What & Why Functional Programming in C# & F#

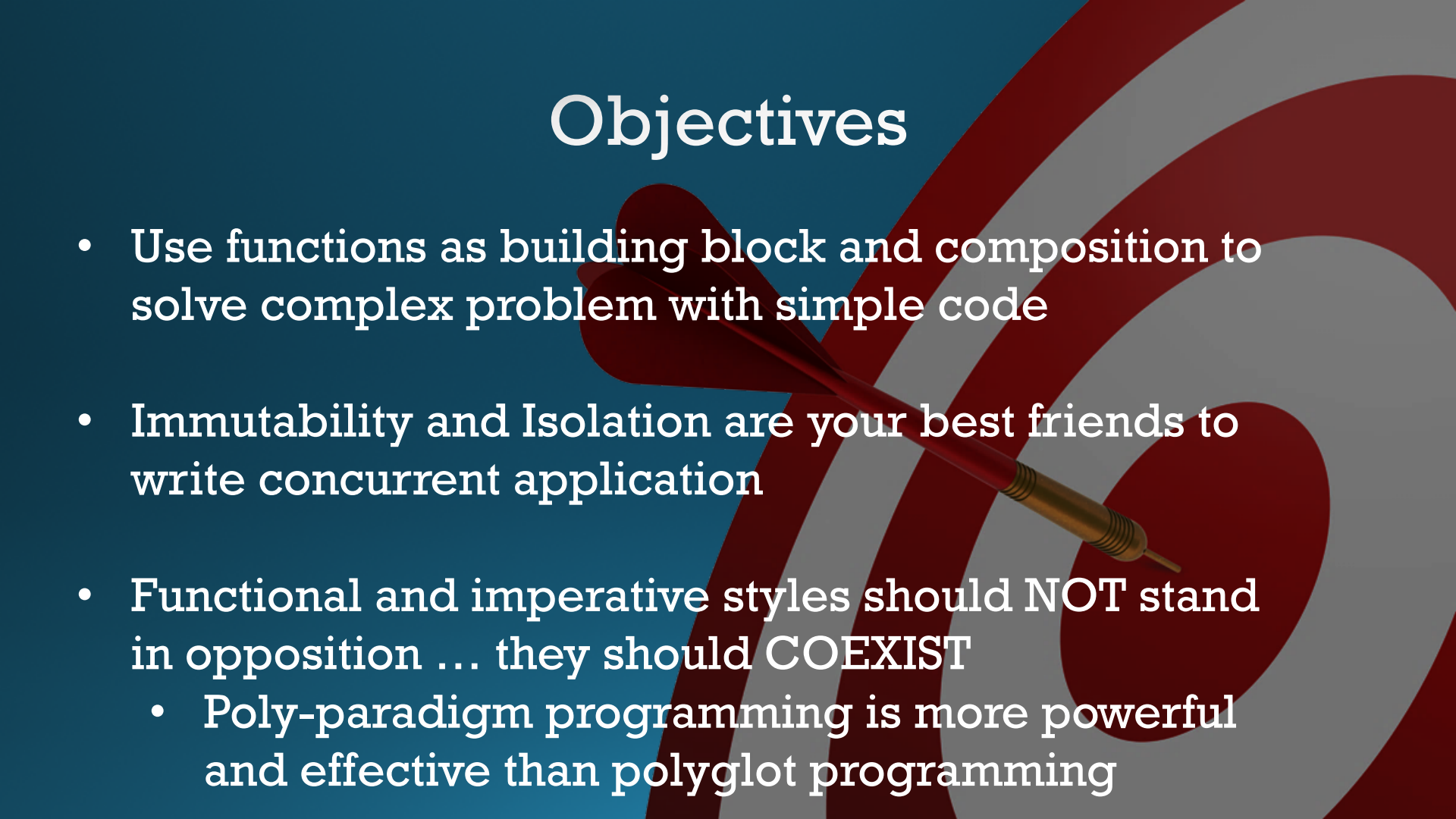
OOP makes code understandable by encapsulating moving parts.
FP makes code understandable by eliminating moving parts.

— *Michael Feathers*
(author of Working with Legacy Code)

Agenda

- History and influence of Functional Programming
- What is Functional Programming
- Why Functional Programming

Objectives



- Use functions as building block and composition to solve complex problem with simple code
- Immutability and Isolation are your best friends to write concurrent application
- Functional and imperative styles should NOT stand in opposition … they should COEXIST
  - Poly-paradigm programming is more powerful and effective than polyglot programming

History

λ

1930

1958

LISP

1962 APL

1966

ISWIM

1973 ML

1977 FP

2005 F#

2007 Clojure

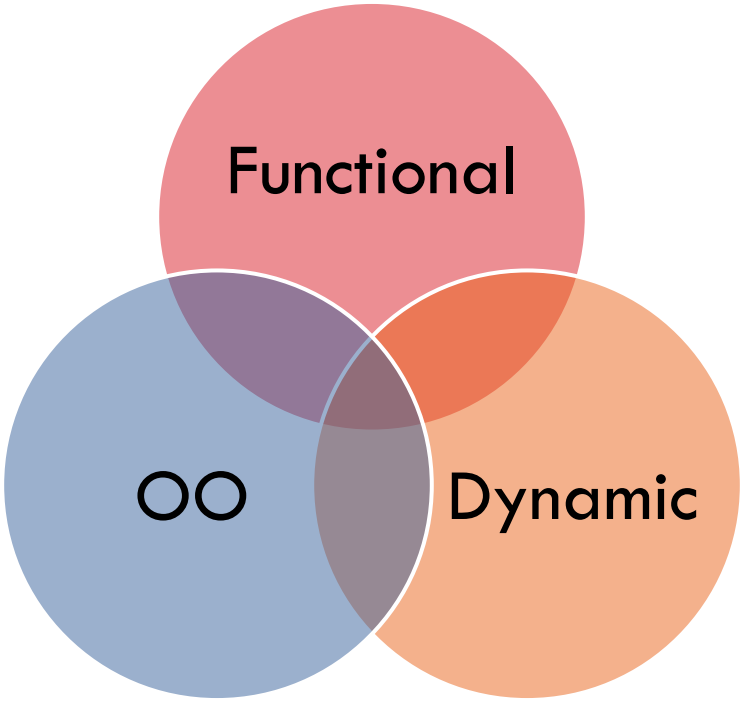
2003 Scala

Ocaml 1996

1988 Haskell

1986 Erlang

Languages go multi-paradigm



Early 1994 support for lambda, filter, map and reduce in Python

Introduced List Comprehension in C# 3.0 with LINQ and Java 8 with Streams

Introduced full support for lambda in Java 8, C#, VB.NET and C++

Functional Programming Influence

- Continue Evolution of GC – Lisp ~1958
- Generics – ML ~ 1973
- In early 1994, support for lambda, filter, map, and reduce was added to Python
- List comprehension - Linq in C# 3.0 and Java 8
- First-class functions were also introduce Visual Basic 9, C#, Java 8 and C++ 11
- Java 8 supports lambda expressions as a replacement for anonymous classes
- Is Lazy Evaluation is part of the main stream languages such as C#
- Type Inference, example is the "var" keyword in C#

Programming in a functional style can also be accomplished in languages that aren't specifically designed for functional programming, Lots of modern languages have elements from FP

What is "Functional Programming?"

Functional programming is just a style, is a programming paradigm that treats computation as the evaluation of functions and avoids state and mutable data…

- *"Functions as primary building blocks"* (first-class functions)
- programming with "*immutable*" variables and assignments, no *state*
  - Programs work by returning values instead of modifying data

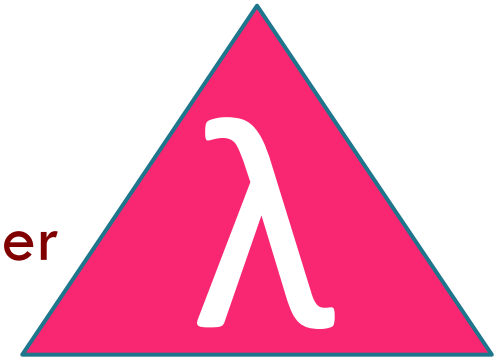
What is "Functional Programming?"

Functional programming is so called because a program consists entirely of functions.

— John Hughes, Why Functional Programming Matters

Declarative

First class
Higher-Order
Functions



Immutability

… is all about functions! Identifying an abstraction and building a function, use existing functions to build more complex abstractions, pass existing functions to other functions to build even more complex abstractions

Functional Programming

It's declarative, stateless, side-effects free and immutable

describes what we want, not how we want it

> *"Functional programming allows developers to describe what they want to do, rather than forcing them to describe how they want to do it."*
>
> Anders Hejlsberg, C# creator

Functional Programming

It's declarative, stateless, side-effects free and immutable

describes what we want, not how we want it

system relys only on inputs, not external state

Functional Programming

It's declarative, stateless, side-effects free and immutable

describes what we want, not how we want it

system relys only on inputs, not external state

once value has been set, we can't override its value

Return new value, instead of altering existing ones

… but why? … *why FP?*

Motivation for Function Programming

- Functional Programming promotes Composition and Modularity
- Declarative programming style
- Concurrency, FP is easy to parallelize
- Conciseness, less code (no null checking)
- Simplicity in life is good: cheaper, easier, faster, better
  - We typically achieve simplicity in software in two ways:
    - By raising the level of abstraction
    - Increasing modularity
- Better composition and modularity == reuse
- The Multi-core (r)evolution! Concurrency without pain
- Data Driven World
  - More and more data: need higher order algorithms
  - and techniques to derive value from data

It's really clear that the imperative style of programming has run its course. ... We're sort of done with that. … However, in the declarative realm we can speculate a 10x improvement in productivity in certain domains.

-Anders Hejlsberg
C# Architect

(from his MIX07 keynote)

“When I code in C, I feel I'm on a knife-edge of ‘state’ — I focus on statements and what they *do*. I'm worried about the behavior of a machine.”

- Richard A. O'Keefe

Twitter Paper

Your Server as a Function

Marius Eriksen

Twitter Inc.

marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a pow-

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

http://monkey.org/~marius/funsrv.pdf

Twitter Paper

Your Server as a Function

Servers in a large-scale setting are required to process tens of thousands, if not hundreds of thousands of requests concurrently; they need to handle partial failures, adapt to changes in network conditions, and be tolerant of operator errors. As if that weren't enough, harnessing off-the-shelf software requires interfacing with a heterogeneous set of components, each designed for a different purpose. These goals are often at odds with creating modular and reusable software [6].

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

Functional characteristics

- Higher Order Function
- Immutability
- Composition
- Curry & Partial Application
- Recursion

Higher-Order functions

A higher-order function is a function that takes another function as a parameter, or a function that returns another function as a value, or a function which does both.

```
Func<Developer, GoodSeniorDeveloper> convertToGSD = d => new
  GoodSeniorDeveloper
{
    FirstName = d.FirstName,
    LastName = d.LastName
};

Func<Developer, bool> isGSD = d => d.YearsOfExperience >= 5 &&
  d.IsFunctional;
      if (predicate(item))
IEnumerable<GoodSeniorDeveloper> goodSeniorDevelopers =
              Map(Filter(developers, isGSD), convertToGSD);
}
```

Higher-Order functions

A higher-order function is a function that takes another function as a parameter, or a function that returns another function as a value, or a function which does both.

```
Func<Developer, GoodSeniorDeveloper> convertToGSD = d => new
  GoodSeniorDeveloper
{
    FirstName = d.FirstName,
    LastName = d.LastName
};

Func<Developer, bool> isGSD = d => d.YearsOfExperience >= 5 &&
  d.IsFunctional;

IEnumerable<GoodSeniorDeveloper> goodSeniorDevelopers =
                developers.Where(isGSD)
                          .Select(convertToGSD);
```

Higher-Order functions

A higher-order function is a function that takes another function as a parameter, or a function that returns another function as a value, or a function which does both.

```
let getGoodSeniorDeveloper developers =
        // Developer -> GoodSeniorDeveloper
        let convertToGSD (d:Developer) = {FirstName = d.FirstName;
          LastName=d.LastName};
        // Developer -> bool
        let isGoodSeniorDeveloper d = d.IsFunctional && d.YearsOfExperience >= 5

        developers
        |> List.filter isGoodSeniorDeveloper
        |> List.map convertToGSD
```

Pure Functions - Properties and Benefits

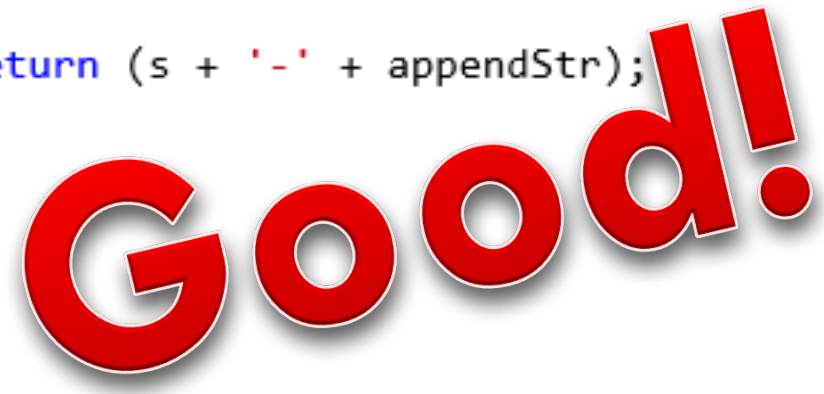
- *A function always gives the same output value for a given input value*
  - *I can then cache the result*
- *A Pure Function has no side effects*
- Pure functions can be executed in parallel
- The function does not rely on any external state
  - *Increased readability and maintainability*
- Pure functions can more easily isolated
  - *easier testing and debugging*

Pure Functions - Properties and Benefits

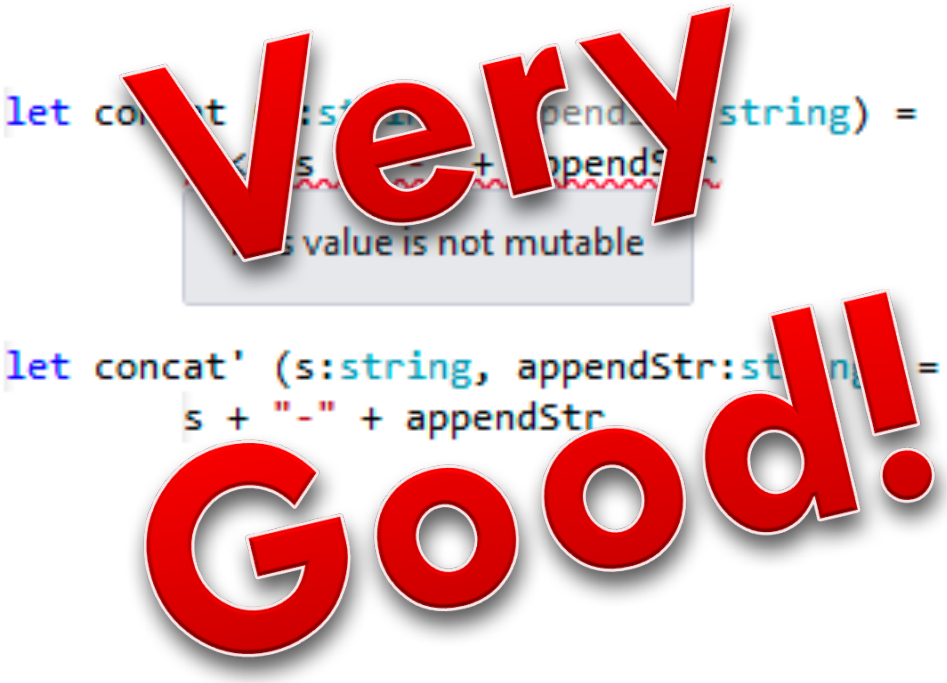
```
private string aMember = "StringOne";

public void Concat(string appendStr)
{
        aMem r = '-' + appendStr;
}
```

Pure Functions - Properties and Benefits

```
public string Concat(string s, string appendStr)
{
        return (s + '-' + appendStr);
}
```



Pure Functions - Properties and Benefits



FP Preachings!

- Avoid Side-Effects!
  - Do not modify variables passed to them
  - Do not modify any global variab
- Avoid Mutation

Reasoning about your code

```
int Sum(List<int> values) { … }

List<int> values = new List<int>{ 1,2,3,4,5 };

int result1 = Sum(values);     // 15

int result2 = Sum(values);     // ??
```

Reasoning about your code

```
int Sum(List<int> values) {

        int sum = 0;
        for(int i = 0;i < arr.Length; i++){
                sum += values[i];
                values[i] = 0;

        }
}
```

Reasoning about your code

```
int Sum(List<int> values) { … }

List<int> values = new List<int>{ 1,2,3,4,5 };

int result1 = Sum(values);     // 15

int result2 = Sum(values);     // ??

int Sum(IEnumerable<int> values) { … }
```

Side Effects

Side effects aren't bad…

Unwanted Side Effects are Evil!

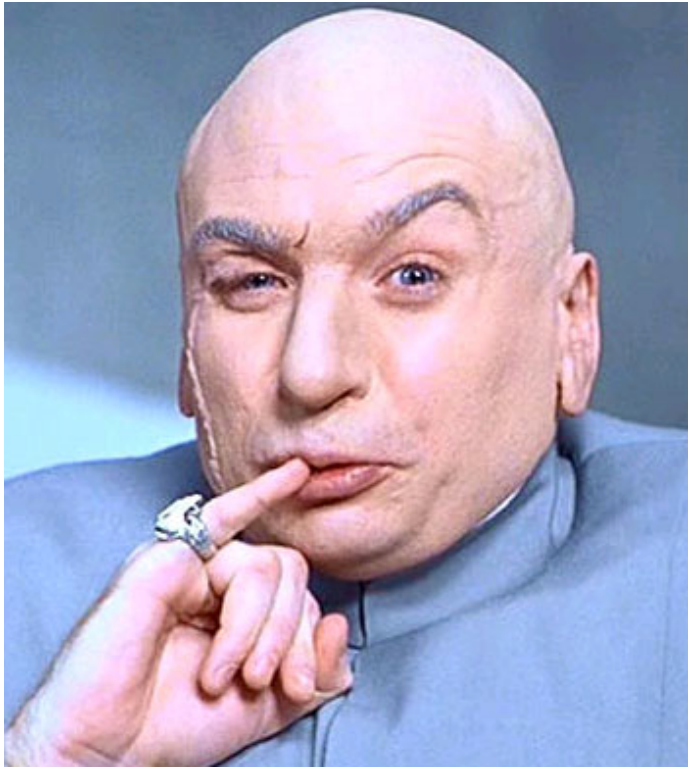
Side effects aren't bad…

unwanted side effects are EVIL

and the root of many bugs!

One of the biggest advantages with functional programming is that the order of execution of "side effect free functions" is not important.

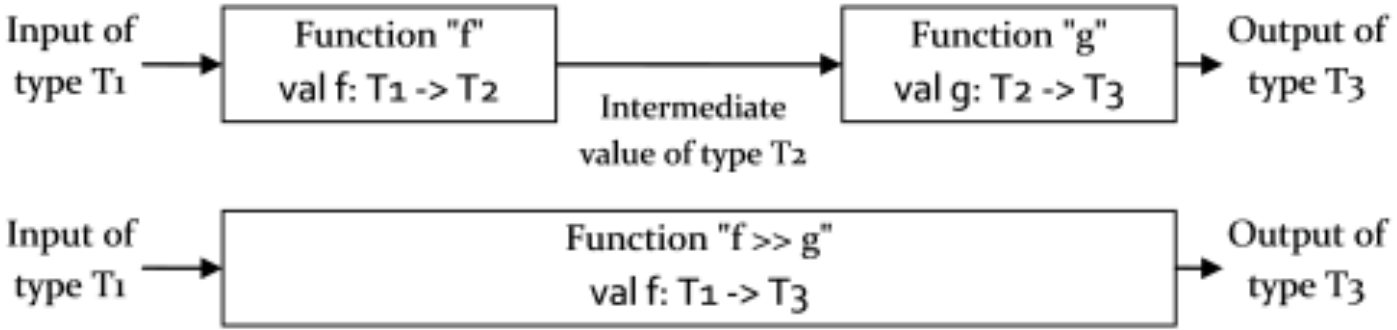
… "side effect free functions" are easy to parallelize



Function Composition

Function Composition - Building with composition. Composition is the 'glue' that allows us build larger systems from smaller ones. This is the very heart of the functional style. Almost every line of code is a composable expression. Composition is used to build basic functions, and then functions that use those functions, and so on.

Composition — in the form of passed parameters plus first-class functions



Using functions as building blocks

```
// string -> bool
let filterDocx = (fun f -> Path.GetExtension(f) = ".docx")
// string -> FileInfo
let toFileInfo = (fun file -> new FileInfo(file))
// FileInfo -> int64
let getLen = (fun (info:FileInfo) -> info.Length)

// string[] -> string[]
let filter = Array.filter filterDocx
// string[] -> FileInfo[]
let mapFileInfo = Array.map toFileInfo
// FileInfo[] -> int64[]
let allLen = Array.map getLen

// string -> int64
let getSizeDirectory =
    getFiles >> filter >> log >> mapFileInfo >> allLen >> Array.sum
```

Using functions as building blocks

```
// string -> bool
let filterDocx = (fun f -> Path.GetExtension(f) = ".docx")
// string -> FileInfo
let toFileInfo = (fun file -> new FileInfo(file))
// FileInfo -> int64
let getLen = (fun (info:FileInfo) -> info.Length)

// string[] -> string[]
let filter = Array.filter filterDocx
// string[] -> FileInfo[]
let mapFileInfo = Array.map toFileInfo
// FileInfo[] -> int64[]
let allLen = Array.map getLen

// string -> int64
let getSizeDirectory =
    getFiles >> filter >> log >> mapFileInfo >> allLen >> Array.sum
```

Function Composition Operators

The forward pipe operator:

```
let (|>) arg func =          Type: 'a -> ('a -> 'b) -> 'b
     func arg
```

Allows you to reorder function calls to put the last argument first:

```
let three = 1 |> add 2
```

Produces function `int -> int`

Invokes `(int -> int)` with 1

Function Composition

```
open System

let evenNumbersAsWords =
    [ 1 .. 15 ]
    |> Seq.filter (fun i -> i % 2 = 0)
    |> Seq.map (fun i -> i.ToWords())
    |> Seq.reduce (sprintf "%s, %s")
```

Creates an F# List with the integers 1 to 15

Partial application of the Seq.filter F# Lambda

Partial application of sprintf function

Map Reduce!

Immutability!

evenNumbersAsWords:
"two, four, six, eight, ten, twelve, fourteen"

Curry and Partial Application

- **Currying** is the process of rewriting function with multiple parameters to a series of functions, each with only one parameter.
- Currying is done **automatically** by the compiler.
- **Partial application** is when function applied with some of the parameters and return new function

Currying and Partial Application

C#

```
public static Func<int, int> Add(int x)
{
    return y => x + y;
}

public static Func<int, int> Add1() => Add(1)

int valueA = Add  (1)(3); // Add  (1:x) (3:y)
int valueB = Add1 (3);    // Add1 (3:y)
```

Currying and Partial Application

F#

```
let add x y = x + y
let add1 = add 1
let value = add1 2
```

Immutability

- Like string in C# **everything** is immutable
- Help in concurrent/parallel code
- Immutability is **default** in F#
- You can choose to go mutable (**use carefully!**)

Referential transparency

- An expression is said to be referentially transparent if it can be **replaced** with its corresponding value **without changing the program's behavior.**

- As a result, evaluating a referentially transparent function gives the same value for same arguments.

- In general it means that the function **don’t have side effect** which may influence its result.

Referential transparency

- Which of the following method are Referential transparency?

C#

```
DateTime Add(DateTime date, TimeSpen duration) =>
                                date.Add(duration);

DateTime AddDay(DateTime date) =>
                                Add(date, TimeSpan.FromDay(1));

DateTime Tomorrow() => AddDay (DateTime.Now)

DateTime AddTrigger(DateTime date, Func<int> f) =>
                                Add(date, TimeSpan.FromDay(f()));
```

Referential transparency

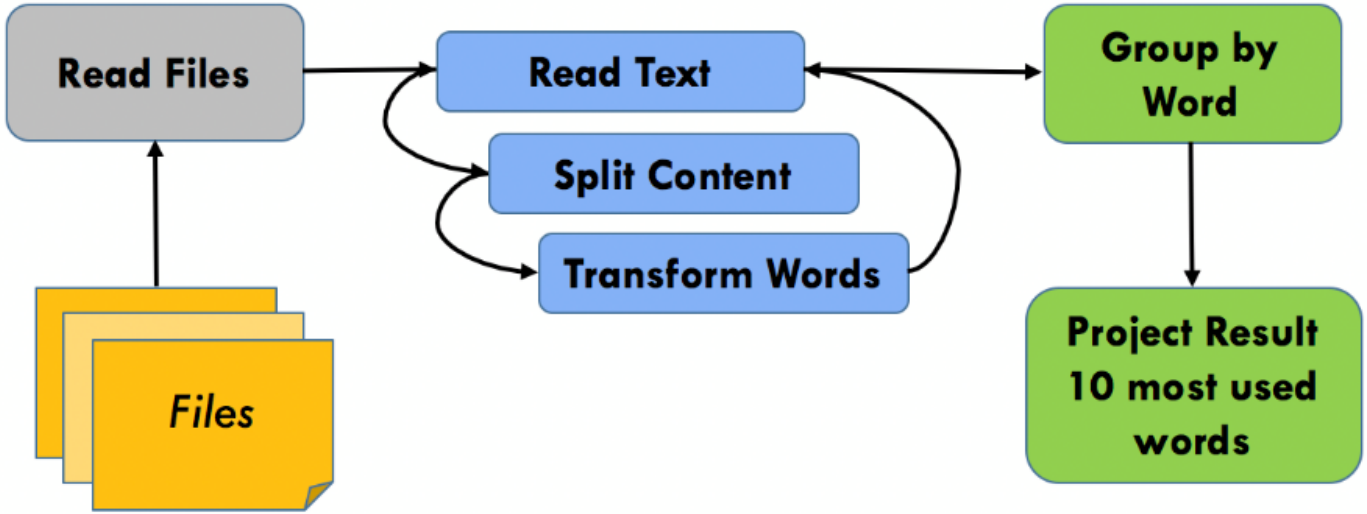
- Which of the following method are Referential transparency?

C#

```
DateTime Add(DateTime date, TimeSpen duration) =>
                                date.Add(duration);

DateTime AddDay(DateTime date) =>
                                Add(date, TimeSpan.FromDay(1));

DateTime Tomorrow() => AddDay (DateTime.Now)

DateTime AddTrigger(DateTime date, Func<int> f) =>
                                Add(date, TimeSpan.FromDay(f()));
```

Avoiding side effects with pure function

Parallel words counter with side effects



Parallel words counter with side effects

```
public static Dictionary<string, int> WordsCounter(string source)
{
    var wordsCount =
            (from filePath in
                Directory.GetFiles(source, "*.txt").AsParallel()
                from line in File.ReadLines(filePath)
                from word in line.Split(' ')
                select word.ToUpper())
        .GroupBy(w => w)
        .OrderByDescending(v => v.Count()).Take(10); //#C
    return wordsCount.ToDictionary(k => k.Key, v => v.Count());
}
```

Why removing without side effects

- Easy to reason about the correctness of your program
- Easy to compose functions for creating new behavior
- Easy to be isolated and, therefore, easy to test and less bug prone
- Easy to be executed in parallel; because pure functions don't have external dependencies, their order of execution (evaluation) doesn't matter

LAB ??
Avoiding side effects with pure function

Parallel words Counter with side effects

```
static Dictionary<string, int> PureWordsPartitioner
                        (IEnumerable<IEnumerable<string>> content) =>
    (from lines in content.AsParallel()
        from line in lines
        from word in line.Split(' ')
        select word.ToUpper())
        .GroupBy(w => w)
        .OrderByDescending(v => v.Count()).Take(10)
        .ToDictionary(k => k.Key, v => v.Count());

static Dictionary<string, int> WordsPartitioner(string source). {
    var contentFiles =
        (from filePath in Directory.GetFiles(source, "*.txt")
            let lines = File.ReadLines(filePath)
            select lines);

    return PureWordsPartitioner(contentFiles);
}
```

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra