F# Code I Love

Don Syme
F# Community Contributor
Researcher @ Microsoft

A stroll through some of the F# code I love…

…and some that I love a little less :)

…and how this relates to the language features

Aside

The Early History of F# - HOPL IV submission (2020)

Draft now available!

fsharp.org/history

WARNING: Opinion!

F# is the open-source, cross-platform functional language for .NET

Get Started with F#

Supported on Windows, Linux, and macOS

www.microsoft.com/net/

Fable

F# |> BABEL

The compiler that emits JavaScript you can be proud of!

Fable is an F# to JavaScript compiler powered by Babel, designed to produce readable and standard code. Try it right now in your browser!

Functional-first programming

Fable brings all the power of F# to the JavaScript ecosystem. Enjoy advanced language features like static typing with type inference, exhaustive pattern matching, immutability by default,

Batteries charged

Fable supports most of the F# core library and some of most commonly used .NET APIs: collections, dates, regular expressions, string formatting, observables, async and even reflection! All of this without adding extra

TL;DR – Where we are now

F# is open, cross-platform, neutral, independent

All F# language and tooling is accepting contributions

The F# community is self-empowered

fsharp.org

Xamarin provide F# tools Android and iOS

VSCode, Visual Studio for Windows, Mac and Azure

VSCode and .NET Core for Linux/OSX

The F# Compiler Service powers many other tools

Fable for F#-to-Javascript

SAFE for F# fullstack

F# 4.6 now complete!

The F# Language Design Process

github.com/fsharp/fslang-design
github.com/fsharp/fslang-suggestions

F# 4.0

- ✔ True shift to cross-platform open engineering
- ✔ Long laundry list of language items
- ✔ Normalized core library
- ✔ Type providers more powerful

- ✔ Better debugging, tooling, performance
- ✔ ~20% compiler perf improvement

https://github.com/fsharp/fslang-design/tree/master/FSharp-4.0

F# 4.1

- ✔ Optional large scope cycles more on this later)
- ✔ Result<T,Error> in standard library
- ✔ Unboxed (struct) tuples
- ✔ Unboxed (struct) records
- ✔ Unboxed (struct) unions

- ✔ More bits and pieces

https://github.com/fsharp/fslang-design/tree/master/FSharp-4.1

F# 4.5

- ✔ Span<T> high perf type-safe non-allocating code
- ✔ Improved async debugging

- ✔ More bits and pieces

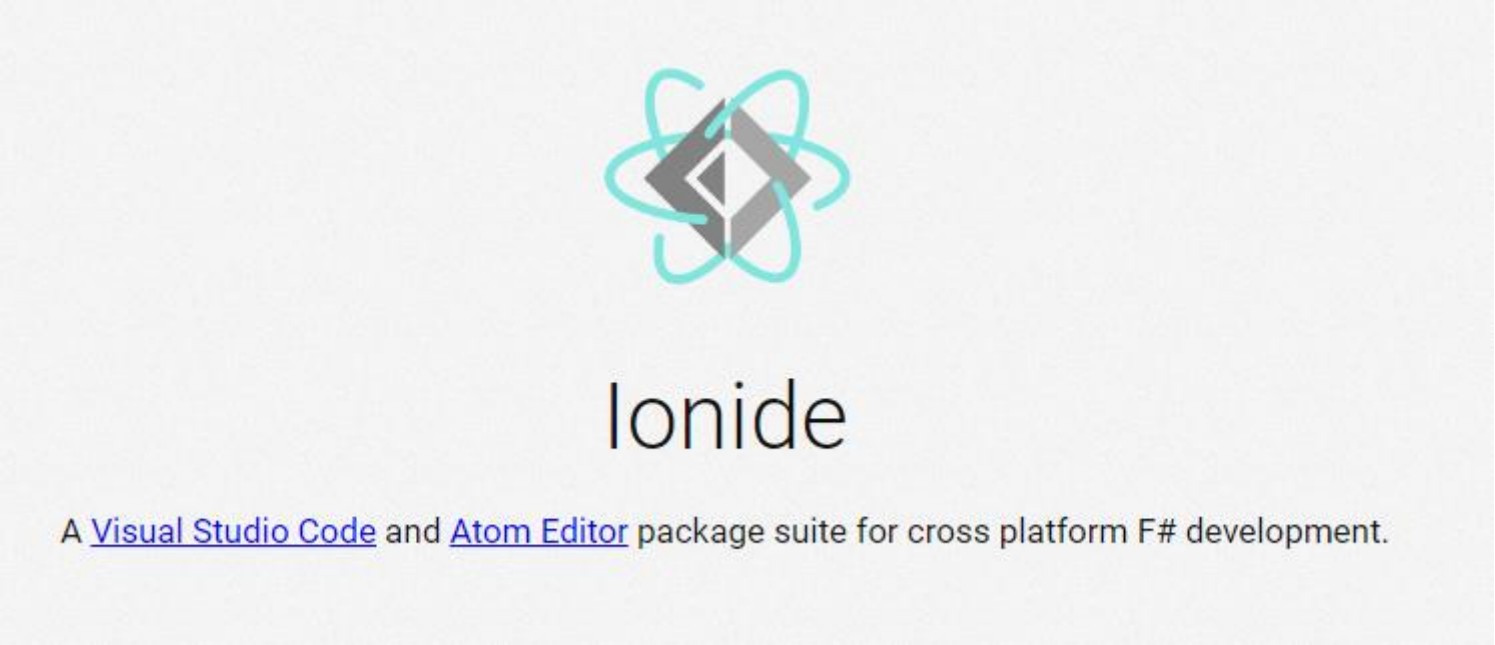
https://github.com/fsharp/fslang-design/tree/master/FSharp-4.5

F# 4.6

- ✔ Anonymous records
- ✔ More bits and pieces

https://github.com/fsharp/fslang-design/tree/master/FSharp-4.5

VSCode + Ionide = Cross-Platform F# Editing Love



F# and .NET Core (Linux, OSX, Windows)

```
dotnet new -lang F#

dotnet build
```

docs.microsoft.com/dotnet/core/

A functional-first approach makes a huge difference in practice

fsharp.org/testimonials

An analysis (Simon Cousins, Energy Sector)

350,000

lines of C# OO
by offshore team

The C# project took five years and peaked at ~8 devs. It never fully implemented all of the contracts.

The F# project took less than a year and peaked at three devs (only one had prior experience with F#). All of the contracts were fully implemented.

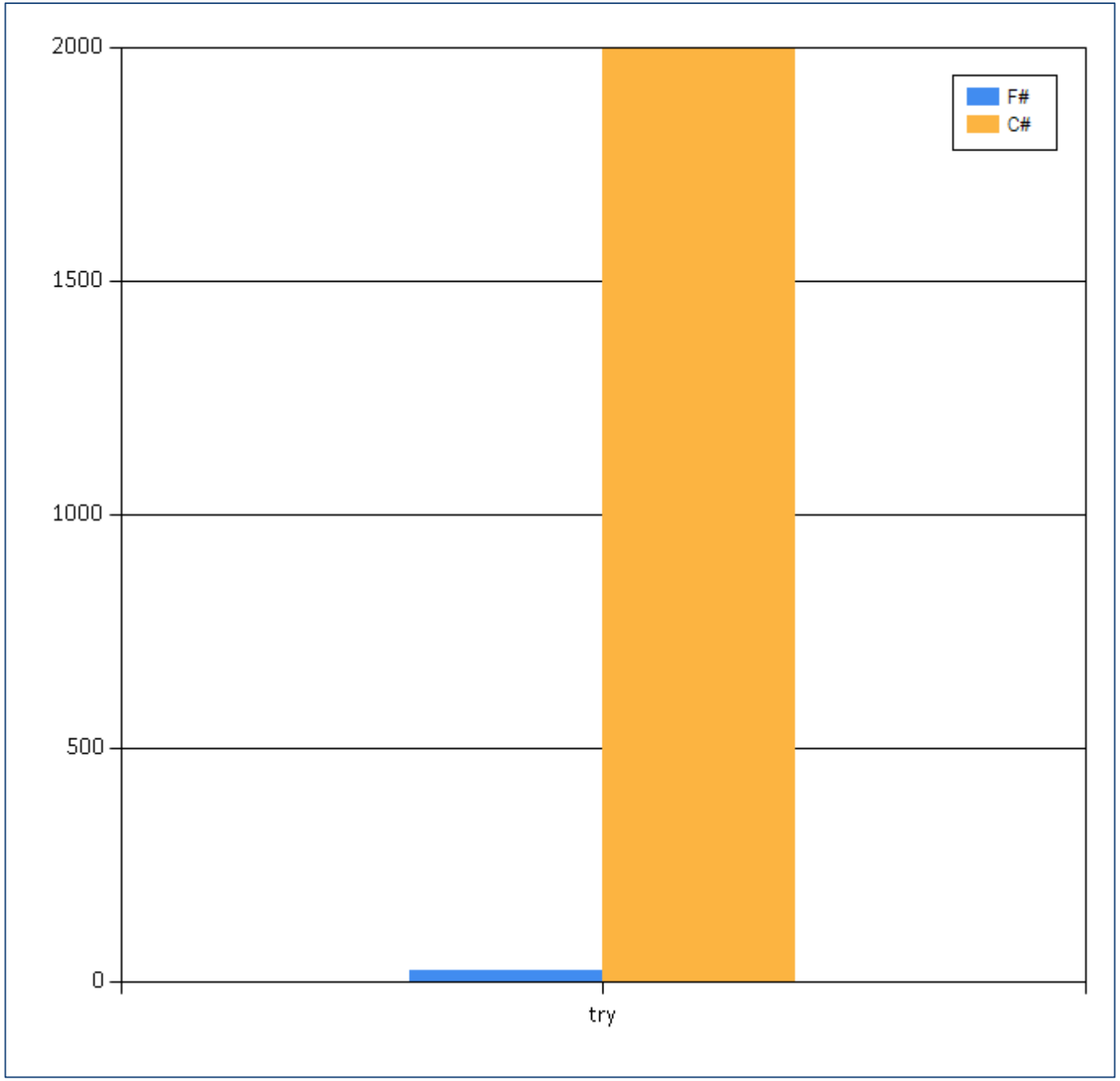
30,000

lines of robust F#, with
parallel +more features

An application to evaluate the revenue due from Balancing Services contracts in the UK energy industry

http://simontcousins.azurewebsites.net/does-the-language-you-use-make-a-difference-revisited/

Implementation	C#	F#
Braces	56,929	643
Blanks	29,080	3,630
Null Checks	3,011	15
Comments	53,270	487
Useful Code	163,276	16,667
App Code	305,566	21,442
Test Code	42,864	9,359
Total Code	348,430	30,801


2000
1500
1000
500
0
try
F#
C#

Simon Cousins, Energy Sector

Zero

bugs in deployed system

"F# is the safe choice for this project,
any other choice is too risky"

An application to evaluate the revenue due from Balancing Services contracts in the UK energy industry

http://simontcousins.azurewebsites.net/does-the-language-you-use-make-a-difference-revisited/

A $3B Unicorn, Built on F#

https://techcrunch.com/2016/08/07/walmart-buys-jet-com-for-3-billion/

News Startups Mobile Gadgets Trending Tesla Google Facebook

Advertising Tech | jet.com | Walmart Labs | Walmart

Walmart is buying Jet.com for $3 billion

Posted Aug 7, 2016 by Jonathan Shieber (@jshieber), John Mannes (@JohnMannes)

Next Story

Walmart Stores is buying Jet.com in a deal worth $3 billion dollars according to a source with direct knowledge of the deal, confirming reports that have been pouring in about the bid for Jet.com all week.

According to our source the signatures for the deal were dry on Friday and will be announced as early as Monday morning — echoing what was reported in both Bloomberg and Recode.

www.managedsolution.com/jet-built-its-entire-e-commerce-platform-including-development-and-del

MANAGED SOLUTION

Home About + Products & Services + Client Support + Contact Us Try Before You Buy!

Jet built its entire e-commerce platform, including development and delivery infrastructure, on Microsoft #Azure.

Jet.com – E-commerce challenger eyes the top spot, runs on the Microsoft cloud

Marc Lore is perhaps best known as the creator of the popular e-commerce site Diapers.com, which was eventually sold to Amazon. Now, the entrepreneur and his team are ready to compete head-on with the e-retailing giant through an innovative online marketplace called Jet.com. To get up and running quickly, Jet built its entire e-commerce platform, including development and delivery infrastructure, on Microsoft Azure, using both .NET and open-source technologies.

Business Challenge

In 2010, Marc Lore sold his company Quidsi (which ran e-retailing sites like Diapers.com and Soap.com) to Amazon for $550 million. Four years later, Marc is competing against Amazon directly—with the creation of a new online marketplace called Jet.com.

There are many reasons to think that Lore might just pull it off. For one, he plans to eliminate any margins from product sales. The company's only source of revenue will come from membership dues, eliminating the kind of mark-ups that Amazon charges and passing the savings on to the customer. In addition, an innovative pricing engine will work to reduce or eliminate costs in the e-commerce value chain, especially fulfillment costs and marketplace commissions.

"Our pricing engine will continually work out the most cost-effective way to fulfill an order from merchant locations closest to the consumer,' explains Lore, Co-Founder and CEO of Jet. "The engine will also figure out which merchants can fulfill most cheaply by putting multiple

OK, I'm the language designer. I could tell you about the features.

But what code do I <u>like</u> and <u>not like</u>?

Reminder:

The F# Advent Calendar

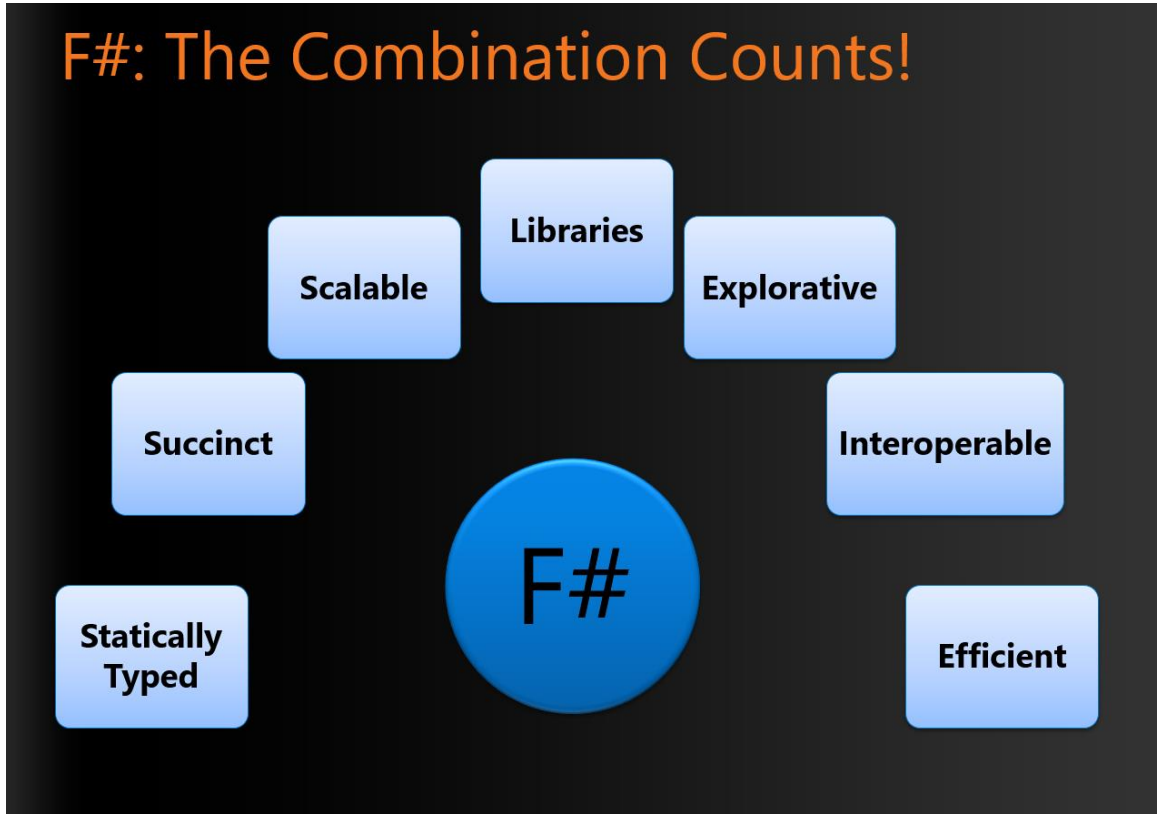
(started by F# users in Japan!)

English 2017, 2016, 2015

Japanese 2016, 2015, 2014, 2013, 2012, 2011, 2010

11	12	13	14	15	16	17
bleis F#に型クラスを入れる実装の話	pocketbe... 2016年時点でF# 用のライブラリを.NET Core対応させ	pocketbe... コンピュテーション式の展開結果を可視化するツール	callmeko... Fsi on Suave	callmeko... F# and Neovim	gab_km 皆さんの期待に応えぬよう頑張ります！ Overwrite	moonmile Android Things 上で Xamarin.Android を動かして
18	**19**	**20**	**21**	**22**	**23**	**24**
pocketbe... Persimmon の.NET Core対応	gorn708 分析者目線でF#なAzure Notebookにトライしてみる	wgag F# Data 型プロバイダの内部について	yanosen_jp UnityでF#を使う（アップデート）	zecl TypeProvider に関するちょっとした小ネタ集	kekyo About Expandable F# Compiler project	matarillo 情報隠蔽とモジュールとシグネチャファイル ～オフラ

Foundations of the F# Design (~2007)



From that, it's fair to say that I love these :)

Code that is **succinct**
Code that is **expressive**
Code that **interoperates**
Code that is **performant**
Code that is **accurate**
Code that is **well-tooled**

Code I love!

```
printfn "hello world"
```

- ✓ Code that is **succinct**
- ✓ Code that is **expressive**
- ✓ Code that **interoperates**
- ✓ Code that is **performant**
- ✓ Code that has **low bug rates**
- ✓ Code that is **well-tooled**

Code I love!
- pipelines

```
x |> f1

x |> f1 |> f2 |> f3 |> …..
```

Code I love!
- pipelines

```
let symbolUses =
    symbolUses
    |> Array.filter (fun symbolUse -> …)
    |> Array.Parallel.map (fun symbolUse -> …)
    |> Array.filter (fun … -> …)
    |> Array.groupBy (fun … -> …)
    |> Array.map (fun … ->….)
```

Code I love!
- pipelines
- domain modelling

Code I love!
- pipelines
- domain modelling

```
/// Represents a parsed expression
type Expr =
    | True
    | And of Expr * Expr
    | Nand of Expr * Expr
    | Or of Expr * Expr
    | Xor of Expr * Expr
    | Not of Expr

+ recursion, evaluation, normalization, analysis,
visualization, …
```

Code I love!
- pipelines
- domain modelling

```
/// Represents information known about a value
type ExprValueInfo =
    | UnknownValue
    | ValValue of ValRef * ExprValueInfo
    | TupleValue of ExprValueInfo[]
    | RecdValue of TyconRef * ExprValueInfo[]
    | UnionCaseValue of UnionCaseRef * ExprValueInfo[]
    | ConstValue of Const * TType
    | CurriedLambdaValue of Unique * Expr * TType
```

Code we love :)
- pipelines
- domain modelling

```
type Status =
    | Online
    | Unresponsive of string
    | Missing of string
    | NotChecked of string
    | Ignored
```

https://lukemerrett.com/fsharp-domain-modelling/

F# has plenty of strengths, many outlined on this outstanding website: F# for Fun and Profit, however I'm increasingly finding the most useful elements are discriminated unions, record types and pattern matching. These 3 combined allow for rapid domain modelling that helps to abstract away complexity and informs terse business logic.

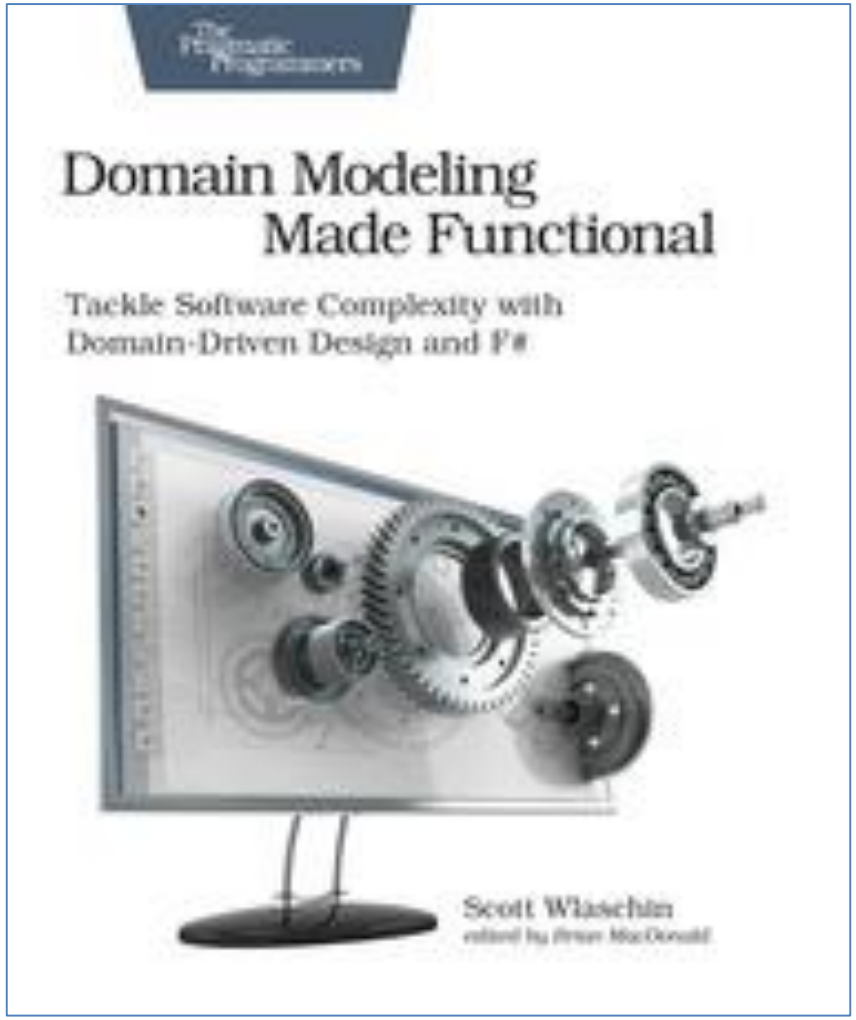
Code we love :)
- pipelines
- domain modelling

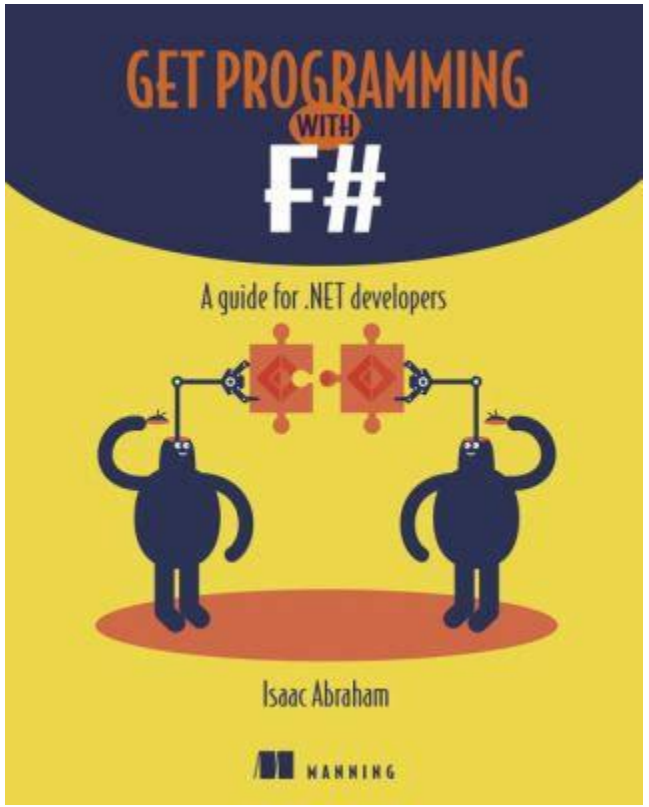
https://medium.com/@odytrice
Ody Mbegbu


```
type Value =
    | Integer of int64
    | String of string
    | Date of DateTime
    | Data of string
    | Bool of bool
    | Dict of list<string * Value>
    | Array of list<Value>
```

It might seem obvious but I'll say it anyway. Your choice of data structures and how you design your domain is crucial when writing code in F# (or in any other language). Screw it up, and you will be walking around in circles. Nail it, and your implementation will be concise, straightforward and probably even trivial.

Code we love :)
- pipelines
- domain modelling
- domain semantics





Code I love :)
- data scripting

```
// Get the nuget stats schema
type NugetStats = HtmlProvider<"https://www.nuget.org/packages/FSharp.Data">

// Load the live package stats for FSharp.Data
let rawStats = NugetStats().Tables.``Version History``

// Group by minor version and calculate download count
let stats =
    rawStats.Rows
    |> Seq.groupBy (fun r -> getMinorVersion r.MinorVersion)
    |> Seq.sortBy fst
    |> Seq.map (fun (k, xs) -> k, xs |> Seq.sumBy (fun x -> x.Downloads))
```

Code I love :)
- model-view-update mobile UIs
- view functions!

A model-view-update mobile app

```
/// The view function giving updated content for the page
let view (model: Model) dispatch =
  if model.Pressed then
    Xaml.Label(text="I was pressed!")
  else
    Xaml.Button(text="Press Me!", command=(fun () -> dispatch Pressed))
```

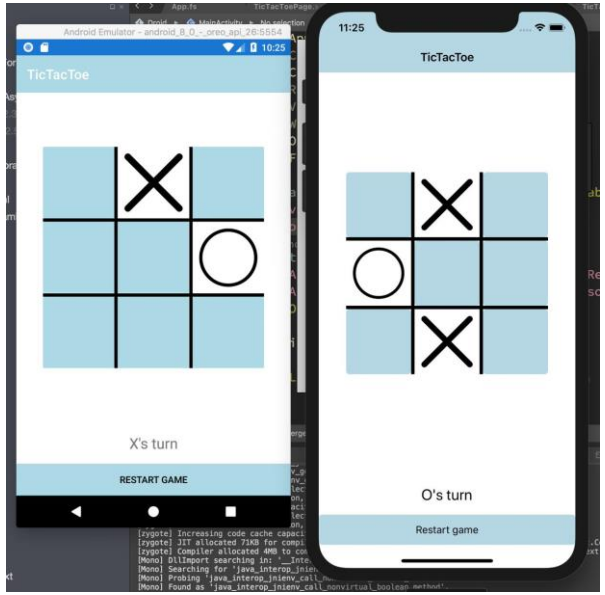
Code I love :)
- model-view-update web UIs
- view functions!

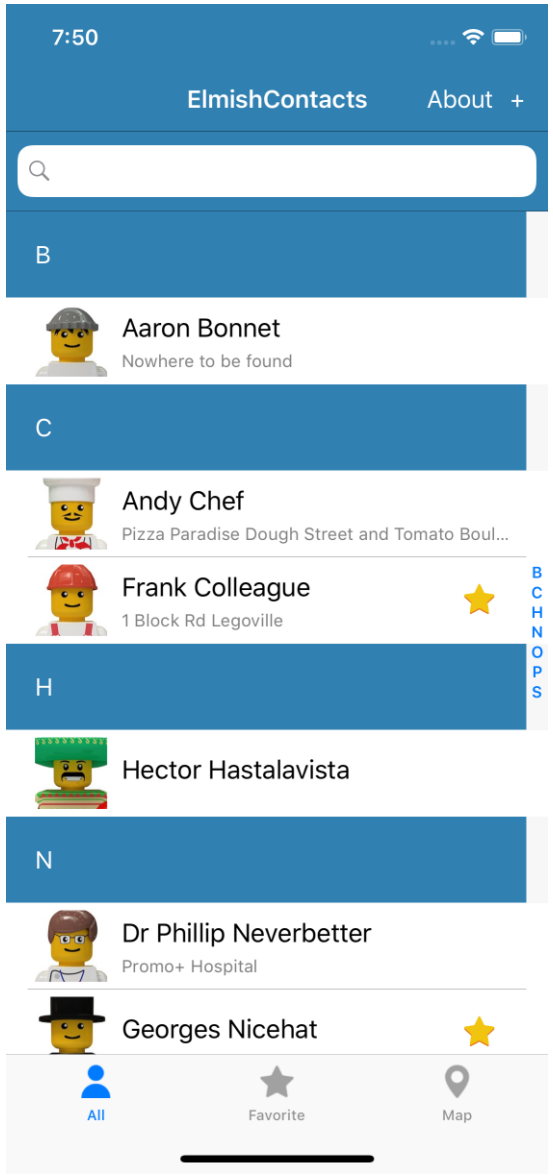
A model-view-update web view

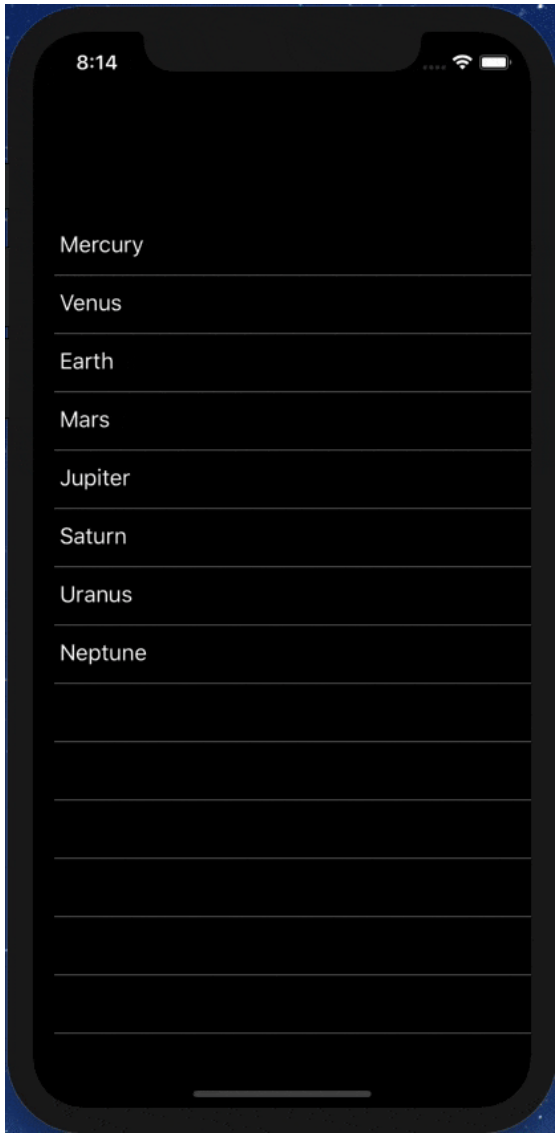
```
/// The view function giving upda
let view model dispatch =
  match model.Text with
  | [| |] ->
    div [] [ div [] [str "Loading..."] ]
  | _ ->
    div [ ClassName "container" ] [
      button [ OnClick (fun _ -> dispatch Faster) ] [ str "Faster" ]
      div [ ClassName "theText" ] [ str model.Text.[model.Index] ]
      button [ OnClick (fun _ -> dispatch Slower) ] [ str "Slower" ]
      div [] [ str (sprintf "Ticks Per Update: %d" model.TicksPerUpdate) ]
    ]
```

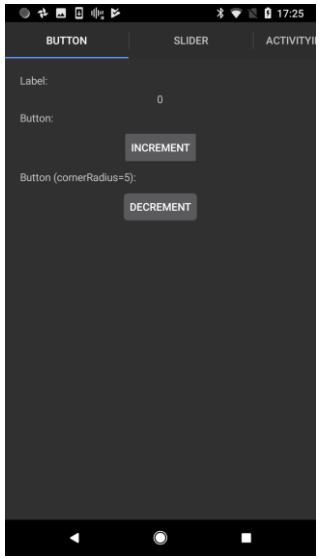
Code I love :)
- reactive, functional UIs
- view functions!

github.com/fsprojects/Fabulous









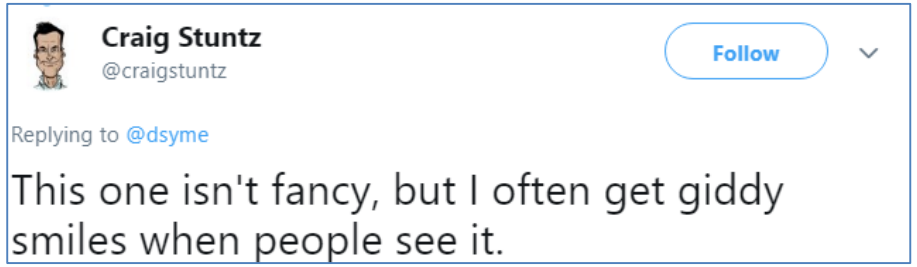
GET IT ON Google Play

Download on the App Store

Code we love :)
- composition

TinyLanguage / TinyLanguage / Compiler.fs

```
let compile =
    Lexer.lex
    >> Parser.parse
    >> Binder.bind
    >> OptimizeBinding.optimize
    >> IlGenerator.codegen
    >> Railway.map OptimizeIl.optimize
    >> Railway.map Il.toAssemblyBuilder
```



Code we love :)

- super-fast compositional web servers

```
let logout : HttpHandler =
    signOut AuthSchemes.cookie
    >=> redirectTo false Urls.index


let webApp : HttpHandler =
    choose [
        GET >=>
            choose [
                route Urls.index >=> index
                route Urls.login >=> login
                route Urls.user >=> authenticate >=> user
                route Urls.logout >=> logout
                route Urls.googleAuth >=> googleAuth
             ]
        notFound ]
```

But….

…not all Functional Code is Good Code…

curry, uncurry

```
let curry f x y = f (x,y)
let uncurry f (x,y) = f x y
```

Too indecipherable, too often

nooo

```
curry String.Compare s1 s2
```

yes

```
String.Compare (s1, s2)
```

nooo

```
let ZipMap f a b =
    Seq.zip a b
    |> Seq.map (uncurry f)
```

yes

```
let ZipMap f a b =
    Seq.zip a b
    |> Seq.map (fun (x,y) -> f x y)
```

<|

```
let (<|) f x = f x
```

Please, never, ever use the <| operator in beginner code

Please, don't **ever** put |> and <| on the same line :)

nooo

```
let testString = "Happy"

let amendedString =
    testString
    |> replace "H" "Cr"
    |> joinWith <| "birthday"
```

yes

```
let testString = "Happy"

let amendedString =
    testString
    |> replace "H" "Cr"
    |> joinWith "birthday"
```

<||, <|||

nooo

```
let (<||) f x y = f x y
let (<|||) f x y z = f x y z
```

Please, always avoid the <|| and <||| operators. They should be deprecated

Point-free is not a virtue

- "Point free" is code without explicit lambdas or let
- Often heavy use of ">>", ">>=", "curry", "uncurry", partial application
- Using and combining existing functions as values is OK
- Please give explicit arguments to functions defined in modules

```
let add10To = List.map((+) 10)
```

nooo

```
let doubleAndIncr = (*) 2 >> (+) 1
```

Please, avoid needless over-use of point-free code

```
let add10To x = x + 10
let doubleAndIncr x = x * 2 + 1
```

yes

"In rare cases there can even be point-free DSLs that are actually legible in the large. However the utility of adopting this approach always carries a big burden of proof, and should not be motivated merely out of stylistic considerations." *Eirik Tsarpalis*

Fold considered harmful

- "Data.fold" is a blunt instrument
- Replace by something more simpler
- Sometimes harder to understand than an imperative while loop

Please, avoid needless use of fold in code if simpler alternatives are available

List/Seq/Array.sumBy
List/Seq/Array.maxBy
List/Seq/Array.choose
List/Seq/Array.tryPick
List/Seq/Array.mapFold
List/Seq/Array.reduce
....

If you fold or mapFold, use ||>

✗
```
List.fold (fun state x -> new-state) state0 xs
```

v.

✓
```
(state0, xs) ||> List.fold (fun state x -> new-state)
```

Records can be bad

- Each time we design a type, we design the **external** view of the type, and the **internal** representation.
- A record is great when these are **the same**. Beware records when they are not.
- Be prepared to make records **private** or **convert records to classes**. Can be painful.

If your record types are not symmetric or representationally simple, then use a class

✘
```
type Program =
   { initial : int
     labelToNode : Map<int, string> ref
     nodeToLabel : Map<string, int> ref
```

✓
```
type Program (parameters) =
    let mutable initial = -1
    let mutable labelToNode = Map.empty
    let mutable nodeToLabel = Map.empty
    let mutable nodeCount = 1
    let mutable transitionCount = 0
    let mutable transitionsArray = …
    let mutable activeTransitions = Set.empty
    let mutable variables = Set.empty
    ...
```

Objects Good, Objects Bad

F# - Objects + Functional

```
type Vector2D (dx:double, dy:double) =

    let d2 = dx*dx+dy*dy

    member v.DX = dx

    member v.DY = dy

    member v.Length = sqrt d2

    member v.Scale(k) = Vector2D (dx*k, dy*k)
```

Inputs to object construction

Object internals

Exported properties

Exported method

Objects

Constructed Class Types

```
type ObjectType(args) =
    let internalValue = expr
    let internalFunction args = expr
    let mutable internalState = expr
    member x.Prop1 = expr
    member x.Meth2 args = expr
```

Object Interface Types

```
type IObject =
  interface ISimpleObject
  abstract Prop1 : type
  abstract Meth2 : type -> type
```

Object Expressions

```
{ new IObject with
    member x.Prop1 = expr
    member x.Meth1 args = expr }

{ new Object() with
    member x.Prop1 = expr
    interface IObject with
      member x.Meth1 args = expr
    interface IWidget with
      member x.Meth1 args = expr }
```

Code I love:

Functional computation of encapsulated tables and summaries

An early example (FsLexYacc):

```
/// Gives an index to each LR(0) kernel
type KernelTable(kernels) =

    let kernelsAndIdxs = List.indexed kernels

    let kernelIdxs = List.map fst kernelsAndIdxs

    let toIdxMap = Map.ofList [ for i,x in kernelsAndIdxs -> x,i ]

    let ofIdxMap = Array.ofList kernels

    member __.Indexes = kernelIdxs

    member __.Index(kernel) = toIdxMap.[kernel]

    member __.Kernel(i) = ofIdxMap.[i]
```

Deconstructing Object Programming

The 20+ features of OO

1. dot notation (x.Length)
2. instance members
3. type-directed name resolution
4. implicit constructors
5. static members
6. indexer notation arr.[x]
7. named arguments
8. optional arguments
9. interface types
10. mutable data
11. defining events
12. defining operators on types
13. auto properties
14. IDisposable, IEnumerable
15. type extensions
16. structs
17. delegates
18. enums
19. implementation inheritance
20. nulls and Unchecked.defaultof<_>
21. method overloading
22. curried method overloads
23. protected members
24. self types
25. wildcard types
26. aspect oriented programming …
27. …

???

Some make F# a better **API language**

Some make F# a better **implementation** language

Some are part of an **interop** standard

Some are **not needed**

Where do we stand?

1. dot notation (`x.Length`)
2. instance members
3. type-directed name resolution
4. implicit constructors
5. static members
6. indexer notation arr.[x]
7. named arguments
8. optional arguments
9. interface types and impl
10. mutable data
11. operators on types
12. auto properties
13. IDisposable, IEnumerable
14. type extensions
15. events
16. structs
17. delegates
18. enums
19. type casting
20. large type hierarchies
21. implementation inheritance
22. nulls and Unchecked.defaultof<_>
23. method overloading
24. curried method overloads
25. protected members
26. self types
27. wildcard types
28. aspect oriented programming …
29. …

Embrace

Use where necessary, use tastefully, use respectfully, use sparingly

Down the object rabbit hole

Not supported

The 20+ features of OO

Love

1. dot notation (`x.Length`)
2. instance members
3. type-directed name resolution
4. implicit constructors
5. static members
6. indexer notation arr.[x]
7. named arguments
8. optional arguments
9. interface types and implementations
10. mutable data
11. operators on types
12. auto properties
13. IDisposable, IEnumerable
14. type extensions
15. events
16. structs
17. delegates
18. enums
19. type casting

Mostly Avoid

20. large type hierarchies
21. implementation inheritance
22. nulls and Unchecked.defaultof<_>
23. method overloading

Tolerate

24. curried method overloads
25. protected members
26. self types
27. wildcard types
28. aspect oriented programming …
29. …

Forget

Object Programming
v.
Object-Oriented Programming

Object Programming focuses on …

succinct coding, notational convenience

API ergonomics

good naming

practical encapsulation

sensible, small, composable abstractions

expression-oriented

making simple things out of (potentially complex) foundations

In the extreme Object-Oriented Programming can be…

objects as a single paradigm

hierarchical classification (Animal, Cat, Dog, AbstractJellyBeanFactoryDelegator)

large abstractions with many holes and failure points

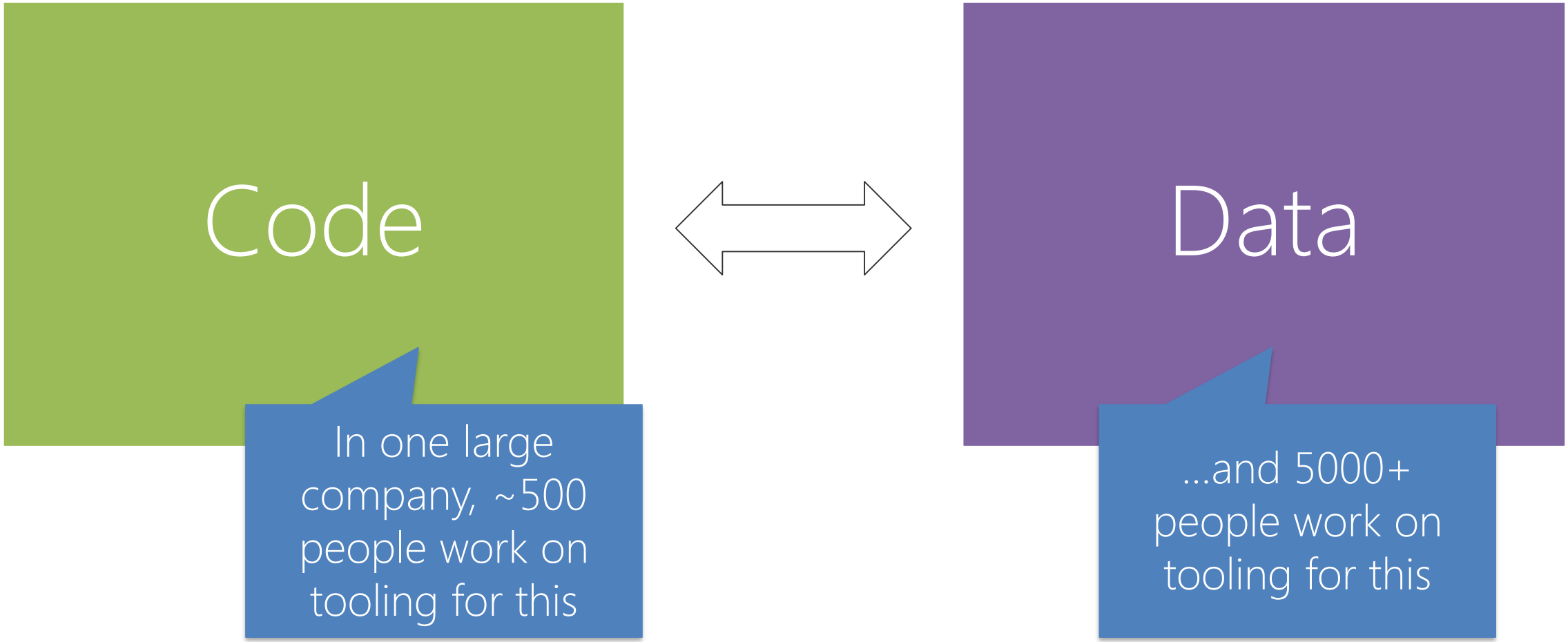
declarations not expressions

composition through… more hierarchies

The F# approach is to **embrace object programming**, make it fit with the expression-oriented typed functional paradigm

but not embrace full "object-**orientation**" (unless you happen to be in a project using that technique)

Code we love :)
- type providers


Code
In one large company, ~500 people work on tooling for this
Data
...and 5000+ people work on tooling for this

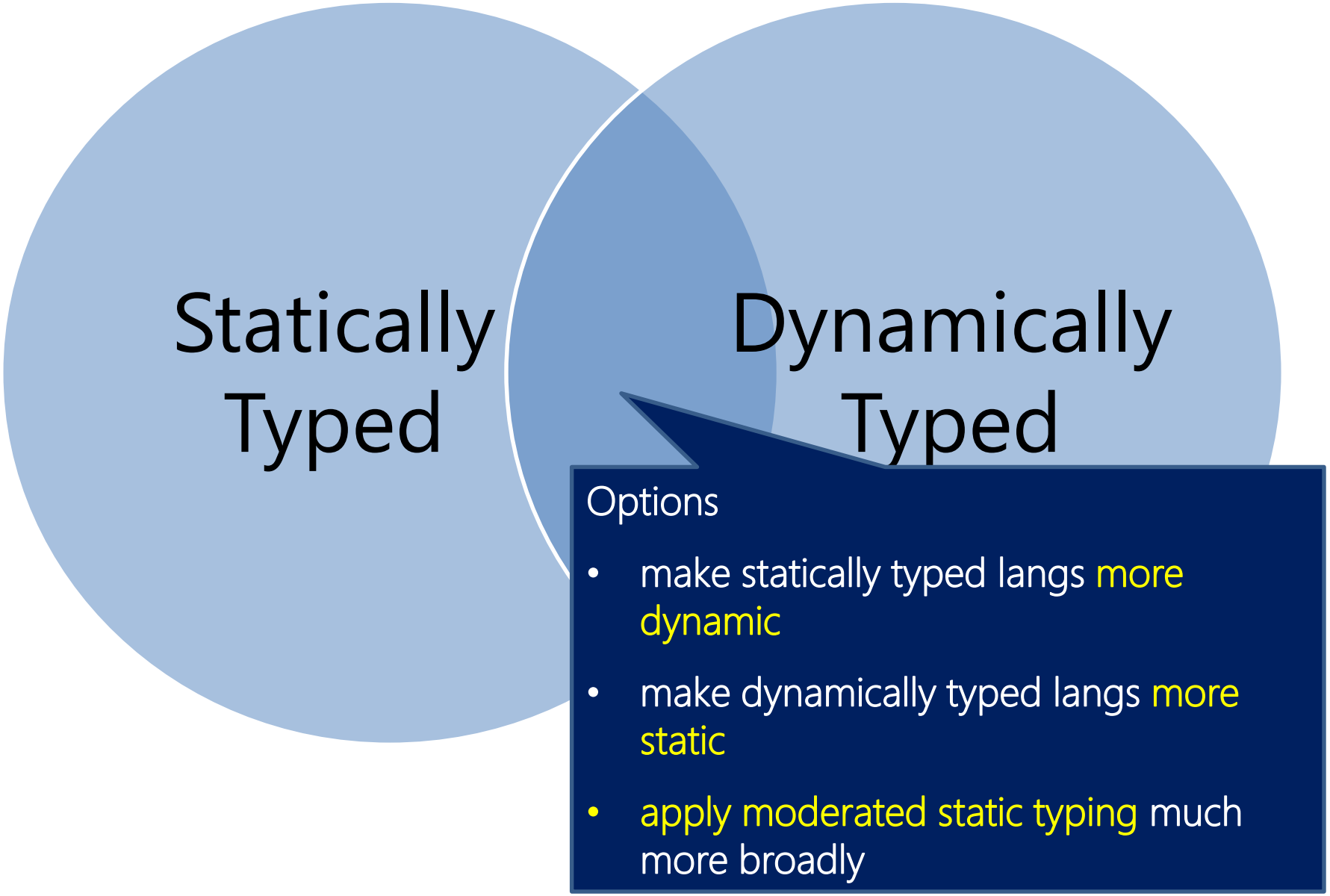
We are living through an information revolution

Data is like water…

We need to bring information **into** the language…

At **internet-scale**, **strongly tooled**, **strongly typed**

Paradigm Locator



Code we love :)
- type providers

http://fsharp.github.io/FSharp.Data/images/json.gif

http://fsharp.github.io/FSharp.Data/images/csv.gif

(but only if they are well engineered, robust, tested, useful, general-purpose)

Code I love: Computation expressions

seq { … }, […], [| … |]

- Many examples, almost every page of code
- Alternative is Seq.append

```
let rec allSymbolsInEntities compGen (entities: FSharpEntitylist) =
   List.concat [
     entities;
     (e.GenericParameters
       |> List.filter (fun gp -> compGen || not gp.IsCompilerGenerated));
     (e.MembersFunctionsAndValues
       |> List.filter (fun x -> compGen || not x.IsCompilerGenerated)
       |> List.collect (fun x ->
           List.cons x
             (x.GenericParameters
              |> List.filter (fun gp -> compGen || not gp.IsCompilerGenerated))));
     e.UnionCases;
     (x.UnionCaseFields
       |> List.filter (fun f -> compGen || not x.IsCompilerGenerated));
     (x.Fields
       |> List.filter (fun f -> compGen || not x.IsCompilerGenerated));
     allSymbolsInEntities compGen e.NestedEntities ]
```

```
      for f in x.UnionCaseFields do
        if compGen || not f.IsCompilerGenerated then
          yield f

      for x in e.FSharpFields do
        if compGen || not x.IsCompilerGenerated then
          yield x

      yield! allSymbolsInEntities compGen e.NestedEntities ]
```

async { ... }

- One example:

```
let server = async { run dotnetCli "watch run" serverPath }
let client = async { run dotnetCli "fable webpack-dev-server" clientPath }
```

```
[ server; client; browser]
|> Async.Parallel
|> Async.RunSynchronously
```

```
[ server; client; browser]
|> Async.Parallel
|> Async.RunSynchronously
```

asyncSeq { … }

- It's a library
- No inversion of control, you think in a "forward" way

```
let withTime =
    asyncSeq {
        do! Async.Sleep 1000 // non-blocking sleep
        yield 1
        do! Async.Sleep 1000 // non-blocking sleep
        yield 2
    }
```

```
let intervalMs (periodMs:int) =
    asyncSeq {
        yield DateTime.UtcNow
        while true do
            do! Async.Sleep periodMs
            yield DateTime.UtcNow
    }
```

https://fsprojects.github.io/FSharp.Control.AsyncSeq/

I love…

- Code that can be debugged
- Code that is commented
- Code that is tested
- Code that is performant
- Code that is under CI
- Code that is readable

Please, implement .ToString() and DebuggerDisplay to aid debugging

Please, use good variable names

Please, use good method names and seek good stack traces

Please, comment your code well

In Closing

F# Emphasises Clear, Code to Solve Real-world Problems

This is the F# Code I love

Not all Functional Code is Good Code

Object Programming

<>

Object-oriented Programming

Thanks! Questions?

Mutation Good, Mutation Bad

Good mutation

F# gives you a lightweight mechanism for encapsulation – use it

- Graphs of data frequently easier with mutation ✗
- Encapsulated, performant data very common
- Please, encapsulate mutable data ✓

```
let addToClosureTable (t:Dictionary<_,_>) (a,b) =
   if not (t.ContainsKey(a)) then
       t.[a] <- HashSet<_>(HashIdentity.Structural)

   t.[a].Add(b)

let closureTableCount (t:Dictionary<_,_>) = t.Count

let closureTableContains (t:Dictionary<_,HashSet<_>>) (a,b) =
    t.ContainsKey(a) && t.[a].Contains(b)
```

```
/// The results of computing the LALR(1) closure of an LR(0) kernel
type Closure1Table() =

    let t = new Dictionary<Item0,HashSet<TerminalIndex>>()

    member __.Add(a,b) =
        if not (t.ContainsKey(a)) then
            t.[a] <- HashSet<_>(HashIdentity.Structural)

        t.[a].Add(b)

    member __.Count = t.Count

    member __.Contains(a,b) = t.ContainsKey(a) && t.[a].Contains(b)
```

“ref” is often bad

- “let mutable x = y” is nearly always better than “let x = ref y” ✘
- Localizes the mutation to a larger expression, type or class ✓
- We are planning to deprecate “!” and “:=” to a compat module in F# 4.5 or 5.0

Please, nearly always avoid using “ref” and just use “let mutable” in an expression or type

```
let kernels =

  let mutable acc = Set.empty

  ProcessWorkList startKernels (fun kernel ->
    if not (acc.Contains(kernel)) then
      acc <- acc.Add(kernel)
    ...)

  acc |> Seq.toList
```

null

- F# heavily biased against it
- F#-defined types do not have null as a normal value

However I have used it for

- Compact memory representations
- Manual implementations of mutating fixups
- Avoiding one indirection for an option type

You can do this (F# 4.1)

```
[<Struct>]
type EntityRefState =
    | Resolved of Entity
    | Unresolved of NonLocalEntityRef

type EntityRef =
    { mutable state: EntityRefState }

    static member Resolved x = { binding=Resolved x }

    static member Unresolved x = { binding=Unresolved x }

    member __.Resolve() = …
```

But you'll see this sort of thing very occasionally

```
type EntityRef =
   { /// Filled in when a entity reference has been resolved
     mutable binding: Entity

     /// Indicates a reference to something in another assembly
     nlr: NonLocalEntityRef }

   static member Resolved x = { binding=x; nlr=Unchecked.defaultof<_> }

   static member Unresolved x = { binding=Unchecked.defaultof<_>; nlr=x }

   member __.Resolve() = …
```