Parallelism 2

COS 326
David Walker
Princeton University

Last Time

Parallel complexity can be described in terms of work and span

Parallel programming interfaces:

- Futures
  - future and force
- Parallel collection interfaces (eg: sequences)
  - tabulate
  - map
  - filter
  - reduce

Implementations: Google map-reduce; Hadoop

Key idea: Parallel functional libraries have sequential semantics

PARALLEL SCAN AND PREFIX SUM

The prefix-sum problem

Sum of Sequence:

input	6	4	16	10	16	14	2	8
output	76							

Prefix-Sum of Sequence:

input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	76

The prefix-sum problem

```
val prefix_sum : int seq -> int seq
```

`input`	6	4	16	10	16	14	2	8
`output`	6	10	26	36	52	66	68	76

The simple sequential algorithm: accumulate the sum from left to right

- Sequential algorithm: Work: $O(n)$, Span: $O(n)$
- Goal: a parallel algorithm with Work: $O(n)$, Span: O(log n)

Parallel prefix-sum

The trick: *Use two passes*

- Each pass has $O(n)$ work and $O(\texttt{log}\ n)$ span
- So in total there is $O(n)$ work and $O(\texttt{log}\ n)$ span

First pass *builds a tree of sums bottom-up*

- the “up” pass

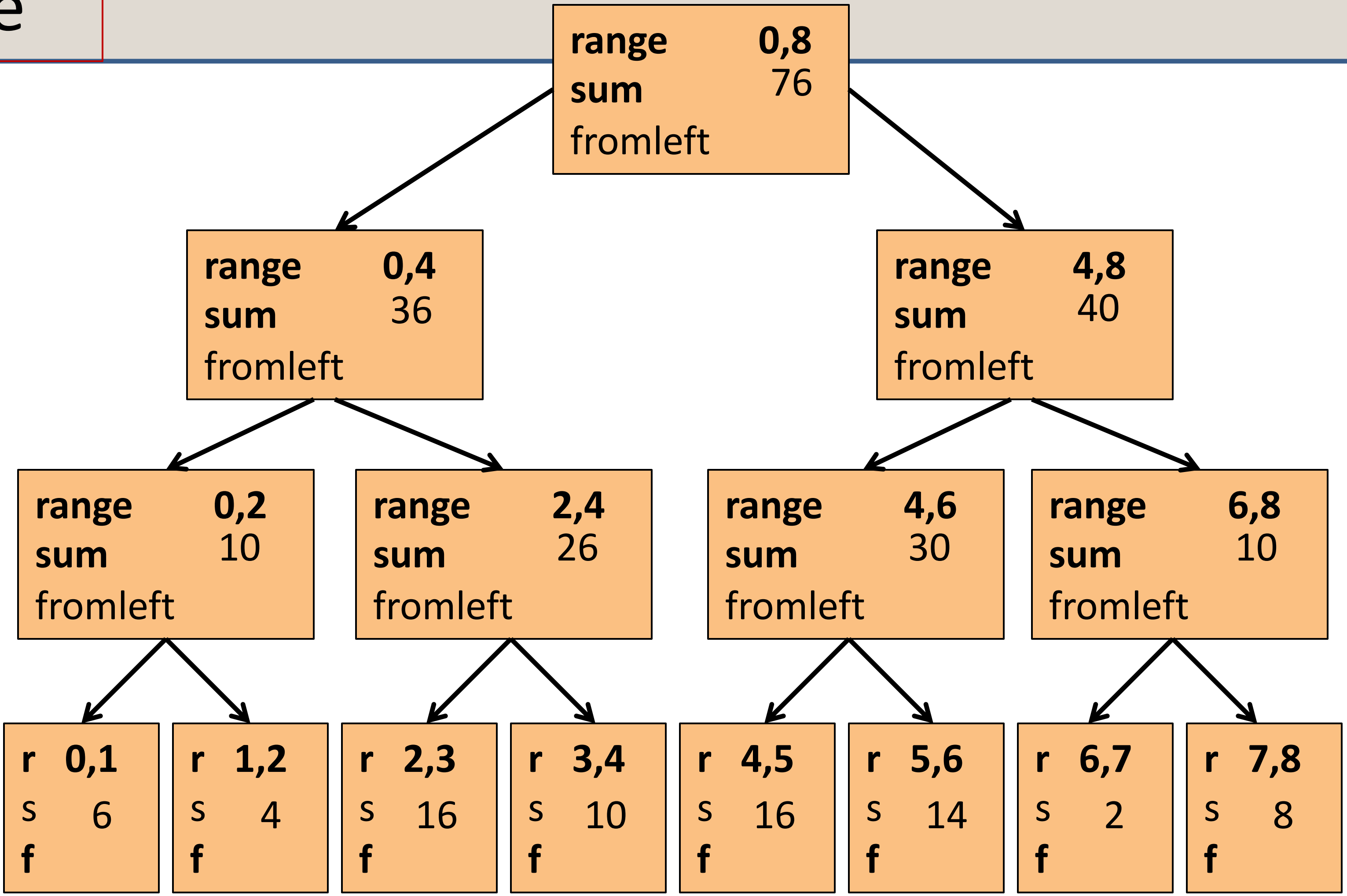
Second pass *traverses the tree top-down to compute prefixes*

- the “down” pass computes the "from-left-of-me" sum

Historical note:

- Original algorithm due to R. Ladner and M. Fischer, 1977

Example



range **0,8**
sum 76
fromleft

range **0,4**
sum 36
fromleft

range **4,8**
sum 40
fromleft

range **0,2**
sum 10
fromleft

range **2,4**
sum 26
fromleft

range **4,6**
sum 30
fromleft

range **6,8**
sum 10
fromleft

r **0,1**
s 6
f

r **1,2**
s 4
f

r **2,3**
s 16
f

r **3,4**
s 10
f

r **4,5**
s 16
f

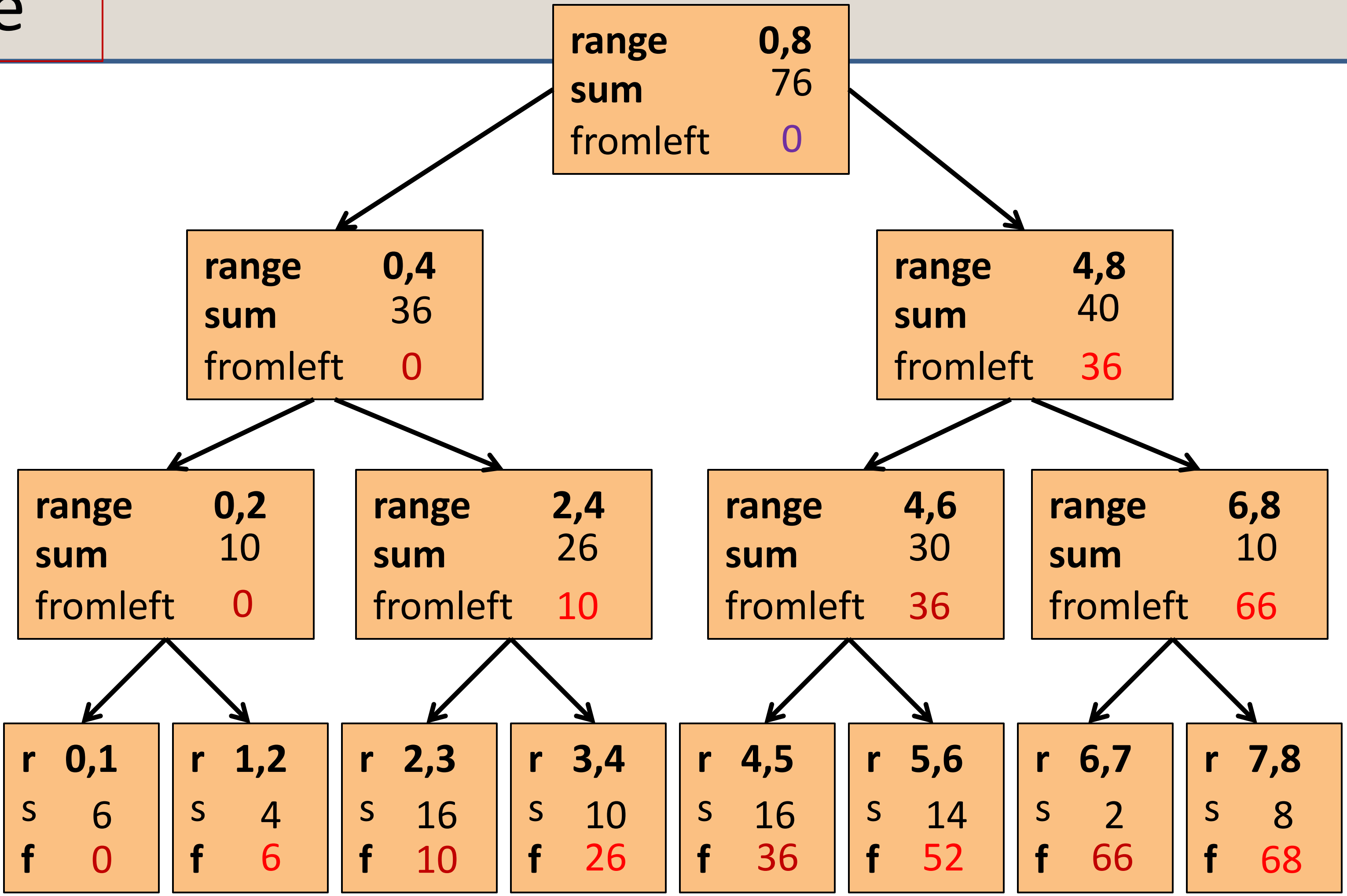
r **5,6**
s 14
f

r **6,7**
s 2
f

r **7,8**
s 8
f

`input`	6	4	16	10	16	14	2	8
`output`								

Example



range **0,8**
sum 76
fromleft 0

range **0,4**
sum 36
fromleft 0

range **4,8**
sum 40
fromleft 36

range **0,2**
sum 10
fromleft 0

range **2,4**
sum 26
fromleft 10

range **4,6**
sum 30
fromleft 36

range **6,8**
sum 10
fromleft 66

r **0,1**
s 6
f 0

r **1,2**
s 4
f 6

r **2,3**
s 16
f 10

r **3,4**
s 10
f 26

r **4,5**
s 16
f 36

r **5,6**
s 14
f 52

r **6,7**
s 2
f 66

r **7,8**
s 8
f 68

`input`	6	4	16	10	16	14	2	8
`output`	6	10	26	36	52	66	68	76

The algorithm, pass 1

1. Up: Build a binary tree where
   - Root has sum of the range [**`x,y`**)
   - If a node has sum of [**`lo,hi`**) and **`hi>lo`**,
     - Left child has sum of [**`lo,middle`**)
     - Right child has sum of [**`middle,hi`**)
     - A leaf has sum of [**`i,i+1`**), i.e., **`nth input i`**

This is an easy parallel divide-and-conquer algorithm: "combine" results by actually building a binary tree with all the range-sums

- Tree built bottom-up in parallel

Analysis: $O(n)$ work, $O(\mathbf{log}\ n)$ span

The algorithm, pass 2

2. Down: Pass down a value **`fromLeft`**
   - Root given a **`fromLeft`** of `0`
   - Node takes its **`fromLeft`** value and
     - Passes its left child the same **`fromLeft`**
     - Passes its right child its **`fromLeft`** plus its left child's **`sum`**
       - as stored in part 1
   - At the leaf for sequence position **`i`**,
     - **`nth output i == fromLeft + nth input i`**

This is an easy parallel divide-and-conquer algorithm:

traverse the tree built in step 1 and produce no result

- Leaves create **`output`**
- Invariant: **`fromLeft`** is sum of elements left of the node's range

Analysis: *O*(*n*) work, *O*(**`log`** *n*) span

Sequential cut-off

For performance, we need a sequential cut-off:

- Up:
  - just a sum, have leaf node hold the sum of a range
- Down:
  - do a sequential scan

Parallel prefix, generalized

Just as map and reduce are the simplest examples of a common pattern, prefix-sum illustrates a pattern that arises in many, many problems

- Minimum, maximum of all elements *to the left of* ***`i`***
- Is there an element *to the left of* ***`i`*** satisfying some property?
- Count of elements *to the left of* ***`i`*** satisfying some property
  - This last one is perfect for an efficient parallel filter …
  - Perfect for building on top of the “parallel prefix trick”

Parallel Scan

scan (o) <x1, ..., xn>

==

<x1, x1 o x2, ..., x1 o ... o xn>

like a fold, except return
the folded prefix at each step

pre_scan (o) base <x1, ..., xn>

==

<base, base o x1, ..., base o x1 o ... o xn-1>

sequence with o applied to all items
to the left of index in input

Parallel Filter

Given a sequence **`input`**, produce a sequence **`output`** containing only elements v such that **`(f v)`** is **`true`**

Example: let f x = x > 10

```
    filter f <17, 4, 6, 8, 11, 5, 13, 19, 0, 24>
 == <17, 11, 13, 19, 24>
```

Parallelizable?

- Finding elements for the output is easy
- *But getting them in the right place seems hard*

Parallel prefix to the rescue

Use parallel map to compute a bit-vector for true elements:

```
input  <17, 4, 6, 8, 11, 5, 13, 19, 0, 24>
bits   <1,  0, 0, 0,  1, 0,  1,  1, 0,  1>
```

Use parallel-prefix sum on the bit-vector:

```
bitsum <1,  1, 1, 1,  2, 2,  3,  4, 4,  5>
```

For each i, if bits[i] == 1 then write input[i] to output[bitsum[i]] to produce the final result:

```
output <17, 11, 13, 19, 24>
```

QUICKSORT

Quicksort review

Recall quicksort was sequential, in-place, expected time $O(n \log n)$

	Best / expected case *work*
1. Pick a pivot element	$O(1)$
2. Partition all the data into:	$O(n)$
A. The elements less than the pivot	
B. The pivot	
C. The elements greater than the pivot	
3. Recursively sort A and C	$2T(n/2)$

How should we parallelize this?

Quicksort

	Best / expected case *work*
1. Pick a pivot element	O(1)
2. Partition all the data into:	O(n)
A. The elements less than the pivot	
B. The pivot	
C. The elements greater than the pivot	
3. Recursively sort A and C	2T(n/2)

Easy: Do the two recursive calls in parallel

- Work: unchanged. Total: $O(n \log n)$
- Span: now $T(n) = O(n) + 1T(n/2) = O(n)$

Doing better

As with mergesort, we get a $O(\texttt{log}\ n)$ speed-up with an *infinite* number of processors. That is a bit underwhelming

- Sort 10^9 elements 30 times faster

(Some) Google searches suggest quicksort cannot do better because the partition cannot be parallelized

- The Internet has been known to be wrong ☺
- But we need auxiliary storage (no longer in place)
- In practice, constant factors may make it not worth it

Already have everything we need to parallelize the partition…

Parallel partition (not in place)

Partition all the data into:

A. The elements less than the pivot
B. The pivot
C. The elements greater than the pivot

This is just two filters!

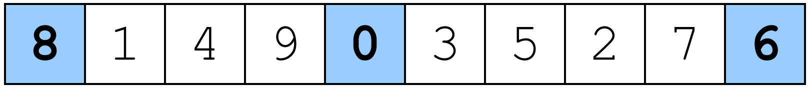
- We know a parallel filter is $O(n)$ work, $O(\log n)$ span
- Parallel filter elements less than pivot into left side of **`aux`** array
- Parallel filter elements greater than pivot into right size of **`aux`** array
- Put pivot between them and recursively sort

With $O(\log n)$ span for partition, the total best-case and expected-case span for quicksort is

$$T(n) = O(\log n) + 1T(n/2) = O(\log^2 n)$$

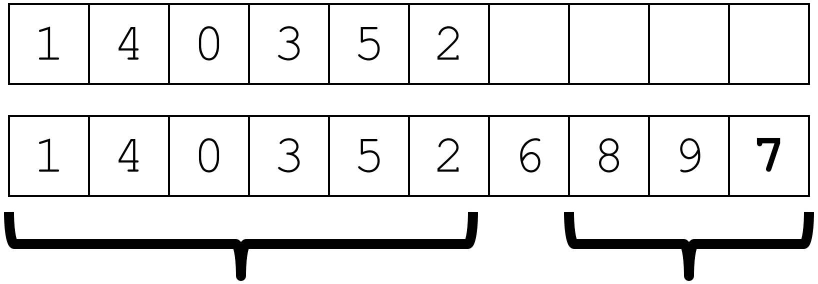
Example

Step 1: pick pivot as median of three



8	1	4	9	**0**	3	5	2	7	**6**

Steps 2a and 2c (combinable): filter less than, then filter greater than into a second array



1	4	0	3	5	2				
1	4	0	3	5	2	6	8	9	**7**

Step 3: Two recursive sorts in parallel

- Can copy back into original array (like in mergesort)

More Algorithms

- To add multiprecision numbers.
- To evaluate polynomials
- To solve recurrences.
- To implement radix sort
- To delete marked elements from an array
- To dynamically allocate processors
- To perform lexical analysis. For example, to parse a program into tokens.
- To search for regular expressions. For example, to implement the UNIX grep program.
- To implement some tree operations. For example, to find the depth of every vertex in a tree
- To label components in two dimensional images.

See Guy Blelloch "Prefix Sums and Their Applications"

Summary

- Parallel prefix sums and scans have many applications
  - A good algorithm to have in your toolkit!

- Key idea: An algorithm in 2 passes:
  - Pass 1: build a "reduce tree" from the bottom up
  - Pass 2: compute the prefix top-down, looking at the left-subchild to help you compute the prefix for the right subchild

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

F# Install

Mac OS

- Follow Option 1 or Option 2:
  - https://fsharp.org/use/mac/
  - Prof Walker used Option 2: Installed Visual Studio for Mac:
    - https://visualstudio.microsoft.com/vs/mac/

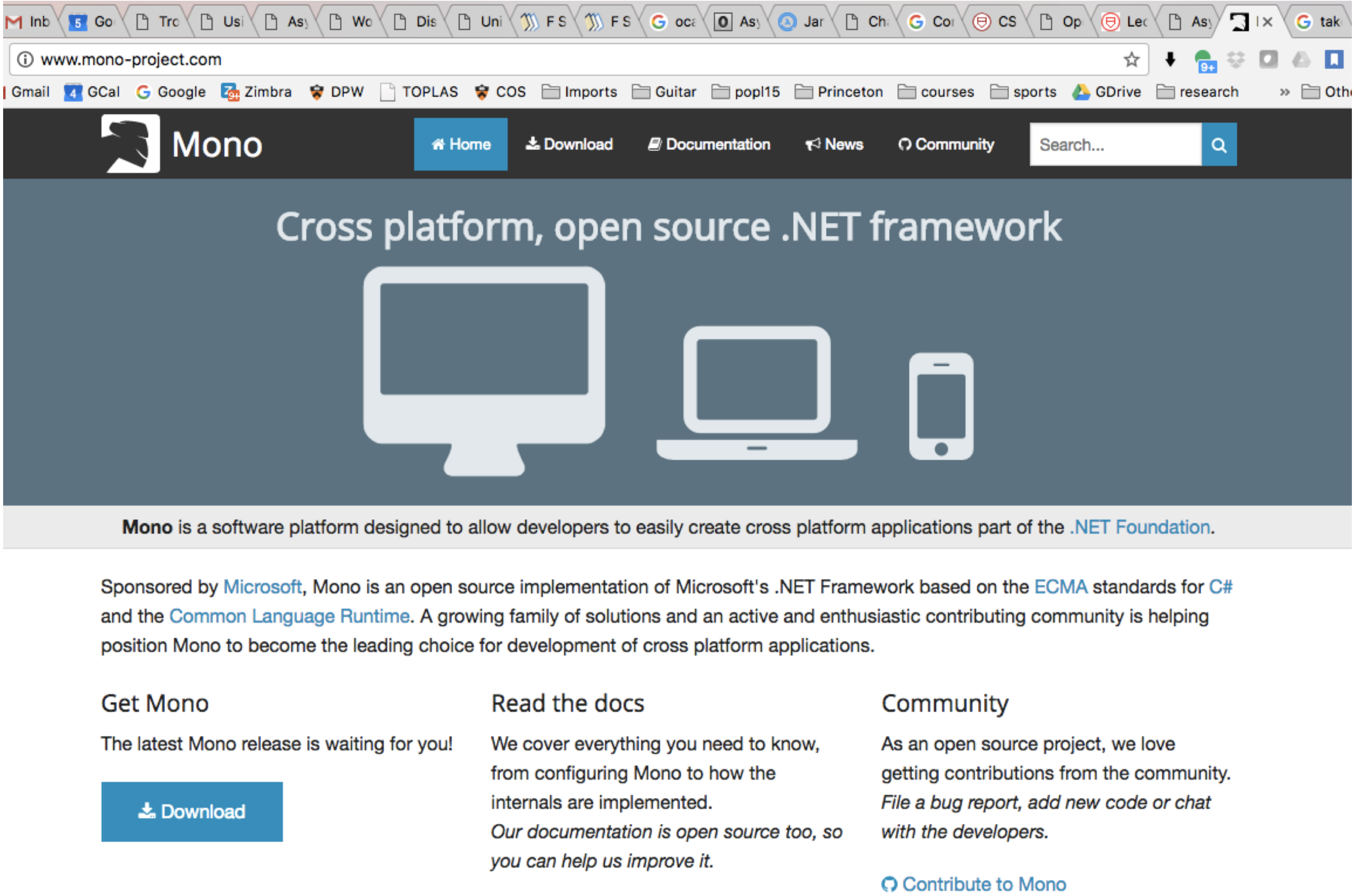
Linux

- Follow the instructions for your distribution:
  - https://fsharp.org/use/linux/

Windows

- Follow Option 1 or Option 2:
  - https://fsharp.org/use/windows/

Step 1 (Mac/Linux): Get Mono



Cross platform, open source .NET framework

Mono is a software platform designed to allow developers to easily create cross platform applications part of the .NET Foundation.

Sponsored by Microsoft, Mono is an open source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime. A growing family of solutions and an active and enthusiastic contributing community is helping position Mono to become the leading choice for development of cross platform applications.

Get Mono

The latest Mono release is waiting for you!

Read the docs

We cover everything you need to know, from configuring Mono to how the internals are implemented. *Our documentation is open source too, so you can help us improve it.*

Community

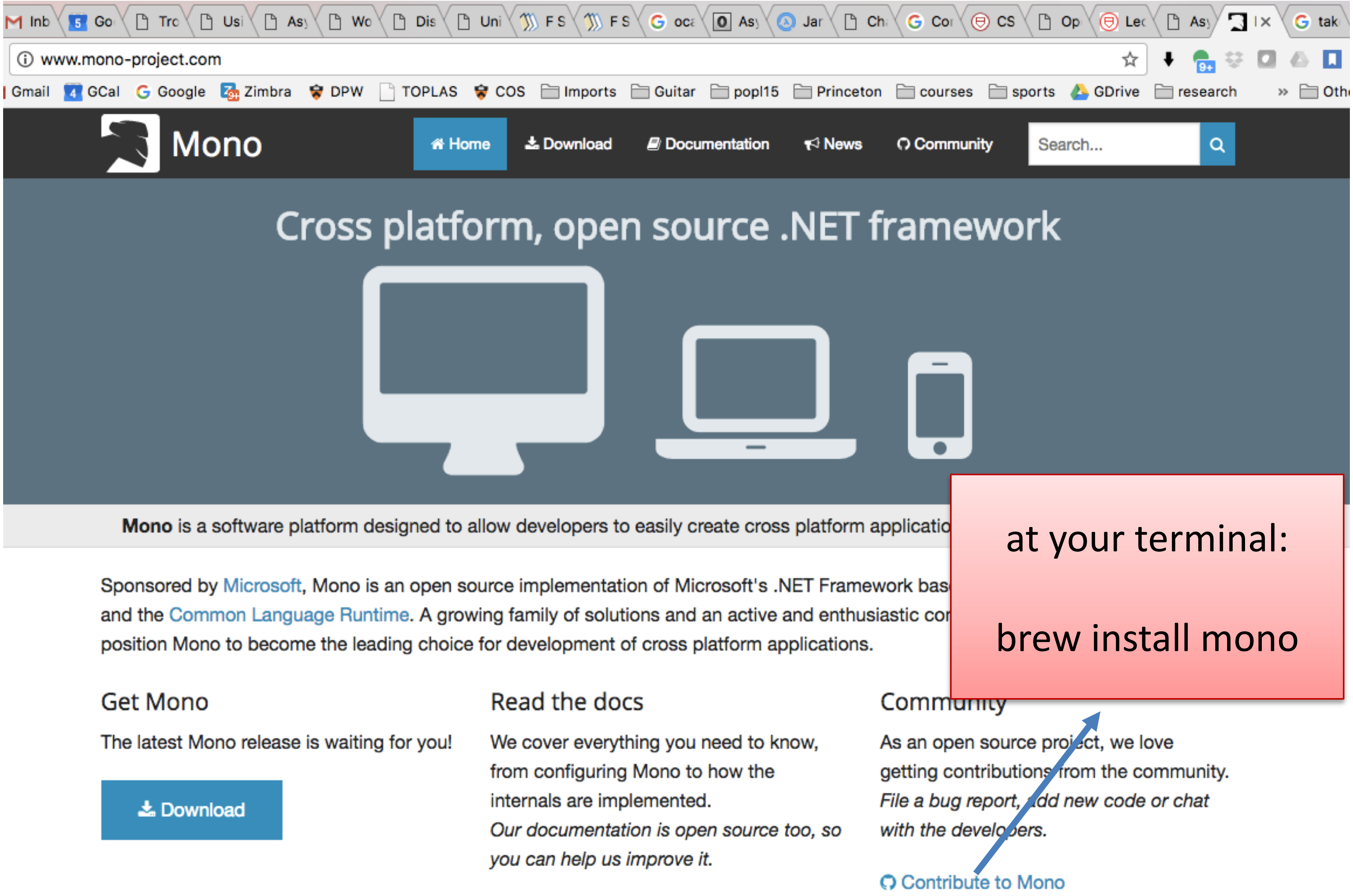
As an open source project, we love getting contributions from the community. *File a bug report, add new code or chat with the developers.*

Contribute to Mono

www.mono-project.com

also via homebrew

Step 1 (Mac/Linux): Get Mono



Cross platform, open source .NET framework

Mono is a software platform designed to allow developers to easily create cross platform applicatio

Sponsored by Microsoft, Mono is an open source implementation of Microsoft's .NET Framework bas and the Common Language Runtime. A growing family of solutions and an active and enthusiastic co position Mono to become the leading choice for development of cross platform applications.

Get Mono

The latest Mono release is waiting for you!

Read the docs

We cover everything you need to know, from configuring Mono to how the internals are implemented.

Our documentation is open source too, so you can help us improve it.

Community

As an open source project, we love getting contributions from the community. *File a bug report, add new code or chat with the developers.*

Contribute to Mono

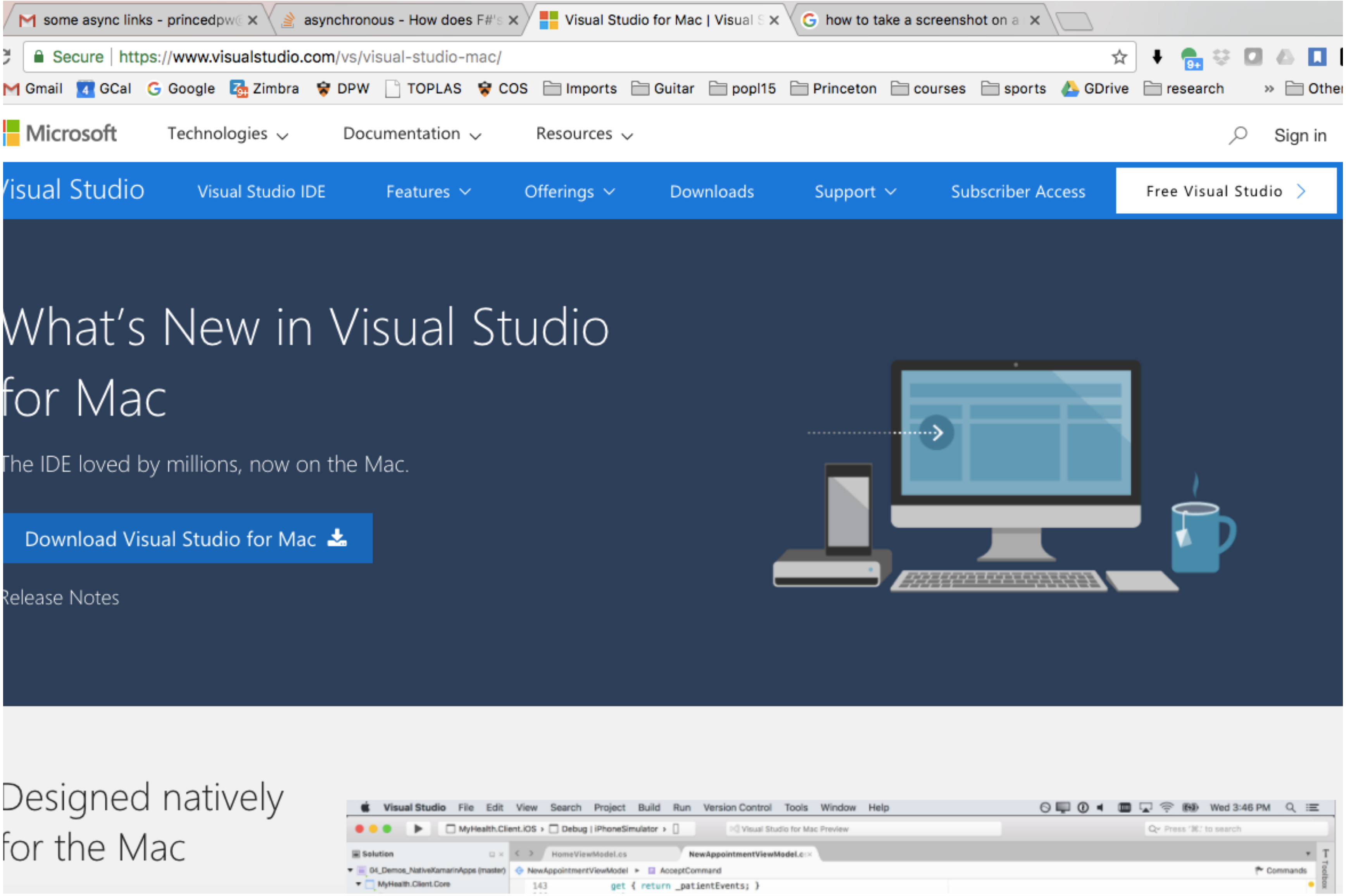
at your terminal:

```
brew install mono
```

www.mono-project.com

also via homebrew

Step 2 (Mac/Linux): Download Visual Studio



What's New in Visual Studio for Mac

The IDE loved by millions, now on the Mac.

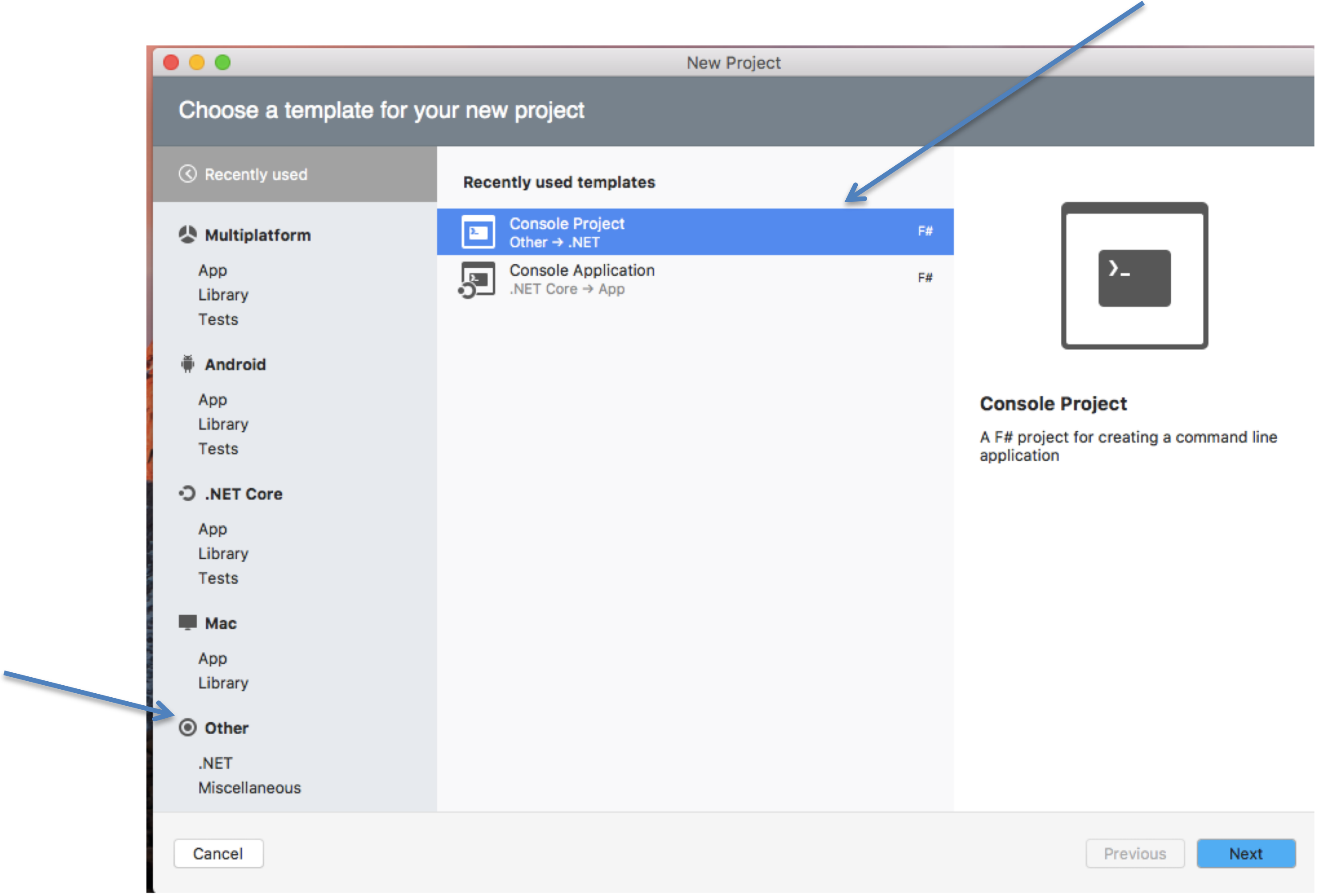
Designed natively for the Mac

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:



New Project

Choose a template for your new project

Recently used templates

Console Project F#
Other → .NET

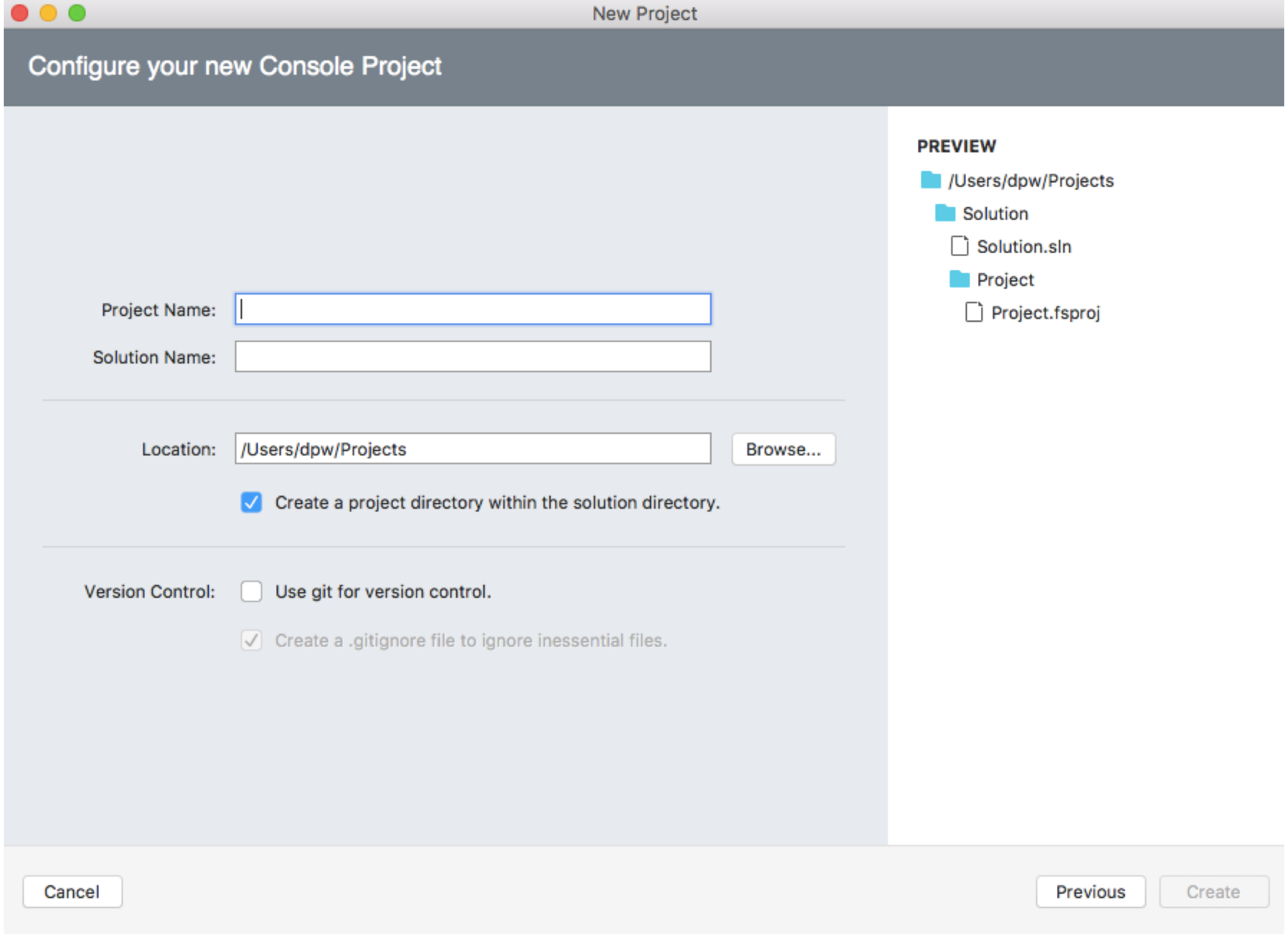
Console Application F#
.NET Core → App

Console Project

A F# project for creating a command line application

Creating a New Solution in VS

3. Choose a name:



Configure your new Console Project

Project Name:

Solution Name:

Location: /Users/dpw/Projects

☑ Create a project directory within the solution directory.

Version Control: ☐ Use git for version control.

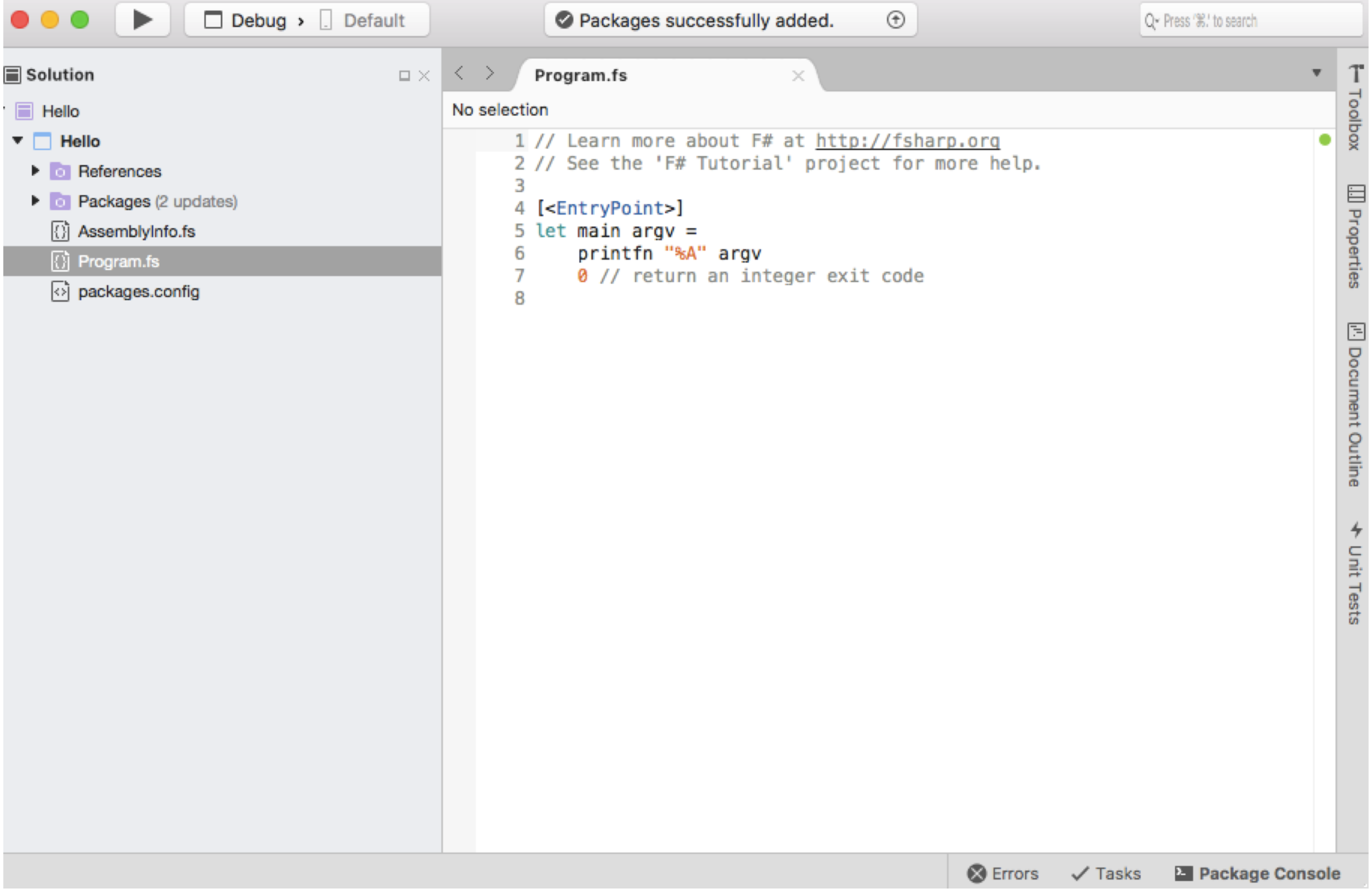
☑ Create a .gitignore file to ignore inessential files.

PREVIEW

/Users/dpw/Projects
  Solution
    Solution.sln
    Project
      Project.fsproj

Creating a New Solution in VS

4. Your first file and boiler plate is generated:



Packages successfully added.

```
// Learn more about F# at http://fsharp.org
// See the 'F# Tutorial' project for more help.

[<EntryPoint>]
let main argv =
    printfn "%A" argv
    0 // return an integer exit code

```

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
```

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 asynchronous computations that will handle many of these issues and more ...

Plus an elegant syntax to boot!

```
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”
```

F# Async

Values with type Async<T> are suspended computations

- that may be run in the background, like futures
- or be 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
   }
```

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

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

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:

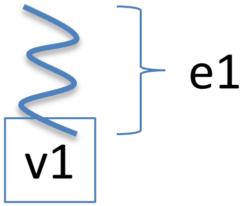
```
let asyncAdd x y =
  async {
   return x + y
  }
```

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>

Visualizing Asyncs

visualization



creation

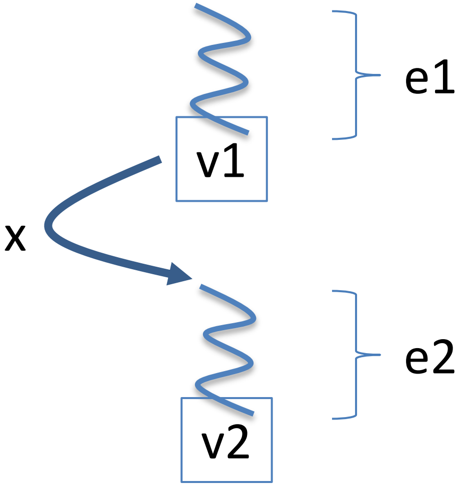
async { return e1 }

type

Async<T>

Visualizing Asyncs

visualization



composition

```
async {
  let! x = return e1
  return e2
}
```

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>

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>

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
```

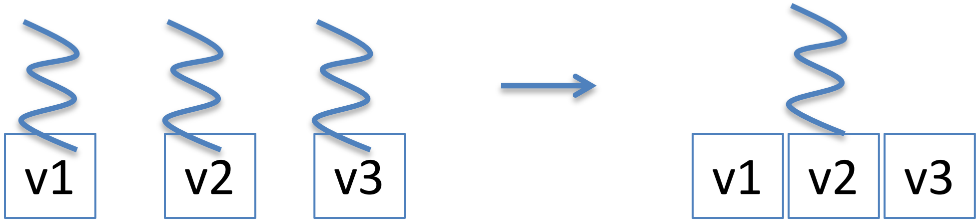
let! waits for the result of asyncAdd before continuing; bind an integer to z

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

Parallelism

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

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

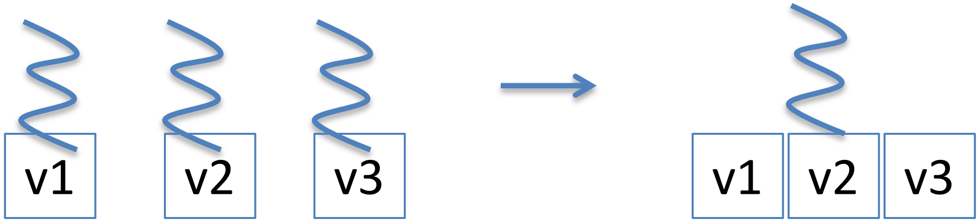


Parallelism

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

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

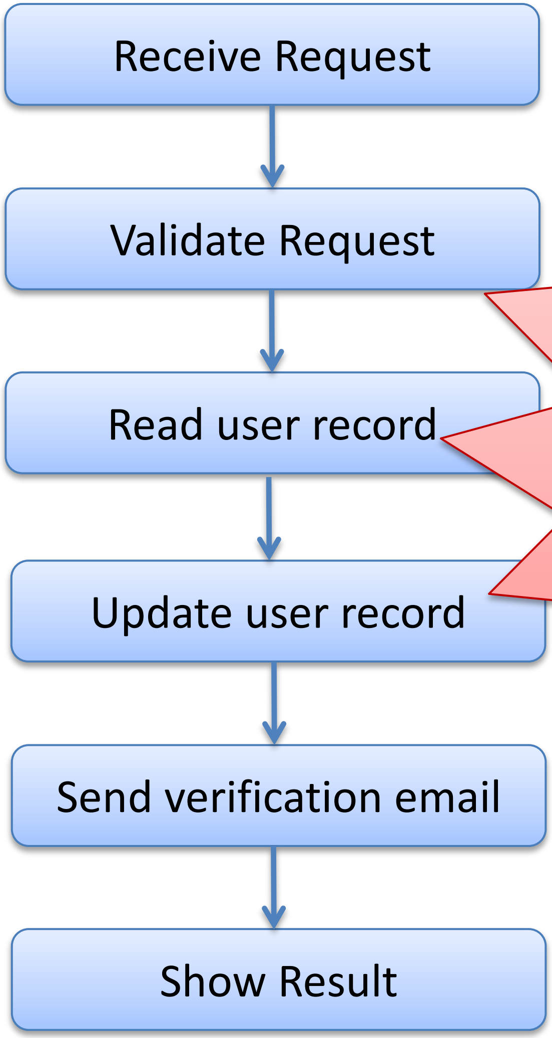
```
expression is 42
expression is 43
expression is 85
```

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

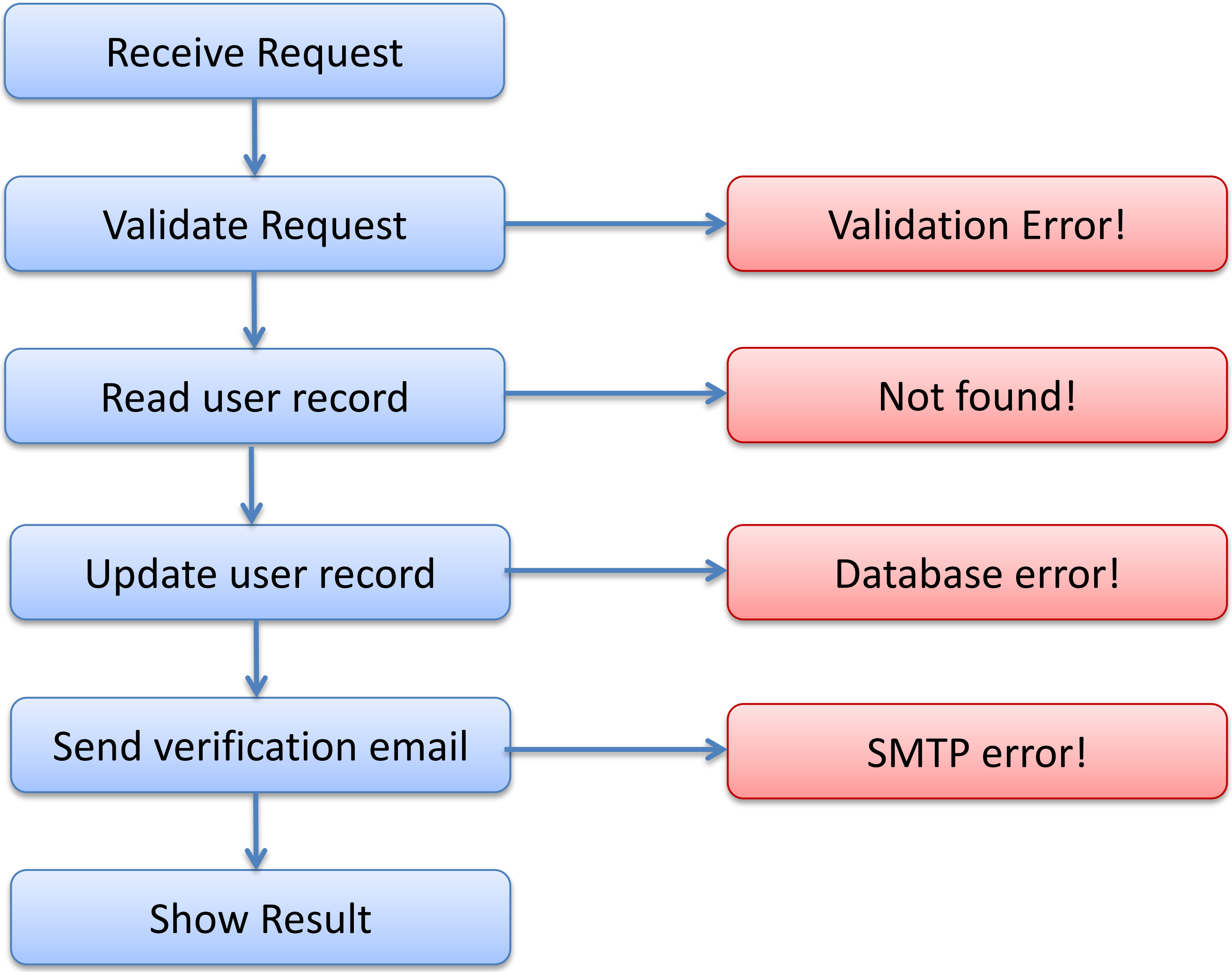
Update user record

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

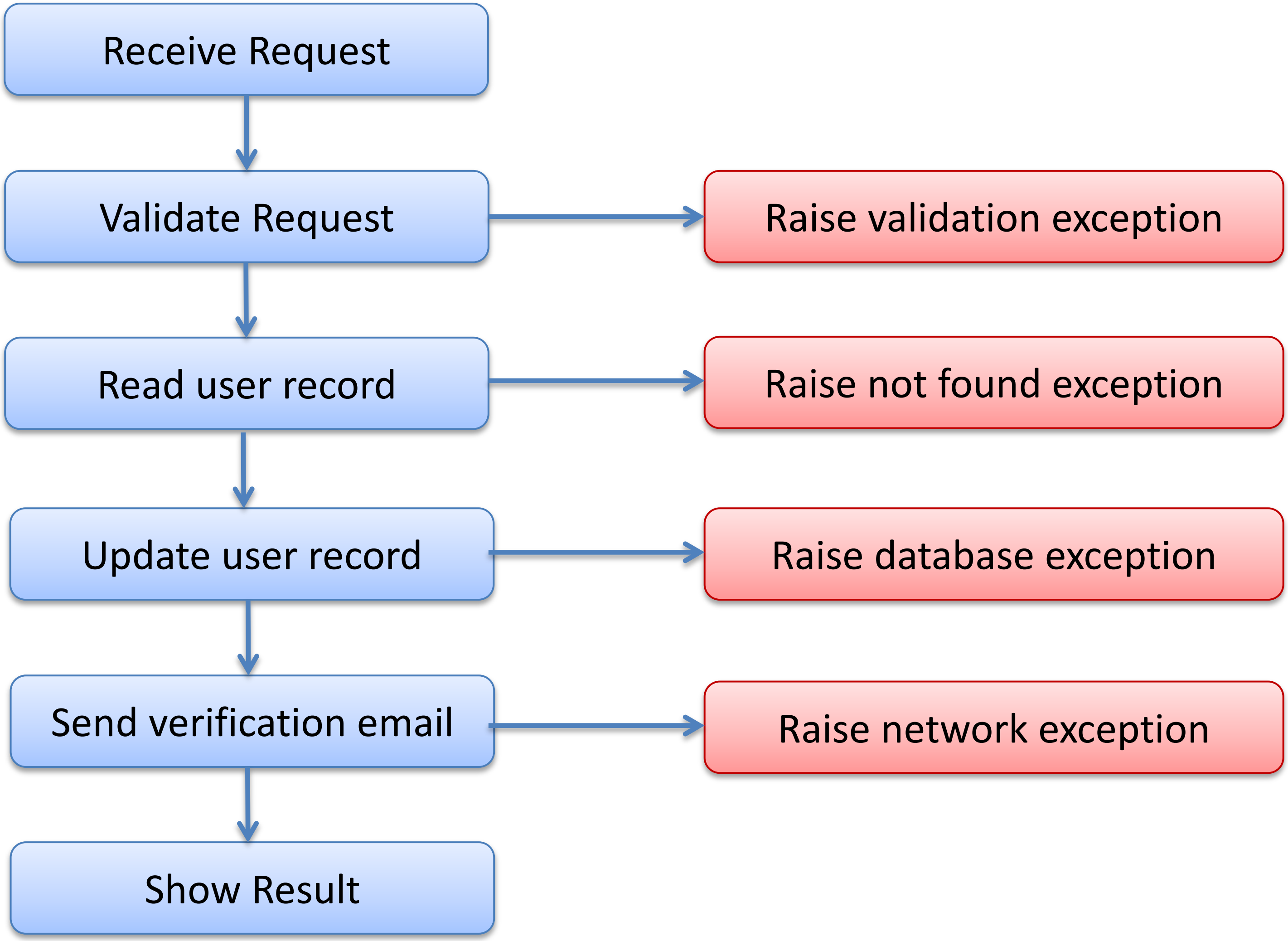
Database error!

Send verification email

SMTP error!

Show Result

One solution



Receive Request

Validate Request

Raise validation exception

Read user record

Raise not found exception

Update user record

Raise database exception

Send verification email

Raise network exception

Show Result

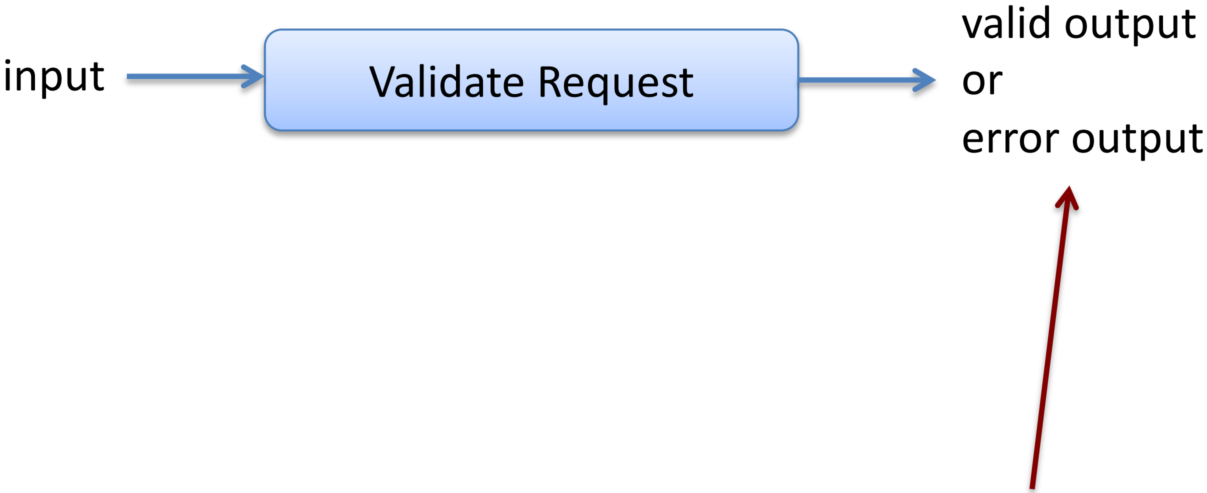
The trouble with exceptions

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



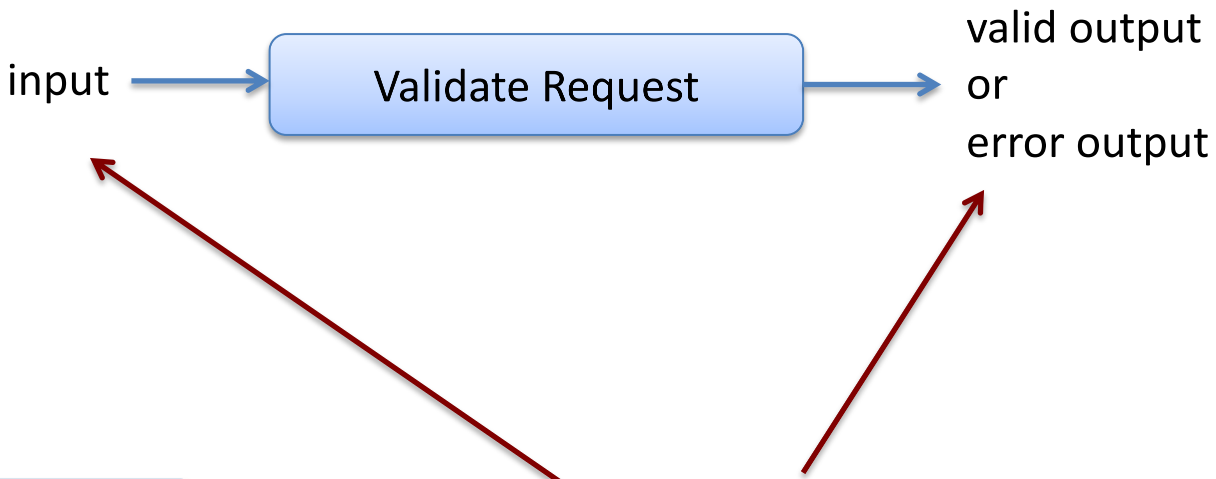
input

Validate Request

valid output
or
error output

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).

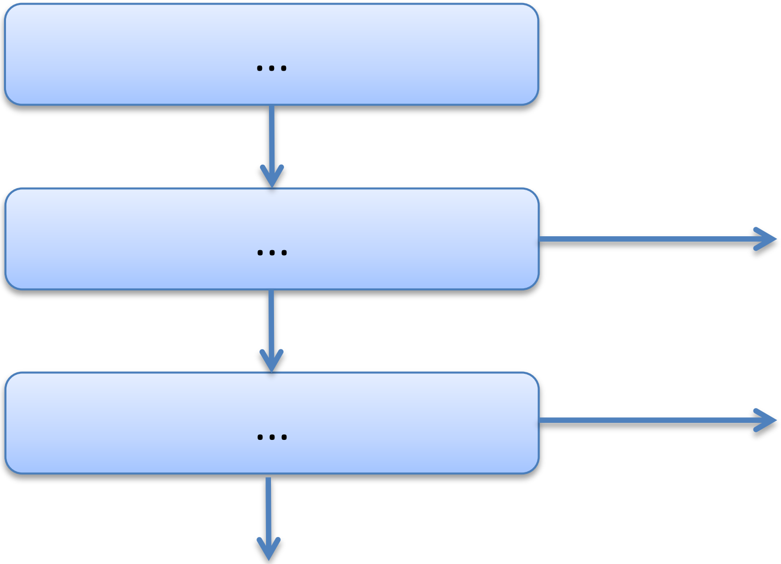
Functional Error Processing



input

Validate Request

valid output
or
error output



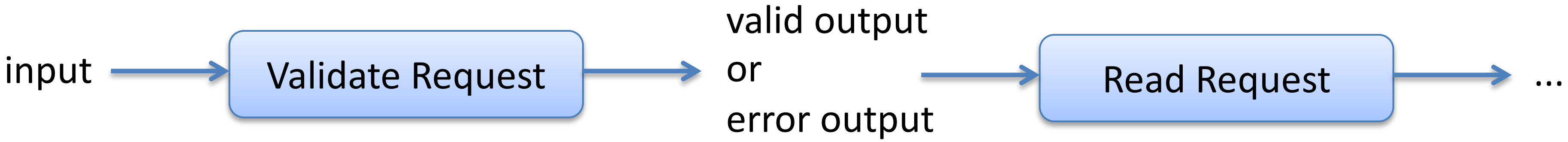
Notice input and output aren’t the same type. On the surface, this makes it look awkward to compose a series of such steps, but:

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

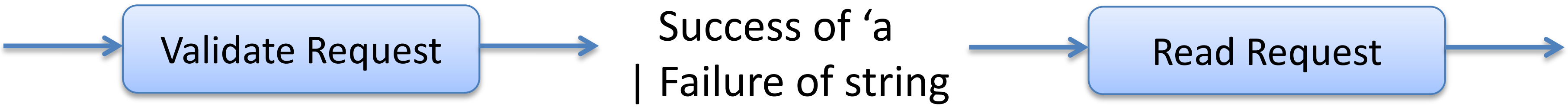


Generic Error Processing

A generic result type:

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

A processing pipeline:



Validate Request

Success of ‘a
| Failure of string

Read Request

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/

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