F# Code I Love

Don Syme

F# Community Contributor, Language Designer, Researcher @ Microsoft

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

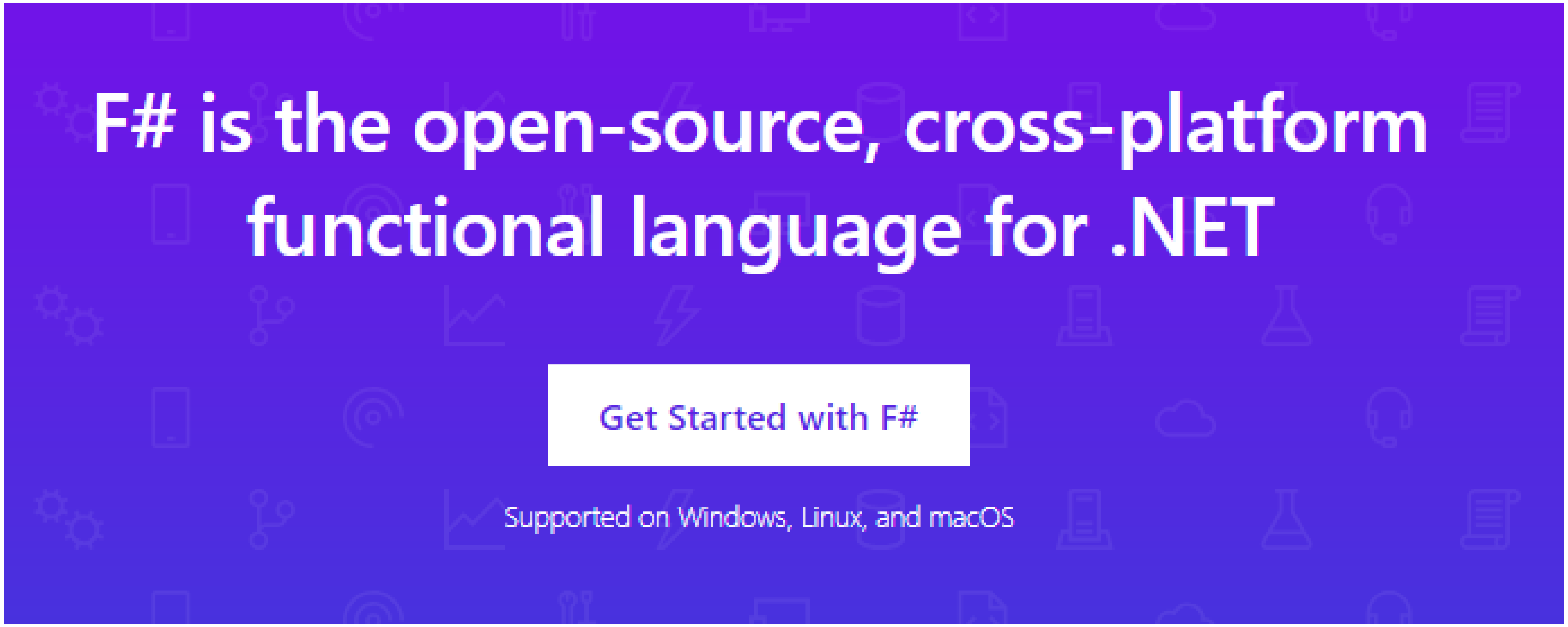
…and some that I love a little less :)

…and how this relates to the language features
and F# 5.0+

Aside

The Early History of F# - HOPL IV (2021)

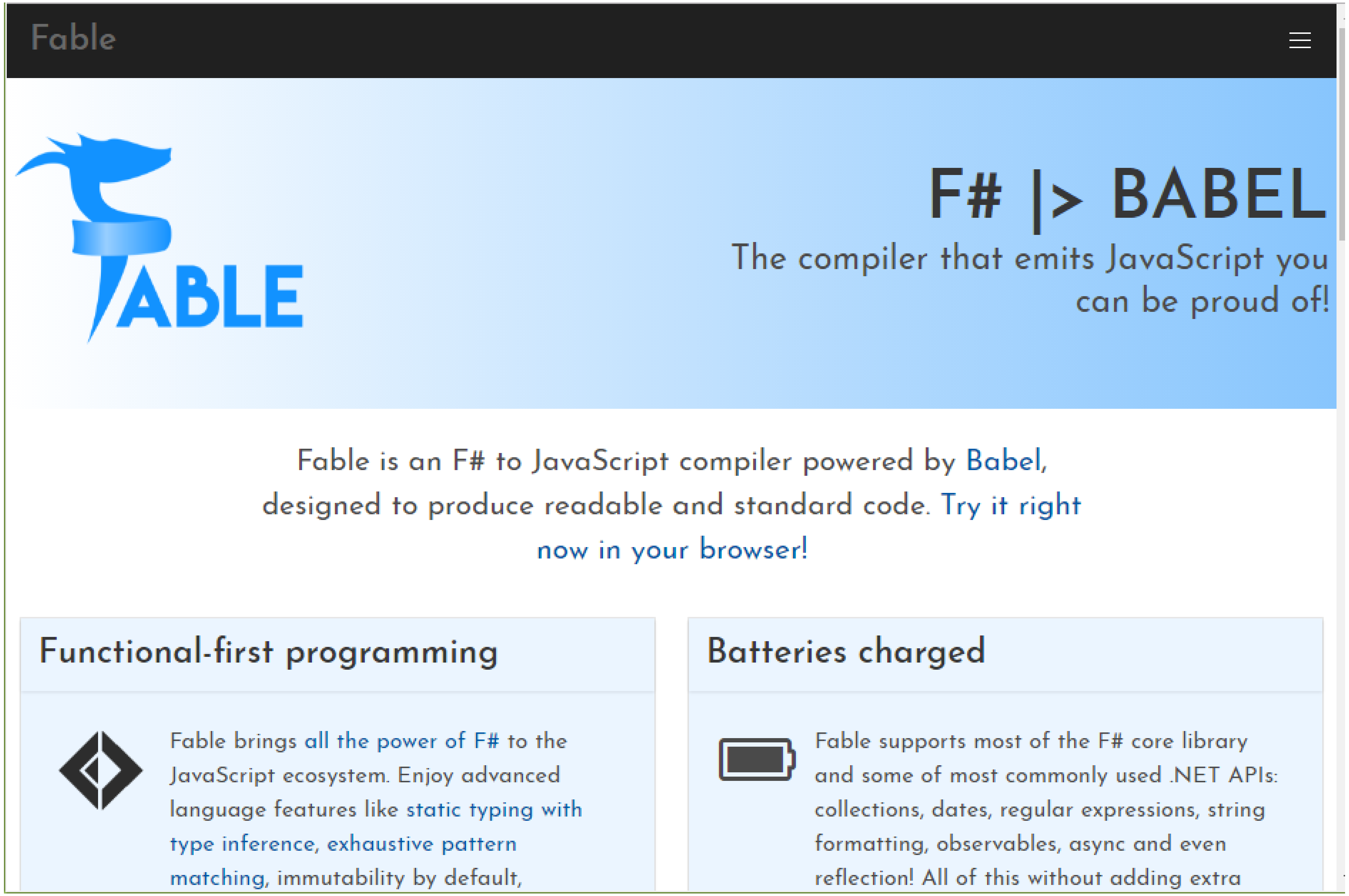
fsharp.org/history



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

Supported on Windows, Linux, and macOS

www.microsoft.com/net/



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

F# get started

```
dotnet new -lang F#

dotnet build
```

F# tools are part of the .NET SDK, available everywhere

F# for the backend

```
dotnet new -i "giraffe-template::*"

dotnet giraffe
```

GIRAFFE

A functional ASP.NET Core micro web framework for building rich web applications.

F# offers extremely high-performance functional-first server-side programming

github.com/giraffe-fsharp/Giraffe

F# for the frontend (JS)

```
dotnet new -i "Fable.Template::*"

dotnet new fable
npm install
npm start
```

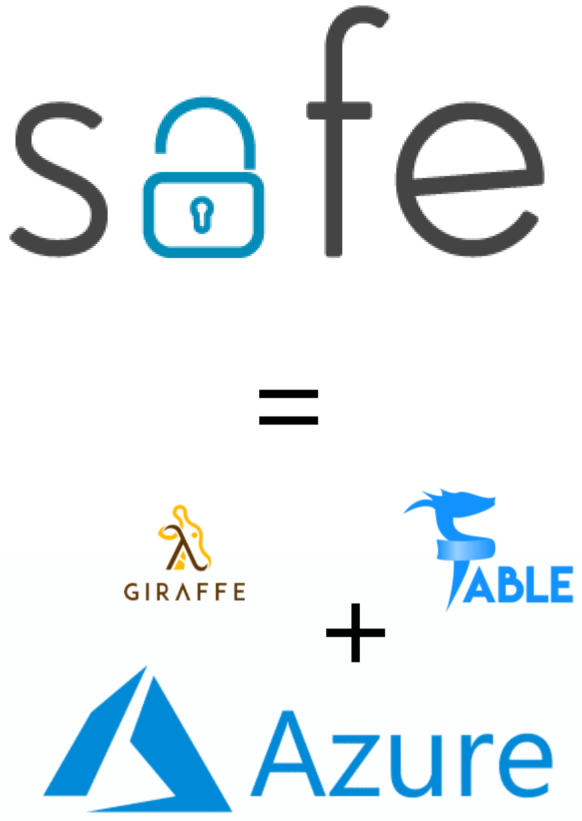
Like Typescript, F# lives happy in the node/npm ecosystem. You can use F# as a Javascript-first language

F# for the frontend (WASM)

```
dotnet new -i Bolero.Templates

dotnet new bolero-app
dotnet run
```

F# for the full stack

```
dotnet new -i SAFE.Template

dotnet new SAFE
dotnet tool restore
dotnet fake run
```



This is the best thing. It powers real businesses including Norway's NRK

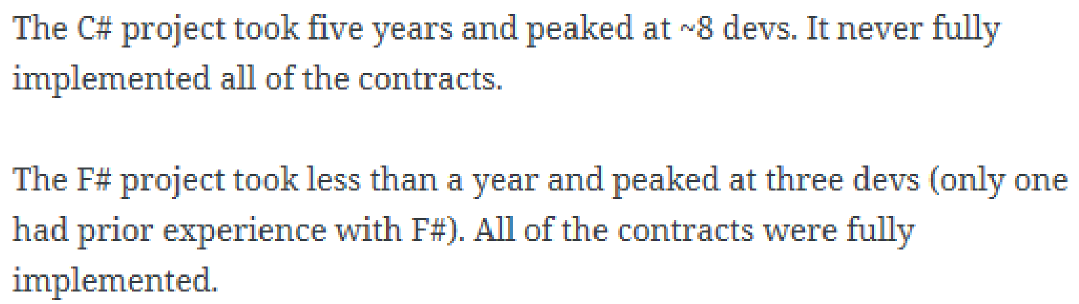
A functional-first approach makes a huge difference in practice

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.

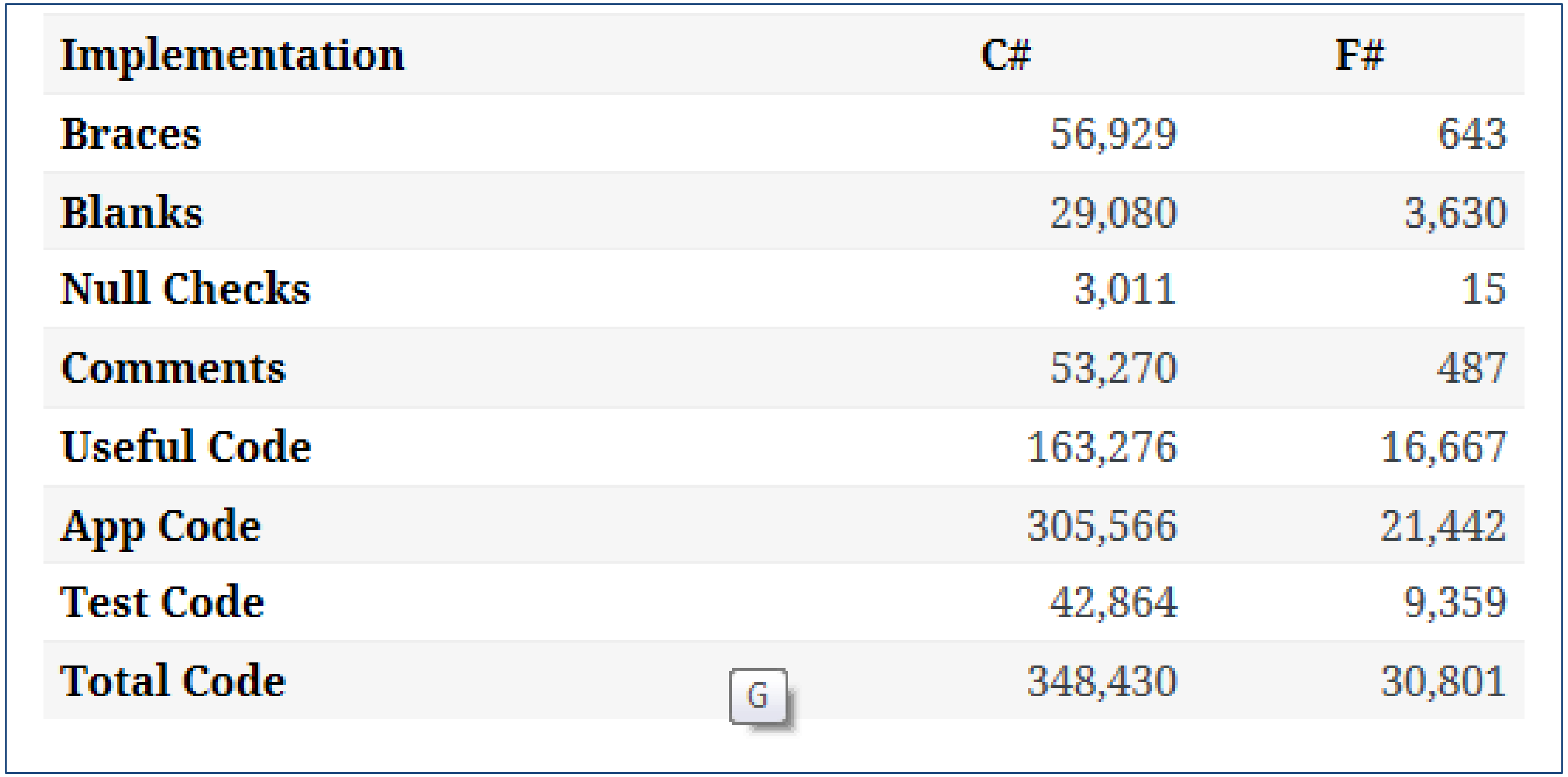
The F# project took less than a year and peaked at three devs (only one had prior experience with F#). All of the contracts were fully implemented.

30,000

lines of robust F#, with
parallel +more features

An application to evaluate the revenue due from [Balancing Services](http://simontcousins.azurewebsites.net/does-the-language-you-use-make-a-difference-revisited/) 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

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/

The Community at the Centre of the Technology

fsharp.org

The F# Language Design Process

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

F# 4.1 (2017)

- ✓ Optional large scope cycles
- ✓ 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 (2018)

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

- ✓ Tooling updates

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

F# 4.6 (2019)

- ✓ Anonymous records
- ✓ Tooling updates

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

F# 4.7 (2019)

- ✓ Implicit yields
- ✓ /langversion
- ✓ indentation relaxations

https://github.com/fsharp/fslang-design/tree/master/FSharp-4.7

F# 5.0! (2021)

- ✓ #r nuget packages in scripts “#r "nuget: Newtonsoft.Json"
- ✓ Jupyter and .NET Interactive notebooks!
- ✓ string interpolation
- ✓ nameof
- ✓ applicatives syntax in computation expressions
- ✓ improved .NET interop
- ✓ improved Map/Set performance + more

https://github.com/fsharp/fslang-design/tree/master/FSharp-5.0

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>?

WARNING: Opinion!

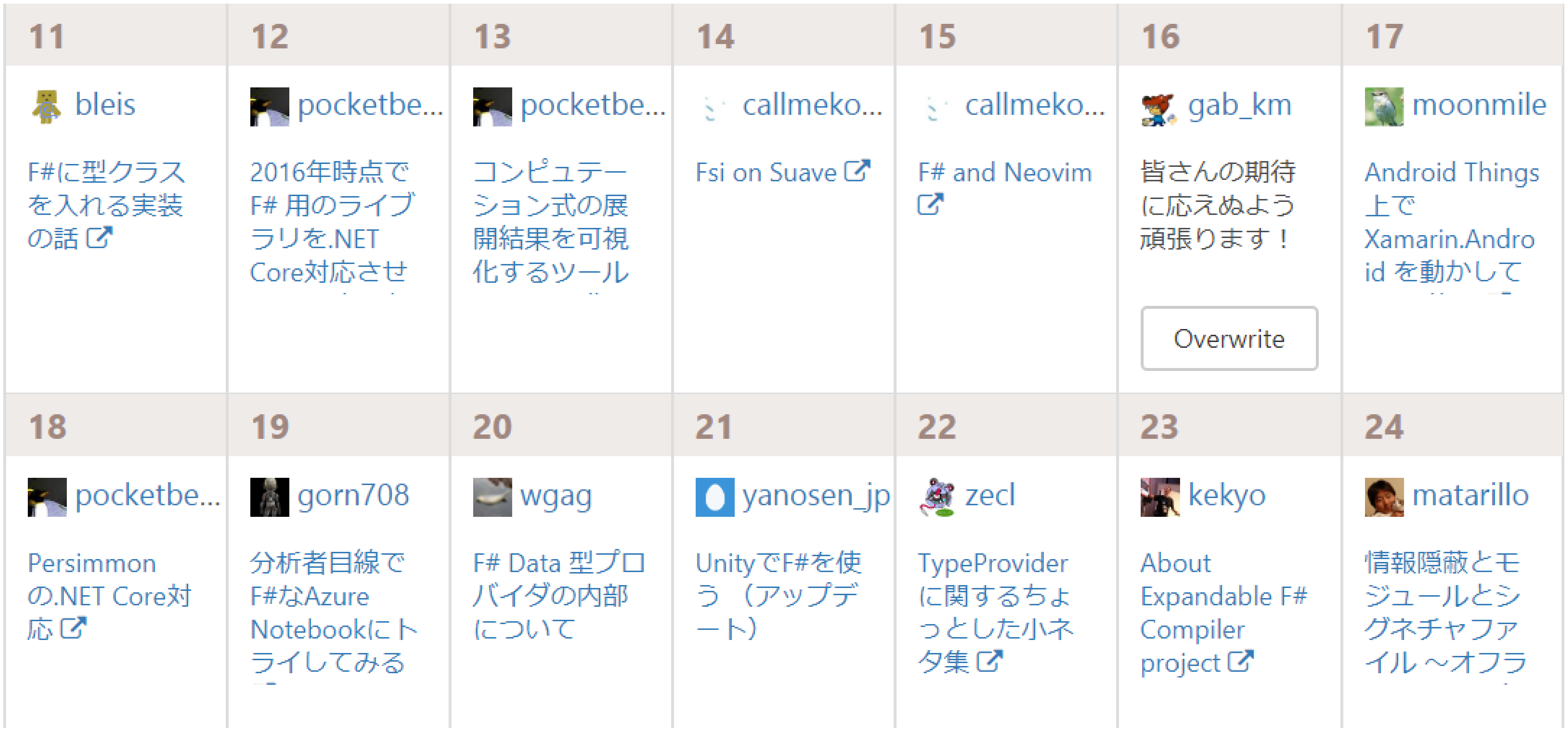
Reminder:

The F# Advent Calendar

(started by F# users in Japan!)

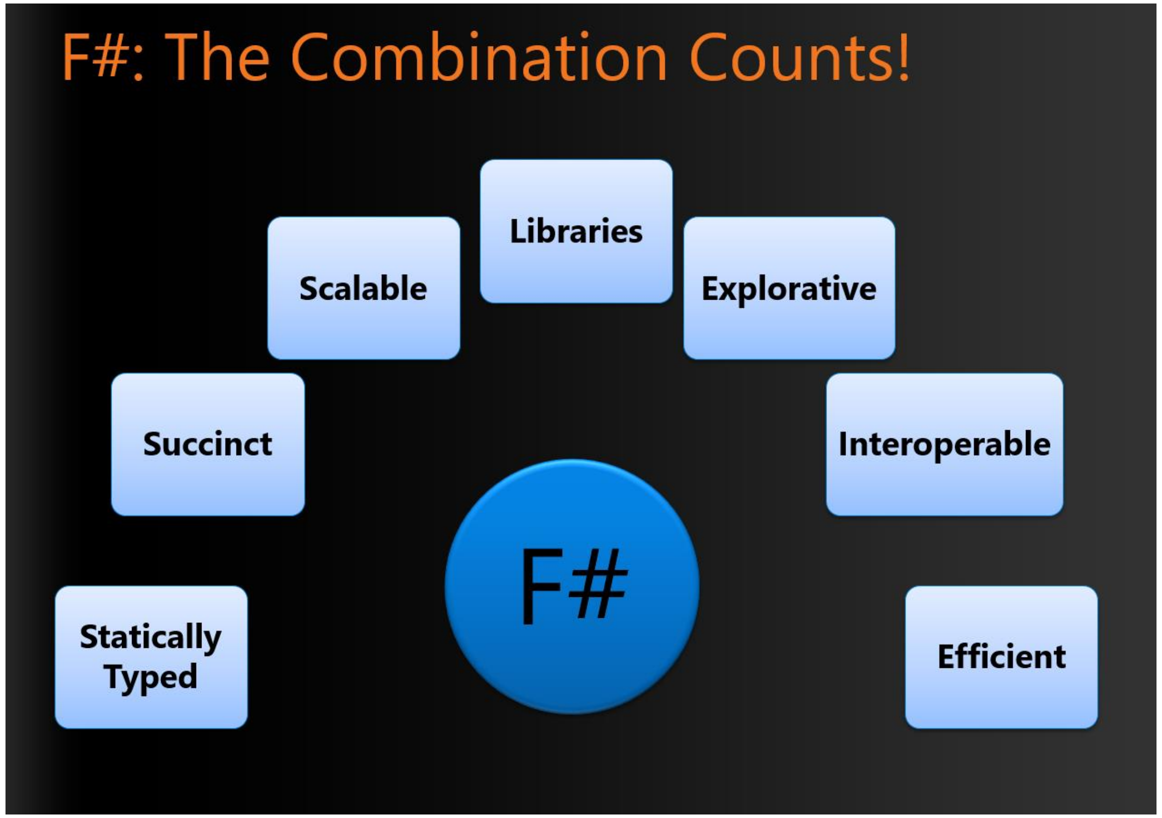
English 2017, 2016, 2015

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



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

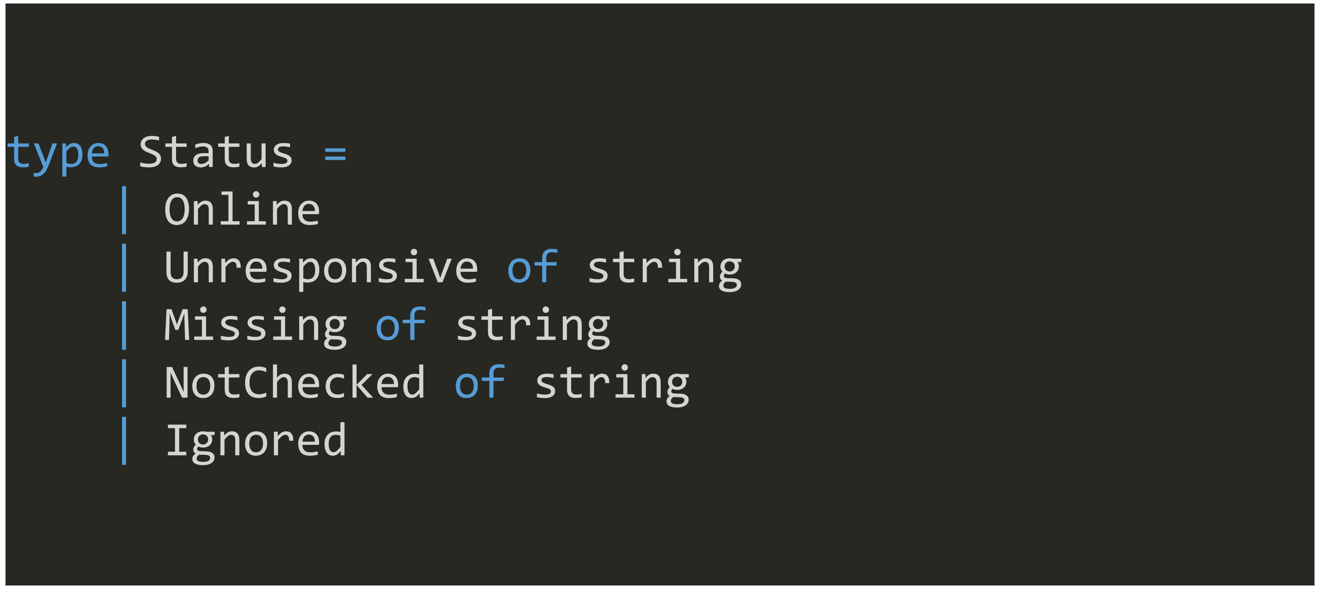
Code I love!
- pipelines
- domain modelling

```
/// 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

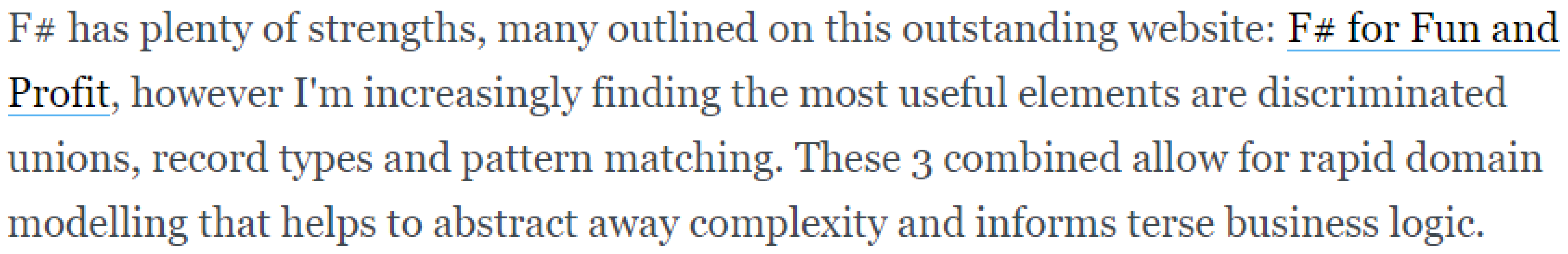
```fsharp
/// Represents information known about a value
type ExprValueInfo =
     | UnknownValue
     | ValValue of ValRef * ExprValueInfo
     | TupleValue of ExprValueInfo[]
     | RecdValue of TyconRef * ExprValueInfo[]
     | UnionCaseValue of UnionCaseRef * ExprValueInfo[]
     | ConstValue of 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.

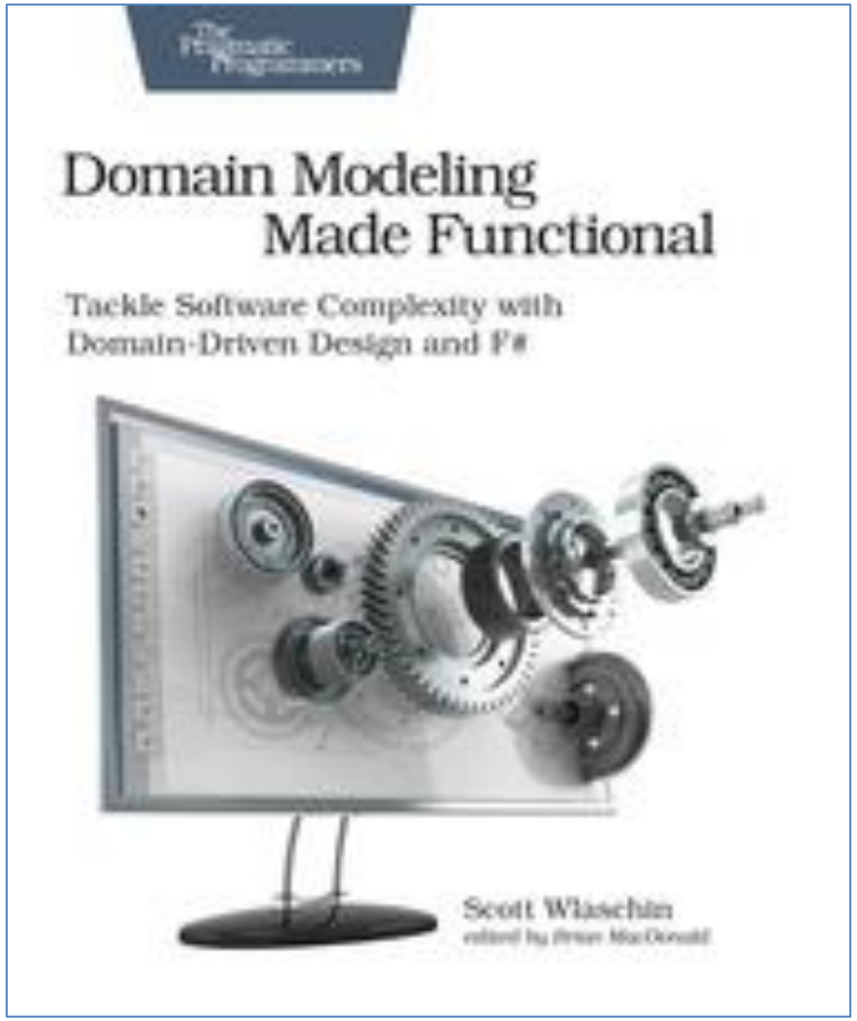
Code we love :)
- pipelines
- domain modelling

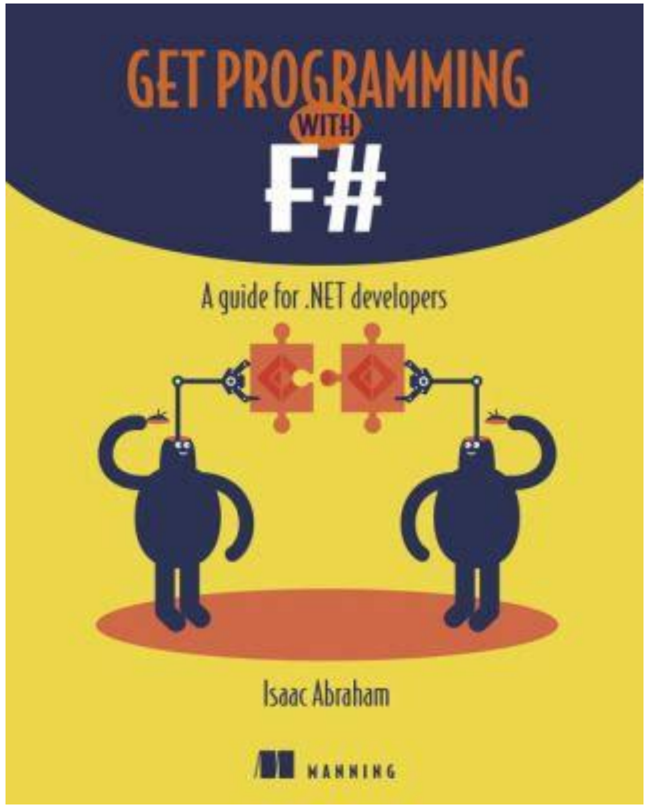
https://medium.com/@odytrice
Ody Mbegbu

```
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


The Pragmatic Programmers
Domain Modeling Made Functional
Tackle Software Complexity with Domain-Driven Design and F#
Scott Wlaschin
edited by Brian MacDonald



Code I love :)
- data scripting + type providers

```
// 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

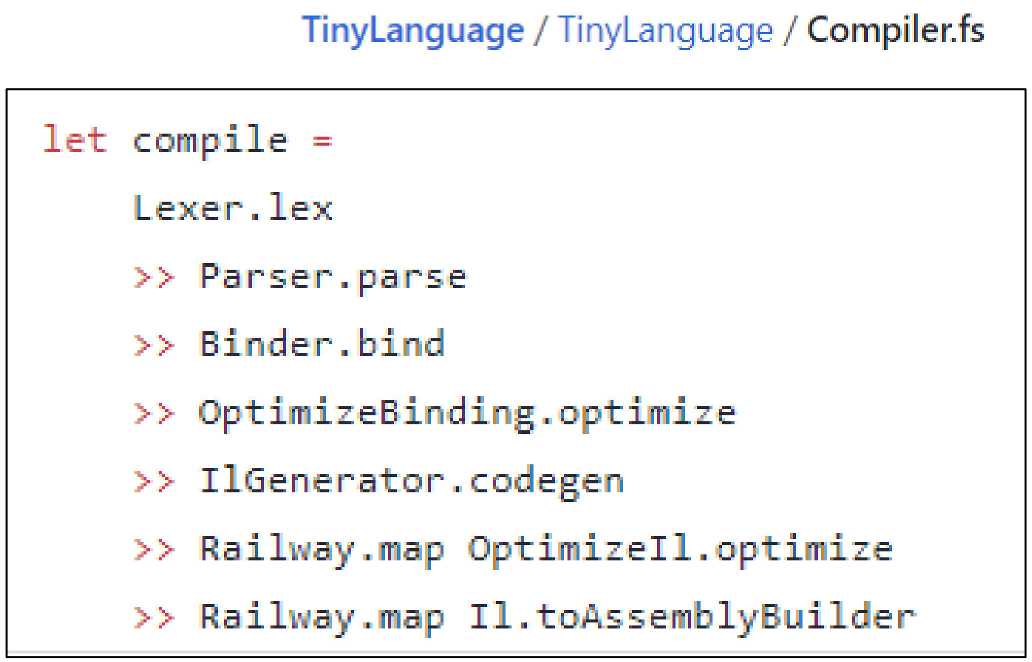
```
/// The view function giving updated content for the page
let view (model: Model) dispatch =
  if model.Pressed then
    Xaml.Label(text="I was pressed!")
  else
    Xaml.Button(text="Press Me!", command=(fun () -> dispatch Pressed))
```

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

A model-view-update web view

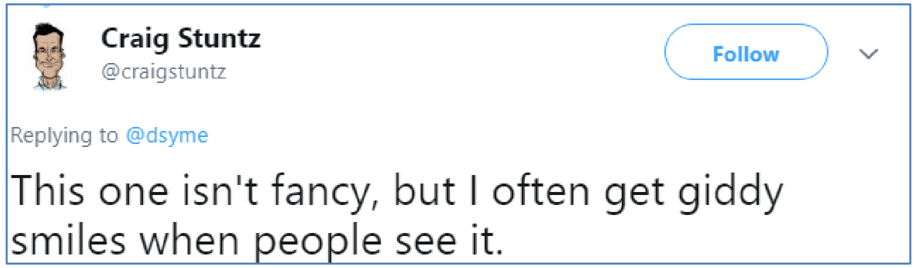
```
/// The view function giving upda
let view model dispatch =
  match model.Text with
  | [| |] ->
    div [] [ div [] [str "Loading..."] ]
  | _ ->
    div [ ClassName "container" ] [
      button [ OnClick (fun _ -> dispatch Faster) ] [ str "Faster" ]
      div [ ClassName "theText" ] [ str model.Text.[model.Index]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

Code 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
```



Craig Stuntz
@craigstuntz

Replying to @dsyme

This one isn't fancy, but I often get giddy smiles when people see it.

Code we love :)

- super-fast compositional web servers

```fsharp
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"
```

<||, <|||

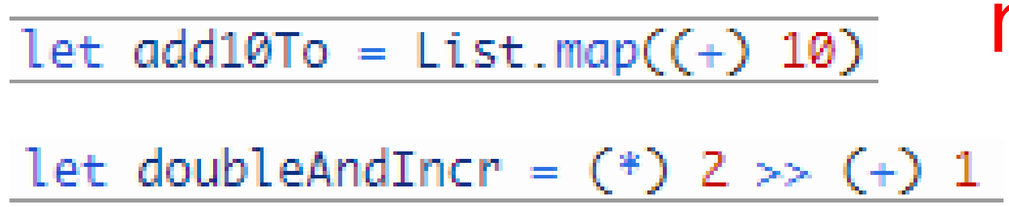
nooo

```
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

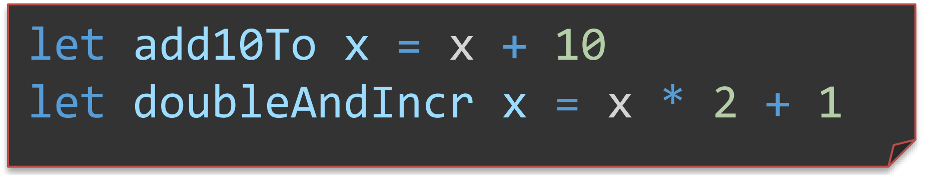
- "Point free" is code without explicit lambdas or let
- Often heavy use of ">>", ">>=", "curry", "uncurry", partial application
- Using and combining existing functions as values is OK
- Please give explicit arguments to functions defined in modules



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

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

nooo

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
```

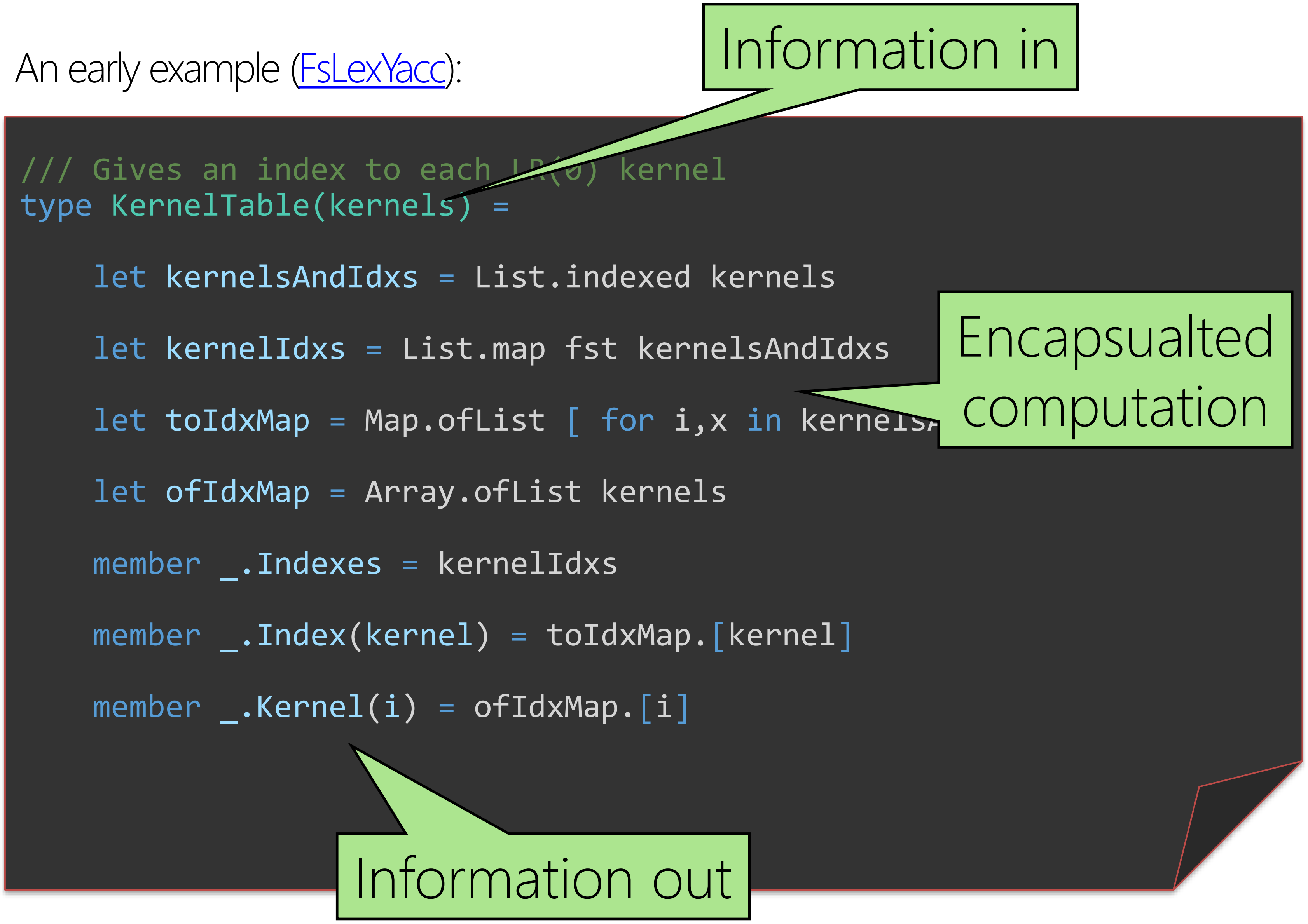
Object Expressions

```
{ 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 kernelsA

    let ofIdxMap = Array.ofList kernels

    member _.Indexes = kernelIdxs

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

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

Deconstructing Object Programming

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 imp
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. pervasive 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

Tolerate

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<_>
3. pervasive method overloading

Forget

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

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

works well with expression-oriented programming

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 I love: computation expressions

“extensible, intuitive, friendly monadic notation on steroids”

seq { … }

[…]

[| … |]

async { … }

option { … }

task { … }

taskSeq { … }

asyncOption { … }

….

seq { … }, […], [| … |]

- Many examples, almost every page of code

- Alternative is explicit append etc

- Typically much more expressive than other comprehension notations

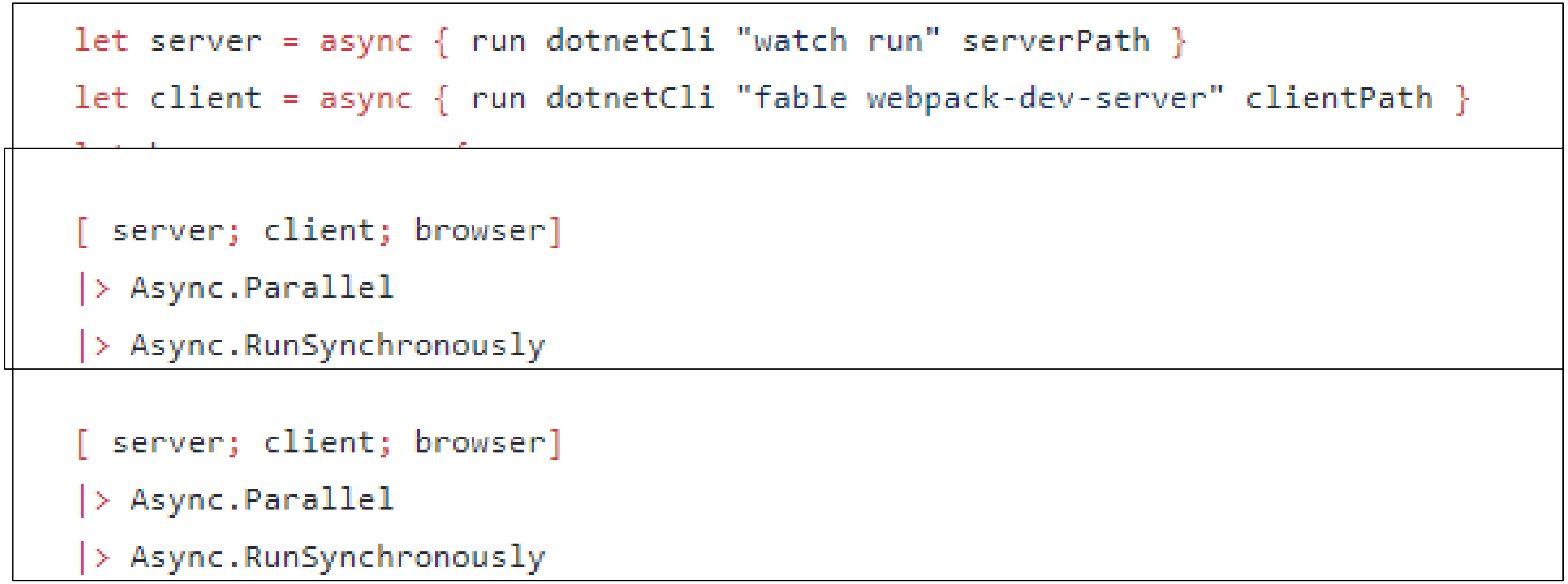
✗

✓

```fsharp
let rec allSymbolsInEntities compGen (entities: FSharpEntitylist) =
    [ yield! entities

      for gp in e.GenericParameters do
        if compGen || not gp.IsCompilerGenerated then
          yield gp

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

        for gp in x.GenericParameters do
          if compGen || not gp.IsCompilerGenerated then
            yield gp

      yield! e.UnionCases

      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

What's coming in F# 5.1/6.0…?

- high-perf computation expressions
- tasks
- anonymous unions
- inline-if-lambda
- additional type-directed conversions for better interop

In Closing

F# Emphasises Clear Code to Solve Real-world Problems

Simple, clear code is the F# Code I Love

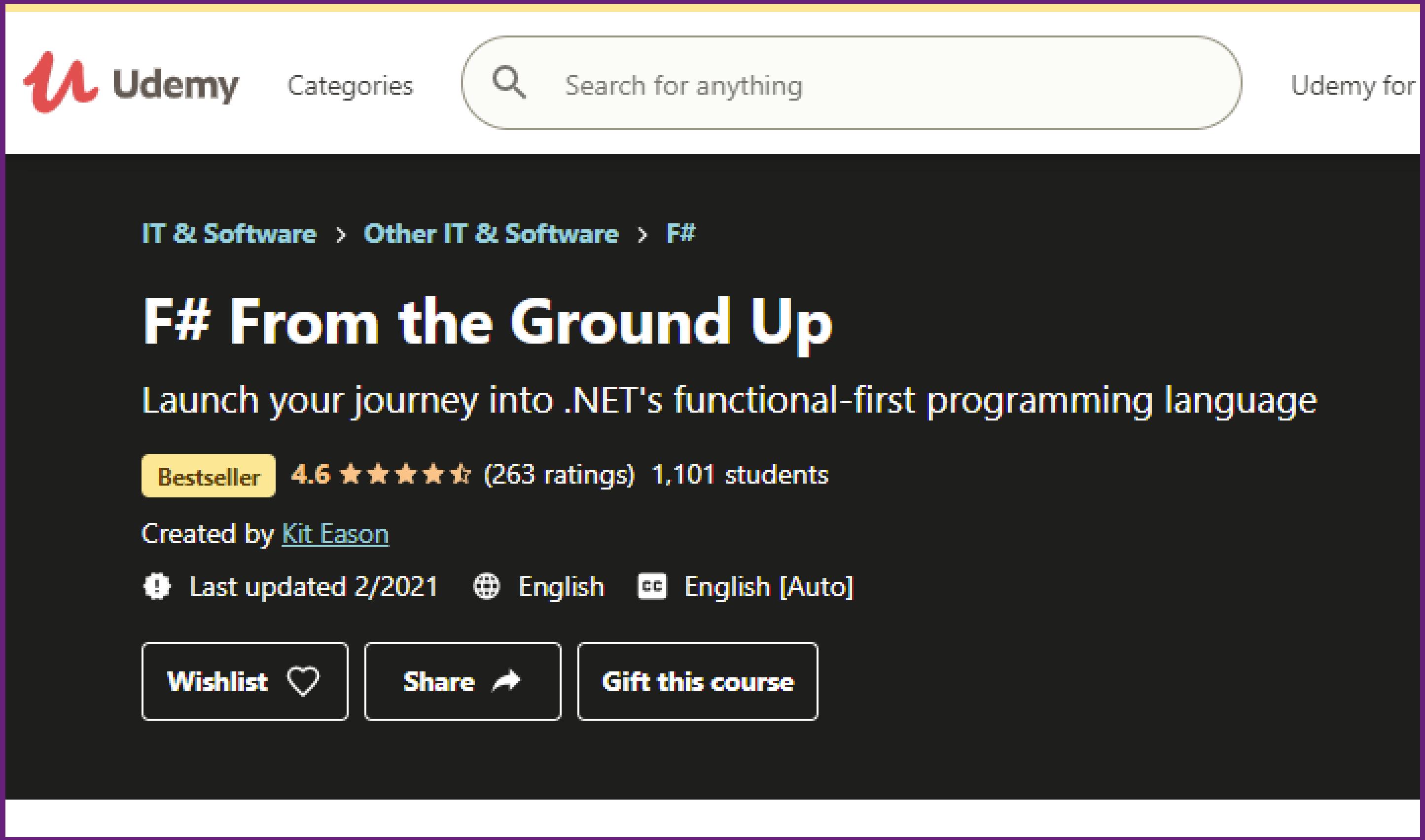
Not all Functional Code is Good Code

Object Programming

< >

Object-Oriented Programming

Thanks! Questions?



IT & Software › Other IT & Software › F#

F# From the Ground Up

Launch your journey into .NET's functional-first programming language

Bestseller 4.6 ★★★★☆ (263 ratings) 1,101 students

Created by Kit Eason

Last updated 2/2021 English English [Auto]