching and looping/recursion
- Pause and resume workflow
- Cancel workflow

# Workflows

- A program that controls a process
- Capture results from earlier actions for use in later actions
- Branching and looping/recursion
- Pause and resume workflow
- Cancel workflow
- Save and restore workflow state

# Alternatives to Free Monads

- List of actions plus an *environment*
- State Machine

# List of Actions

- Just a list of actions that are executed one at a time
- Need an *environment*
  - Map of keys to values that results of actions can be stored in
- Problems:
  - Environment is not type safe
  - Branching and looping makes code hard to understand

# State Machine

- Encode a state machine as a DU type plus *update* function that matches on the case of the DU

- Problems:
  - Complex workflows will require nested DU state and nested matching
    - Hard to understand
  - Too hard to extract reusable building blocks
    - End up building every workflow from scratch

# Better Way

**When faced with nested matching apply a computation expression**

# Better Way

**When faced with nested matching apply a computation expression**

```
let wait = workflow {
    // set the device state to "good"
    do! setDeviceState Device.Good
    // wait 5 seconds
    do! wait 5<seconds>
    // set the device state to "bad"
    do! setDeviceState Device.Bad
    // wait for the data to increment 5 times (you'll need to get the current data value)
    let! curData = getCurrentData ()
    let targetData = curData + 5
    let! finalData = waitForData targetData
    // add a device message saying what the data value is at the end of the wait
    do! addDeviceMsg (sprintf "final data = %d" finalData)
    // set the device state to "ugly"
    do! setDeviceState Device.Ugly
    // get input from the device
    let! input = getDeviceInput "Waiting for input"
    // add a control message saying what the input is
    do! addControlMsg (sprintf "Got input = %s" input)
    // set the device state to "good"
    do! setDeviceState Device.Good
}
```

# Free Monads?

# Free Monads?

A free monad is just a free monoid in the category of endofunctors

# Free Monads?

~~A free monad is just a free monoid in the category of endofunctors~~

An easy way to embed a custom DSL in another program

# Workflow Demo

- https://github.com/bretthall/workflow-demo
- Three console apps and two libraries
    - *device*: The *device* that the workflow will control
    - *control:* The app that executes the workflow
    - *workflow:* The library that contains the workflow implementation
    - *test:* Workflow unit tests
    - *common*: Common types used to message between *control* and *device*

# Exercise 1

- Build a simple workflow
  - Checkout *exercise1* branch
  - See workflows.fs in the workflow project
    - Actions are defined in the Free.Actions module (see Free.fs)
- Run your workflow
  - Run the *device*: dotnet run -p device/device.fsproj
  - Run the *control*: dotnet run -p control/control.fsproj
  - Select the *wait* workflow in control and then select *start*

# Exercise 1

**exercise1-solution branch**

```
let wait = workflow {
    // set the device state to "good"
    do! setDeviceState Device.Good
    // wait 5 seconds
    do! wait 5<seconds>
    // set the device state to "bad"
    do! setDeviceState Device.Bad
    // wait for the data to increment 5 times (you'll need to get the current data value)
    let! curData = getCurrentData ()
    let targetData = curData + 5
    let! finalData = waitForData targetData
    // add a device message saying what the data value is at the end of the wait
    do! addDeviceMsg (sprintf "final data = %d" finalData)
    // set the device state to "ugly"
    do! setDeviceState Device.Ugly
    // get input from the device
    let! input = getDeviceInput "Waiting for input"
    // add a control message saying what the input is
    do! addControlMsg (sprintf "Got input = %s" input)
    // set the device state to "good"
    do! setDeviceState Device.Good
}
```

# Interpreters

- The computation expression creates an AST for our DSL
  - Called WorkflowProgram in this case
- To actually do anything useful you need an interpreter to *run the program*
  - Real Workflow Run
  - Workflow replay
  - State Restore
  - Unit Testing

# Exercise 2

- Write a unit test for the workflow from exercise 1
  - Checkout exercise2 branch
    - Search for TODO in test.fs in the test project
    - Build list of ExpectedInstruction objects
      - Be sure to include a Pure case at the end
- Run the test: dotnet run -p test/test.fsproj

# Exercise2

**exercise2-solution branch**

```
testCase "wait" <| fun _ ->
    //TODO: Exercise 2: Fill in expected so that this test will test the wait workflow from exercise 1
    let expected = [
        yield SetDeviceState (Device.Good, ())
        yield Wait (5<Free.seconds>, ())
        yield SetDeviceState (Device.Bad, ())
        let curData = 5
        yield GetCurrentData ((), curData)
        let targetData = curData + 5
        let finalData = targetData + 1
        yield WaitForData (targetData, finalData)
        yield AddDeviceMsg ((sprintf "final data = %d" finalData), ())
        yield SetDeviceState (Device.Ugly, ())
        let input = "testing 1 2 3"
        yield GetDeviceInput ("Waiting for input", input)
        yield AddControlMsg ((sprintf "Got input = %s" input), ())
        yield SetDeviceState (Device.Good, ())
        yield Pure ()
    ]

    interpretTest expected Workflows.wait
```

# Branching

- Workflows can contain branches
  - if/then
  - match
- If you want workflow state restore to work then the branching conditions must be based on either:
  - fixed inputs that will be the same when restoring the workflow
  - results from previous workflow actions

# Exercise 3

- Implement a branching workflow
  - Checkout exercise3 branch
  - See workflows.fs in the workflow project
- Run your workflow
  - Run the device: dotnet run -p device/device.fsproj
  - Run the control: dotnet run -p control/control.fsproj
  - Select the *choice* workflow in control and then select start

# Exercise 3

**exercise3-solution branch**

```
//TODO: Exercise 3: Implement the branching workflow below
let choice = workflow {
    // Get device input, prompt should ask for path A or B
    let! input = getDeviceInput "Path A or B?"
    if input = "A" then
        // if path A is chosen set the device state to "ugly"
        do! setDeviceState Device.Ugly
    else if input = "B" then
        // if path B is chosen reset the device data
        do! resetData ()
    else
        // if another path is chosen add a control message saying what was entered
        do! addControlMsg (sprintf "Chose other path: %s" input)
}
```

# Loops/Recursion

- For and While loops can be implemented
  - Since mutable variables don’t work well in computation expressions For and While loops aren’t that useful
- Use recursion instead
  - Use return! along with a recursive function the returns WorkflowProgram

# Exercise 4

- Implement a recursive workflow
  - Checkout exercise4 branch
  - See workflows.fs in the workflow project
- Run your workflow
  - Run the device: dotnet run -p device/device.fsproj
  - Run the control: dotnet run -p control/control.fsproj
  - Select the *recurse* workflow in control and then select start

# Exercise 4

**exercise4-solution branch**

```
//TODO: Exercise 4: Implement the recursive workflow below
let recurse =
    // Make a recursive workflow that asks for input until the input is "stop".
    let rec getInput index = workflow {
        let! input = getDeviceInput "Input (\"stop\" to stop)"
        if input.ToLower () <> "stop" then
            // A control message containing the input should be added for each input received.
            do! addControlMsg (sprintf "Input %d: %s" index input)
            return! getInput (index + 1)
        else
            return index - 1
}
workflow {
    let! num = getInput 1
    // When "stop" is received a device message should be added saying how many inputs were received.
    do! addDeviceMsg (sprintf "Got %d inputs" num)
}
```

# Abstracted Recursion

- Can implement the usual abstracted recursion schemes
  - fold
  - map
  - unfold

# Exercise 5

- Implement a recursive workflow using fold
  - Checkout exercise5 branch
  - See workflows.fs in the workflow project
    - fold is already implemented in Free.fs
- Run your workflow
  - Run the device: dotnet run -p device/device.fsproj
  - Run the control: dotnet run -p control/control.fsproj
  - Select the *fold* workflow in control and then select start

# Exercise 5

**exercise5-solution branch**

```
//TODO: Exercise 5: Implement the workflow below using fold (from the Free module)
WorkflowProgram<unit>
let fold = workflow {
    // Use fold to iterate the good, bad, and ugly device states setting each one in the device
    // waiting 5 seconds after each change. While doing the fold count how many changes are made
    // and report that value in a control message when the fold is done.
    let! num =
        (0, [Device.Good; Device.Bad; Device.Ugly]) ||> fold (
            fun s t ->
                workflow {
                    do! setDeviceState t
                    do! Free.Actions.wait 5<seconds>
                    return (s + 1)
                }
        )
    do! addControlMsg (sprintf "Did %d state changes" num)
}
```

# Implementation

- Instructions
- Program
- Computation Expression
- Convenience functions
- Interpreter

# Instructions

**DU with one case for each instruction**

```
type Instruction<'a> =
    | Instruction1 of argument * (result -> 'a)
    | Instruction2 of argument * (result -> 'a)
    | Instruction3 of argument * (result -> 'a)
```

# Instructions

**Turn Instruction into a functor by implementing map**

```
// ('a -> 'b) -> Instruction<'a> -> Instruction<'b>
let mapI f instruction =
    match instruction with
    | Instruction1 (x, next) -> Instruction1 (x, next >> f)
    | Instruction2 (x, next) -> Instruction2 (x, next >> f)
    | Instruction3 (x, next) -> Instruction3 (x, next >> f)
```

# Programs

**Container type that combines instructions**

```
type Program<'a> =
    | Free of Instruction<Program<'a>>
    | Pure of 'a
```

# Programs

**Container type that combines instructions**

```
type Program<'a> =
    | Free of Instruction<Program<'a>>
    | Pure of 'a
```

**For a Program ‘a is the result of the program**

# Programs

**Container type that combines instructions**

```
type Program<'a> =
    | Free of Instruction<Program<'a>>
    | Pure of 'a
```

**For a Program ‘a is the result of the program**

```
type Instruction<'a> =
    | Instruction1 of argument * (result -> 'a)
    | Instruction2 of argument * (result -> 'a)
    | Instruction3 of argument * (result -> 'a)
```

**For an instruction ‘a will be the rest of the program**

# Programs

**Need to be able to combine smaller programs into larger programs,**
**~~monads~~ computation expressions to the rescue**

```
// ('a -> Program<'b>) -> Program<'a> -> Program<'b>
let rec bind f program =
    match program with
    | Free x -> x |> mapI (bind f) |> Free
    | Pure x -> f x

type WorkflowBuilder () =
    member this.Bind (x, f) = bind f x
    member this.Return x = Pure x
    member this.ReturnFrom x = x
    member this.Zero () = Pure ()

let workflow = WorkflowBuilder ()
```

# Program Example

```
workflow {
    let! r1 = Free Instruction1 (1, Pure)
    let! r2 = Free Instruction2 (r1, Pure)
    return r2
}
```

```
Free Instruction1 (1, Pure)
|> bind (fun r1 ->
    Free Instruction2 (r1, Pure)
    |> bind (fun r2 ->
        Pure r2
    )
)
```

# Program Example

```
Free Instruction1 (1, Pure)
|> bind (fun r1 ->
    Free Instruction2 (r1, Pure)
    |> bind (fun r2 ->
        Pure r2
    )
)
```

```
Free Instruction2 (r1, Pure) |> bind (fun r2 -> Pure r2)
Instruction2 (r1, Pure) |> mapI (bind (fun r2 -> Pure r2)) |> Free
Instruction2 (r1, Pure >> bind (fun r2 -> Pure r2)) |> Free
Instruction2 (r1, fun r -> Pure r |> bind (fun r2 -> Pure r2)) |> Free
Instruction2 (r1, fun r -> Pure r) |> Free
Free (Instruction2 (r1, Pure))
```

# Program Example

```
Free Instruction1 (1, Pure)
|> bind (fun r1 ->
    // Free Instruction2 (r1, Pure)
    // |> bind (fun r2 ->
    //     Pure r2
    //)
    Free Instruction2 (r1, Pure)
)
```

```
Free Instruction1 (1, Pure) |> bind (fun r1 -> Free Instruction2 (r1, Pure))
Instruction1 (1, Pure) |> mapI (bind (fun r1 -> Free Instruction2 (r1, Pure))) |> Free
Instruction1 (1, Pure >> bind (fun r1 -> Free Instruction2 (r1, Pure))) |> Free
Instruction1 (1, fun r -> Pure r |> bind (fun r1 -> Free Instruction2 (r1, Pure))) |> Free
Instruction1 (1, fun r -> Free Instruction2 (r, Pure)) |> Free
Free (Instruction1 (1, fun r1 -> Free Instruction2 (r1, fun r2 -> Pure r2)))
```

# Convenience Functions

```
workflow {
    let! r1 = Free Instruction1 (1, Pure)
    let! r2 = Free Instruction2 (r1, Pure)
    return r2
}
```

```
let instruction1 arg = Free (Instruction1 (arg, Pure))
let instruction2 arg = Free (Instruction2 (arg, Pure))
let instruction3 arg = Free (Instruction3 (arg, Pure))
```

```
workflow {
    let! r1 = instruction1 1
    let! r2 = instruction2 r1
    return r2
}
```

# Interpreter

```
let rec interpret program =
    match program with
    | Pure x -> x
    | Free (Instruction1 (arg, next)) -> arg |> doStuff1 |> next |> interpret
    | Free (Instruction2 (arg, next)) -> arg |> doStuff2 |> next |> interpret
    | Free (Instruction3 (arg, next)) -> arg |> doStuff3 |> next |> interpret
```

# Interpreter

```
let rec interpret program =
    match program with
    | Pure x -> x
    | Free (Instruction1 (arg, next)) -> arg |> doStuff1 |> next |> interpret
    | Free (Instruction2 (arg, next)) -> arg |> doStuff2 |> next |> interpret
    | Free (Instruction3 (arg, next)) -> arg |> doStuff3 |> next |> interpret
```

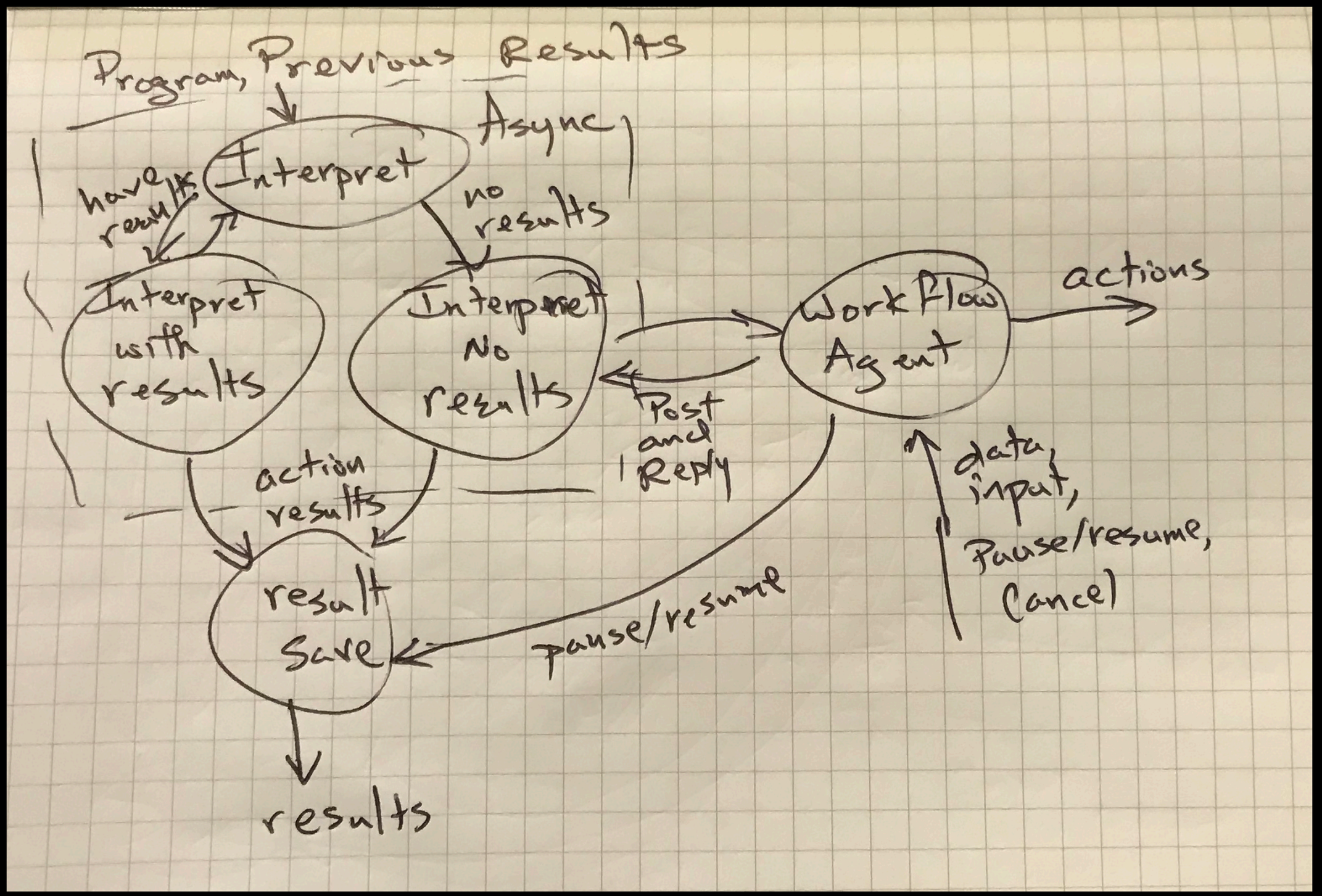
**Interpret the workflow from a few slides ago**

```
Free (Instruction1 (1, fun r1 -> Free Instruction2 (r1, fun r2 -> Pure r2))) |> interpret
let r1 = doStuff1 1
r1 |> (fun r1 -> Free Instruction2 (r1, fun r2 -> Pure r2)) |> interpret
Free Instruction2 (r1, fun r2 -> Pure r2) |> interpret
let r2 = doStuff2 r1
r2 |> (fun r2 -> Pure r2) |> interpret
Pure r2 |> interpret
r2
```

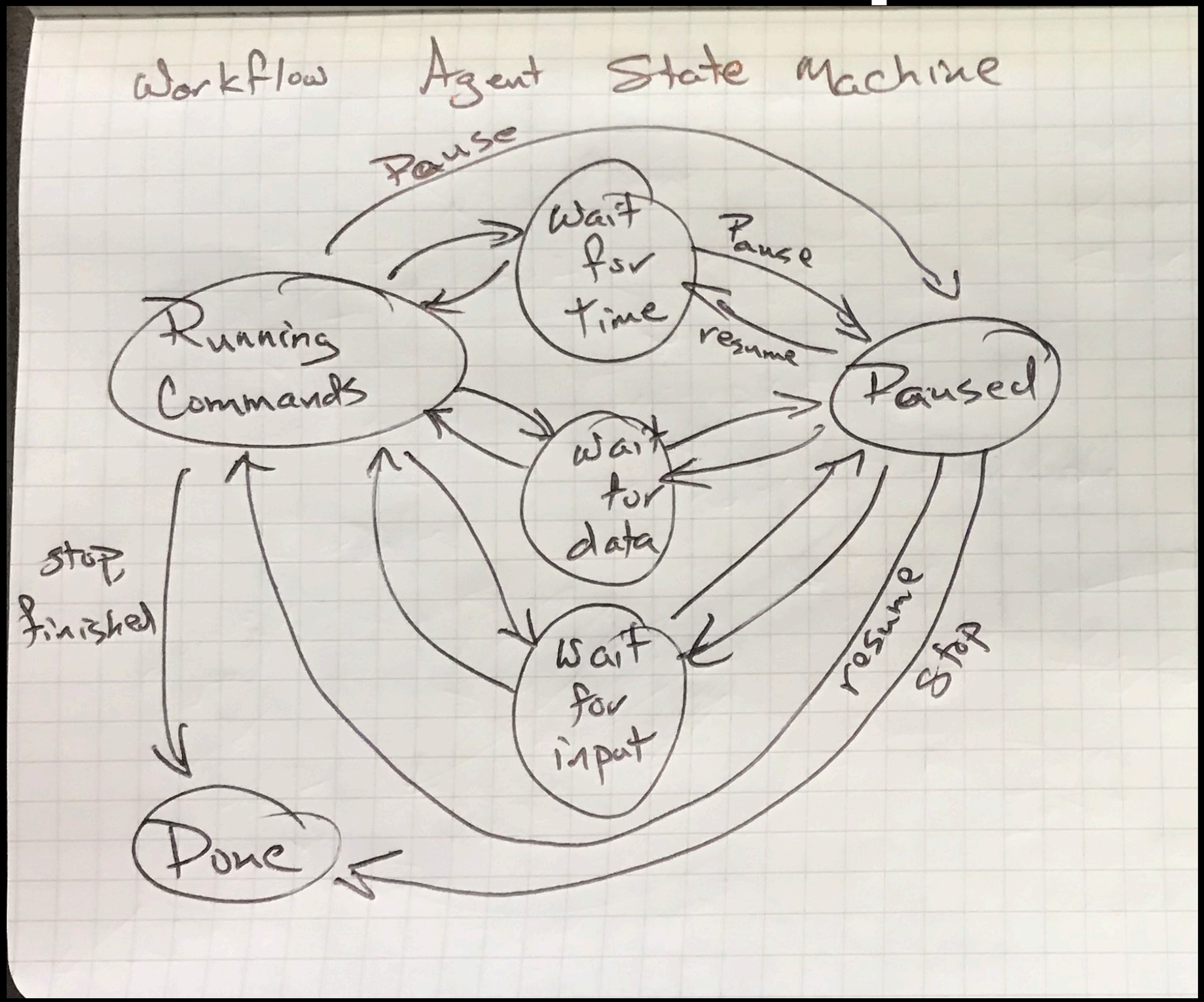
# Test Interpreter

```
ExpectedInstruction<'a> list -> Free.WorkflowProgram<'a> -> unit
let rec interpretTest expectedProgram program =
    let expected = List.head expectedProgram
    let getExpectedTail () =
        let tail = List.tail expectedProgram
        Expect.isNonEmpty tail "Expecting there to be more program"
        tail
    match program with
    | Free.Pure x -> Expect.equal (Pure x) expected "Expected to get the correct final value"
    | Free.Free (Free.SetControlState (x, next)) ->
        match expected with
        | SetControlState (state, res) ->
            Expect.equal x state "Expected to get correct argument for SetControlState"
            res |> next |> (getExpectedTail () |> interpretTest)
        | _ ->
            failtest (sprintf "Expected %A but got %A instead" expected program)
| Free.Free (Free.SetDeviceState (x, next)) ->
    match expected with
    | SetDeviceState (arg, res) ->
        Expect.equal x arg "Expected to get correct argument for SetDeviceState"
        res |> next |> (getExpectedTail () |> interpretTest)
    | _ ->
        failtest (sprintf "Expected %A but got %A instead" expected program)
| Free.Free (Free.AddControlMsg (x, next)) ->
    match expected with
    | AddControlMsg (arg, res) ->
        Expect.equal x arg "Expected to get correct argument for AddControlMsg"
        res |> next |> (getExpectedTail () |> interpretTest)
```

# Workflow Interpreter



# Workflow Interpreter



# Workflow Results

```
// Used to capture the results of workflow actions so that the workflow can be played back later.
type WorkflowResult =
    | SetControlState of unit
    | SetDeviceState of unit
    | AddControlMsg of unit
    | ClearControlMsgs of unit
    | AddDeviceMsg of unit
    | ClearDeviceMsgs of unit
    | GetDeviceInput of string
    | CancelDeviceInput of unit
    | WaitStart of start:DateTime * duration:int<Free.seconds>
    | Wait of unit
    | WaitForData of int
    | ResetData of unit
    | GetCurrentData of int
    | Pause of start:DateTime
```

# Workflow Interpreter

## When we don't have previous results

```
WorkflowResult list -> Free.WorkflowProgram<unit> -> Async<unit>
and interpretNoResults results program =
    async {
        match program with
        | Free.Pure x ->
            interpLog.Info (sprintf "Pure %A" x)
            //Don't bother saving state here, workflow is done
            return x
        | Free.Free (Free.SetControlState (state, next)) ->
            interpLog.Info (sprintf "SetControlState %A" state)
            let! res =  workflowAgent.PostAndAsyncReply (fun r -> RunnerAgentMsg.Action (ActionMsg.SetControlState (state, r)))
            interpLog.Info (sprintf "SetControlState %A result: %A" state res)
            let! newResults = saveResults results (WorkflowResult.SetControlState ())
            return! res |> next |> interpretNoResults newResults
        | Free.Free (Free.SetDeviceState (state, next)) ->
            interpLog.Info (sprintf "SetDeviceState %A" state)
            do! workflowAgent.PostAndAsyncReply (fun r -> RunnerAgentMsg.Action (ActionMsg.SetDeviceState (state, r)))
            interpLog.Info (sprintf "SetControlState %A done" state)
            let! newResults = saveResults results (WorkflowResult.SetDeviceState ())
            return! () |> next |> interpretNoResults newResults
        | Free.Free (Free.AddControlMsg (msg, next)) ->
            interpLog.Info (sprintf "AddControlMsg %A" msg)
            do! workflowAgent.PostAndAsyncReply (fun r -> RunnerAgentMsg.Action (ActionMsg.AddControlMsg (msg, r)))
            interpLog.Info (sprintf "AddControlMsg %A done" msg)
            let! newResults = saveResults results (WorkflowResult.AddControlMsg ())
            return! () |> next |> interpretNoResults newResults
        | Free.Free (Free.ClearControlMsgs ((), next)) ->
            interpLog.Info "ClearControlMsgs"
            do! workflowAgent.PostAndAsyncReply (fun r -> RunnerAgentMsg.Action (ActionMsg.ClearControlMsgs r))
            interpLog.Info "ClearControlMsgs done"
            let! newResults = saveResults results (WorkflowResult.ClearControlMsgs ())
            return! () |> next |> interpretNoResults newResults
        | Free.Free (Free.AddDeviceMsg (msg, next)) ->
            interpLog.Info (sprintf "AddDeviceMsg %A" msg)
            do! workflowAgent.PostAndAsyncReply (fun r -> RunnerAgentMsg.Action (ActionMsg.AddDeviceMsg (msg, r)))
            interpLog.Info (sprintf "AddDeviceMsg %A done" msg)
```

# Workflow Interpreter

## When we do have previous results

```
WorkflowResult list -> WorkflowResult list -> Free.WorkflowProgram<unit> -> Async<unit>
and interpretWithResults prevResults curResults program =
    async {
        match program with
        | Free.Pure x ->
            interpLog.Info (sprintf "saved Pure %A" x)
            return x
        | Free.Free (Free.SetControlState (x, next)) ->
            match prevResults with
            | (WorkflowResult.SetControlState res) :: rest ->
                interpLog.Info (sprintf "SetControlState %A saved result: %A" x res)
                let! newResults = saveResults curResults (WorkflowResult.SetControlState res)
                return! res |> next |> interpret rest newResults
            | result :: _ -> failwith (sprintf "Was expecting SetControlState saved result but got %A" result)
            | [] -> failwith "Was expecting SetControlState saved result but got no result"
        | Free.Free (Free.SetDeviceState (x, next)) ->
            match prevResults with
            | (WorkflowResult.SetDeviceState res) :: rest ->
                interpLog.Info (sprintf "SetDeviceState %A saved result: %A" x res)
                let! newResults = saveResults curResults (WorkflowResult.SetDeviceState res)
                return! res |> next |> interpret rest newResults
            | result :: _ -> failwith (sprintf "Was expecting SetDeviceState saved result but got %A" result)
            | [] -> failwith "Was expecting SetDeviceState saved result but got no result"
        | Free.Free (Free.AddControlMsg (x, next)) ->
            match prevResults with
            | (WorkflowResult.AddControlMsg res) :: rest ->
                interpLog.Info (sprintf "AddControlMsg %A saved result: %A" x res)
                let! newResults = saveResults curResults (WorkflowResult.AddControlMsg res)
                return! res |> next |> interpret rest newResults
            | result :: _ -> failwith (sprintf "Was expecting AddControlMsg saved result but got %A" result)
            | [] -> failwith "Was expecting AddControlMsg saved result but got no result"
        | Free.Free (Free.ClearControlMsgs (x, next)) ->
            match prevResults with
            | (WorkflowResult.ClearControlMsgs res) :: rest ->
                interpLog.Info (sprintf "ClearControlMsgs %A saved result: %A" x res)
                let! newResults = saveResults curResults (WorkflowResult.ClearControlMsgs res)
```

# Exercise 6

- Add a new workflow action
  - Checkout exercise6 branch
  - See Free.fs in the workflow project
    - Start by adding a case to the WorkflowInstruction type and then follow the instructions above that type
- Run your workflow
  - Run the device: dotnet run -p device/device.fsproj
  - Run the control: dotnet run -p control/control.fsproj
  - Select the R*eset Device* workflow in control and then select start

# Exercise 6

**exercise6-solution branch, search for "//Exercise 6:"**

**Free.fs**

```
type WorkflowInstruction<'a> =
    | SetControlState of state:string * (unit -> 'a)
    | SetDeviceState of state:Device.State * (unit -> 'a)
    | AddControlMsg of msg:string * (unit -> 'a)
    | ClearControlMsgs of unit * (unit -> 'a)
    | AddDeviceMsg of msg:string * (unit -> 'a)
    | ClearDeviceMsgs of unit * (unit -> 'a)
    | GetDeviceInput of prompt:string * (string -> 'a)
    | CancelDeviceInput of unit * (unit -> 'a)
    | Wait of duration:int<seconds> * (unit -> 'a)
    | WaitForData of minDataValue:int * (int -> 'a)
    | ResetData of unit * (unit -> 'a)
   //Exercise 6: Added new instruction
    | ResetDataWithMsg of unit * (unit -> 'a)
    | GetCurrentData of unit * (int -> 'a)
```

```
('a -> 'b) -> WorkflowInstruction<'a> -> WorkflowInstruction<'b>
let private mapI f = function
    | SetControlState (x, next) -> SetControlState (x, next >> f)
    | SetDeviceState (x, next) -> SetDeviceState (x, next >> f)
    | AddControlMsg (x, next) -> AddControlMsg (x, next >> f)
    | ClearControlMsgs (x, next) -> ClearControlMsgs (x, next >> f)
    | AddDeviceMsg (x, next) -> AddDeviceMsg (x, next >> f)
    | ClearDeviceMsgs (x, next) -> ClearDeviceMsgs (x, next >> f)
    | GetDeviceInput (x, next) -> GetDeviceInput (x, next >> f)
    | CancelDeviceInput (x, next) -> CancelDeviceInput (x, next >> f)
    | Wait (x, next) -> Wait (x, next >> f)
    | WaitForData (x, next) -> WaitForData (x, next >> f)
    | ResetData (x, next) -> ResetData (x, next >> f)
   //Exercise 6: added handler for ResetDataWithMsg
    | ResetDataWithMsg (x, next) -> ResetDataWithMsg (x, next >> f)
    | GetCurrentData (x, next) -> GetCurrentData (x, next >> f)
```

```
module Actions =
    // ...

    //Exercise 6: Added convenience function for ResetDataWithMsg
    unit -> WorkflowProgram<unit>
    let resetDataWithMsg () = Free (ResetDataWithMsg ((), Pure))

    // ...
```

# Exercise 6

**exercise6-solution branch, search for “//Exercise 6:”**

**test.fs**

```
type ExpectedInstruction<'a> =
    | SetControlState of state:string * unit
    | SetDeviceState of state:Device.State * unit
    | AddControlMsg of msg:string * unit
    | ClearControlMsgs of unit * unit
    | AddDeviceMsg of msg:string * unit
    | ClearDeviceMsgs of unit * unit
    | GetDeviceInput of prompt:string * string
    | CancelDeviceInput of unit * unit
    | Wait of duration:int<Free.seconds> * unit
    | WaitForData of minDataValue:int * int
    | ResetData of unit * unit
    //Exercise 6: Added case for ResetDataWithMsg
    | ResetDataWithMsg of unit * unit
    | GetCurrentData of unit * int
    | Pure of 'a
```

```
ExpectedInstruction<'a> list -> Free.WorkflowProgram<'a> -> unit
let rec interpretTest expectedProgram program =
    let expected = List.head expectedProgram
    let getExpectedTail () =
        let tail = List.tail expectedProgram
        Expect.isNonEmpty tail "Expecting there to be more program"
        tail
    match program with
    | Free.Pure x -> Expect.equal (Pure x) expected "Expected to get the correct final value"
    //Exercise 6: Added handler for ResetDataWithMsg
    | Free.Free (Free.ResetDataWithMsg (x, next)) ->
        match expected with
        | ResetDataWithMsg (arg, res) ->
            Expect.equal x arg "Expected to get correct argument for ResetDataWithMsg"
            res |> next |> (getExpectedTail () |> interpretTest)
        | _ ->
            failtest (sprintf "Expected %A but got %A instead" expected program)
    | Free.Free (Free.SetControlState (x, next)) ->
        match expected with
        | SetControlState (state, res) ->
            Expect.equal x state "Expected to get correct argument for SetControlState"
            res |> next |> (getExpectedTail () |> interpretTest)
        | _ ->
            failtest (sprintf "Expected %A but got %A instead" expected program)
    | Free.Free (Free.SetDeviceState (x, next)) ->
```

# Exercise 6

**exercise6-solution branch, search for “//Exercise 6:”**

**WorkflowRunner.fs**

```fsharp
// Used to capture the results of workflow actions so t
type WorkflowResult =
    //Exercise 6: Added case for ResetDataWithMsg
    | ResetDataWithMsg of unit
    | SetControlState of unit
    | SetDeviceState of unit
    | AddControlMsg of unit
    | ClearControlMsgs of unit
    | AddDeviceMsg of unit
    | ClearDeviceMsgs of unit
    | GetDeviceInput of string
```

```fsharp
WorkflowResult list -> WorkflowResult list -> Free.WorkflowProgram<unit> -> Async<unit>
and interpretWithResults prevResults curResults program =
    async {
        match program with
        | Free.Pure x ->
            interpLog.Info (sprintf "saved Pure %A" x)
            return x
        //Exercise 6: Added handler for ResetDataWithMsg
        | Free.Free (Free.ResetDataWithMsg (x, next)) ->
            match prevResults with
            | (WorkflowResult.ResetDataWithMsg res) :: rest ->
                interpLog.Info (sprintf "ResetDataWithMsg %A saved result: %A" x res)
                let! newResults = saveResults curResults (WorkflowResult.ResetDataWithMsg res)
                return! res |> next |> interpret rest newResults
            | result :: _ -> failwith (sprintf "Was expecting ResetDataWithMsg saved result but got %A" result)
            | [] -> failwith "Was expecting ResetDataWithMsg saved result but got no result"
        | Free.Free (Free.SetControlState (x, next)) ->
            match prevResults with
            | (WorkflowResult.SetControlState res) :: rest ->
                interpLog.Info (sprintf "SetControlState %A saved result: %A" x res)
                let! newResults = saveResults curResults (WorkflowResult.SetControlState res)
```

# Exercise 6

**exercise6-solution branch, search for “//Exercise 6:”**

## WorkflowRunner.fs

```
WorkflowResult list -> Free.WorkflowProgram<unit> -> Async<unit>
and interpretNoResults results program =
    async {
        match program with
        | Free.Pure x ->
            interpLog.Info (sprintf "Pure %A" x)
            //Don't bother saving state here, workflow is done
            return x
        //Exercise 6: Added handler for ResetDataWithMsg
        | Free.Free (Free.ResetDataWithMsg ((), next)) ->
            interpLog.Info "ResetDataWithMsg"
            do! workflowAgent.PostAndAsyncReply (fun r -> RunnerAgentMsg.Action (ActionMsg.ResetData r))
            do! workflowAgent.PostAndAsyncReply (fun r -> RunnerAgentMsg.Action (ActionMsg.AddDeviceMsg ("reset data", r)))
            interpLog.Info "ResetDataWithMsg done"
            let! newResults = saveResults results (WorkflowResult.ResetDataWithMsg ())
            return! () |> next |> interpretNoResults newResults
        | Free.Free (Free.SetControlState (state, next)) ->
            interpLog.Info (sprintf "SetControlState %A" state)
            let! res =  workflowAgent.PostAndAsyncReply (fun r -> RunnerAgentMsg.Action (ActionMsg.SetControlState (state, r)))
            interpLog.Info (sprintf "SetControlState %A result: %A" state res)
            let! newResults = saveResults results (WorkflowResult.SetControlState ())
```

## workflows.fs

```
WorkflowProgram<unit>
let reset = workflow {
    do! setDeviceState Device.Good
    do! cancelDeviceInput ()
    do! clearDeviceMsgs ()
    do! clearControlMsgs ()
    //Exercise 6: converted resetData to resetDataWithMsg (also moved to end so the message doesn't get cleared)
    do! resetDataWithMsg ()
}
```

# Possible Issues

- *Free Monads Considered Harmful*
  - https://markkarpov.com/post/free-monad-considered-harmful.html
  - Can be hard to inspect, follow through the debugger
  - Efficiency
  - Composability

# Possible Issues

- *F# Free Monad Recipe*
  - https://blog.ploeh.dk/2017/08/07/f-free-monad-recipe/
  - Should you use this?
  - Be ready to answer questions about what and how

# Longer-Term Workflows

- Make things more “web app” friendly:
  - Probably don’t need pause/resume functionality
  - Serialize previous results to database for each user
  - Use previous results to restore workflow state next time user is doing something under the workflow
    - allows migration to another machine
    - might need to suspend at start of each action and restore state when action result is ready
  - Probably need to version workflows in order to deal with old results in database

# Resources

- https://blog.ploeh.dk/2017/08/07/f-free-monad-recipe/
- F# for fun and profit (computation expressions)
  - https://fsharpforfunandprofit.com/series/computation-expressions.html
- Free monads in category theory (if you're interested, **not required**)
  - https://joa.sh/posts/2016-03-23-free-monads.html
  - *Category Theory for Programmers* (Bartosz Milewski)
    - https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/
  - *Category Theory, 2nd Edition* (Steve Awodey)