F#

COS 326
David Walker
Princeton University

Slide credits: Material drawn from:
https://fsharpforfunandprofit.com/posts/computation-expressions-intro/
https://fsharpforfunandprofit.com/posts/concurrency-async-and-parallel/
https://en.wikibooks.org/wiki/F_Sharp_Programming/Async_Workflows

OCaml --> F#

Xavier Leroy
OCaml

Don Syme
F#

F# Design Goals

- Implement a great functional language
  - They chose core OCaml
- That interoperates with all of the Microsoft software
  - ie: allow seemless use of any C# .Net libraries
  - this involved integrating .Net objects into OCaml
  - this involved some compromises
- To avoid too much complexity, throw away some things
  - Simple module system
- And steal a few good ideas from other functional languages
  - eg: monads from Haskell

PS: Scala is similar

- Implement a great functional language
- That interoperates with all of the ~~Microsoft~~ Java software
  - ie: allow seemless use of any ~~C# .Net~~ Java libraries
  - this involved integrating ~~.Net~~ Java objects into a functional language
  - this involved some compromises
- ~~To avoid~~ too much complexity
- And steal a few good ideas from other functional languages
  - eg: monads from Haskell, type classes, ...
- And then throw in more stuff! https://www.scala-lang.org/

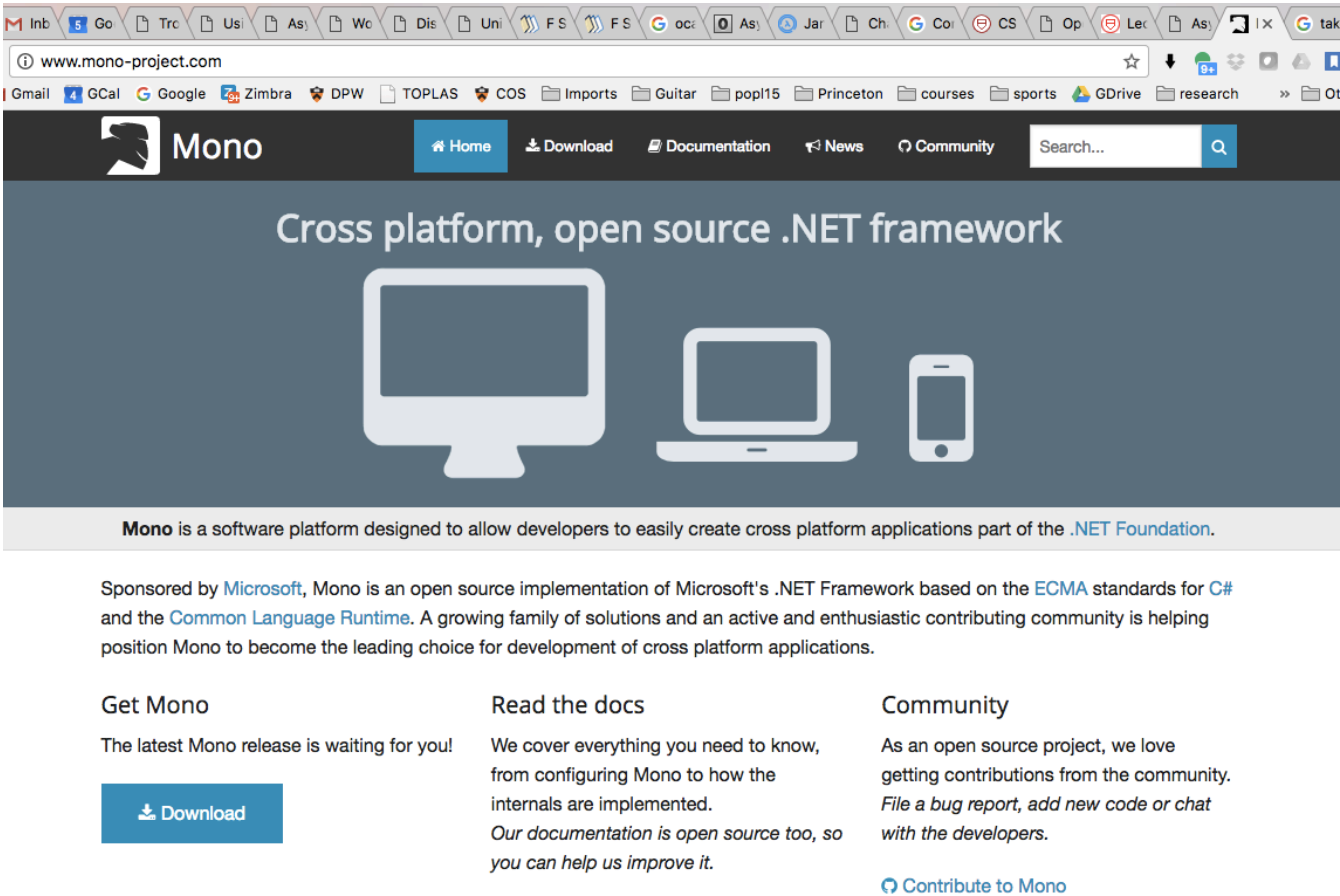
Some References

- A great blog on F# programming idioms:
  - https://fsharpforfunandprofit.com/
  - lots of lessons apply to any functional programming language

- A wikibook
  - https://en.wikibooks.org/wiki/F_Sharp_Programming
  - lots of details and examples
  - can help with minor variations in syntax from OCaml

F# INSTALL

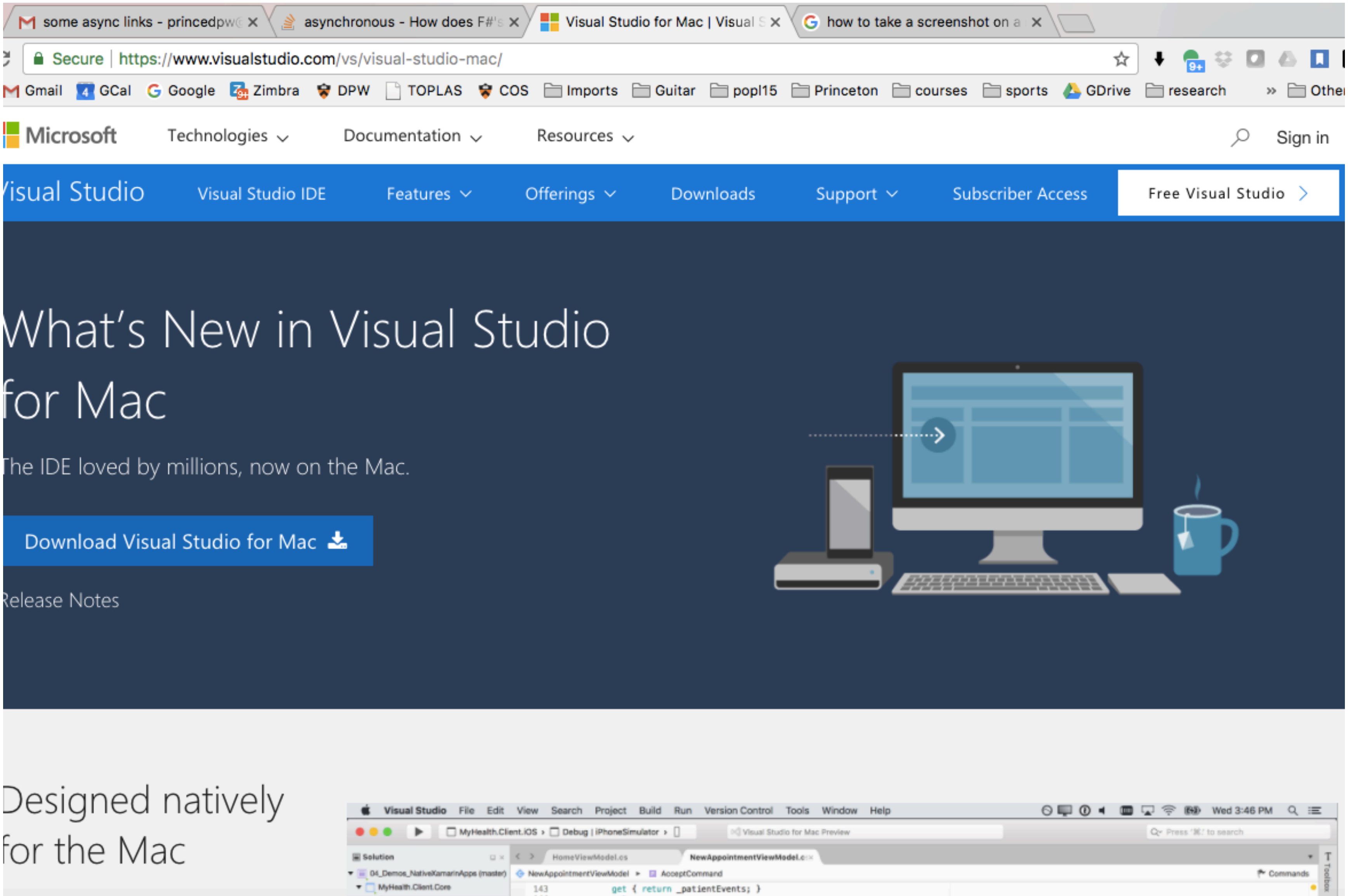
Step 1 (Mac/Linux): Get Mono



www.mono-project.com

also via homebrew

Step 2 (Mac/Linux): Download Visual Studio

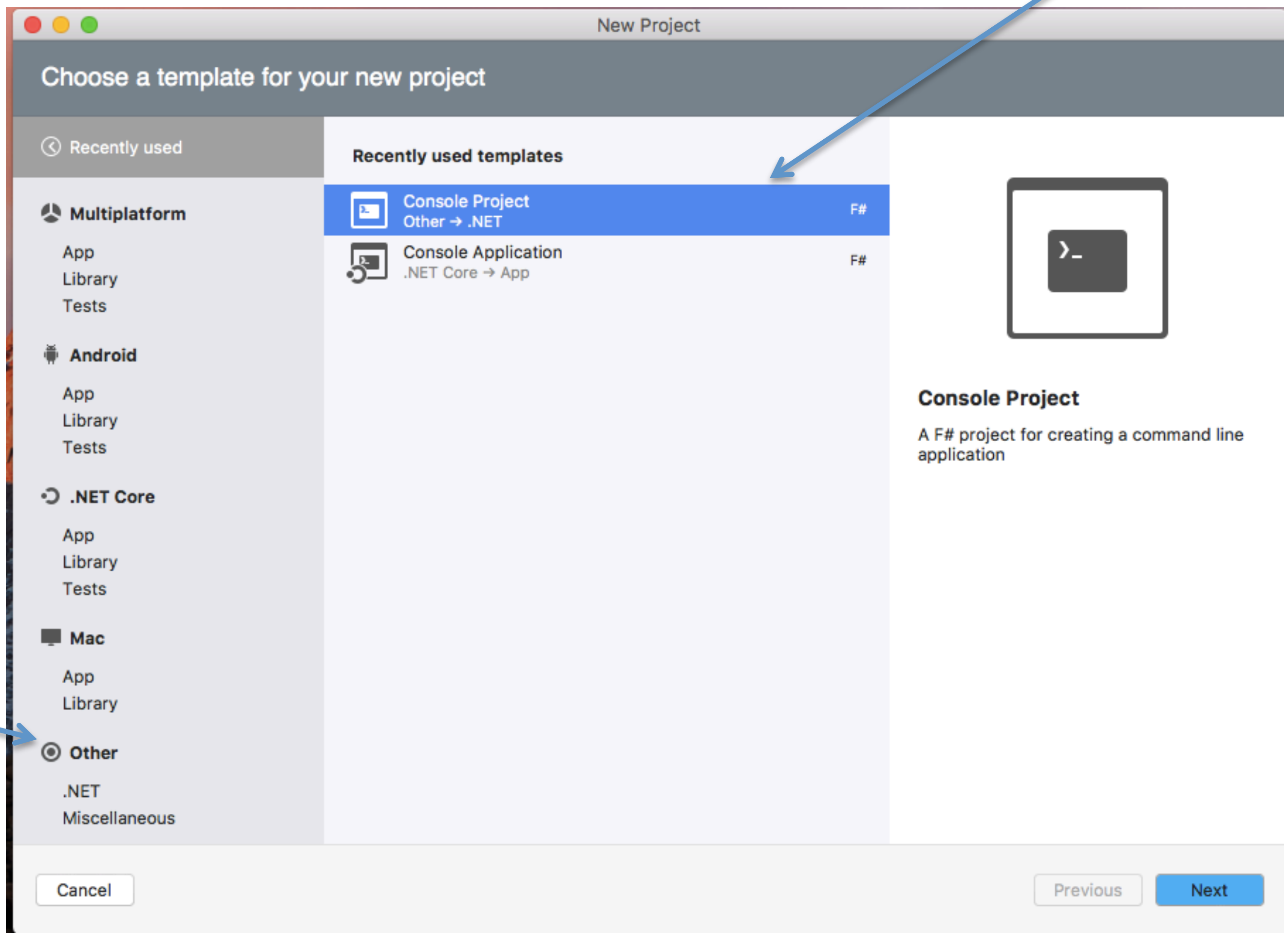


www.visualstudio.com/vs/visual-studio-mac

F# HELLO WORLD

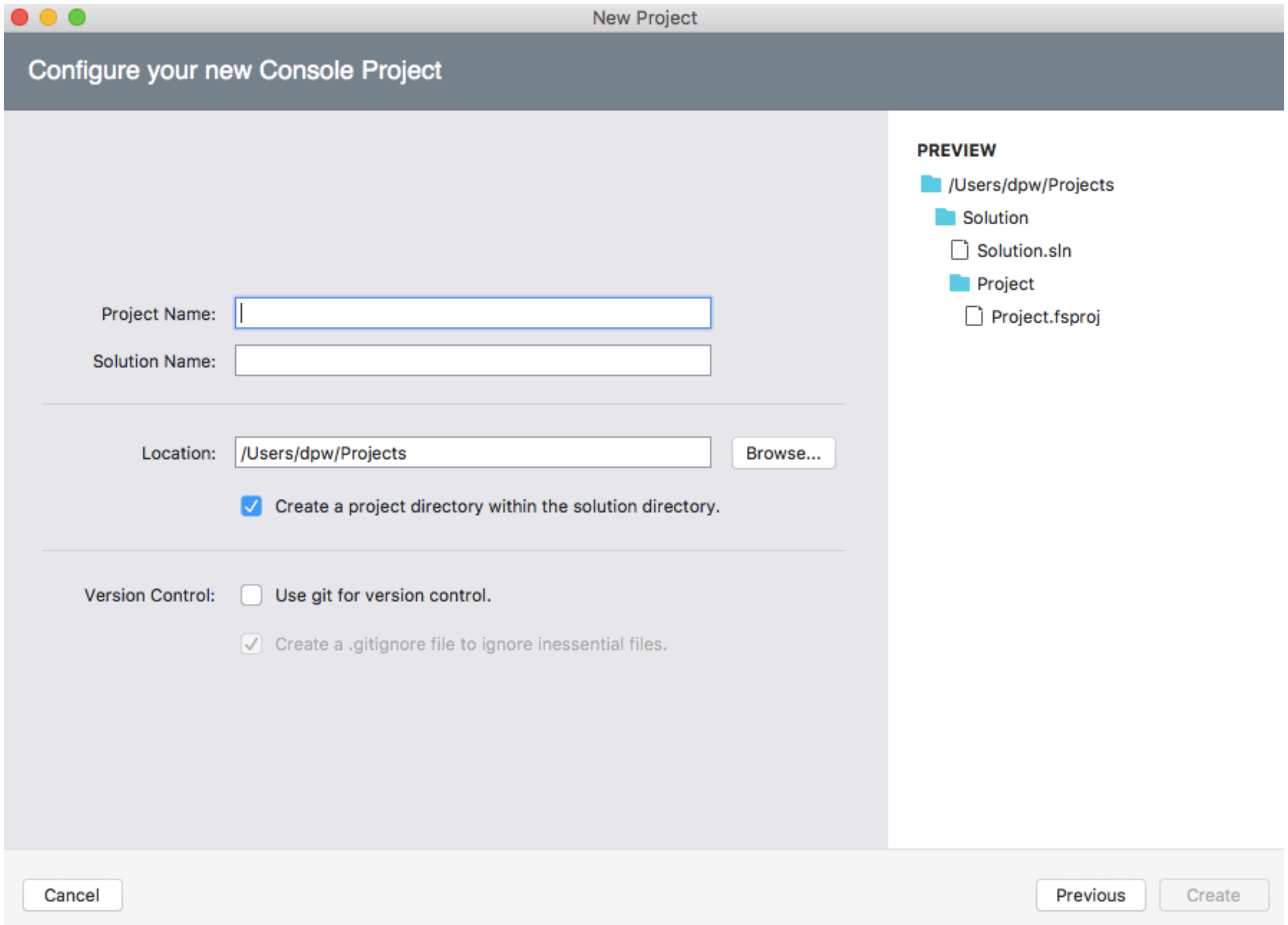
Creating a New Solution in VS

1. File Menu: "New Solution"
2. Choose a template for your new project:



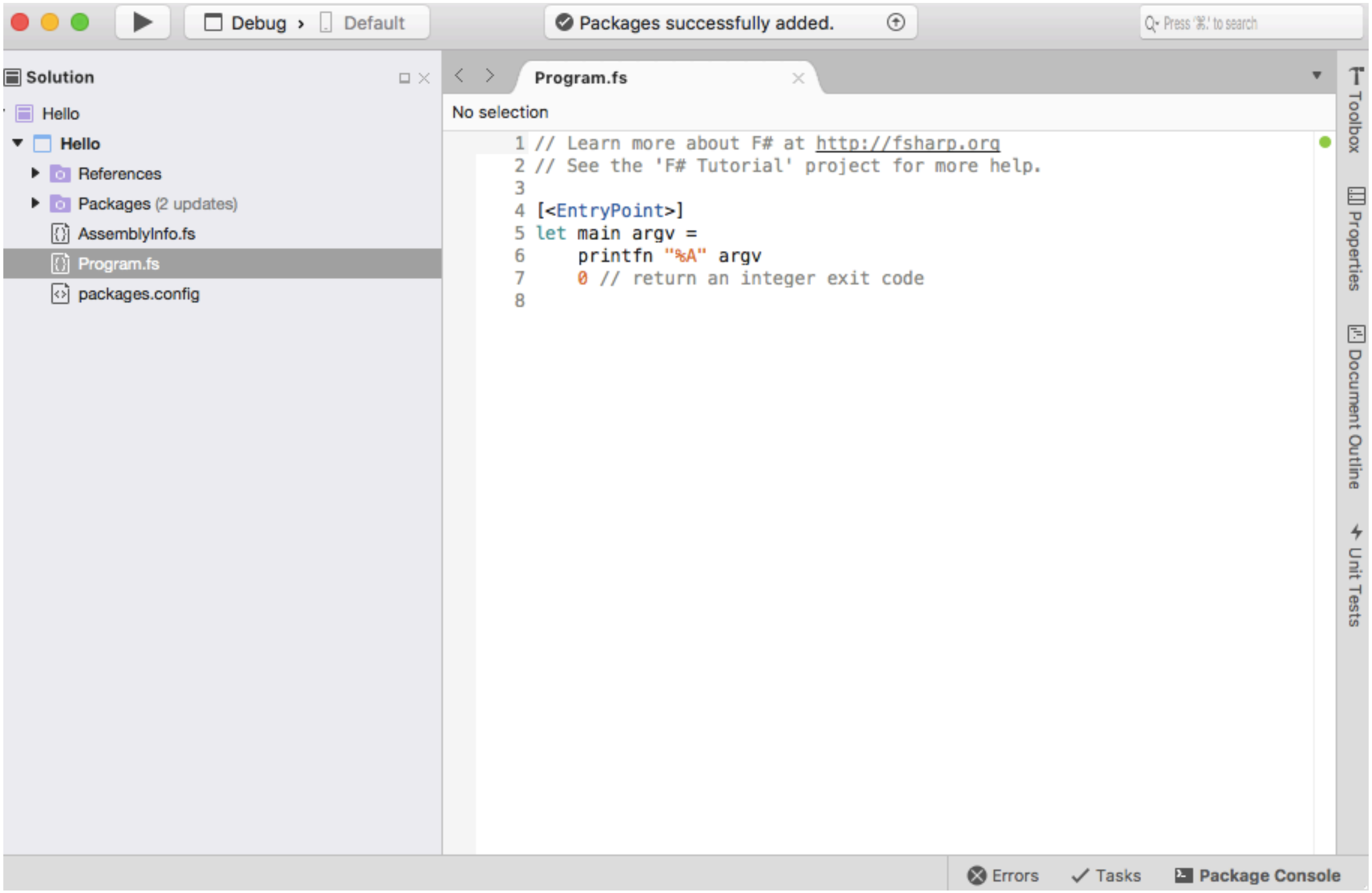
Creating a New Solution in VS

3. Choose a name:



Creating a New Solution in VS

4. Your first file and boiler plate is generated:



DEMO

PARALLEL & CONCURRENT PROGRAMMING IN F#

Recall Futures

```
module type FUTURE =
sig
  type 'a future
  val future : ('a->'b) -> 'a -> 'b future
  val force : 'a future -> 'a
end
```

```
let future f x =
 let r = ref None
 let t = Thread.create (fun _ -> r := Some(f ())) in
 let y = g() in
    Thread.join t ;
    match !r with
    | Some v ->
    | None -> failwith "impossible"
```

Recall Futures

```
module type FUTURE =
sig
  type 'a future
  val future : ('a->
  val force : 'a futu
end
```

```
let future f x =
 let r = ref Non
 let t = Thread.
 let y = g() in
    Thread.join t ;
    match !r with
    | Some v ->
    | None -> failwith "impossible"
```

Naive:

- creates a new thread every time, rather than use a thread pool
- does not handle exceptions
- does not allow for cancellation of futures
- no support for event-driven programming
- and besides, no real parallel execution

F# has a library for asyncronous computations that will handle many of these issues and more ...

Plus an elegant syntax to boot!

F# Async

Values with type Async<T> are suspended computations

- that may be run in the background, like futures
- or composed and executed in sequence, while avoiding blocking
- or executed in parallel

F# Async

Values with type Async<T> are suspended computations

- that may be run in the background, like futures
- or composed and executed in sequence, while avoiding blocking
- or executed in parallel

A function that returns a suspended computation:

```
let asyncAdd x y =
    async {
     return x + y
    }
```

F# Async

Values with type Async<T> are suspended computations

- that may be run in the background, like futures
- or composed and executed in sequence, while avoiding blocking
- or executed in parallel

A function that returns a suspended computation:

```
let asyncAdd x y =
   async {
    return x + y
   }
```

the code in here has a special syntax. It is called a *computation expression*

let's the compiler know we are beginning the construction of a suspended (async) computation with type Async<T>

F# Async

Values with type Async<T> are suspended computations

- that may be run in the background, like futures
- or composed and executed in sequence, while avoiding blocking
- or executed in parallel

A function that returns a suspended computation:

```
let asyncAdd x y =
   async {
    return x + y
   }
```

the simplest kind of async is one that does nothing but return a value

"return" is not the same as the "return" keyword in C/Java
think of it as a function with type T -> Async<T>

F# Async

Chaining asynchronous computations:

```
let asyncAdd (x:int) (y:int) : Async<int> =
    async {
      return x + y
    }

let compositeAsync () =
  async {
     let! z = asyncAdd 1 2
     let! w = asyncAdd z 1
    printfn "answer: %i" (z + w)
    return ()
  }

let main () =
       compositeAsync()
   |> Async.RunSynchronously
```

let! waits for the result of asyncAdd before continuing; bind an integer to z

allows other threads to continue in the meantime; doesn't take up resources

Async Typing

let! extracts the final value from an async computation:

```
let! x = e1
...
```

x has type T
in the following code

e1 has type Async<T>

Async Typing

let! extracts the final value from an async computation:

```
let! x = e1
...
```

x has type T
in the following code

e1 has type Async<T>

Compare with typing let:

```
let x = e1
...
```

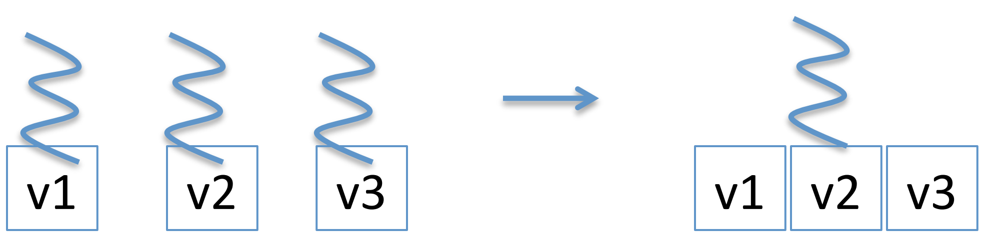
x has type Async<T>
in the following code

e1 has type Async<T>

Parallelism

Async.Parallel : seq<Async<T>> -> Async<T []>

converts a sequence of Async computations
into
an Async of an array of results

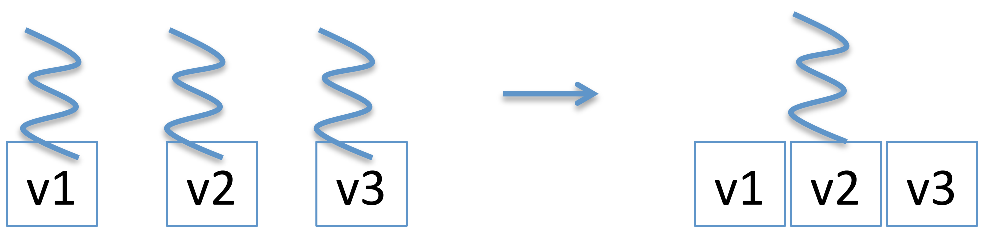


Parallelism

Async.Parallel : seq<Async<T>> -> Async<T []>

in F#, many concrete types can be viewed as a sequence: lists, arrays, ... F# uses *objects* more pervasively than OCaml

converts a sequence of Async computations
into
an Async of an array of results



A More Interesting Example

```
// Fetch the contents of a web page asynchronously
let fetchUrlAsync url =
    async {
        let req = WebRequest.Create(Uri(url))
        let! resp = req.AsyncGetResponse()
        let stream = resp.GetResponseStream()
        let reader = new IO.StreamReader(stream)
        let html = reader.ReadToEnd()
        printfn "finished downloading %s" url
        }
```

A More Interesting Example

```
// Fetch the contents of a web page asynchronously
let fetchUrlAsync url =
    async {
        let req = WebRequest.Create(Uri(url))
        let! resp = req.AsyncGetResponse()
        let stream = resp.GetResponseStream()
        let reader = new IO.StreamReader(stream)
        let html = reader.ReadToEnd()
        printfn "finished downloading %s" url
        }
```

Notice that AsyncGetResponse returns an Async.

let! causes this Async to be executed while the rest of the computation is suspended, wasting no CPU resources until the response is returned.

A More Interesting Example

```
// Fetch the contents of a web page asynchronously
let fetchUrlAsync url =
    async {
        let req = WebRequest.Create(Uri(url))
        let! resp = req.AsyncGetResponse()
        let stream = resp.GetResponseStream()
        let reader = new IO.StreamReader(stream)
        let html = reader.ReadToEnd()
        printfn "finished downloading %s" url
        }
```

Notice that AsyncGetResponse returns an Async.

let! causes this Async to be executed while the rest of the computation is suspended, wasting no CPU resources until the response is returned.

Without the special let! syntax, we would have to program with continuations, which would be ugly. *We will come back to this.*

A More Interesting Example

```
// Fetch the contents of a web page asynchronously
let fetchUrlAsync (url:string) : Async<string> =  ...

let sites = ["http://www.bing.com";
           "http://www.google.com";
           "http://www.microsoft.com";
           "http://www.amazon.com";
           "http://www.yahoo.com"]

let runParallel () =
        sites
    |> List.map fetchUrlAsync      // make a list of async tasks
    |> Async.Parallel              // set up the tasks to run in parallel
    |> Async.RunSynchronously  // start them off
    |> ignore
```

Background Work

Sequential operation:

```
finished downloading http://www.microsoft.com
finished downloading http://www.google.com
finished downloading http://www.bing.com
finished downloading http://www.yahoo.com
finished downloading http://www.amazon.com
1365.457700
```

Parallel operation:

```
finished downloading http://www.bing.com
finished downloading http://www.google.com
finished downloading http://www.microsoft.com
finished downloading http://www.amazon.com
finished downloading http://www.yahoo.com
528.371000
```

COMPUTATION EXPRESSIONS

What is this?

```
async {

  ...

  }
```

```
let! x = v
e
```

A special syntax for a commonly appearing paradigm

- In F#: A *computation expression*
- In Haskell: A *monad*

The concurrency monad is but one kind of monad.

There are many others.

Monads

A monad are just abstract data types with a particular interface:

monad interface

```
type M<T>

return : T -> M<T>

bind : M<T> -> (T -> M<T>) -> M<T>
```

Monads

A monad are just abstract data types with a particular interface:

monad interface

type M<T>

return : T -> M<T>

bind : M<T> -> (T -> M<T>) -> M<T>

```
async {
    ...
  }
```

"start using the async monad now with its special syntax"

Monads

A monad are just abstract data types with a particular interface:

monad interface

```
type M<T>

return : T -> M<T>

bind : M<T> -> (T -> M<T>) -> M<T>
```

```
let! x = e1
e2
```

translated to

```
bind e1 (fun x -> e2)
```

the neat bit about a monad is that bind does some interesting "behind the scenes" work for you. It's a "programmable semi-colon"

Monads

A monad are just abstract data types with a particular interface:

```
let! x = v
e
```

translated to

```
bind v (fun x -> e)
```

```
let! x1 = f1 a
let! x2 = f2 b
let! x3 = f3 c
let! x4 = f4 d
e
```

translated to

```
bind (f1 a) (fun x1 ->
  bind (f2 b) (fun x2 ->
    bind (f3 c) (fun x3 ->
      bind (f4 d) (fun x4 -> e)
```

prettier

Monads

A monad are just abstract data types with a particular interface:

```
let! x = v
e
```

translated to

```
bind v (fun x -> e)
```

```
let! x1 = f1 a
let! x2 = f2 b
let! x3 = f3 c
let! x4 = f4 d
e
```

translated to

```
bind (f1 a) (fun x1 ->
  bind (f2 b) (fun x2 ->
    bind (f3 c) (fun x3 ->
      bind (f4 d) (fun x4 -> e)
```

prettier

(note: F# has quite a few more bits of syntax: do!, use!, ...
that may be present in computation expressions, making them
a little more than just pure monads, and even nicer sometimes)

A Logger

```
let log p = printfn "expression is %A" p

let loggedWorkflow =
    let x = 42
    log x
    let y = 43
    log y
    let z = x + y
    log z
   z
```

A Logger

```
let log p = printfn "expression is %A" p

let loggedWorkflow =
    let x = 42
    log x
    let y = 43
    log y
    let z = x + y
    log z
   z
```

output

```
expression is 42
expression is 43
expression is 85
```

A Logger

```
let log p = printfn "expression is %A" p

let loggedWorkflow =
    let x = 42
    log x
    let y = 43
    log y
    let z = x + y
    log z
   z
```

lots of repeated code

output

```
expression is 42
expression is 43
expression is 85
```

A Logger

```
type LoggingBuilder() =
    let log p = printfn "expression is %A" p

    member this.Bind(x, f) =
        log x
        f x

    member this.Return(x) =
        x
```

f# object

Bind method

Return method

output

```
expression is 42
expression is 43
expression is 85
```

A Logger

```
type LoggingBuilder() =
    let log p = printfn "expression is %A" p
    member this.Bind(x, f) = log x; f x
    member this.Return(x) = x

let logger = new LoggingBuilder()

let loggedWorkflow =
  logger {
    let! x = 42
    let! y = 43
    let! z = x + y
    z
    }
```

output

```
expression is 42
expression is 43
expression is 85
```

A Logger

```
type LoggingBuilder() =
    let log p = printfn "expression is %A" p
    member this.Bind(x, f) = log x; f x
    member this.Return(x) = x

let logger = new LoggingBuilder()

let loggedWorkflow =
  logger {
    let! x = 42
    let! y = 43
    let! z = x + y
    z
    }
```

```
let x = 42
log x
let y = 43
log y
let z = x + y
log z
z
```

output

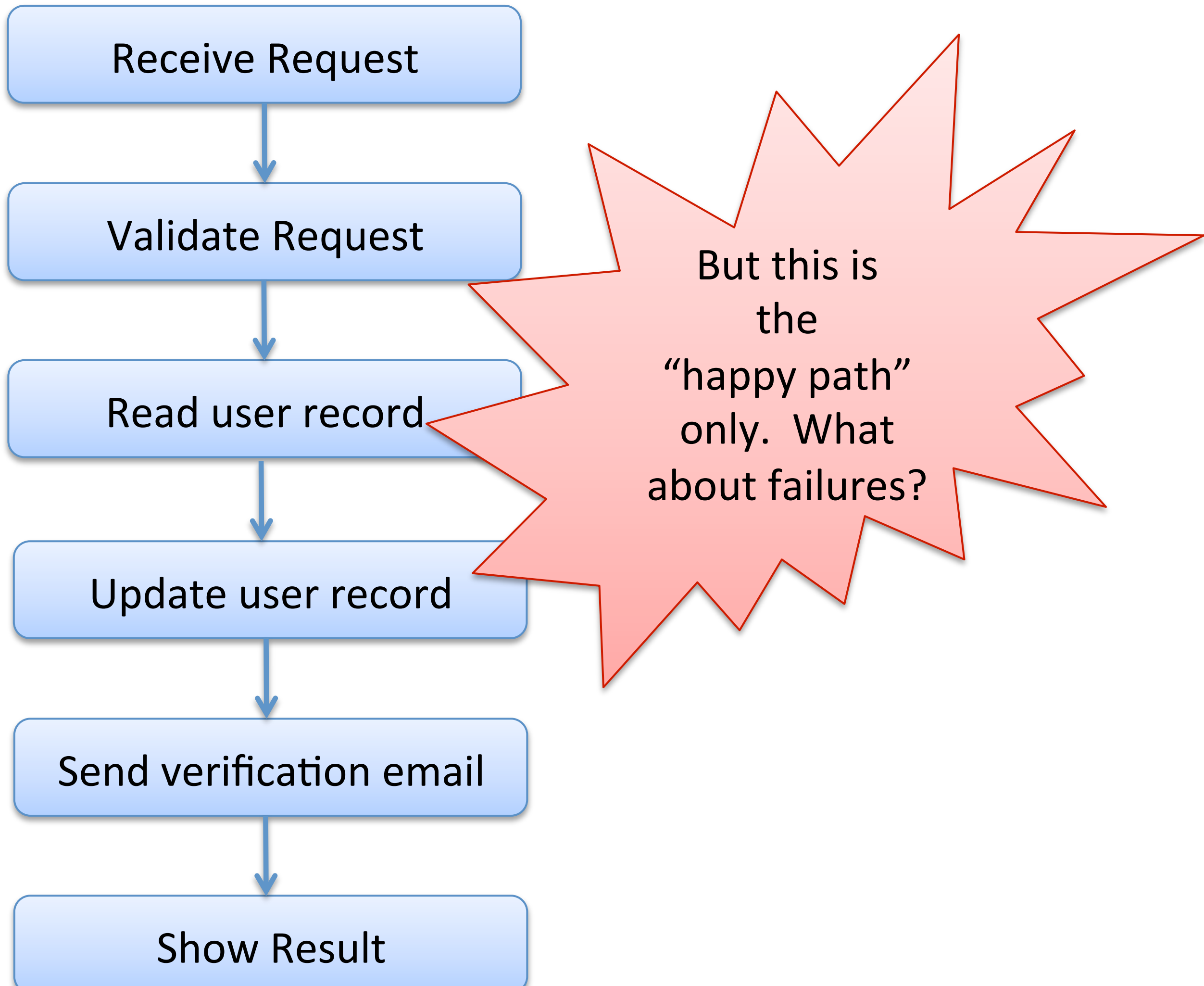
```
expression is 42
expression is 43
expression is 85
```

Another Example

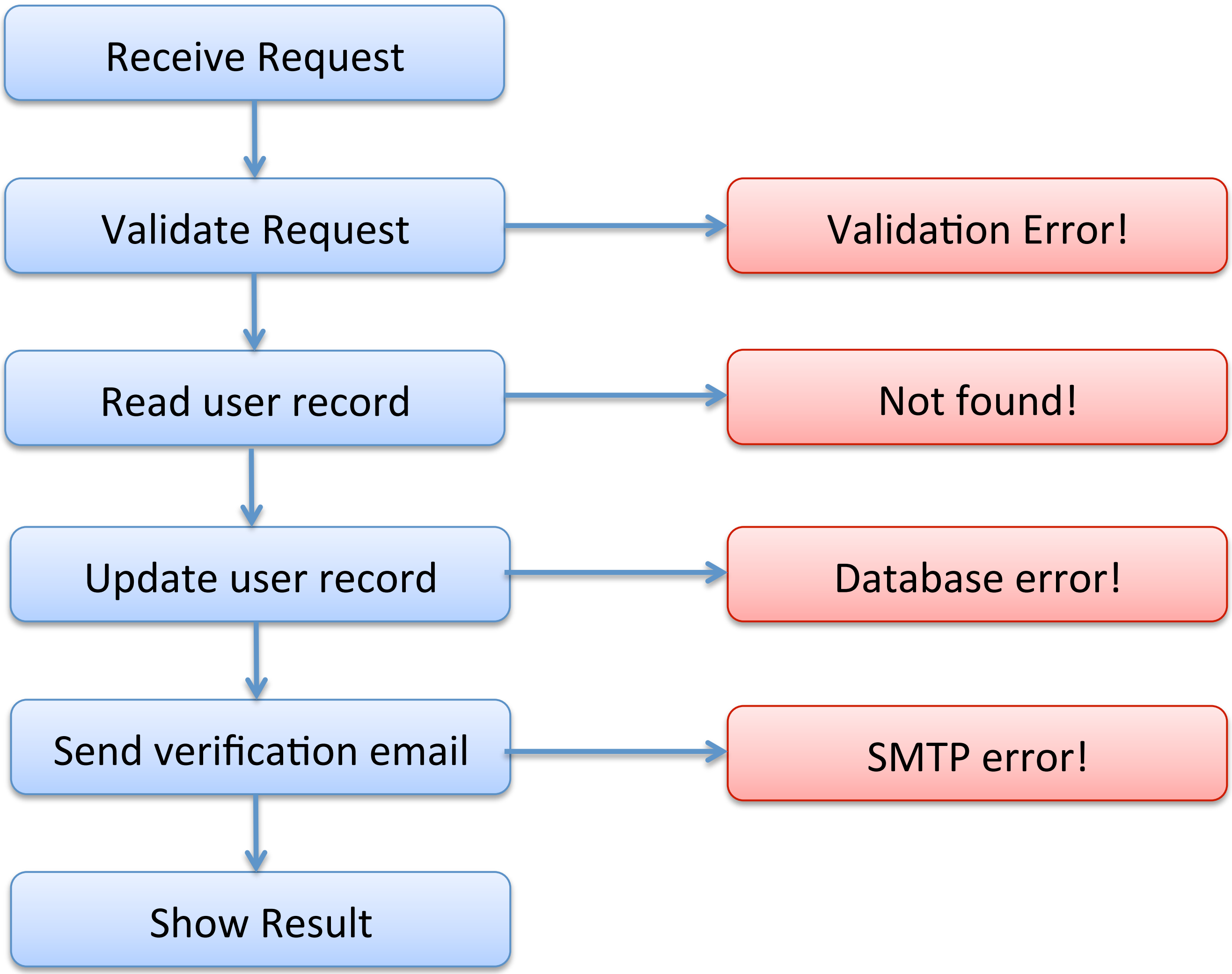
Imagine you are designing a front end for a database that takes update requests.

- A user submits some data (userid, name, email)
- Check for validity of name, email
- Update user record in database
- If email has changed, send verification email
- Display end result to user

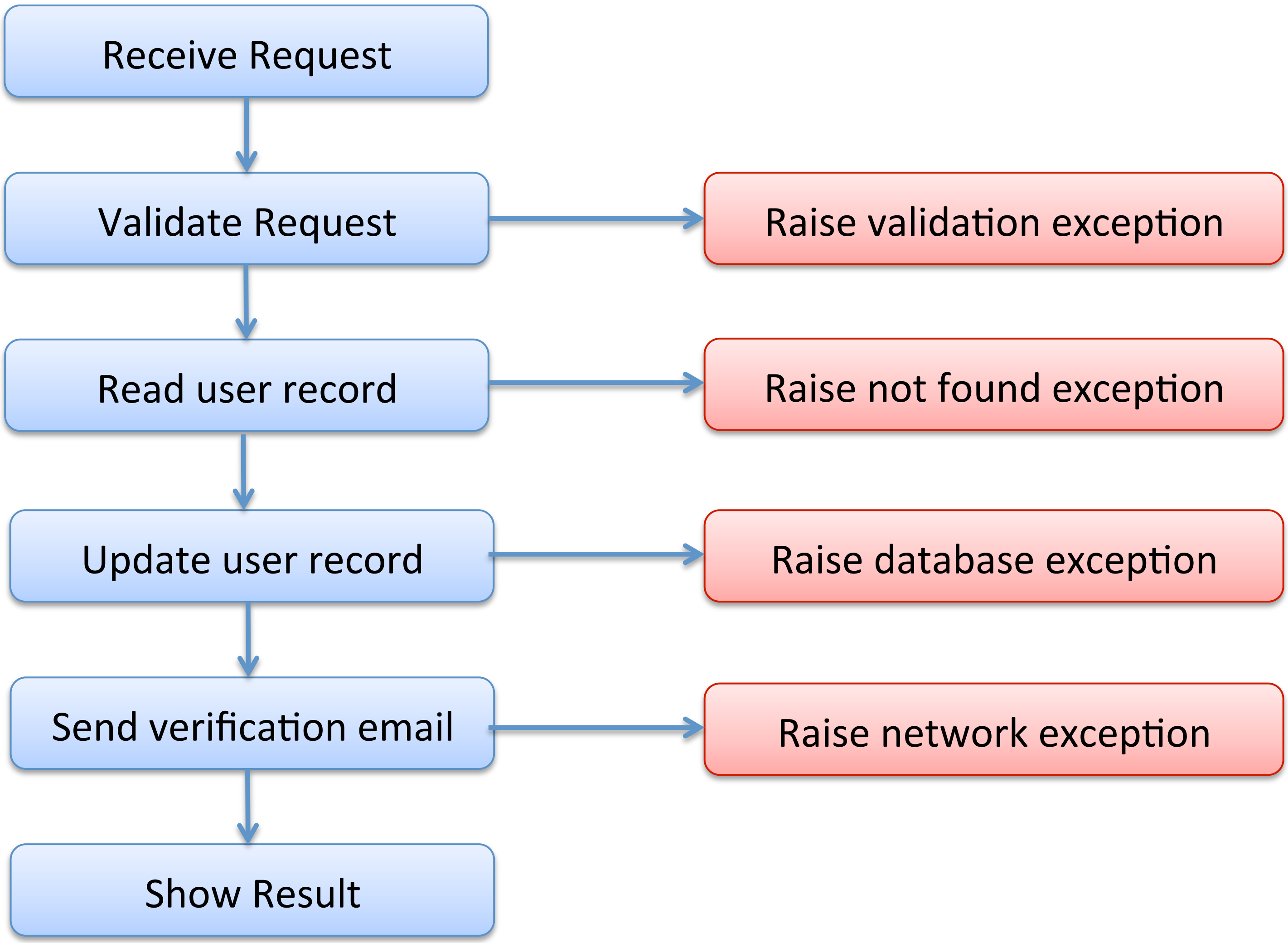
In Pictures


Receive Request
Validate Request
Read user record
Update user record
Send verification email
Show Result
But this is the "happy path" only. What about failures?

In Pictures


Receive Request
Validate Request
Validation Error!
Read user record
Not found!
Update user record
Database error!
Send verification email
SMTP error!
Show Result

One solution



The trouble with exceptions

People forget to catch them!

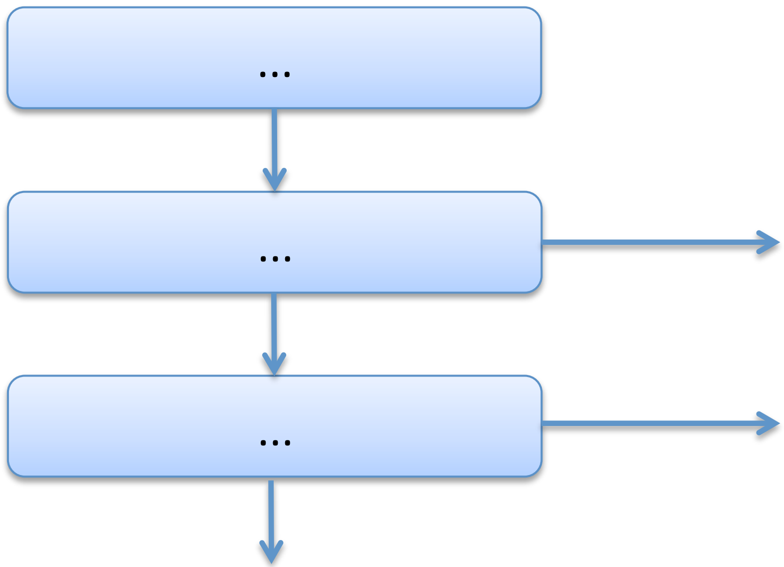
- applications fail
- *sadness* ensues
- See *A type-based analysis of uncaught exceptions*
  - by Pessaux and Leroy.
  - Uncaught exceptions: a big problem in OCaml (and Java!)
  - (not a big problem in C. Why not? ☹)

In a more functional approach, the full behavior of a program is determined exclusively *by the value it returns*, not by its “effect”

Functional Error Processing

Explicitly return "good" result or error. If we use OCaml data types to represent the two possibilities we will force the client code to process the error (or get a warning from the OCaml type checker).

Functional Error Processing



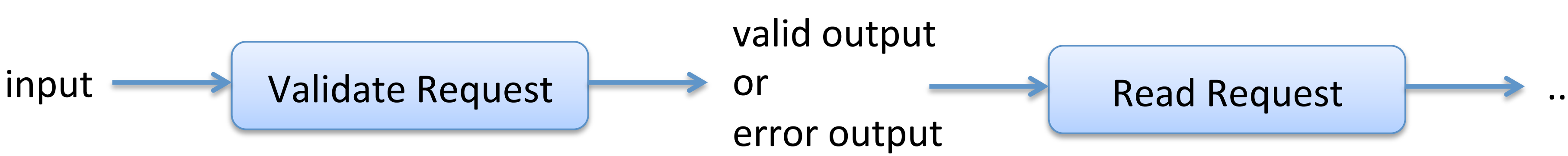
Notice input and output aren't the same type. On the surface, this makes it look awkward to compose a series of such steps, but:

Good abstractions are compositional ones.

Let's design a generic library for error processing that is *highly reuseable* and *compositional*.

Functional Error Processing

The Challenge: Composition

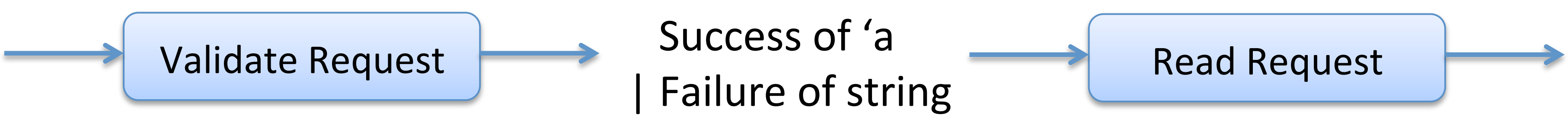


Generic Error Processing

A generic result type:

```
type 'a result =
      Success of  ‘a
    | Failure of string
```

A processing pipeline:



Validation Functions

```
type Result<'a> = Success of 'a | Failure of string
type Request = {name:string; email:string}

let validate1 (input:Request) : input Result =
  if input.name = "" then Failure "Name must not be blank"
  else Success input

let validate2 (input:Request) : input Result =
  if input.name.Length > 50 then Failure "Name must not be > 50 char"
  else Success input

let validate3 (input:Request) : input Result =
  if input.email = "" then Failure "Email must not be blank"
  else Success input
```

Validation Functions

```
type Result<'a> = Success of 'a | Failure of string
type Request = {name:string; email:string}

val validate1 : Request -> Request Result
val validate2 : Request -> Request Result
val validate3 : Request -> Request Result

let validationWorkflow input =
  match validate input with
    | Failure s -> Failure s
    | Success i2 ->
        match validate2 i2 with
          | Failure s -> Failure s
          | Success i3 ->
              match validate3 i3 with
                | Failure s -> Failure s
                | Success i4 -> Success i4
```

Validation Functions

```
type Result<'a> = Success of 'a | Failure of string
type Request = {name:string; email:string}

val validate1 : Request -> Request Result
val validate2 : Request -> Request Result
val validate3 : Request -> Request Result

let validationWorkflow input =
  match validate input with
   | Failure s -> Failure s
   | Success i2 ->
       match validate2 i2 with
        | Failure s -> Failure s
        | Success i3 ->
            match validate3 i3 with
             | Failure s -> Failure s
             | Success i4 -> Success i4
```

horrible boilerplate code

so much repetition

easy to make mistakes

ugly to read.

You can't pay people enough money to read this code carefully!

Validation Functions

```
type Result<'a> = Success of 'a | Failure of str
type Request = {name:string; email:string}

val validate1 : Request -> Request Result
val validate2 : Request -> Request Result
val validate3 : Request -> Request Result

let validationWorkflow input =
  match validate input with
    | Failure s -> Failure s
    | Success i2 ->
        match validate2 i2 with
          | Failure s -> Failure s
          | Success i3 ->
              match validate3 i3 with
                | Failure s -> Failure s
                | Success i4 -> Success i4
```

```
type FailureBuilder() =

    member this.Bind(x, f) =
        match x with
        | Failure s -> Failure s
        | Success a -> f a

    member this.Return(x) =
        Success x

let failure = new FailureBuilder()
```

Validation Functions

```
type Result<'a> = Success of 'a | Failure of str
type Request = {name:string; email:string}

val validate1 : Request -> Request Result
val validate2 : Request -> Request Result
val validate3 : Request -> Request Result

let validationWorkflow input =
  match validate1 input with
    | Failure s -> Failure s
    | Success i2 ->
        match validate2 i2 with
          | Failure s -> Failure s
          | Success i3 ->
              match validate3 i3 with
                | Failure s -> Failure s
                | Success i4 -> Success i4
```

```
type FailureBuilder() =

    member this.Bind(x, f) =
        match x with
        | Failure s -> Failure s
        | Success a -> f a

    member this.Return(x) =
        Success x

let failure = new FailureBuilder()
```

```
let validationWorkflow input =
  let! i2 = validate1 input
  let! i3 = validate2 input
  let! i4 = validate3 input
  return i4
```

Finally, Async Calls Again

```
open System.Net
let req1 = HttpWebRequest.Create("http://fsharp.org")
let req2 = HttpWebRequest.Create("http://google.com")
let req3 = HttpWebRequest.Create("http://bing.com")

req1.BeginGetResponse((fun r1 ->
   let resp1 = req1.EndGetResponse(r1)
   printfn "Downloaded %O" resp1.ResponseUri

   req2.BeginGetResponse((fun r2 ->
      let resp2 = req2.EndGetResponse(r2)
      printfn "Downloaded %O" resp2.ResponseUri

      req3.BeginGetResponse((fun r3 ->
         let resp3 = req3.EndGetResponse(r3)
         printfn "Downloaded %O" resp3.ResponseUri

         ),null) |> ignore
      ),null) |> ignore
   ),null) |> ignore
```

Finally, Async Calls Again

```
open System.Net
let req1 = HttpWebRequest.Create("http://fsharp.org")
let req2 = HttpWebRequest.Create("http://google.com")
let req3 = HttpWebRequest.Create("http://bing.com")

req1.BeginGetResponse((fun r1 ->
   let resp1 = req1.EndGetResponse(r1)
   printfn "Downloaded %O" resp1.ResponseUri

   req2.BeginGetResponse((fun r2 ->
      let resp2 = req2.EndGetResponse(r2)
      printfn "Downloaded %O" resp2.ResponseUri

      req3.BeginGetResponse((fun r3 ->
         let resp3 = req3.EndGetResponse(r3)
         printfn "Downloaded %O" resp3.ResponseUri

         ),null) |> ignore
      ),null) |> ignore
   ),null) |> ignore
```

Horrible boilerplate.

Lots of continuations (ie callbacks) inside continuations!

Finally, Async Calls Again

```
open System.Net
let req1 = HttpWebRequest.Create("http://fsharp.org")
let req2 = HttpWebRequest.Create("http://google.com")
let req3 = HttpWebRequest.Create("http://bing.com")

req1.BeginGetResponse((fun r1 ->
   let resp1 = req1.EndGetResponse(r1)
   printfn "Downloaded %O" resp1.ResponseUri

   req2.BeginGetResponse((fun r2 ->
      let resp2 = req2.EndGetResponse(r2)
      printfn "Downloaded %O" resp2.ResponseUri

      req3.BeginGetResponse((fun r3 ->
         let resp3 = req3.EndGetResponse(r3)
         printfn "Downloaded %O" resp3.ResponseUri

         ),null) |> ignore
      ),null) |> ignore
   ),null) |> ignore
```

```
open System.Net
let req1 = HttpWebRequest.Create("http://fsharp.org")
let req2 = HttpWebRequest.Create("http://google.com")
let req3 = HttpWebRequest.Create("http://bing.com")

async {
   let! resp1 = req1.AsyncGetResponse()
   printfn "Downloaded %O" resp1.ResponseUri

   let! resp2 = req2.AsyncGetResponse()
   printfn "Downloaded %O" resp2.ResponseUri

   let! resp3 = req3.AsyncGetResponse()
   printfn "Downloaded %O" resp3.ResponseUri

   } |> Async.RunSynchronously
```

Monads, Technically

A *monad* is a (*set of values*, *bind*, *return*) that satisfies these equational laws:

```
bind (return a, f)  ==  f a
```

```
bind(m, return)  ==  m
```

```
bind(m, (fun x -> bind(k x,  h))  ==  bind (bind(m, k), h)
```

In Haskell, the compiler could actually use such laws to optimize a program (in theory ... not sure if it does this in practice).

But programmers expect these kinds of laws to be true and may rearrange their programs with them in mind

Monads, Technically

Monads are particularly important in Haskell because:

- functions with type a -> b do not have effects!*
- they are pure!*
- they don't print, or use mutable references!*
- the type system enforces this property*

Haskell does have effectful computations

- they have type IO b
  - where IO b is the "IO monad"
  - when you run this kind of computation at the top level, effects happen
- lots of Haskell functions have type a -> M b
  - they are "pure" functions, that produce a computation
- lots of times in this class, we have said "this equational law only applies when we are working with pure functions"
  - Haskell actually enforces the caveat with its type system!*

Monads, Technically

Monads are particularly important in Haskell because:

- functions with type a -> b do not have effects!*
- they are pure!*
- they don't print, or use mutable references!*
- the type system enforces this property*

Haskell does have effectful computations

- they have type IO b
  - where IO b is the "IO monad"
  - when you run this kind of computation at the top level, effects happen
- lots of Haskell functions have type a -> M b
  - they are "pure" functions, that produce a computation
- lots of times in this class, we have said "this equational law only applies when we are working with pure functions"
  - Haskell actually enforces the caveat with its type system!*

* There is a function called PerformUnsafeIO ... you can guess what it does :-)
But people avoid using it most of the time.

More Computation Expressions(!)

Construct	De-sugared Form
let pat = expr in cexpr	let pat = expr in cexpr
let! pat = expr in cexpr	b.Bind(expr, (fun pat -> cexpr))
return expr	b.Return(expr)
return! expr	b.ReturnFrom(expr)
yield expr	b.Yield(expr)
yield! expr	b.YieldFrom(expr)
use pat = expr in cexpr	b.Using(expr, (fun pat -> cexpr))
use! pat = expr in cexpr	b.Bind(expr, (fun x -> b.Using(x, fun pat -> cexpr))
do! expr in cexpr	b.Bind(expr, (fun () -> cexpr))
for pat in expr do cexpr	b.For(expr, (fun pat -> cexpr))
while expr do cexpr	b.While((fun () -> expr), b.Delay(fun () -> cexpr))
if expr then cexpr1 else cexpr2	if expr then cexpr1 else cexpr2
if expr then cexpr	if expr then cexpr else b.Zero()
try cexpr with patn -> cexprn	b.TryWith(expr, fun v -> match v with (patn:ext) -> cexprn | _ raise exn)
try cexpr finally expr	b.TryFinally(cexpr, (fun () -> expr))
cexpr1 cexpr2	b.Combine(cexpr1, b.Delay(fun () -> cexpr2))

One More Example

```
let map1 = [ ("1","One"); ("2","Two") ]                        |> Map.ofList
let map2 = [ ("A","Alice"); ("B","Bob") ]                      |> Map.ofList
let map3 = [ ("CA","California"); ("NY","New York") ] |> Map.ofList

let multiLookup key =
   match map1.TryFind key with
   | Some result1 -> Some result1   // success
   | None ->                        // failure
          match map2.TryFind key with
          | Some result2 -> Some result2 // success
          | None ->                      // failure
                 match map3.TryFind key with
                 | Some result3 -> Some result3  // success
                 | None -> None                  // failure
```

One More Example

```
let map1 = [ ("1","One"); ("2","Two") ]
let map2 = [ ("A","Alice"); ("B","Bob") ]
let map3 = [ ("CA","California"); ("NY","New Yo

let multiLookup key =
    match map1.TryFind key with
    | Some result1 -> Some result1   // success
    | None ->
            match map2.TryFin
            | Some result2 -> S
            | None ->
                    match ma
                    | Some re
                    | None ->
```

```
let multiLookup key =
  orElse    {
    return! map1.TryFind key
    return! map2.TryFind key
    return! map3.TryFind key
    }
```

```
type OrElseBuilder() =
    member this.ReturnFrom(x) = x
    member this.Combine (a,b) =
        match a with
        | Some _ -> a  // a succeeds -- use it
        | None -> b    // a fails -- use b instead
    member this.Delay(f) = f()

let orElse = new OrElseBuilder()
```

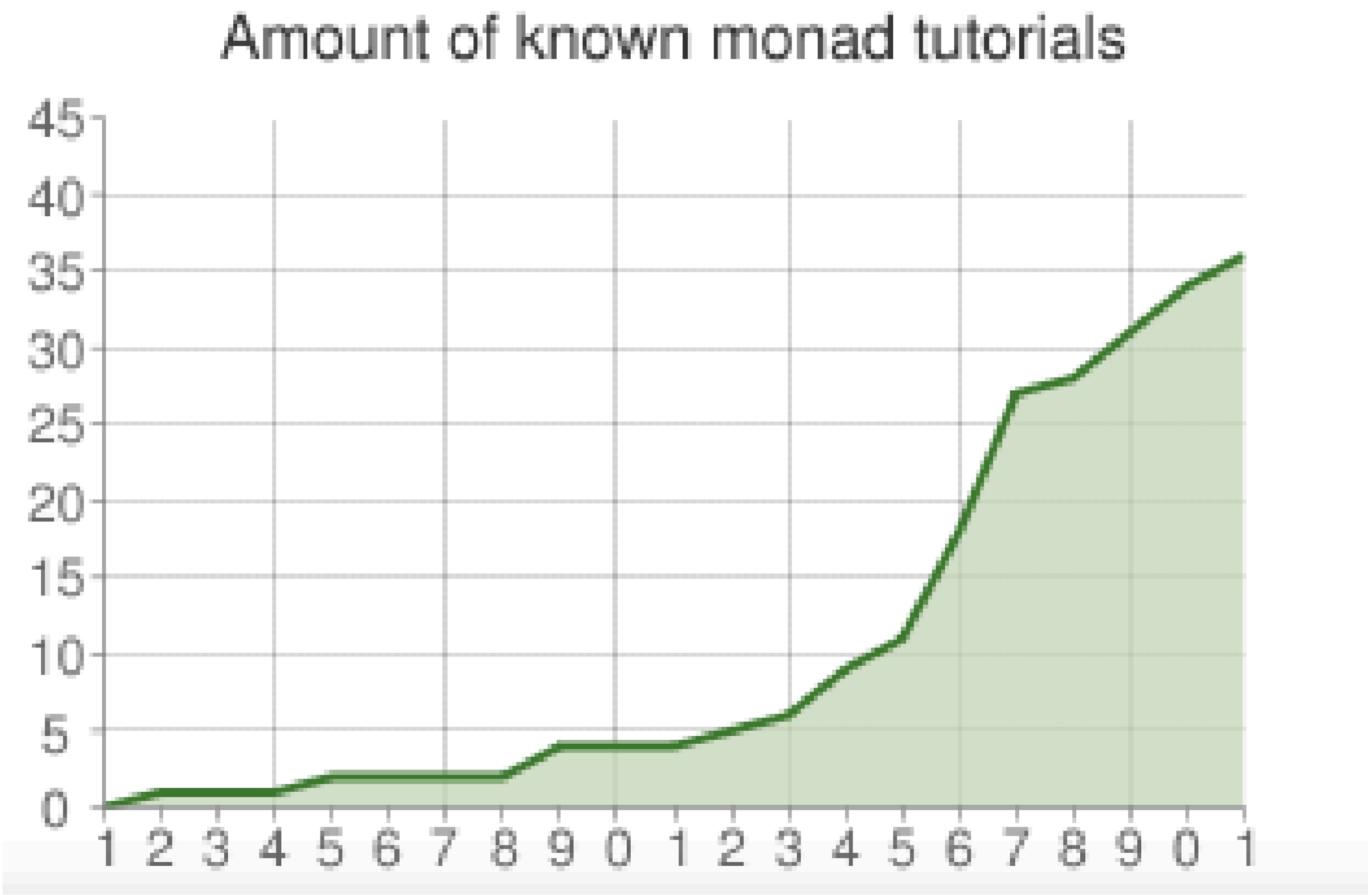
More Monads & Computation Expressions

Monads for:

- parsing elegantly
- transactional software memory (a concurrency paradigm)
- error handling
- imperative state (mutable data)
- database programming
- ...

More computation expressions

- https://fsharpforfunandprofit.com/posts/computation-expressions-intro/

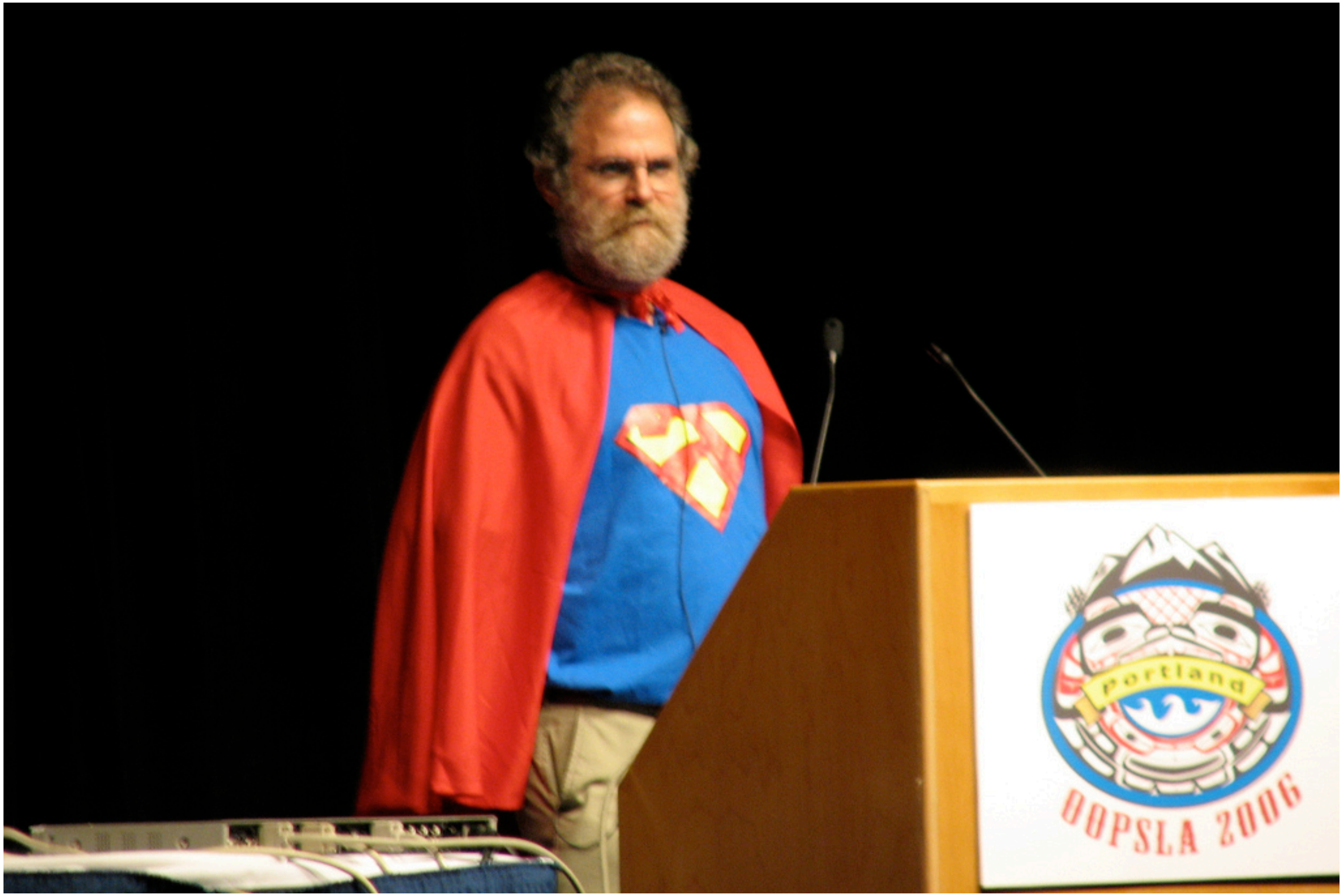


(Picture from Wadler)

An academic paper: Comprehending Monads. Phil Wadler.

https://ncatlab.org/nlab/files/WadlerMonads.pdf

OOPSLA 2006



Phil Wadler at a conference on *object-oriented* programming (OOPSLA) advocating for *functional* programming

Assignment #7

- Parallel algorithms in F#
  - Async.Parallel
- GO TO PRECEPT THIS WEEK! I THINK IT WILL HELP!
  - if you get stuck installing F# over holiday break and did not go to precept, we will have little pity for you.
- I RARELY USE ALLCAPS ON MY SLIDES
- CONSIDER THIS A HINT
- Before precept, install F# on your laptop