Introduction to Functional Programming using F#

Chris Lomont, PhD

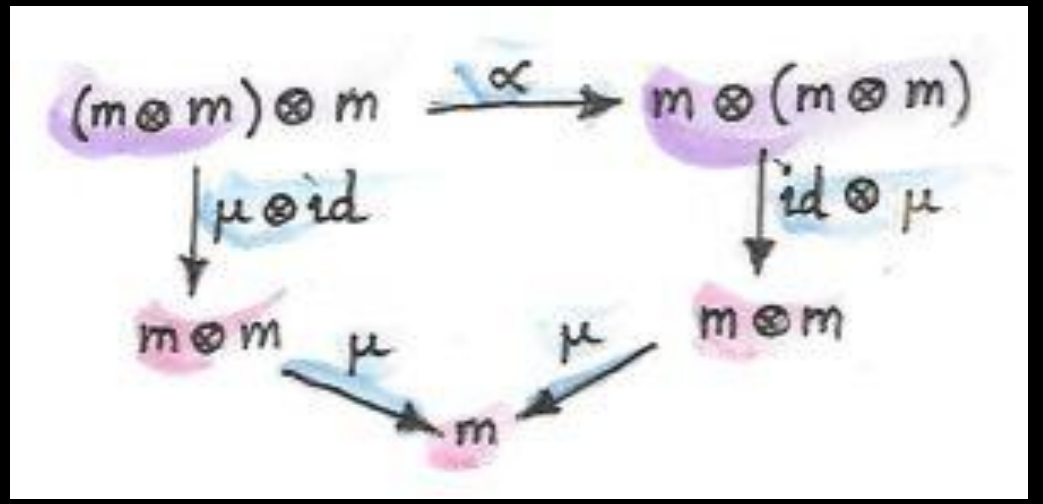
Senior Principal Engineer

Logikos Inc. Fort Wayne, IN

http://www.logikos.com

clomont@logikos.com

chris@lomont.org



About me

- Programming since 4th grade (TI-55!)
  - Soon moved to TRS-80 Color Computer
- Studied math, CS, physics, economics
- Worked in too many areas to cover
  - Some: 3D, games, science, DoD, security, algorithms

- Co-founder of Hypnocube, LLC
- Functional Programming since Mathematica 1.0 in 1988
- Works for Logikos, Inc, Fort Wayne, IN

About Logikos

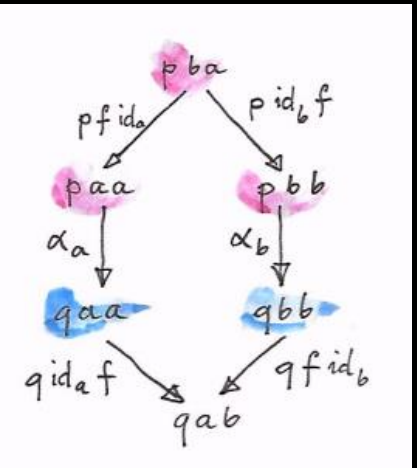
- Software Services
  - Fort Wayne, IN
  - Founded 1978
- Areas
  - Embedded
  - Desktop
  - Web Applications and Design
  - Mobile Applications
  - Software Testing

- Clients
  - Automotive
  - Commercial Services
  - Consumer Electronics
  - Govt and DoD
  - Medical
  - Retail
  - Others

Talk Purpose

- Illustrate the power of functional programming

- Functional programming languages claim to be
  - ***quicker*** *for development*
  - ***less errors***
  - ***smaller*** *code*
  - *easily* ***maintainable***



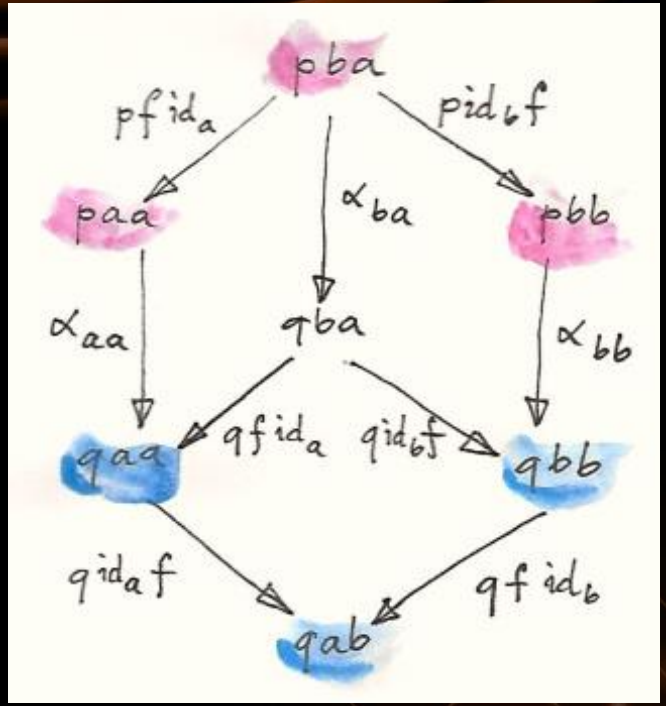
- Illustrate functional programming features via F# .NET

Talk Overview

I. Functional programming overview

II. F# syntax and examples

III. Mind-bending topics

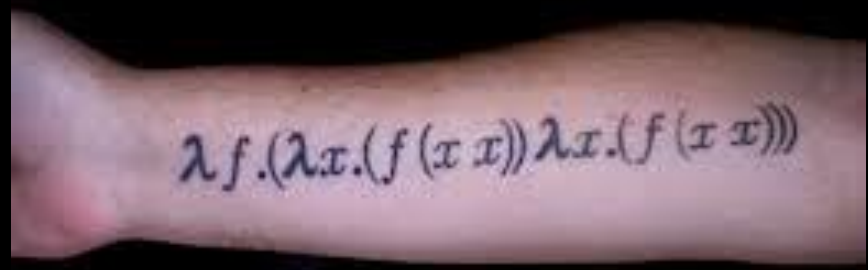


Functional Overview

Theoretical Foundations

- The ***Lambda Calculus***
  - Alonzo Church (1903-1995), 1930 **"What is computable?"**
  - Simplifies math expressions into lambda expressions
  - Church-Turing Thesis
    - Any computation can be done Turing machine
    - Any computation can be done with general recursive functions (in lambda sense)

- ***Lambda functions*** (named from the λ)
  - $\lambda x.x + 1$
  - Means take x as input, output x+1
  - Called pure functions
  - Have no state
  - Also known as ***anonymous functions***



A very famous lambda function defining recursion

Functional Programming at a High level

Functional	OO, Imperative
Stateless functions	Functions mutable
No side effects	Side effects everywhere
Thread safe	Hard to make thread safe
Scalable	Harder to scale
Easy to test	Harder to test
Composable	Not designed to be composable
Higher order functions (functions as data)	Functions and data separate
Immutable data (persistent data structures)	Mutable items default

- Evolved from ***pure functions***
- Goal: separate pure and impure parts of program

Example: Roslyn C# Compiler

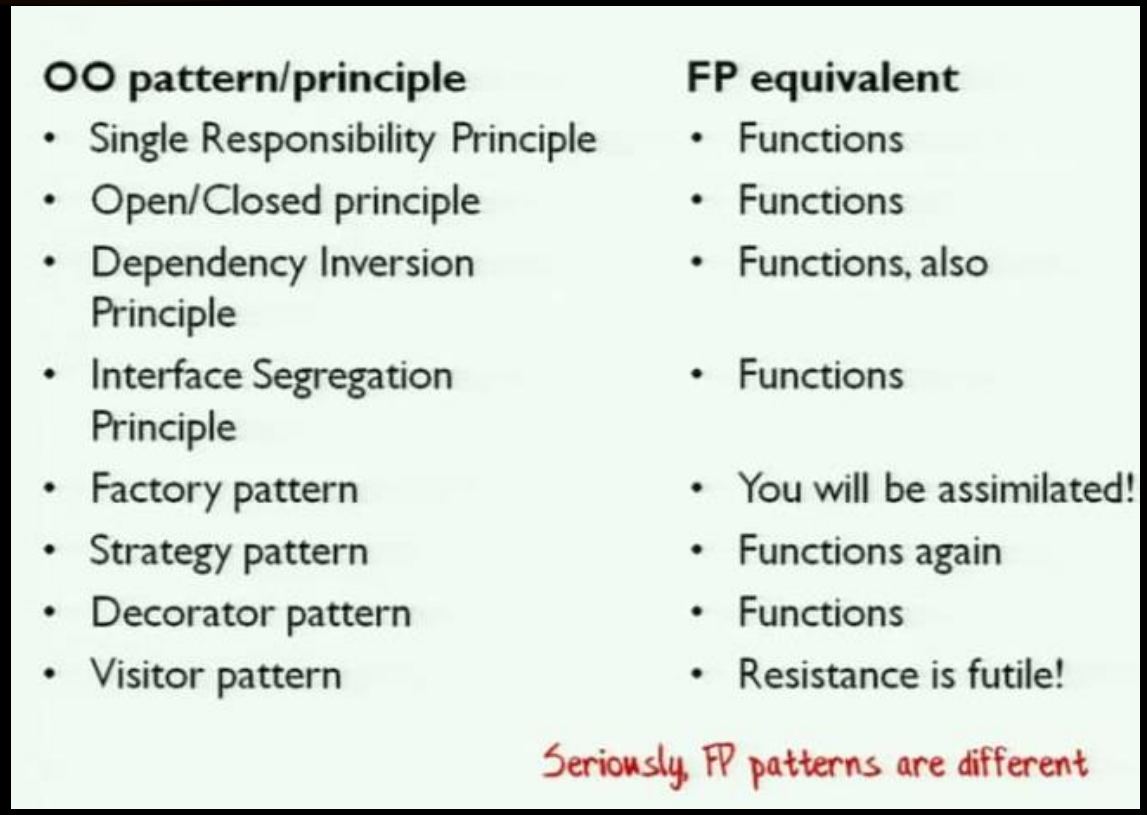
- C# compiler, written in C#
  - https://github.com/dotnet/roslyn
- Provides compiler as a service
- Main data structures are functional programming based
- Syntax Tree (red-green tree, from maker board colors used)
  - Immutable
  - Persistent – allows keeping most of tree when edit made to buffer
    - https://blogs.msdn.microsoft.com/ericlippert/2012/06/08/persistence-facades-and-roslyns-red-green-trees/

Example: Design Patterns

- Gang of Four – 23 patterns

- "Patterns are simply workarounds for C++ not being expressive" - Paul Graham ,2002

- Peter Norvig demonstrated 16 of the 23 are simplified or eliminated in Lisp or Dylan

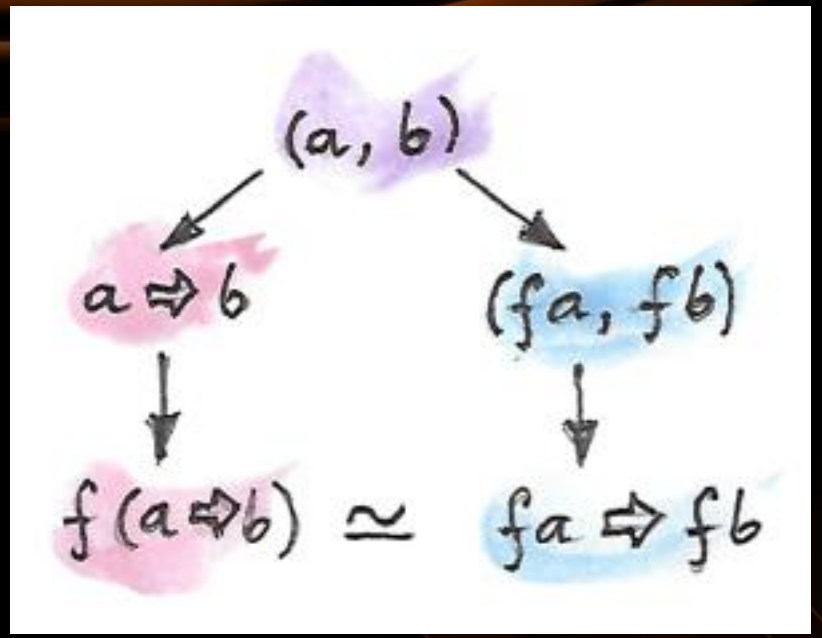


Scott Wlaschin

Functional Programming Languages

- Functional languages
  - Lisp, Haskell, SML, Clojure, Scala, Erlang, F#
- Functional with OO features
  - OCaml, F#, Scala
- Imperative languages with functional features
  - Python, Ruby, C#, Java

"Modern functional languages are nontrivial embellishments of the lambda calculus"
– Paul Hudak (1989)



F# Syntax and Features

What is F#?

- Statically typed ***functional first*** language
- Took OCaml and added C# and .NET glue
- Mixed Language
  - Supports Functional, OO, Imperative
- F# is to C# (CLR) what Scala is to Java (JVM)

```
module List =
   let rec insertions x = function
   | [] -> [[x]]
   | (y :: ys) as l -> (x::l)::(List.map (fun x -> y::x) (insertions x ys))

let rec permutations = function
   | [] -> seq [ [] ]
   | x :: xs -> Seq.concat (Seq.map (insertions x) (permutations xs))
```

F# History

- Created by Cambridge MS Research, led by Don Syme (Benevolent Dictator for Life)
  - Theorem proving => ML (Meta Language)
  - ML => CAML
  - CAML => OCaml
  - **OCaml + C# + .NET + ??? => F#**

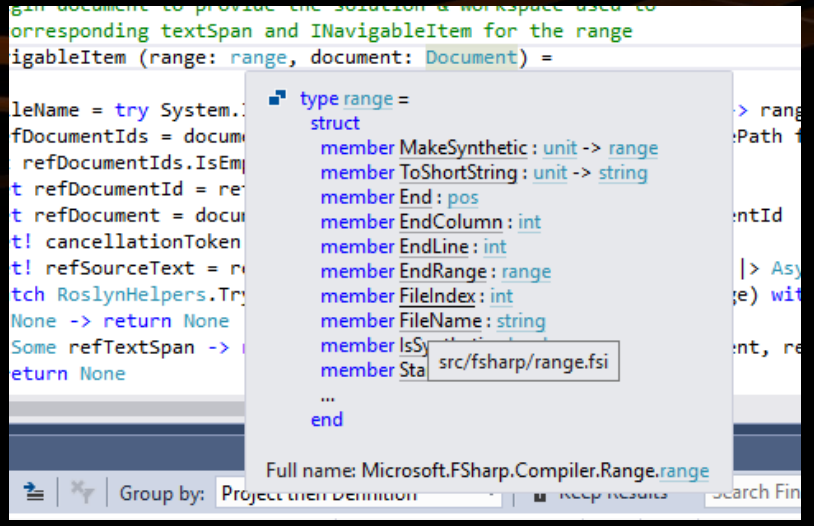
- Versions
  - 1.0 VS2005, separate download
    - Functional, types, modules
  - 2.0 VS2010, built in
    - Active patterns, Units of Measure, async
  - 3.0 VS2012, built in
    - Type Providers
  - 4.0 VS2015, built in
    - printf interpolation
  - 4.1 VS2017, built in
    - Struct tuples, numeric underscores

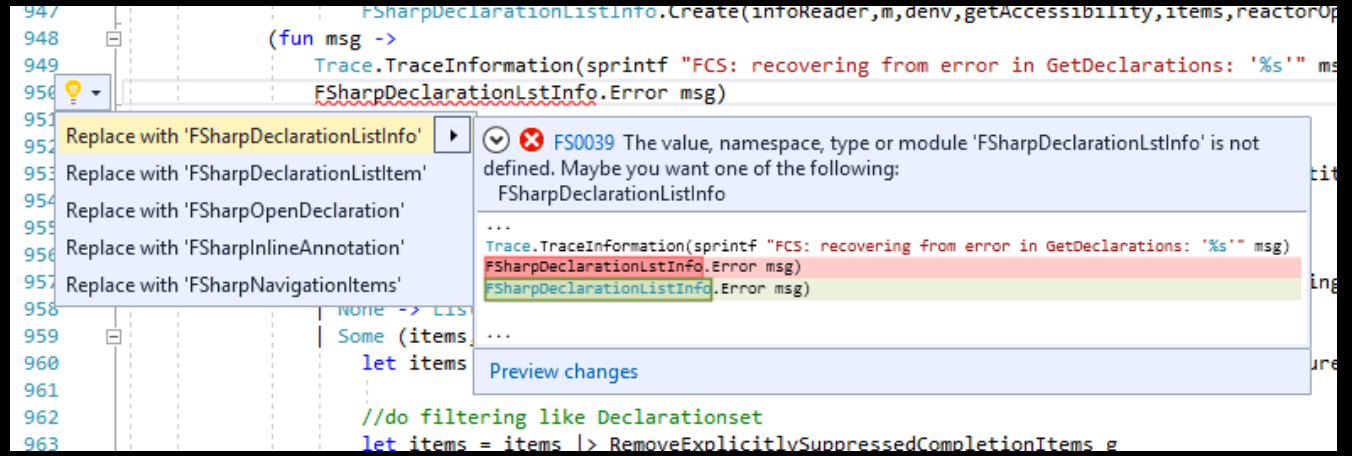
F# (and functional) Features Migrating to C# and other languages

- F# 1.0 ***implicitly typed locals*** (var in C# 3.0, auto in C++ 11, others)
- F# 1.0 ***lambda functions*** added to C# v2.0, C++ 11, PHP 5.3.0, Delphi 2009
- F# 2007 ***async workflows*** added to C# 2012
  - Migrated to many languages since over since
- ***Functionals***
- ***LINQ***
- ***Expression valued items***
- ***Pattern matching***
- ***Tuple construction/deconstruction*** in C # 7
- ***Expression bodied members***
- ***Exception filters***
- ***Auto property initializers***
- ***Task Parallel Library***
- ***Non-nullable types*** (scheduled for C# 8.0?)

Why Use F#?

- F# benefits
  - Works well with .NET , easy to integrate
  - Modern IDE
    - Debugger support
    - Intellisense
  - F# REPL window in VS
- Power of multicore





C# vs F# Example 1

- C#

```
namespace Math
{
    public static class Helpers
    {
        public static int Add( int x, int y)
        {
            return x + y;
        }
    }
}
```

- F#

```
let add x y = x + y
```

C# vs F# Example 2 : Map / Reduce

```csharp
// map / reduce C#
public static
IEnumerable<R> Map<T,R>
(this IEnumerable<T> xs, Func<T,R> f)
{
    foreach (var x in xs)
        yield return f(x);
}

public static R Reduce<T,r>
(this IEnumerable<T> xs, R init, Func<R,T,R> f)
{
    var current = init;
    foreach (var x in xs)
        current = f(furrent,x);
    return current;
}
```

```fsharp
// map / reduce F#
let map f xs = seq {
    for x in xs do
        yield f x
    }

let reduce f init items =
    let mutable current = init
    for item in items do
        current <- f current item
    current
```

F# Syntax

- Comments

```
(* comment between *)
// comment to end of line
/// XML doc comments here
```

- Variables
  - type inference
  - immutable

```
let result = 1 + 8 // auto type inference
let s = "bob"
let pi = 3.14159265

let b : byte = 123 // can specify type

let a, b, c = 1.0, "text" , result*4
```

Lists and Arrays

- Lists
  - Singly linked
  - Immutable
- Arrays
  - Fixed size
  - Mutable

```
let list1 = [ "a"; "b" ]
let list2 = "c" :: list1    // :: is prepending
let list3 = list1 @ list2   // @ is concat
let list4 = []              // empty list
let list5 = 2 :: [3; 5; 7; 11] @ [13; 17; 19; 23]
```

```
let array1 = [| "a"; "b" |]
let first = array1.[0] // Indexed using dot
// slicing
let tenSquares = [| for i in 1..10 -> (i,i*i) |]
let firstHalf = tenSquares.[..5]
let secondHalf = tenSquares.[6..]
let mid = tenSquares.[2..6]
```

Sequences

- Sequences
  - Lazy
  - Unbounded
  - Enumerable

```
let seq1 = {1..10}      // seq[1;2;3;...;10]
let seq2 =
    seq {
        yield 1         // adds element
        yield! [5..10] // adds subsequence
    }
```

Range and comprehensions

- Range
  - Start, stop
  - Step size

```
let as = [ 5..8 ]    // [5;6;7;8]

let xs = [ 1..2..9 ] // [1;3;5;7;9]

let ys = [| for i in 0..4 -> 2 * i + 1 |]

let zs = List.init 5 (fun i -> 2 * i + 1)
```

- Comprehension
  - Construct list from function

Tuples

- Containers for a few things
- Typed groups of values
- Trivial to make and deconstruct

```
// Tuple construction
let wizard = ("Merlin",87)
// type is string * int

// deconstruction
let (name,age) = wizard
```

Records

- Named type
- Sealed
- Immutable
- Patterns
- Structural ==

```
// Declare a record type
type Person = { Name : string; Age : int }

// Create a value via record expression
let paul = { Name = "Paul"; Age = 28 }

// 'Copy and update' record expression
let paulsTwin = { paul with Name = "Jim" }

let isPaul person =
    match person with
    | { Name = "Paul" } -> true
    | _ -> false
```

Functions

- Are values
- Type inference
- Composable

```
// functions return the last item in the function
let negate x = x * -1
let square x = x * x

// functions have types
let add x y = x + y // type int -> int -> int
let add3 = add 3    // type int -> int

let sumSquares x y =
    let sq z = z*z  // local functions
    sq x + sq y
```

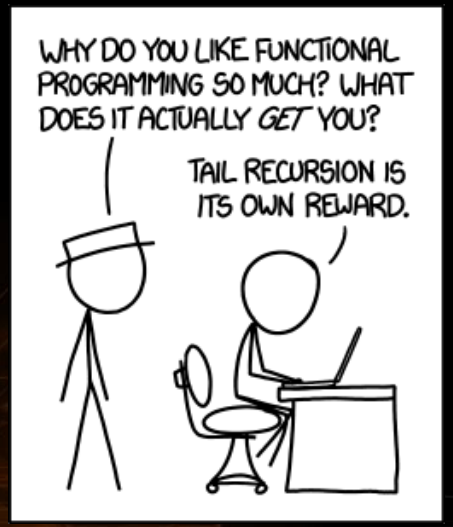
Recursive functions

- Tail recursion

```
let rec factorial x =
    if x < 1 then 1
    else x * factorial (x - 1)
```

- List example

```
let rec sum list =  // decompose (::) operator
    match list with // pattern matching
    | [] -> 0
    | x :: xs -> x + sum xs
```



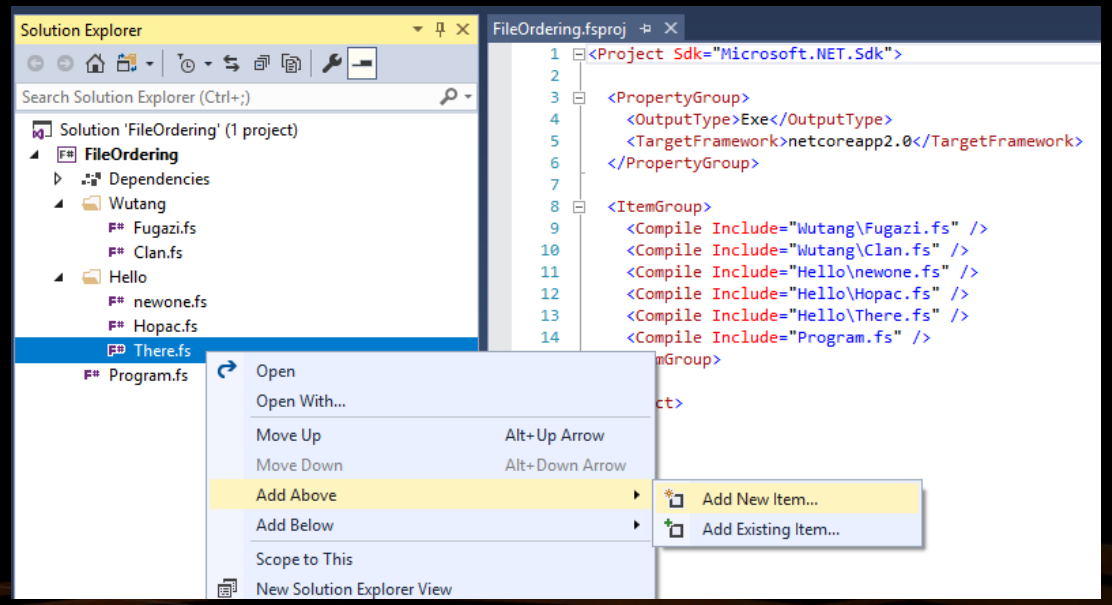
XKCD #1270

Type inference

- Generic
- Automagic

```
let pack a b = (a, b)
// type 'a -> 'b -> 'a * 'b
```

- File order in compilation important
  - Deduces types
  - No circular dependencies



Options

- option > nulls

```
// Built in 'option' widely used
type 'T option =
    | None
    | Some of 'T
```

- Prevents ignoring bad return values

- Compiler forces pattern matching to cover every case

Pattern Matching

- Very powerful
  - Switch statement on steroids
- Complete! Cannot ignore None in an Option

```
let checkList lst =
    match lst with
    | []              -> printfn "Empty"
    | [e]             -> printfn "One element: %O" e1
    | 1 :: xs         -> printfn "Start with 1"
    | _ :: x :: xs    -> printfn "Second element %O" x
    | _               -> () // do nothing
```

Active Patterns

- Extend matching programatically

```
(* parameterized active patterns *)
let (|DivisibleBy|_|) by n =
    if n % by = 0 then Some DivisibleBy else None

let fizzBuzz = function
    | DivisibleBy 3 & DivisibleBy 5 -> "FizzBuzz"
    | DivisibleBy 3 -> "Fizz"
    | DivisibleBy 5 -> "Buzz"
    | i -> string i
```

Discriminated Unions

- Disjoint unions

```
type Tree<'T> =
    | Tree of Tree<'T> * 'T * Tree<‘T>
    | Leaf of 'T

let rec depth tree =
    match tree with
    | Tree(l, _, r) -> 1 + max (depth l) (depth r)
    | Leaf(_) -> 1

let rec size = function
    | Tree(l, _, r) -> 1 + size l + size r
    | Leaf(_) -> 1
```

Pipeline, partial application, composition

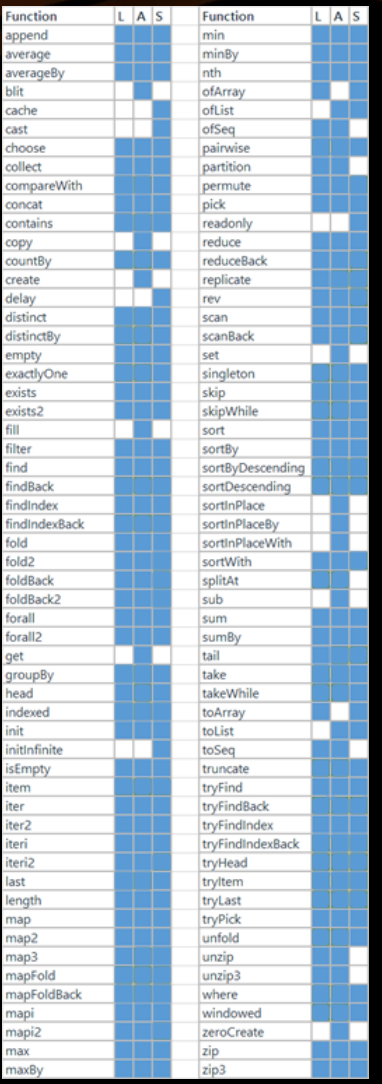
- Composes functions

```
// pipe |>
let ``square, negate, then print`` x =
    x |> square |> negate |> print

let sumOfLengths (xs : string []) =
    xs
    |> Array.map (fun s -> s.Length)
    |> Array.sum
```

Useful list, array, seq functions

- Rich set of (mostly) orthogonal helpers
  - find, map, filter, partition, zip, reduce, exists, …

```
(* list, seq, array functions *)
let prod  = [1..10] |> List.fold (*) 1
let vals  = {1..10} |> Seq.map (fun x->x*x+1)
let odds  = [1..10] |> List.filter (fun x-> (x%1) = 0)
let (o,e) = [1..10] |> List.partition (fun x-> (x%1) = 0)
let all   = List.zip odds evens
```



Function	L	A	S	Function	L	A	S
append	■	■	■	min	■	■	■
average	■	■	■	minBy	■	■	■
averageBy	■	■	■	nth	■	■	■
blit		■		ofArray	■		■
cache			■	ofList		■	■
cast			■	ofSeq	■	■	
choose	■	■	■	pairwise	■	■	■
collect	■	■	■	partition	■	■	
compareWith	■	■	■	permute	■	■	■
concat	■	■	■	pick	■	■	■
contains	■	■	■	readonly			■
copy		■		reduce	■	■	■
countBy	■	■	■	reduceBack	■	■	■
create		■		replicate	■	■	■
delay			■	rev	■	■	■
distinct	■	■	■	scan	■	■	■
distinctBy	■	■	■	scanBack	■	■	■
empty	■	■	■	set		■	
exactlyOne	■	■	■	singleton	■	■	■
exists	■	■	■	skip	■	■	■
exists2	■	■	■	skipWhile	■	■	■
fill		■		sort	■	■	■
filter	■	■	■	sortBy	■	■	■
find	■	■	■	sortByDescending	■	■	■
findBack	■	■	■	sortDescending	■	■	■
findIndex	■	■	■	sortInPlace		■	
findIndexBack	■	■	■	sortInPlaceBy		■	
fold	■	■	■	sortInPlaceWith		■	
fold2	■	■	■	sortWith	■	■	■
foldBack	■	■	■	splitAt	■	■	
foldBack2	■	■	■	sub		■	
forall	■	■	■	sum	■	■	■
forall2	■	■	■	sumBy	■	■	■
get		■		tail	■	■	■
groupBy	■	■	■	take	■	■	■
head	■	■	■	takeWhile	■	■	■
indexed	■	■	■	toArray	■		■
init	■	■	■	toList		■	■
initInfinite			■	toSeq	■	■	
isEmpty	■	■	■	truncate	■	■	■
item	■	■	■	tryFind	■	■	■
iter	■	■	■	tryFindBack	■	■	■
iter2	■	■	■	tryFindIndex	■	■	■
iteri	■	■	■	tryFindIndexBack	■	■	■
iteri2	■	■	■	tryHead	■	■	■
last	■	■	■	tryItem	■	■	■
length	■	■	■	tryLast	■	■	■
map	■	■	■	tryPick	■	■	■
map2	■	■	■	unfold	■	■	■
map3	■	■	■	unzip	■	■	
mapFold	■	■	■	unzip3	■	■	
mapFoldBack	■	■	■	where	■	■	■
mapi	■	■	■	windowed	■	■	■
mapi2	■	■	■	zeroCreate		■	
max	■	■	■	zip	■	■	■
maxBy	■	■	■	zip3	■	■	■

Conditionals

- if/then/else
- Don’t use ‘if’
  - Use ‘match’

```
// bad
let f x =
    if x = 1
    then "a"
    else "b"

// good
let f x =
    match x with
    | 1 -> "a"
    | _ -> "b"
```

Loops

- Loops

- Use recursion or functions instead

```
let weirdo =
    for i = 1 to 49 do
        printfn "%d" i
    let mutable j = 50
    while j <= 100 do
        printfn "%d" j
        j <- j + 1
```

Exceptions

- try/with/finally
- Exception filtering

```
let divide x y =
    try
        try
            (x+1) / y
        finally
            printf "this will always be printed"
    with
    | :? System.DivideByZeroException as ex ->
        printfn "%s" ex.Message; 0
```

Classes

- Inheritance

```
type Animal() =
    member __.Rest() = ()
type Dog() =
    inherit Animal()
    member __.Run() =
        base.Rest()
```

- Upcasting
  - Compile time
- Downcasting
  - Run time

```
// upcasting :>
let dog = Dog()
let animal = dog :> Animal
// downcasting :?>
let shouldBeADog = animal :?> Dog
```

Example : Quicksort

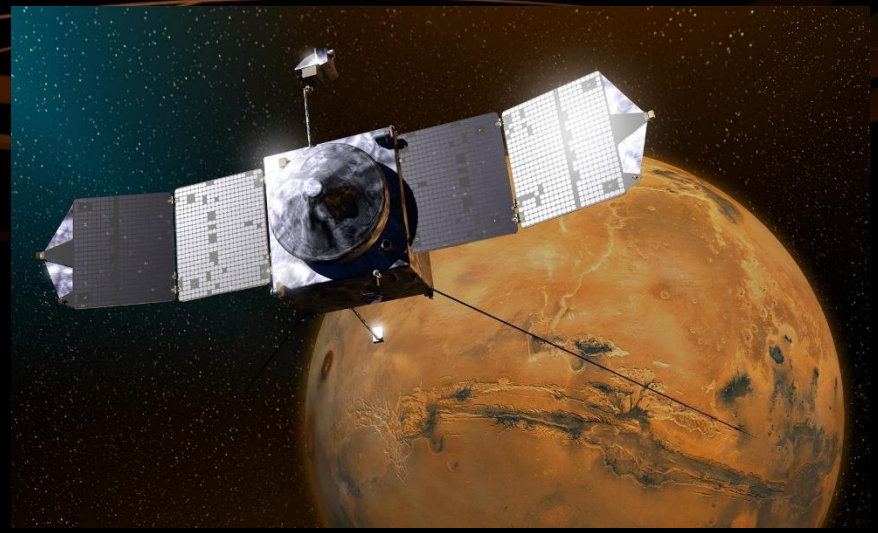
```
let rec sort(lst: int list) =
    match lst with
    | []     -> lst
    | [x]    -> lst
    | hd:tl -> let less,greater = List.partition ( (>=) hd ) tl
               [sort(less)] @ [hd] @ [sort(greater)]
```

F# Features : Type Providers

- Code generation
- Working cleanly with structured data automatically
- Hooks into the compiler and IDE
  - Intellisense, code errors
  - Static typing of items in the structure
- Takes a while to see the power
- Many existing
  - CSV, HTML, XML, JSON, HTTP, Semantic Web, World Bank, web services, data markets, much more

F# Features : Units of Measure

- Mars climate orbiter fail
  - Failed to translate English units into metric

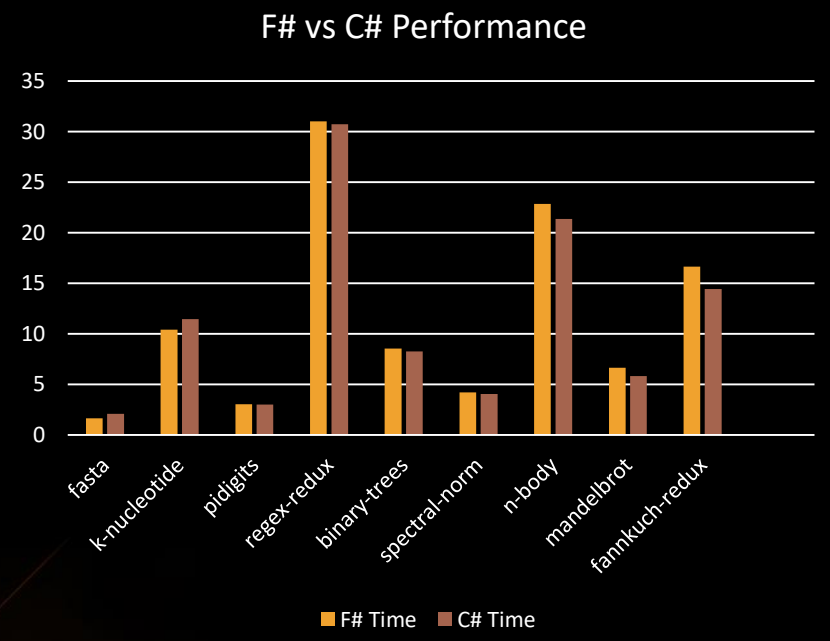


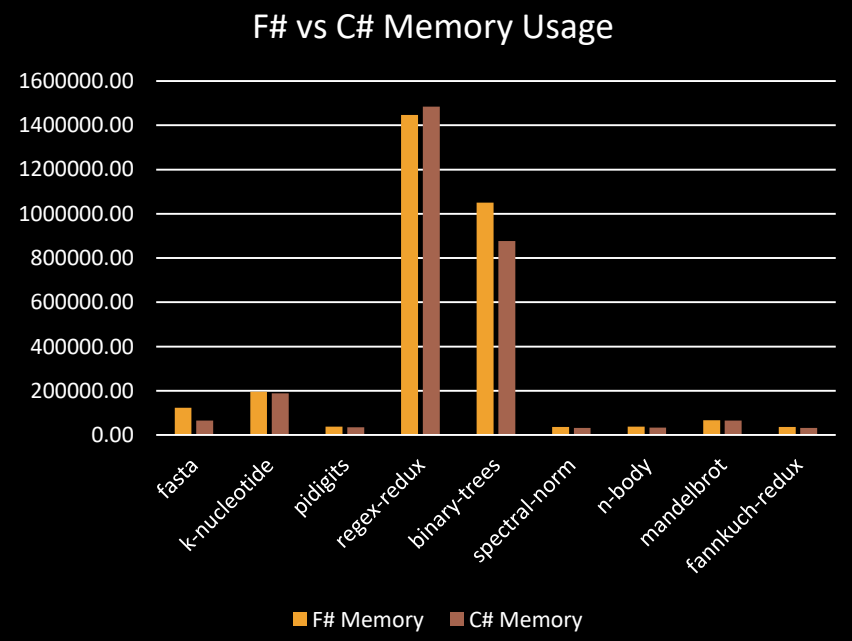
- Units of Measure
  - Enforces unit consistency

```
[<Measure>] type degC
[<Measure>] type degF
let convertDegCToF c =
c * 1.8<degF/degC> + 32.0<degF>

let f = convertDegCToF 0.0<degC>
// type float<degC> -> float<degF>
```

Performance

- F# code performs similarly to C#



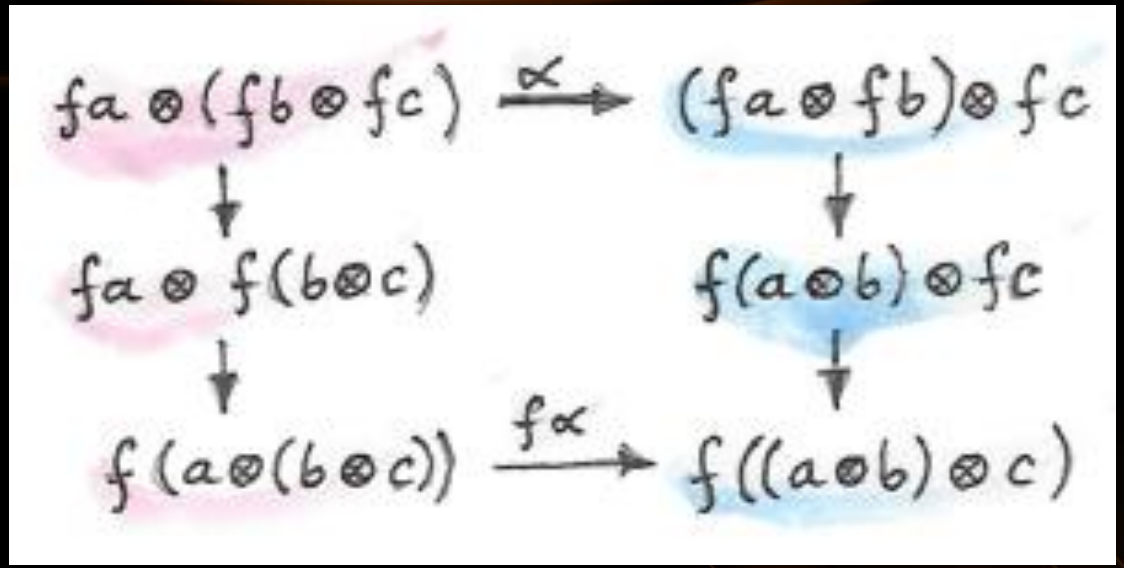


- From https://benchmarksgame.alioth.debian.org

Dos and Don’ts

- Don’t use mutable keyword
- Don’t use for loops
  - Use recursion instead
- Don’t use if then else
  - Use pattern matching instead
- Don’t use dot notation
  - Use little functions, pipes
- Don’t use classes
  - Use tuples, records, unions
- Don’t rely on debugger
  - Compiler finds more errors for you

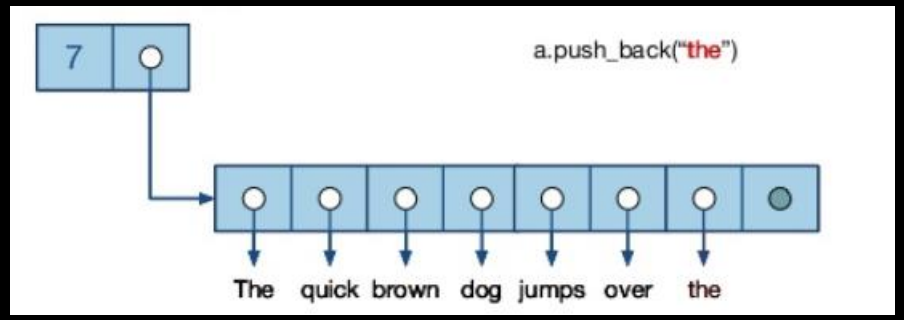
- Do create little types
  - Especially unions
- Do use list and seq types
  - Learn fold and map well!
- Eventually replace recursion with fold
- Do use pipe |>
- Do use composition >>
- Do develop incrementally, using interactive window

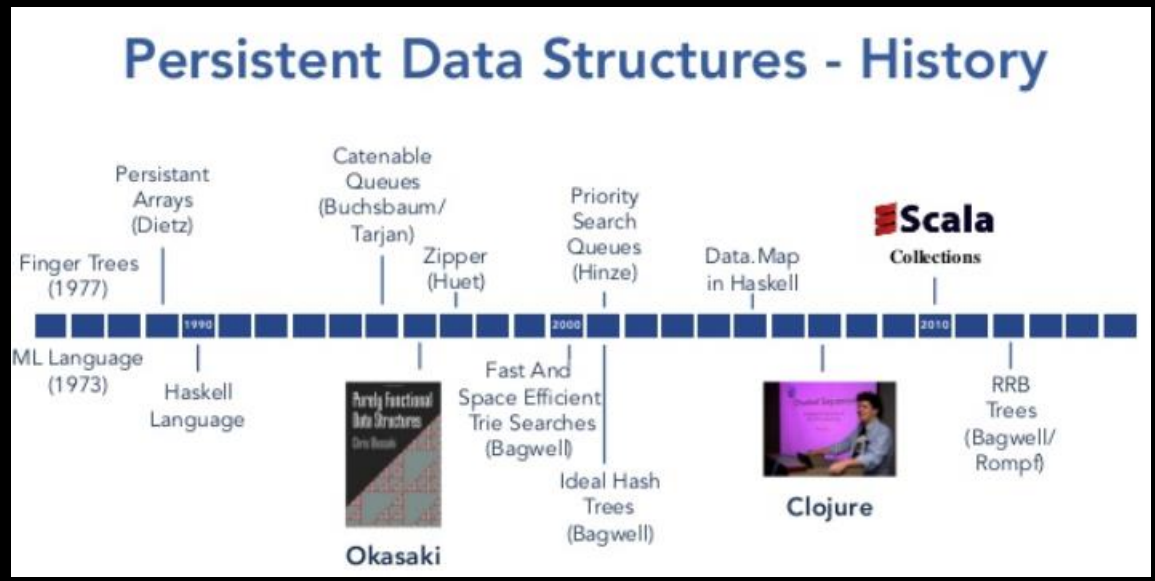


Mind Bending

Data structures

- How to make ***immutable*** lists? Trees?
- Example: appendable vector

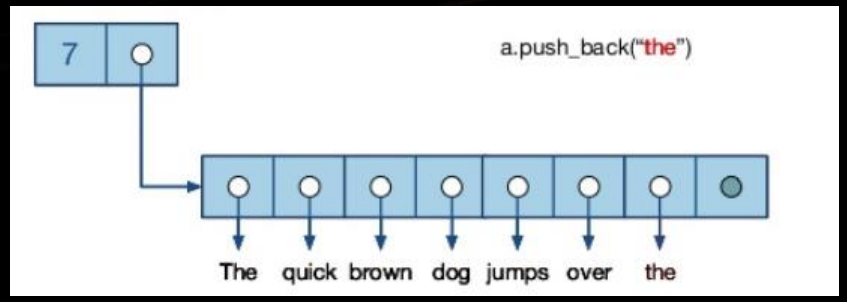




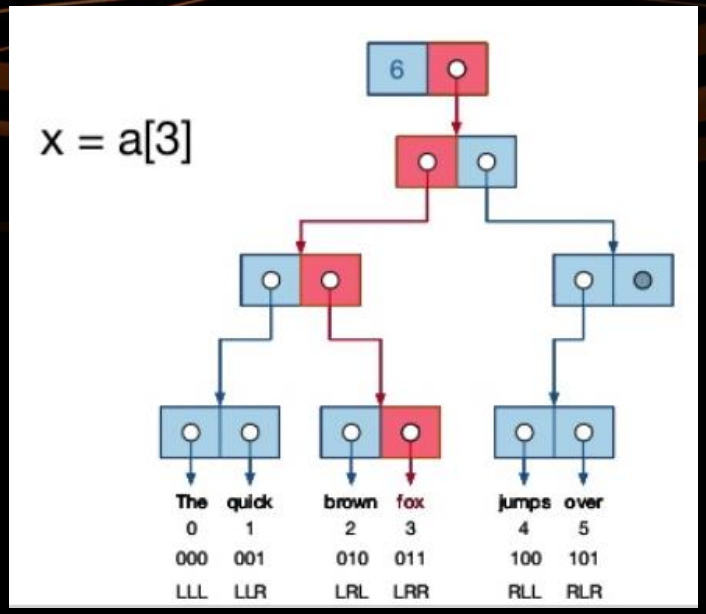
- Want to push back, want immutable!
- Idea: make a new one! (but slow, too much memory!)
- Solution: Wizardry!

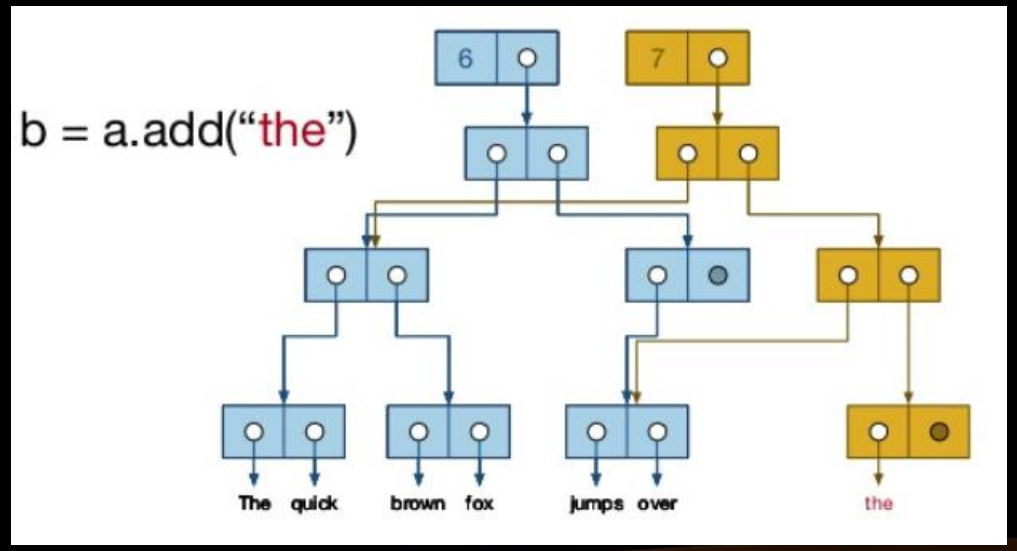
Data structures

- Replace vector with tree:



- But trees $O(\log n)$, not $O(1)$
- Use 32-way trees, then
  - $O(\log_{32} n) \sim O(1)$
  - More tricks to make faster
    - Local mutable, global immutable…
  - No locking, parallelizable





Currying

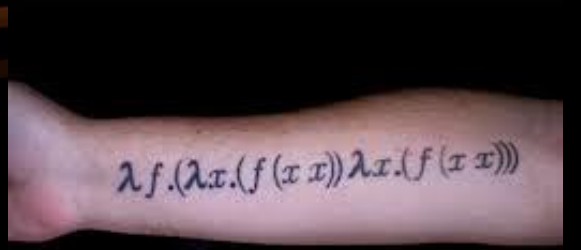
- Converts function of multiple arguments into function of one argument
- Then everything can be thought of as a function of one variable
- Named from Haskell Curry (1900-1982)

- $(x, y) \Rightarrow x + y$ curried into $x \Rightarrow (y \Rightarrow x + y)$

- Allows reusing pieces in a much more composable manner

Y-Combinator

- Functional that recurses any function
- Discovered by Haskell Curry



$$Y \coloneqq \lambda f.\left(\lambda x.\, f\,(x\ x)\right)\left(\lambda x.\, f\ (x\ x)\right)$$

$$\begin{aligned} Y\,g &= \lambda f.\left(\lambda x.\, f\,(x\ x)\right)\left(\lambda x.\, f\ (x\ x)\right) g && (1)\\ &= \left(\lambda x.\, g\,(x\ x)\right)\left(\lambda x.\, g\ (x\ x)\right) && (2)\\ &= g\left(\left(\lambda x.\, g\,(x\ x)\right)\left(\lambda x.\, g\,(x\ x)\right)\right)\\ &= g\,(Y\ g)\\ &= g\,(Y\ g) = g\big(g\,(Y\ g)\big) \ldots = g\,(\ldots g\,(Y\ g)\ \ldots) \end{aligned}$$

(1) apply function: f gets replaced with g

(2) apply first function to second arg: get $g\,(x\ x)$ with $x \Rightarrow \left(\lambda x.\, g\ (x\ x)\right)$

Let's you write a Y-Combinator function, that recurses on any function fed to it

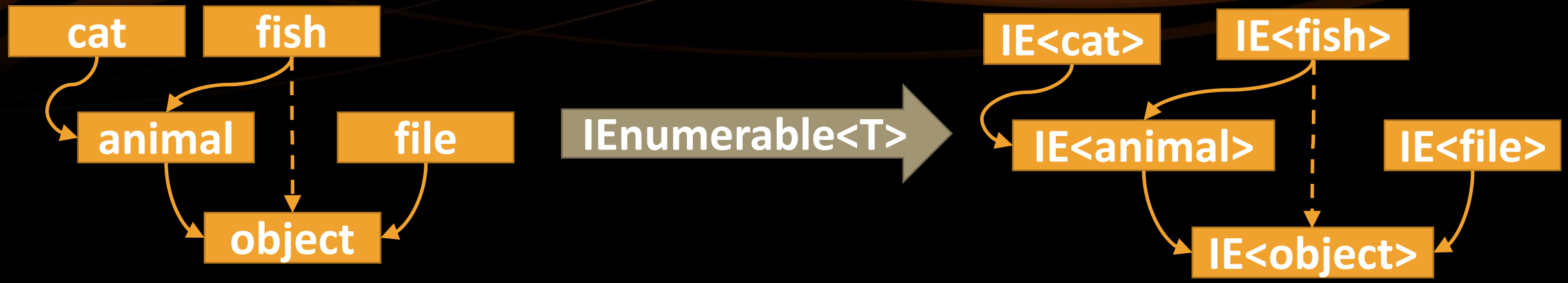
Contravariant/Covariant

- C# uses the concepts of ***contravariance*** and ***covariance***
- How parameters do up/down casting, how assignments work

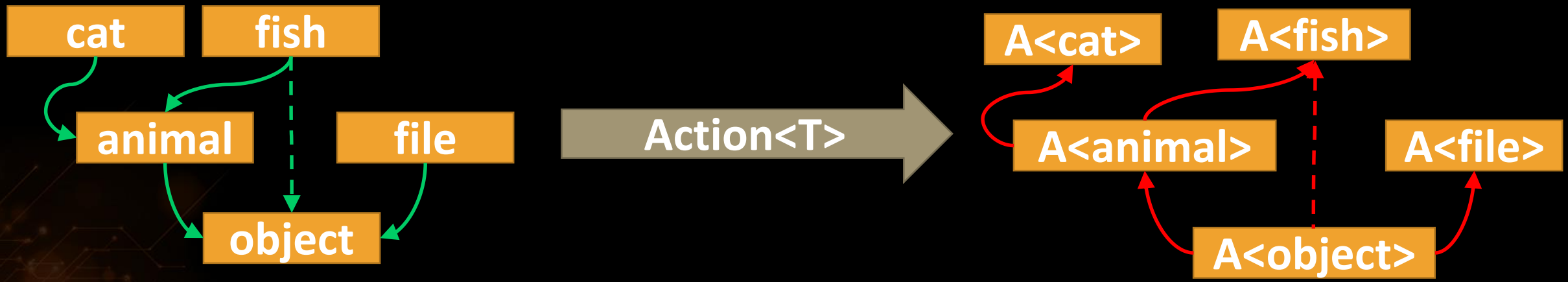
```
// type relations are an "arrow" : derived
from, assignable
// allowed, object <- string
object obj = "string"

// covariance preserves arrows
// Allowed: IE<obj> <- IE<string>
IEnumerable<object> objects = List<string>
// ERROR! Disallowed! Cannot do IE<obj> ->
IE<string>
IEnumerable<string> strings = List<obj>
```

```
// contravariace reverses arrows
Action<object> objAct = function taking
object
Action<string> strAct = funtion taking a
string;
// Allowed: A<obj> -> A<str>
strAct = objAct;
// ERROR! Disallowed! Cannot do A<obj> <-
A<string>
objAct = strAct;
```

Covariant/Contravariant



*Co*variant = preserves arrows



Contravariant = reverses arrows

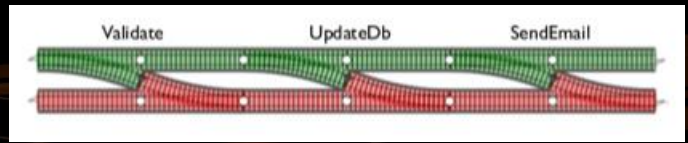
Contravariant/Covariant

- This leads to the concept of ***category theory***
- A ***category*** is a collection of things with arrows that compose
- IEnumerable and Action are ***functors***
  - ***functors***: map things with arrows to things with arrows
  - Many, many things in programming are functors
- Understanding some of this gives powerful tools to reason about systems

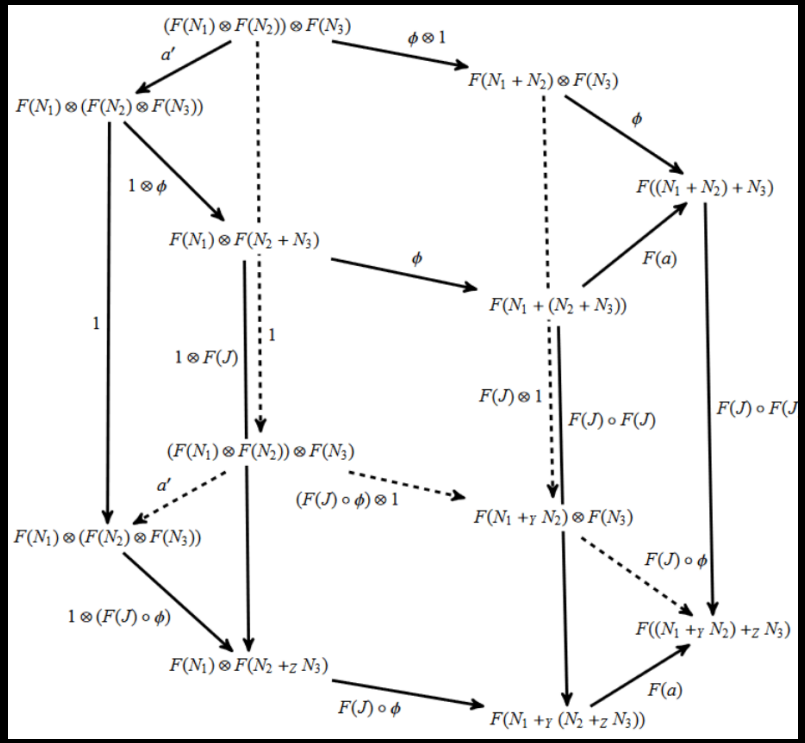
"Category theory is type theory" – Abraham Lincoln

Category Theory

- Gives large expressive power if you build interfaces “correctly”
- Example:
- Monoids - set with binary operation and unit
  - Integers, +, 0
  - Integers, *, 1
  - Strings, +, “”
  - Lists, concat, []
  - Generalized…..
- Lots more terms ☺
  - Duality, morphisms,
  - Functors galore
  - Products – simply a pair of items
  - Coproducts – the “dual” of a product
  - Monads
  - Kliesi arrows
- Railway Oriented Programming



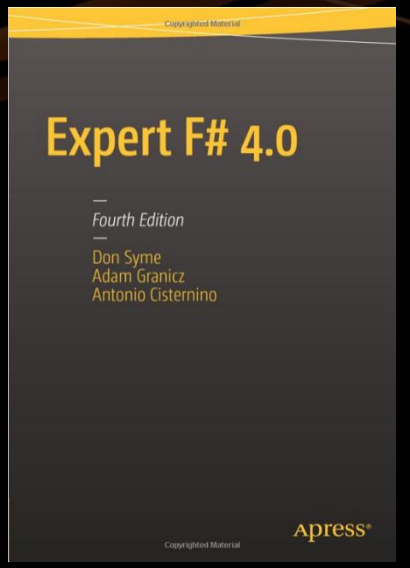
Questions?

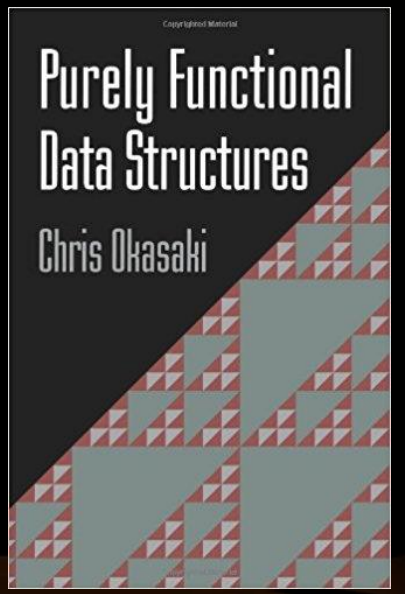


A Bicategory of Decorated Cospans

Sources and Resources

- Good talk on F# : https://channel9.msdn.com/blogs/pdc2008/tl11
- https://fsharpforfunandprofit.com/
- ***Expert F# 4.0*** – Don Syme book
- ***Purely Functional Data Structures*** - Chris Okasaki book
- ***Category Theory for Programmers*** – Bartosz Milewski https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/
- ***Railway Oriented Programming*** - Scott Wlaschin: https://www.slideshare.net/ScottWlaschin/railway-oriented-programming
- ***Efficient Immutable Data Structures*** – Tom Faulhaber, https://www.slideshare.net/tomfaulhaber/efficient-immutable-data-structures-okasaki-for-dummies
- Slide theme from SoftUni Team: http://softuni.bg





TODO – extra slides

- Books and sources,
- Add pics to slides (category stuff)
- Example translating C# to F#
- TODO if can find
- Quick example of reducing C# code to F# by removing unnecessary syntax
- Books, bring them
- Data structures example
- Explain how works (lists, trees, issues)

Examples : Perfect tic-tac-toe

```
let wins = [[1;2;3]; [4;5;6]; [7;8;9]; [1;4;7]; [2;5;8]; [3;6;9]; [1;5;9]; [3;5;7]]
let Contains number = List.exists (fun n -> n = number)
let ContainsList list = List.forall (fun n -> list |> Contains n)
let Except list = List.filter (fun n -> not (list |> Contains n))
let Available (p : int list) (o : int list) = [1..9] |> Except (List.append p o)
let IsWin (squares: int list) = wins |> List.exists (fun w -> ContainsList squares w)
let IsDraw player opponent = List.isEmpty (Available player opponent)

let rec Score (player: int list) (opponent: int list) =
   if (IsWin player) then 1
   else if (IsDraw player opponent) then 0
   else
      let opponentsBestMove    = BestMove opponent player
      let opponentsNewPosition = opponentsBestMove::opponent
      - Score opponentsNewPosition player

and BestMove (player: int list) (opponent: int list) = Available player opponent
   |> List.maxBy (fun m -> Score (m::player) opponent)
```

Options

- Example

```
let people = [
    ("Adam", None),
    ("Eve" , None),
    ("Cain", Some("Adam", "Eve")),
    ("Abel", Some("Adam", "Eve"))
    ]

let showParents (name, parents) =
    match parents with
    | Some(dad, mom) ->
        printfn "%s dad %s and mom %s" name dad mom
    | None -> printfn "%s has no parents" name
```