# F# Code I Love

Don Syme, F# Community Contributor
Researcher @ Microsoft Mobile Tools

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

# WARNING: Opinion!

# Reminder:

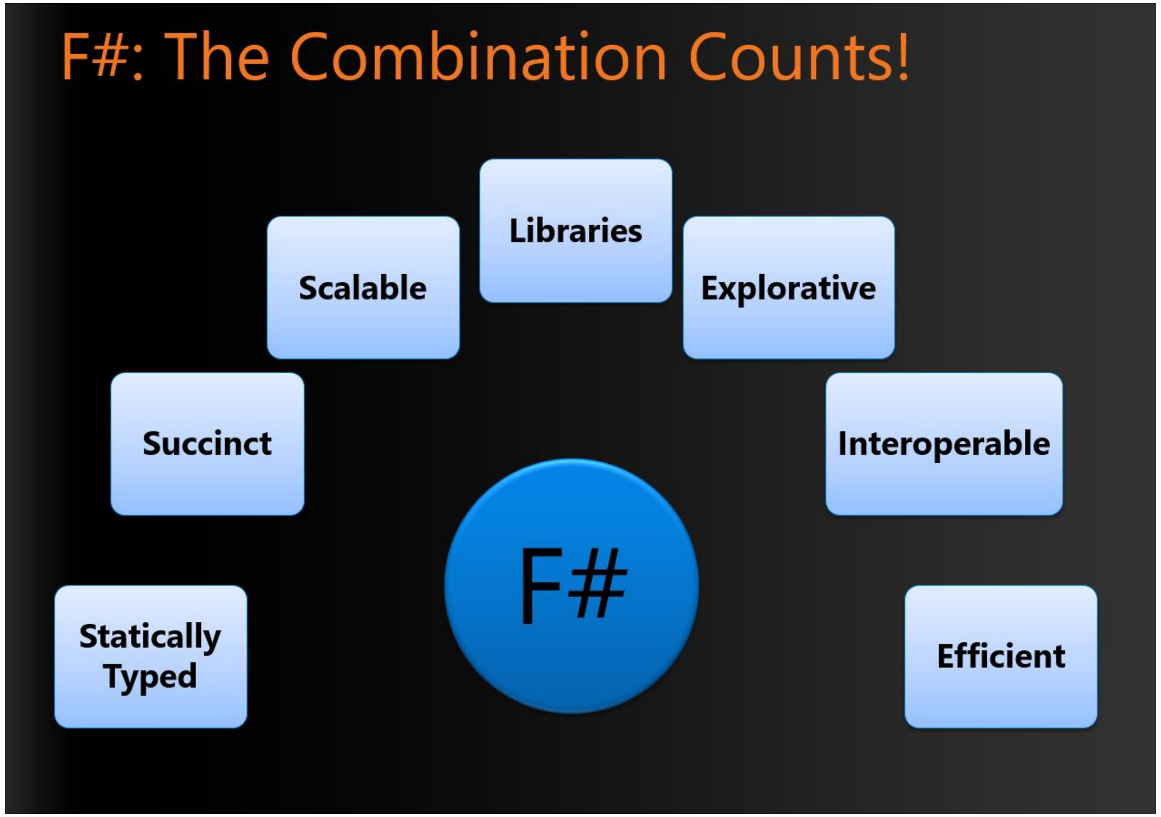
# The F# Advent Calendar

(started by F# users in Japan!)

English 2017, 2016, 2015

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

# Foundations of the F# Design (~2007)

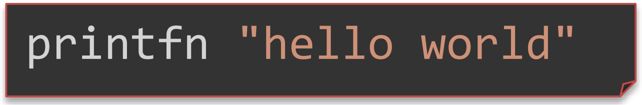


F#: The Combination Counts!

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

# 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

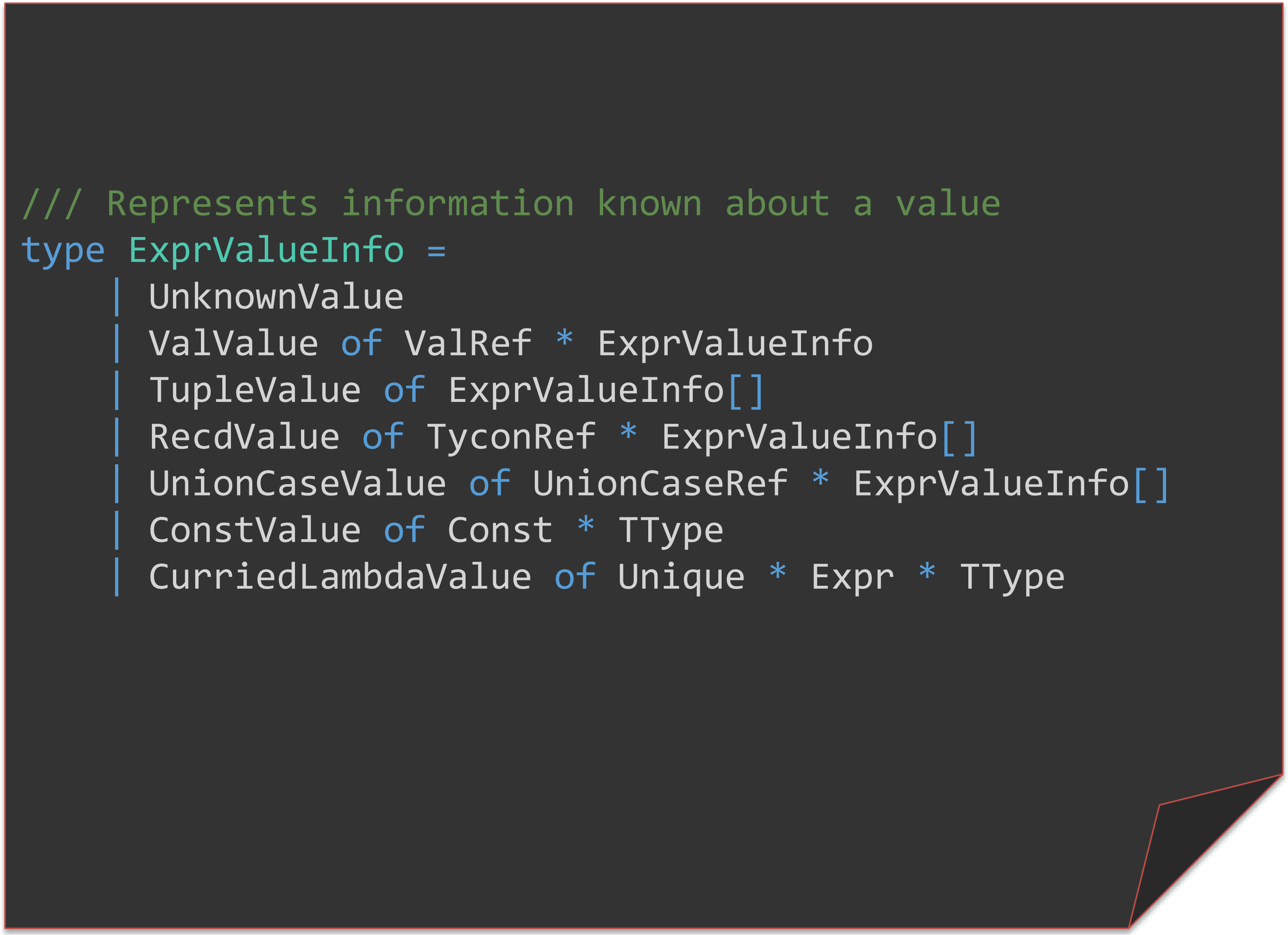
# Code I love!

- 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!

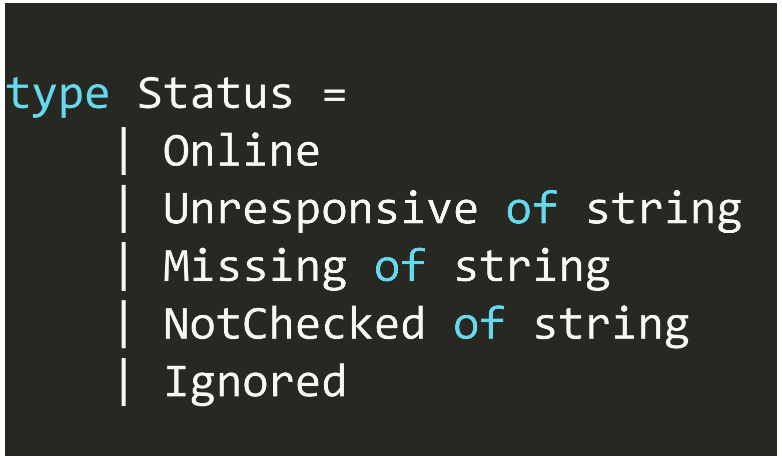
- pipelines
- domain modelling



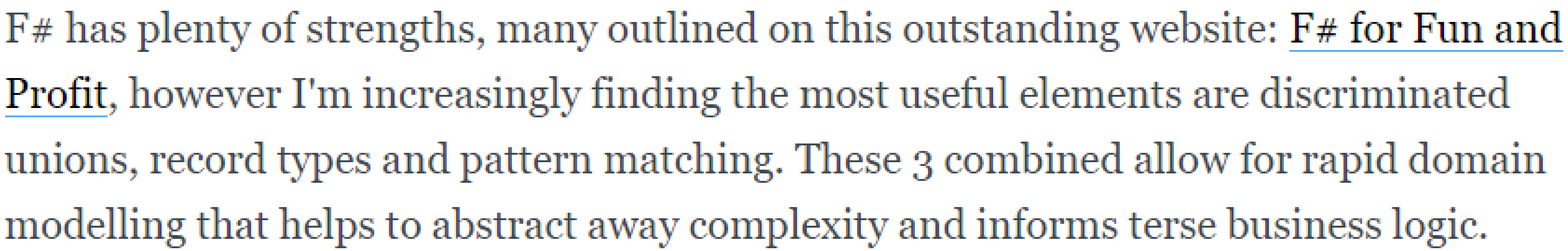
```
/// Represents information known about a value
type ExprValueInfo =
      | UnknownValue
      | ValValue of ValRef * ExprValueInfo
      | TupleValue of ExprValueInfo[]
      | RecdValue of TyconRef * ExprValueInfo[]
      | UnionCaseValue of UnionCaseRef * ExprValueInfo[]
      | ConstValue of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

# Code we love :)
- pipelines
- domain modelling

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



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



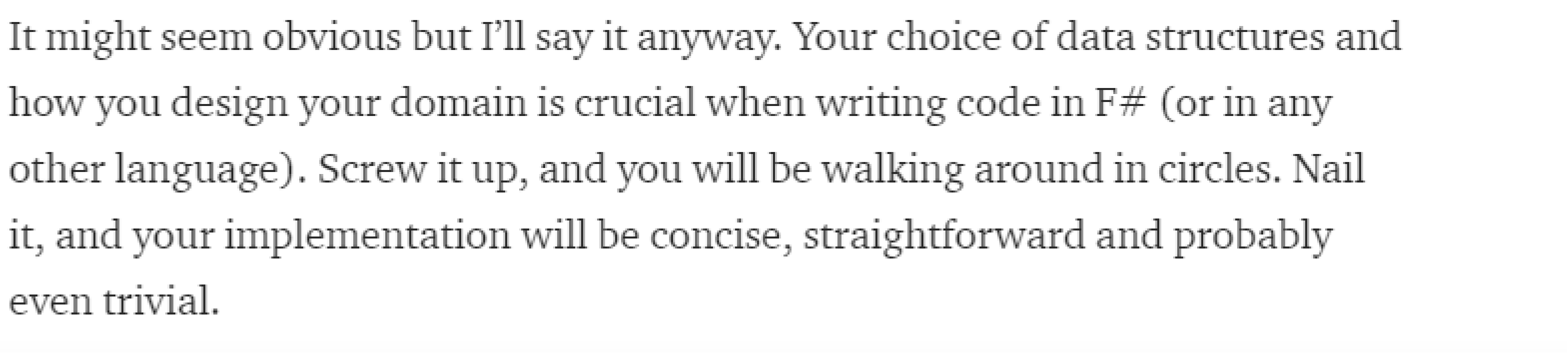
F# has plenty of strengths, many outlined on this outstanding website: F# for Fun and Profit, however I'm increasingly finding the most useful elements are discriminated unions, record types and pattern matching. These 3 combined allow for rapid domain modelling that helps to abstract away complexity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

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

https://medium.com/@odytrice

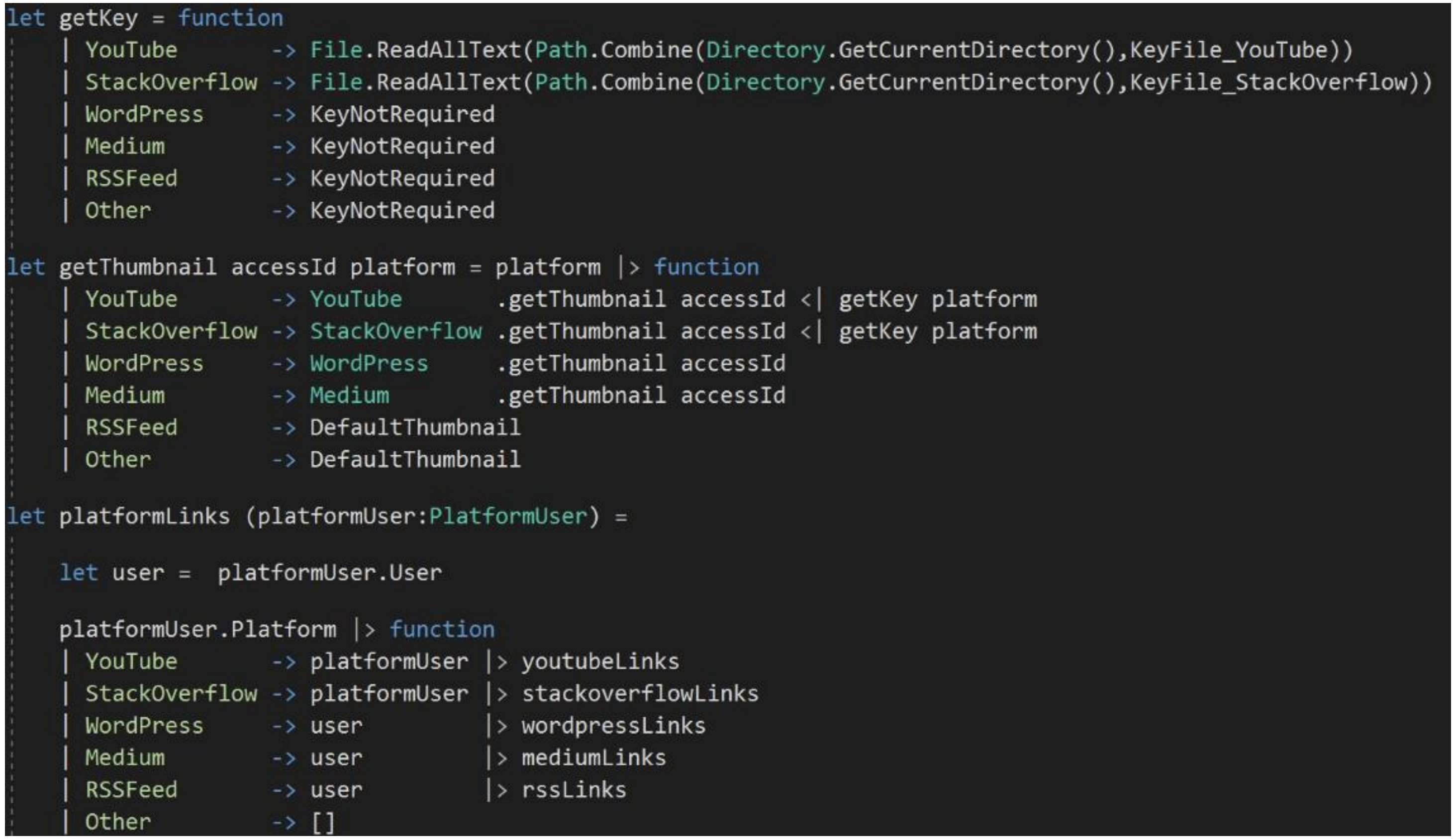
Ody Mbegbu

```
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.

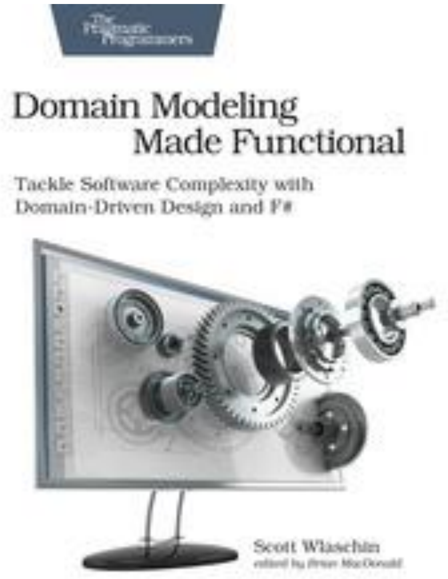
# Code we love :)
- pipelines
- domain modelling
- domain semantics

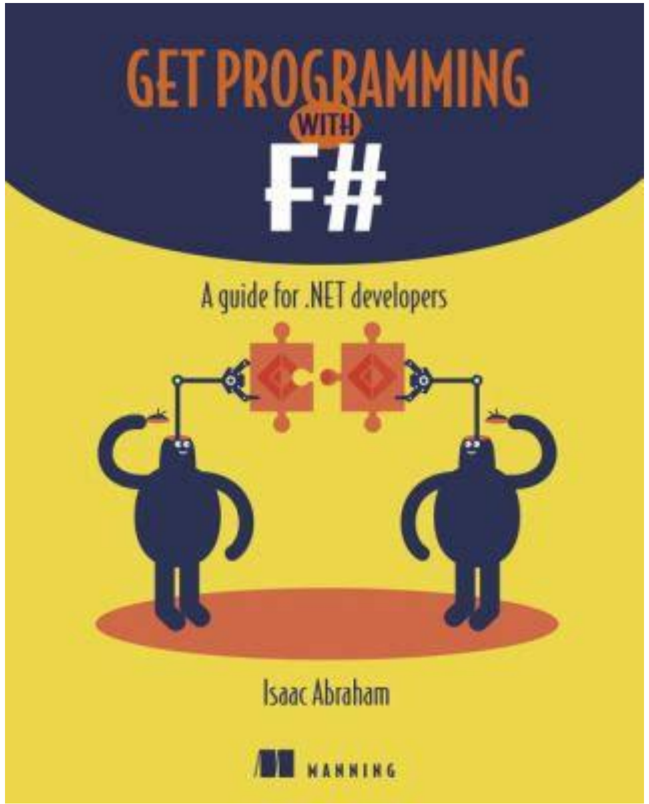


```
let getKey = function
    | YouTube       -> File.ReadAllText(Path.Combine(Directory.GetCurrentDirectory(),KeyFile_YouTube))
    | StackOverflow -> File.ReadAllText(Path.Combine(Directory.GetCurrentDirectory(),KeyFile_StackOverflow))
    | WordPress     -> KeyNotRequired
    | Medium        -> KeyNotRequired
    | RSSFeed       -> KeyNotRequired
    | Other         -> KeyNotRequired

let getThumbnail accessId platform = platform |> function
    | YouTube       -> YouTube       .getThumbnail accessId <| getKey platform
    | StackOverflow -> StackOverflow .getThumbnail accessId <| getKey platform
    | WordPress     -> WordPress     .getThumbnail accessId
    | Medium        -> Medium        .getThumbnail accessId
    | RSSFeed       -> DefaultThumbnail
    | Other         -> DefaultThumbnail

let platformLinks (platformUser:PlatformUser) =

    let user =  platformUser.User

    platformUser.Platform |> function
    | YouTube       -> platformUser |> youtubeLinks
    | StackOverflow -> platformUser |> stackoverflowLinks
    | WordPress     -> user         |> wordpressLinks
    | Medium        -> user         |> mediumLinks
    | RSSFeed       -> user         |> rssLinks
    | Other         -> []
```

Scott Nimrod
https://github.com/bizmonger

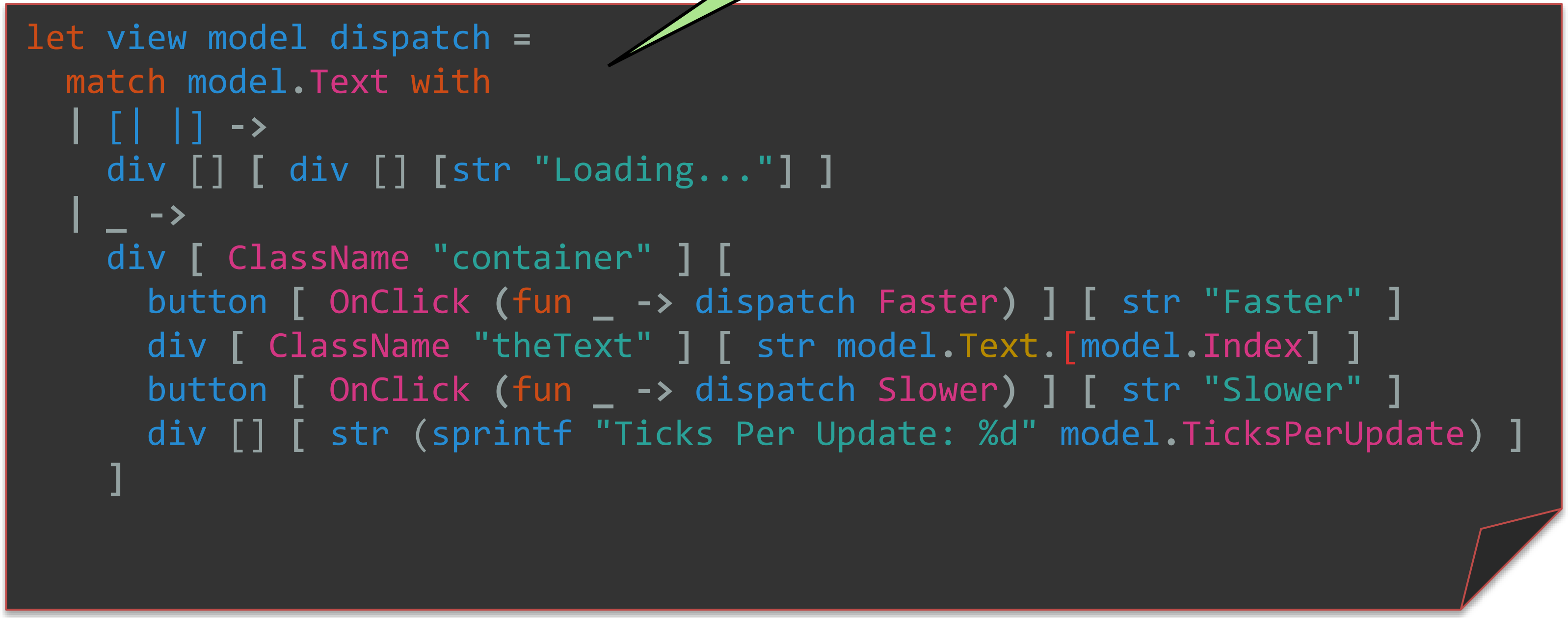
Code we love :)
- pipelines
- domain modelling
- domain semantics


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



# Code we love :)
# - the update/view functions in Fable/Elmish apps

The UI can completely change!



```
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) ]
    ]
```

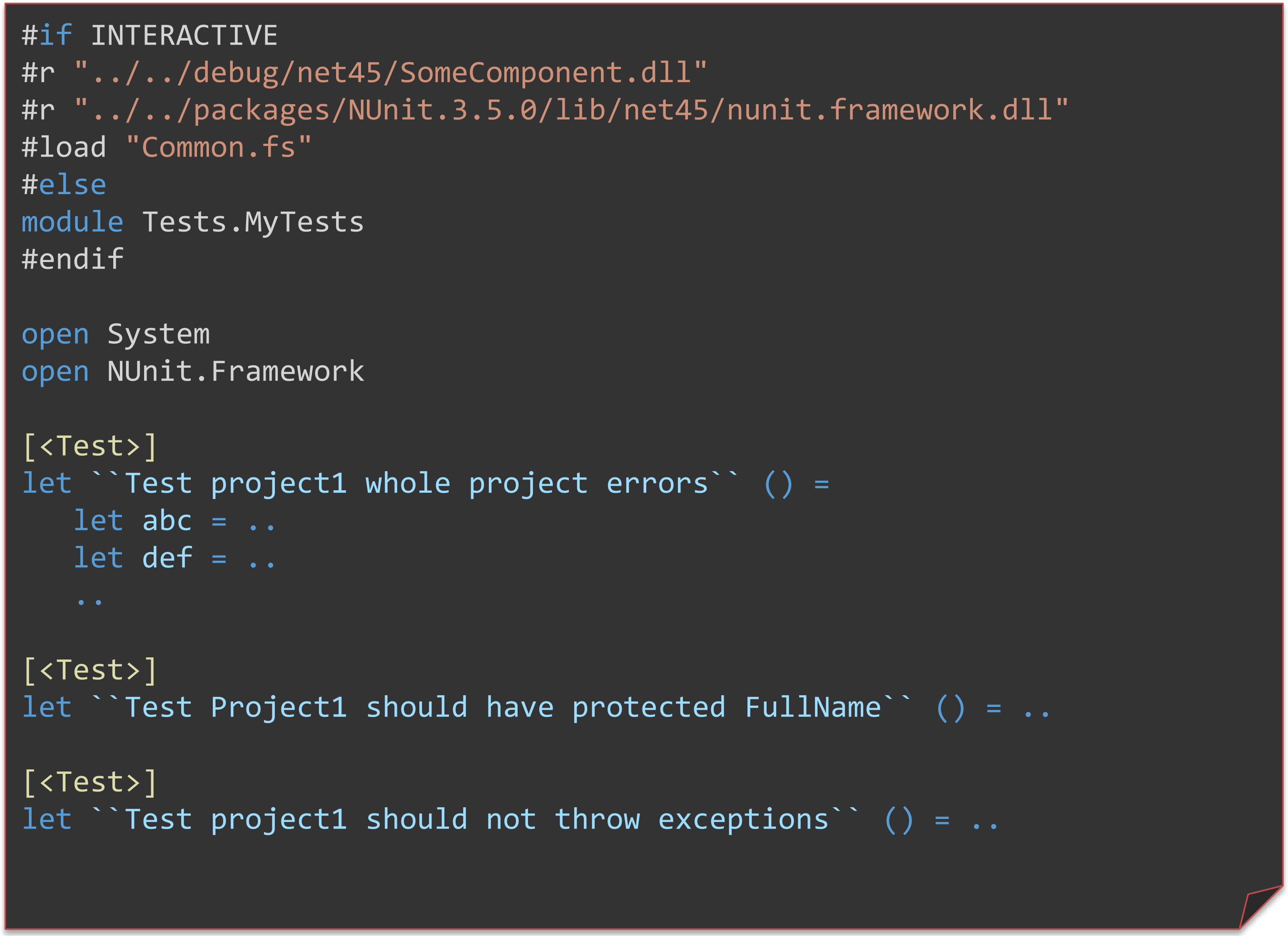
# Code we love :)

- scripts

# Code I love :)
# - scripted tests

Note, I don't like [<TestData(...)>] because it doesn't work well with F# scripting

Ctrl-A, Alt-Enter and you can start debugging and developing individual tests

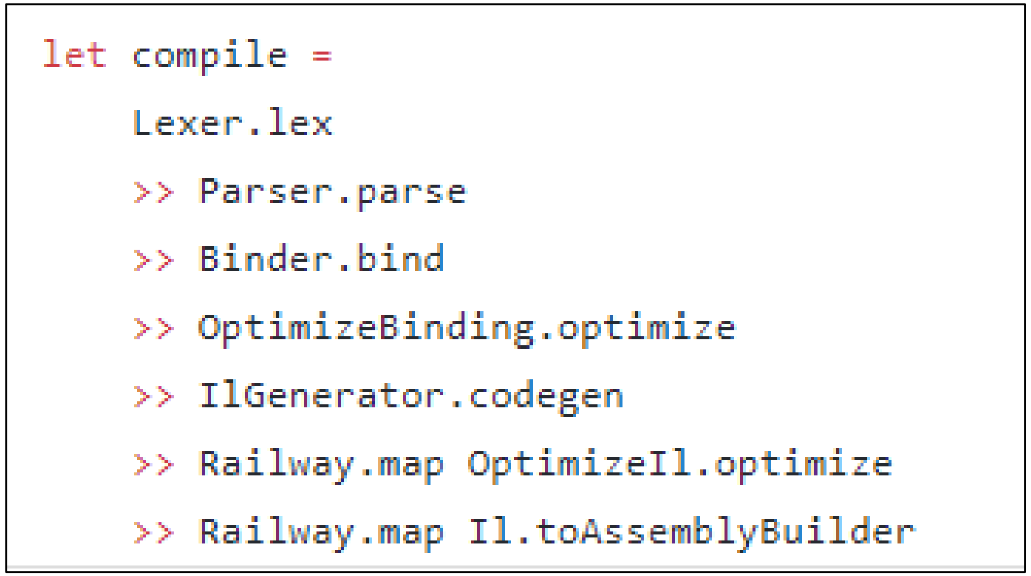


```
#if INTERACTIVE
#r "../../debug/net45/SomeComponent.dll"
#r "../../packages/NUnit.3.5.0/lib/net45/nunit.framework.dll"
#load "Common.fs"
#else
module Tests.MyTests
#endif

open System
open NUnit.Framework

[<Test>]
let ``Test project1 whole project errors`` () =
   let abc = ..
   let def = ..
   ..

[<Test>]
let ``Test Project1 should have protected FullName`` () = ..

[<Test>]
let ``Test project1 should not throw exceptions`` () = ..
```

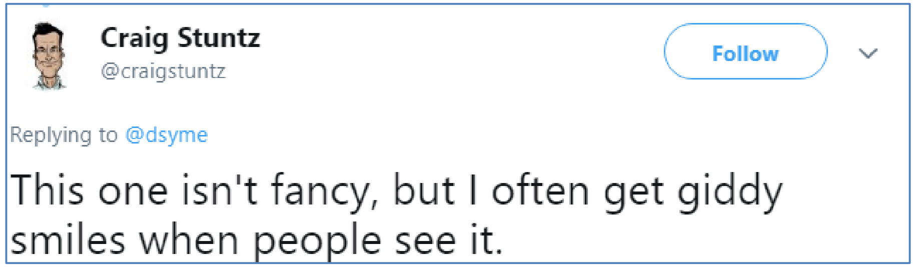
https://github.com/Microsoft/visualfsharp/blob/master/tests/service/ProjectAnalysisTests.fs

# 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 :)
# - type providers

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

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

# But not all Functional Code is Good Code

# curry, uncurry

nooo

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

yes

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

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

nooo

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

Too indecipherable,
too often

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"
```

# <||, <|||

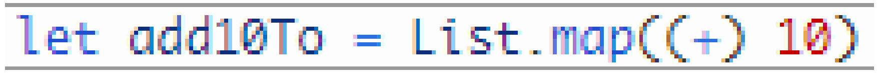
nooo

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

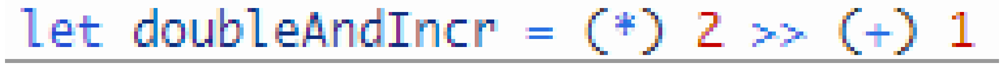
Please, always avoid the <|| and <||| operators. They should be deprecated

# 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



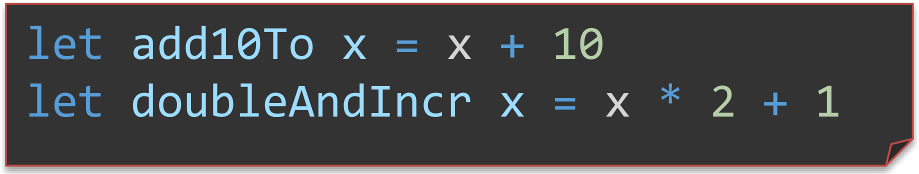
```
let add10To = List.map((+) 10)
```



```
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

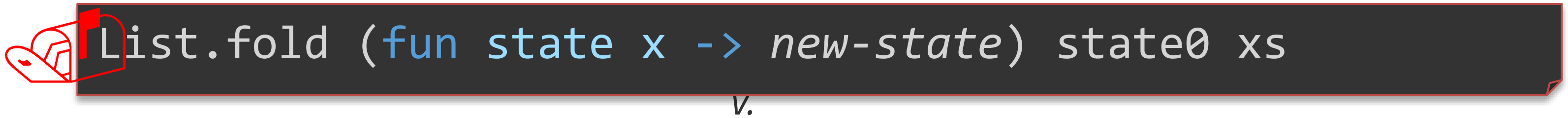
# Fold considered harmful

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

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

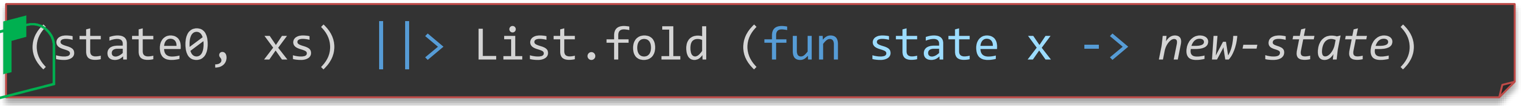
List/Seq/Array.sumBy
List/Seq/Array.maxBy
List/Seq/Array.choose
List/Seq/Array.tryPick
List/Seq/Array.mapFold
List/Seq/Array.reduce
….

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

+
```

```
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

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<_>
23. 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

# 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

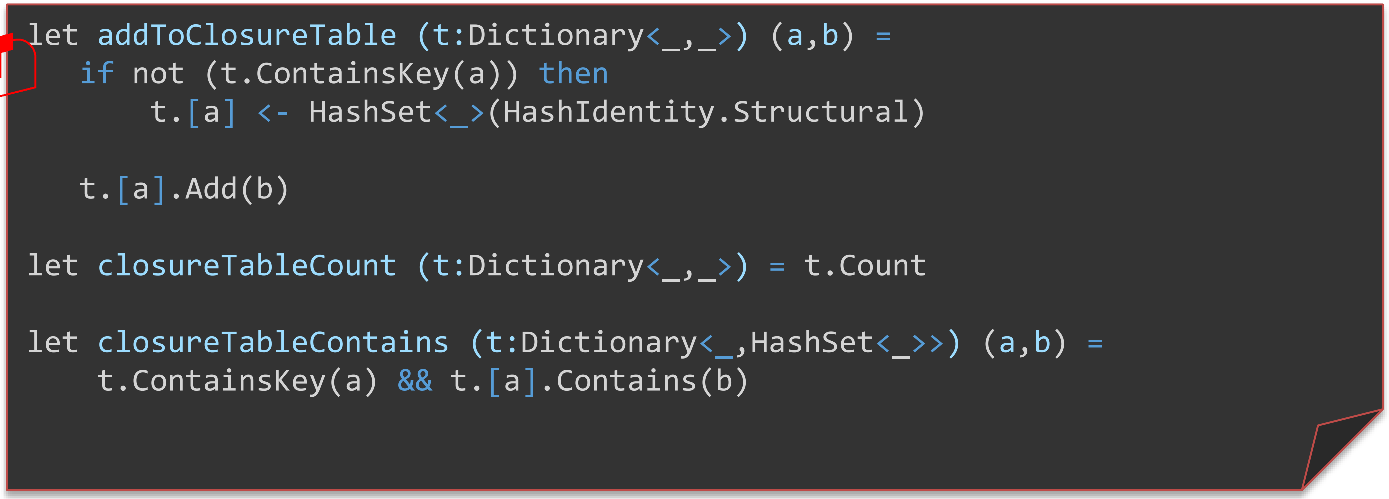
but not embrace full “object-**orientation**” (unless you happen to be in a project using that technique)

# Mutation Good, Mutation Bad

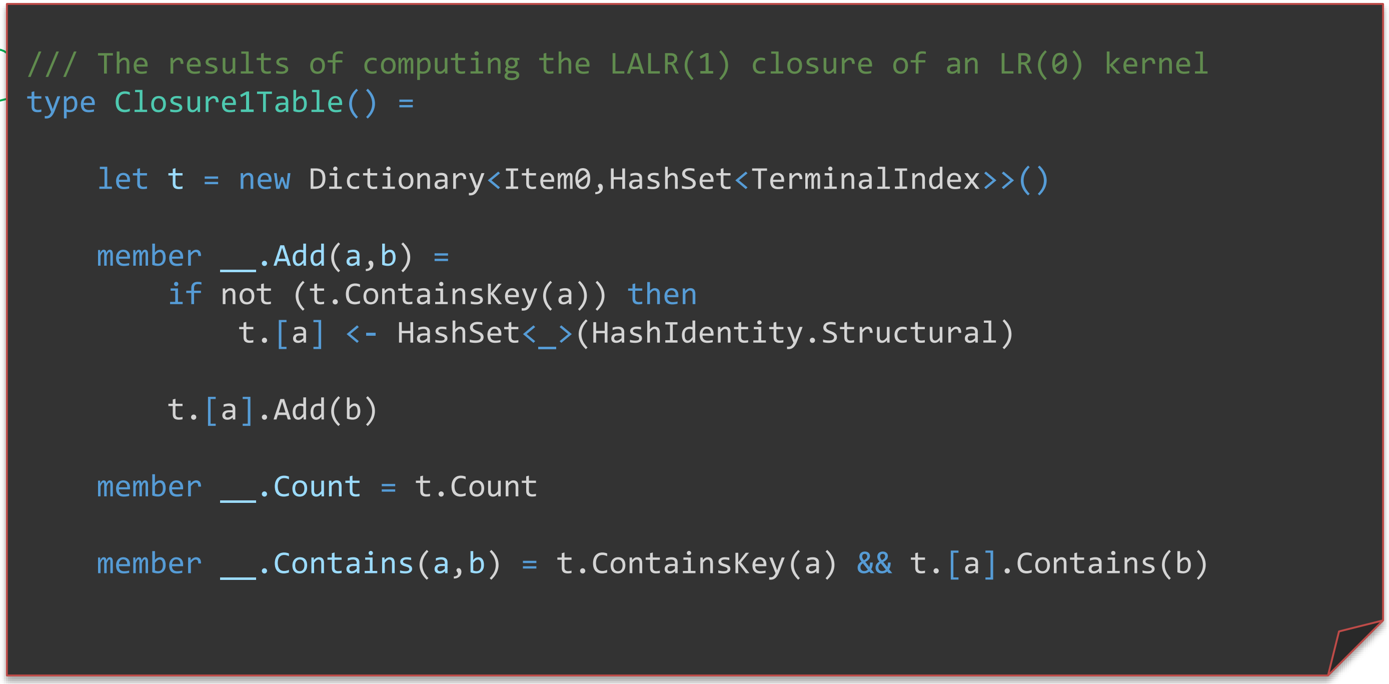
# Good mutation

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
```

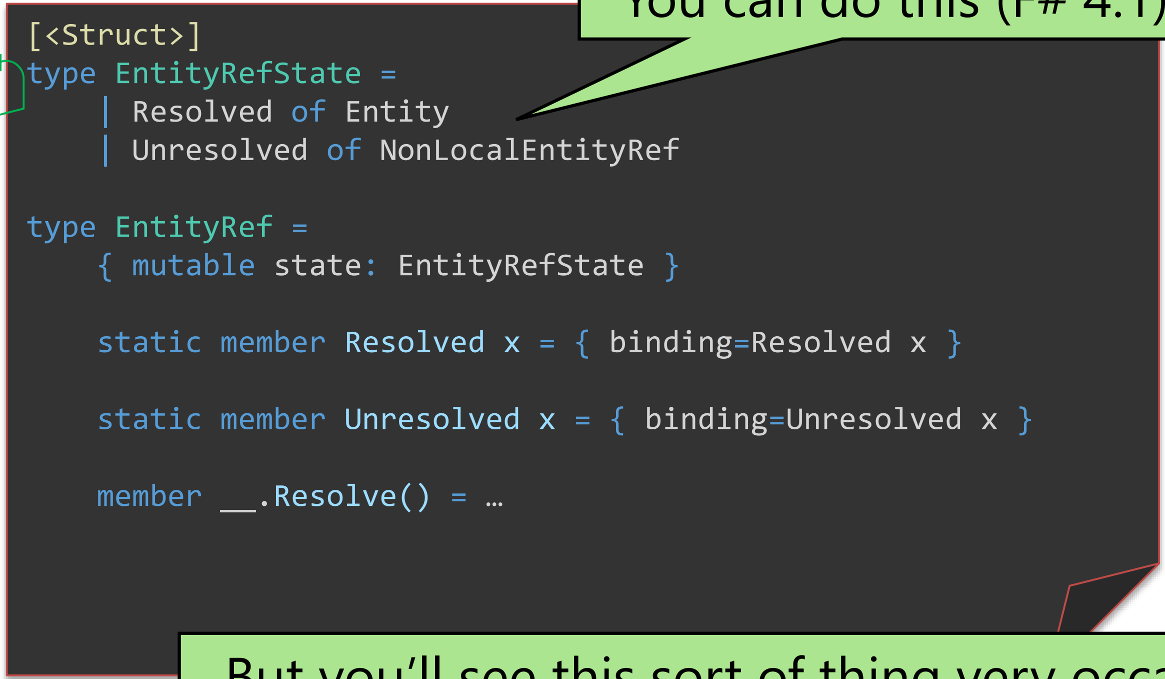
# null

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

However I have used it for

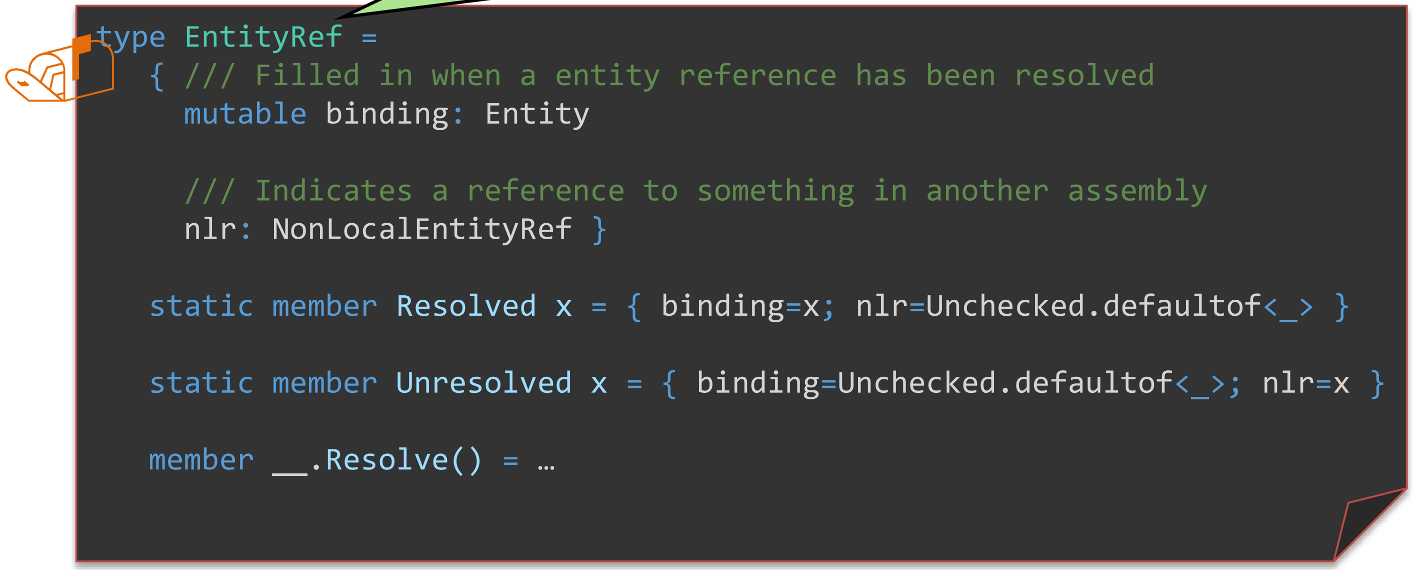
- Compact memory representations
- Manual implementations of mutating fixups
- Avoiding one indirection for an option type

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



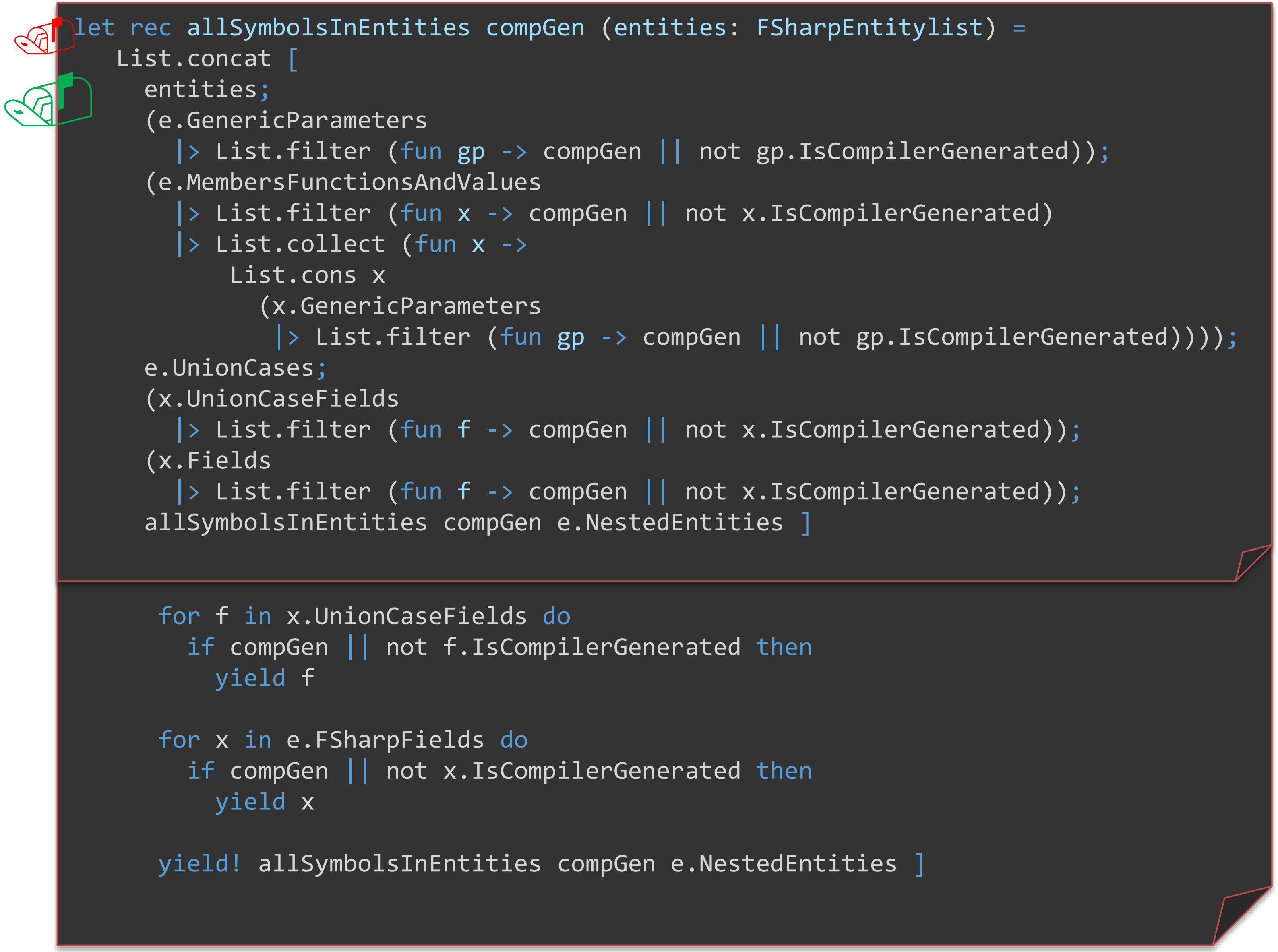
```
type EntityRef =
   { /// Filled in when a entity reference has been resolved
     mutable binding: Entity

     /// 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() = …
```

# Code I love: Computation expressions

# seq { … }, [ … ], [| … |]

- Many examples, almost every page of code

- Alternative is Seq.append

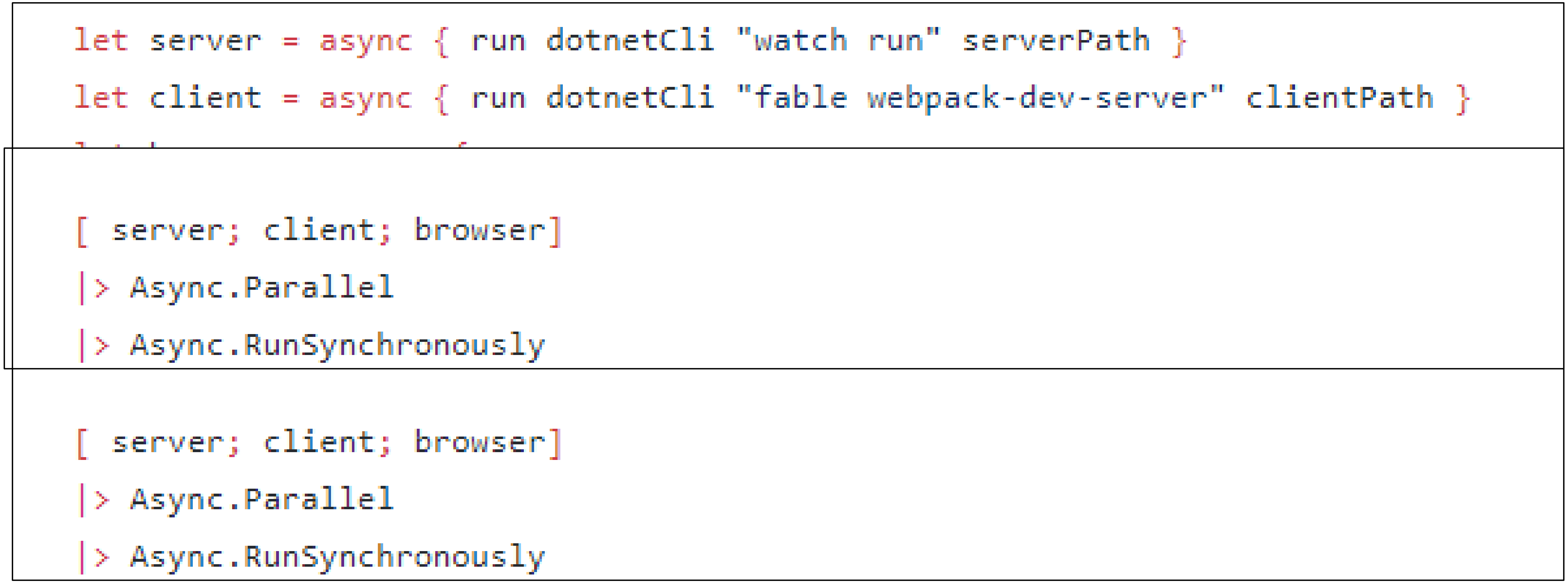


```
let rec allSymbolsInEntities compGen (entities: FSharpEntitylist) =
   List.concat [
     entities;
     (e.GenericParameters
       |> List.filter (fun gp -> compGen || not gp.IsCompilerGenerated));
     (e.MembersFunctionsAndValues
       |> List.filter (fun x -> compGen || not x.IsCompilerGenerated)
       |> 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 { … }

- F# already supports async sequences, it's a library
- I love this style of "reactive" code.
- "asynchronous pull" (AsyncSeq) v. "synchronous push" (IObservable)
- No inversion of control, you think in a "forward" way
- Makes a lovely compositional animation language

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

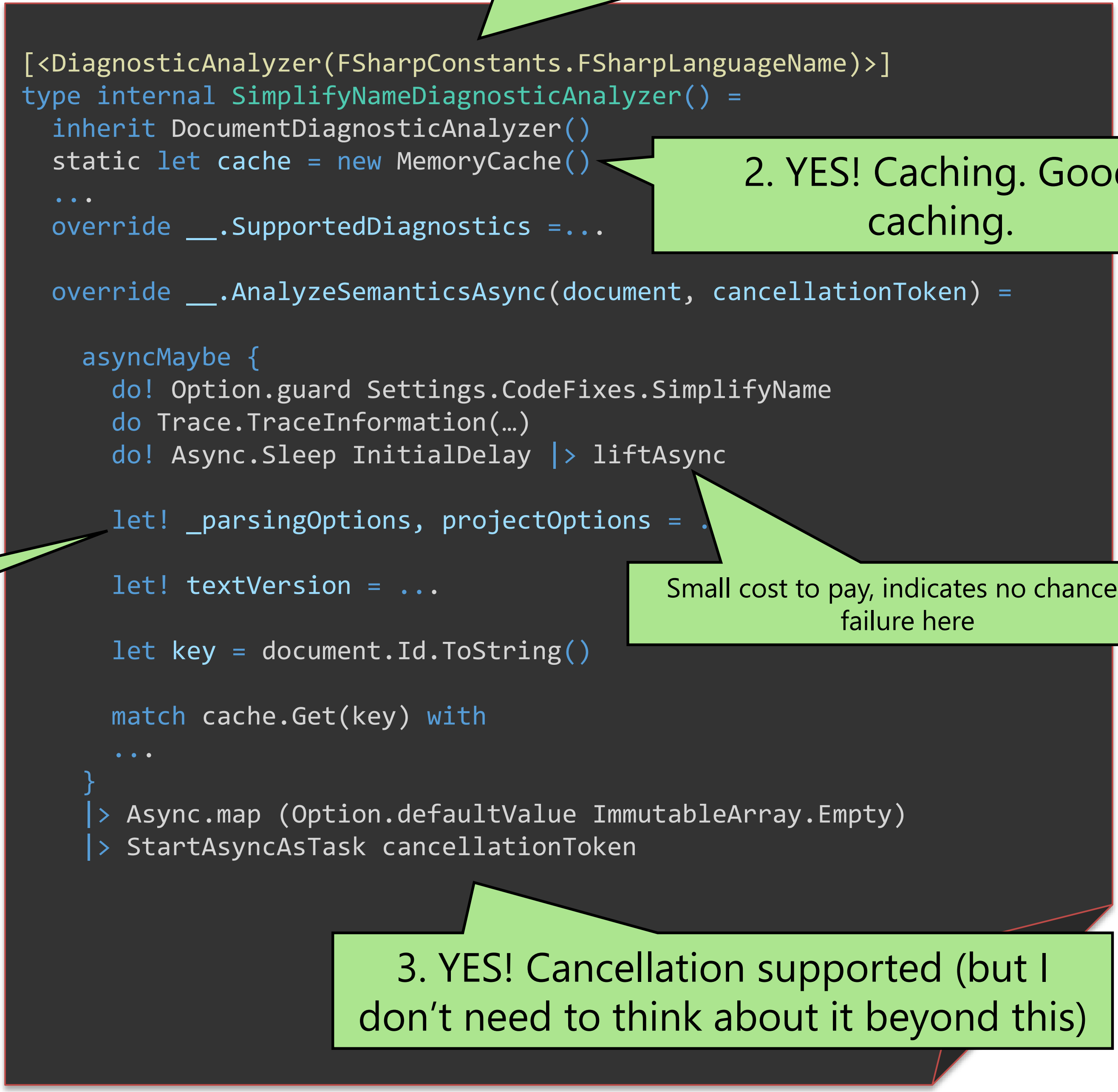
```
let intervalMs (periodMs:int) = asyncSeq {
    yield DateTime.UtcNow
    while true do
        do! Async.Sleep periodMs
        yield DateTime.UtcNow
}
```

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

# asyncMaybe { … }

- I absolutely love the uses of this CE in the FSharp.Editor implementation by Vasily Kirichenko
- This helps makes some of the clearest, most declarative, most robust editor implementation code I know

1. OK, this implements a C# framework abstraction



```
[<DiagnosticAnalyzer(FSharpConstants.FSharpLanguageName)>]
type internal SimplifyNameDiagnosticAnalyzer() =
  inherit DocumentDiagnosticAnalyzer()
  static let cache = new MemoryCache()
  ...
  override __.SupportedDiagnostics =...

  override __.AnalyzeSemanticsAsync(document, cancellationToken) =

    asyncMaybe {
      do! Option.guard Settings.CodeFixes.SimplifyName
      do Trace.TraceInformation(…)
      do! Async.Sleep InitialDelay |> liftAsync

      let! _parsingOptions, projectOptions = .

      let! textVersion = ...

      let key = document.Id.ToString()

      match cache.Get(key) with
      ...
    }
    |> Async.map (Option.defaultValue ImmutableArray.Empty)
    |> StartAsyncAsTask cancellationToken
```

2. YES! Caching. Good caching.

Small cost to pay, indicates no chance of failure here

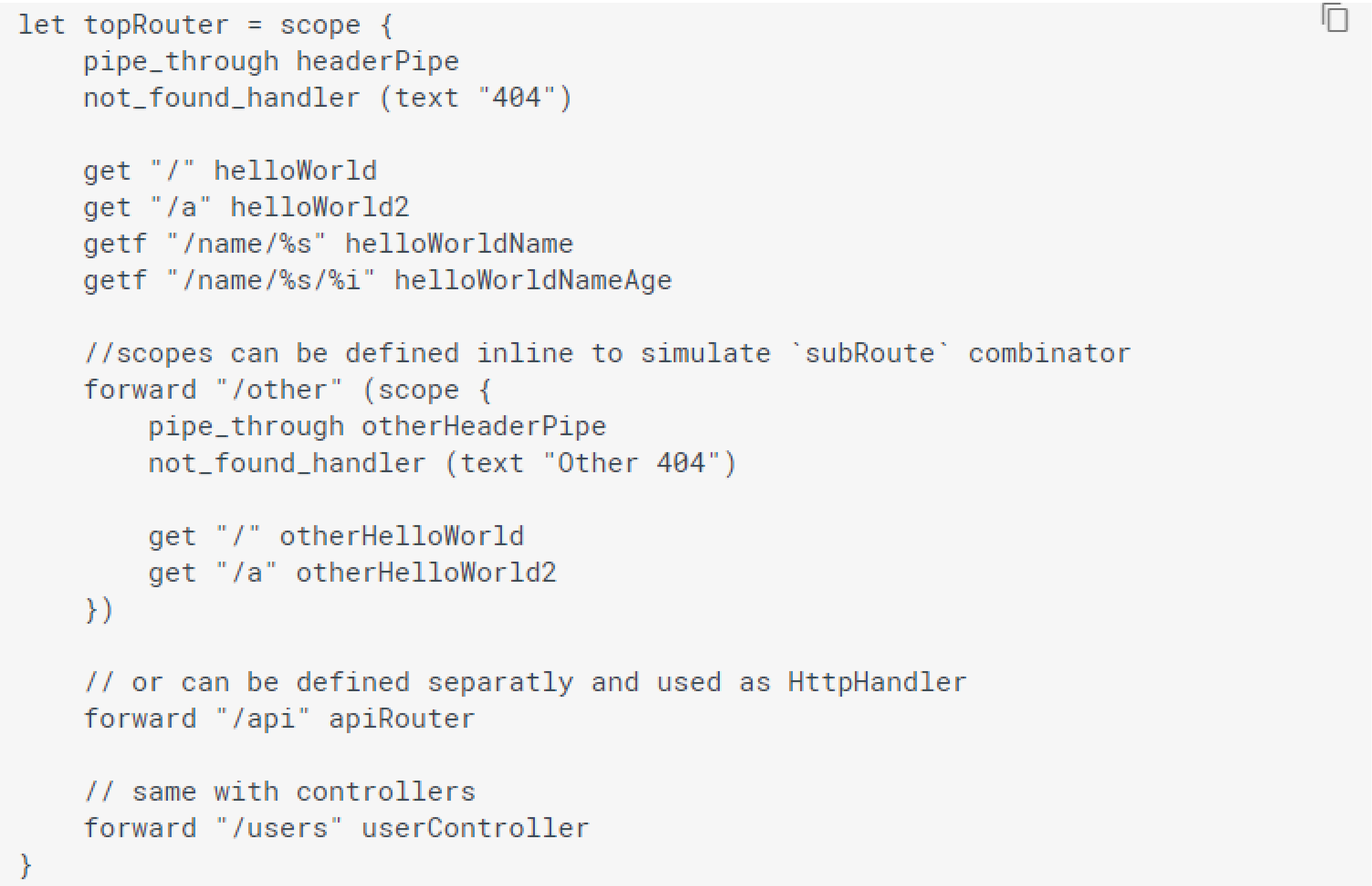
4. YES! clear failure/stop points

3. YES! Cancellation supported (but I don’t need to think about it beyond this)

# freyaMachine { … }, Saturn scope { … }

- Composition languages for web server components

- Saturn scope { … } implements HttpHandler in ASP.NET Core/Giraffe



```
let topRouter = scope {
    pipe_through headerPipe
    not_found_handler (text "404")

    get "/" helloWorld
    get "/a" helloWorld2
    getf "/name/%s" helloWorldName
    getf "/name/%s/%i" helloWorldNameAge

    //scopes can be defined inline to simulate `subRoute` combinator
    forward "/other" (scope {
        pipe_through otherHeaderPipe
        not_found_handler (text "Other 404")

        get "/" otherHelloWorld
        get "/a" otherHelloWorld2
    })

    // or can be defined separatly and used as HttpHandler
    forward "/api" apiRouter

    // same with controllers
    forward "/users" userController
}
```

# In Closing

# 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

# Historical archaeology: Some code that inspired early F#

- Forte FL – An Intel internal toolchain for formal verification using a functional language. Reinforced how powerful functional programming is for practical symbolic manipulation

- HOL Lite – A brilliant development of a theorem prover using Caml Light. Taught me the immense practical power of the core ML language.

- SPiM – Stochastic Pi Machine, plus user interface elements. Beautiful, simple core,

- Static Driver Verifier – verification toolchain for Windows drivers. Taught me many good things including how bad some FP-only features can be in practice (e.g. unencapsulated mutable records)

# Summary

- F# is full of delightful moments

- Constructs need to be used with moderation

- Functional, Object and other features can be misused

- Please share experiences and help improve coding standards

Thanks!

# Thanks! Questions?