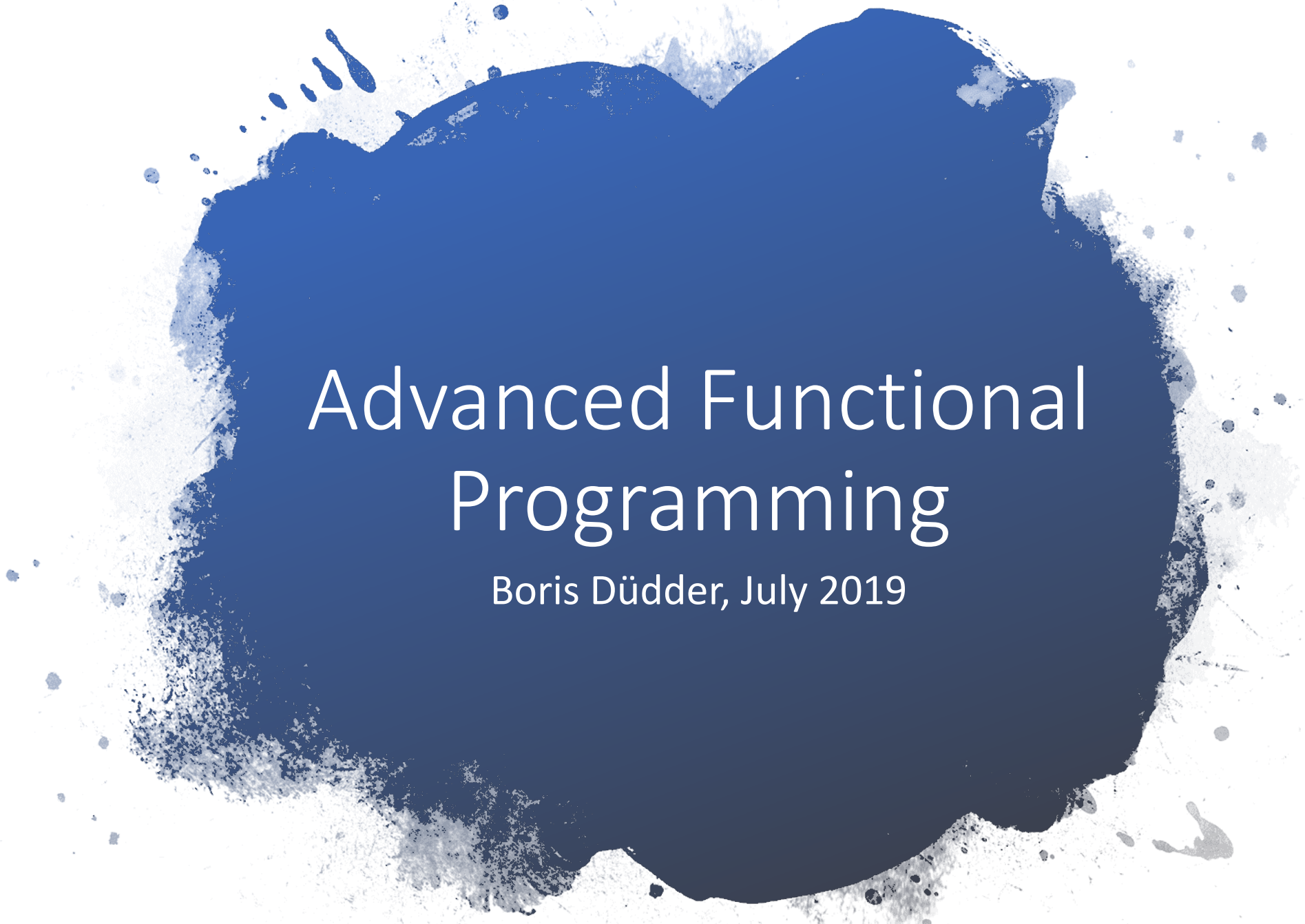

Advanced Functional Programming
Boris Düdder, July 2019

Introduction to Functional Programming

Teachers:

- Boris Düdder, Assistant Professor, UCPH
- Sarah Hyatt, Teaching Assistant, UCPH
- Ma Mengfan, Teaching Assistant, UESTC
- Tao Binglin Teaching Assistant, UESTC
- Li Mengying, Teaching Assistant, UESTC

F#

Notes:

- ”Learning to program with F#”
- “Report template”
- Assignments + code + data

Hand-ins:

- Exercise 1: Groups of 2
- Exercise 2: Groups of 2

Basic rules

1. In time submission of assignments
2. Submission via email as a zip-file
    1. Containing one single solution file
    2. One report as PDF(!)-document
    3. Include all files that solution is runnable (exclude BIN and other folders)
    4. Include UESTC ID in subject of email, report, and source code
3. Genuine solutions (plagiarism leads to direct fail)

Schedule

Time	Monday	Tuesday	Wednesday	Thursday	Friday
09:00-9:45	Introduction to F#	Solutions to assignments 1-3	Solutions to assignment 4 Report writing	Solutions to assignment 5 Open question session	Assignment 6
10:00-10:45	Introduction Lambda Calculus 1	User defined types	Assignment 5	Assignment 6	Assignment 6
10:00-11:55	Assignments 1-3	Working with trees	Assignment 5	Assignment 6	Assignment 6
14:00-14:45	Introduction Lambda Calculus 2	Higher-order functions			
15:00-17:00	Assignments 1-3	Assignment 4			
20:00			Assignment 5 deadline		Assignment 6 deadline

Programming Paradigms















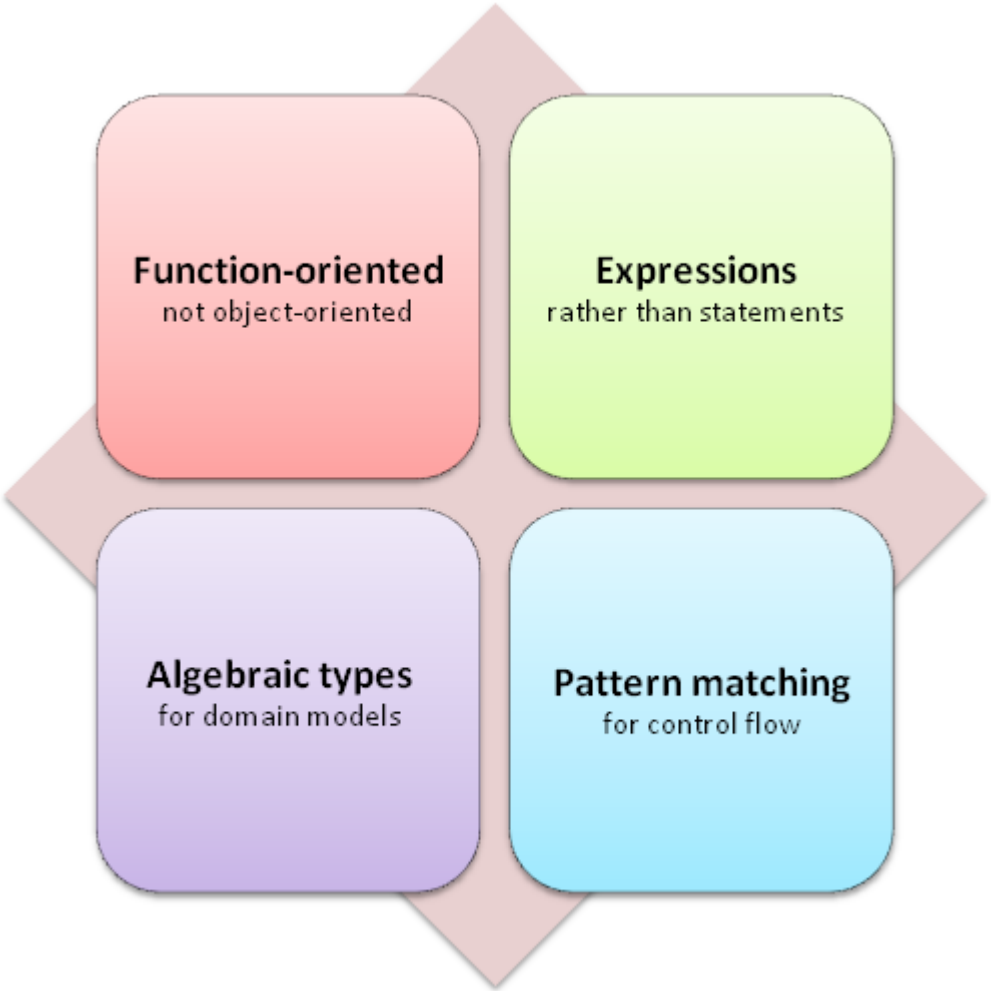



F# Scheme


Why Should I Use Functional Programming?

- Fewer Bugs
- Code Simpler/More Maintainable Code
- No Side Effects
- Easy to Parallelize & Scale
- Mathematically Provable
- Its been around a while

Functional Core Concepts



Terms to Know

- Immutable Data
- First Class Functions
- Recursion
- Tail Call Optimization
- Mapping
- Reducing
- Pipelining
- Currying
- Higher Order Functions
- Lazy Evaluation

Monad: “A Monad is just a monoid in the category of endofunctors, what’s the problem?

Lambda Calculus

Functional Programming

```
a = 0
b = 2
sum = 0
def add():
    global sum
    sum = a + b
```

```
def add(a, b):
    return a + b
```

Higher Order Functions (Map)

```
y = [0, 1, 2, 3, 4]
ret = []
for i in y:
    ret.append(i ** 2)
print(ret)
```

```
y = [0, 1, 2, 3, 4]
squares = map(lambda x: x * x, y)
print(squares)
```

Higher Order Functions (Filter)

```
x = np.random.rand(10,)
for i in range(len(x)):
        if(x[i] % 2):
                y[i] = x[i] * 5
        else:
                y[i] = x[i]
```

```
x = np.random.rand(10,)
y = map(lambda v : v * 5,
                filter(lambda u : u % 2, x))
```

Higher Order Functions (Reduce)

```
x = [0, 1, 2, 3, 4]
sum(x)
```

```
import functools

x = [0, 1, 2, 3, 4]
ans = functools.reduce(lambda a, b: a + b, x)
```

```
x = [0, 1, 2, 3, 4]
ans = reduce(lambda a, b: a + b, x)
```

Tail Call Recursion

```
def factorial(n, r=1) :
    if n <= 1 :
        return r
    else :
        return factorial(n-1, n*r)
```

```
def factorial(n):
    if n==0 :
        return 1
    else :
        return n * factorial(n-1)
```

Partial Functions

Consider a function f(a, b, c);

Maybe you want a function g(b, c) that's equivalent to f(1, b, c);

This is called "partial function application".

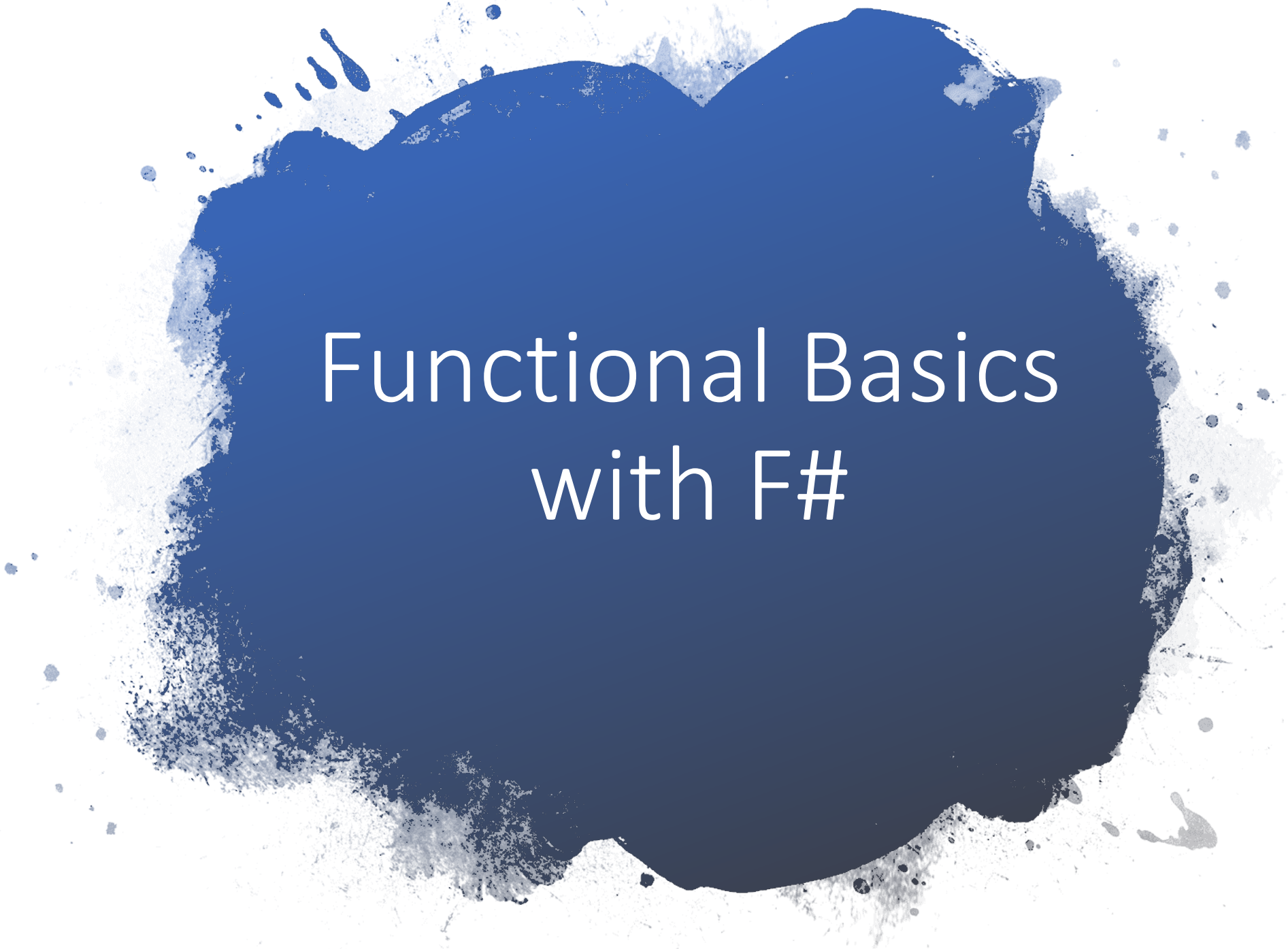
```
import functools

def log(message, subsystem):
    """Write the contents of 'message' to the specified subsystem."""
    print('%s: %s' % (subsystem, message))
    ...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

DEMO

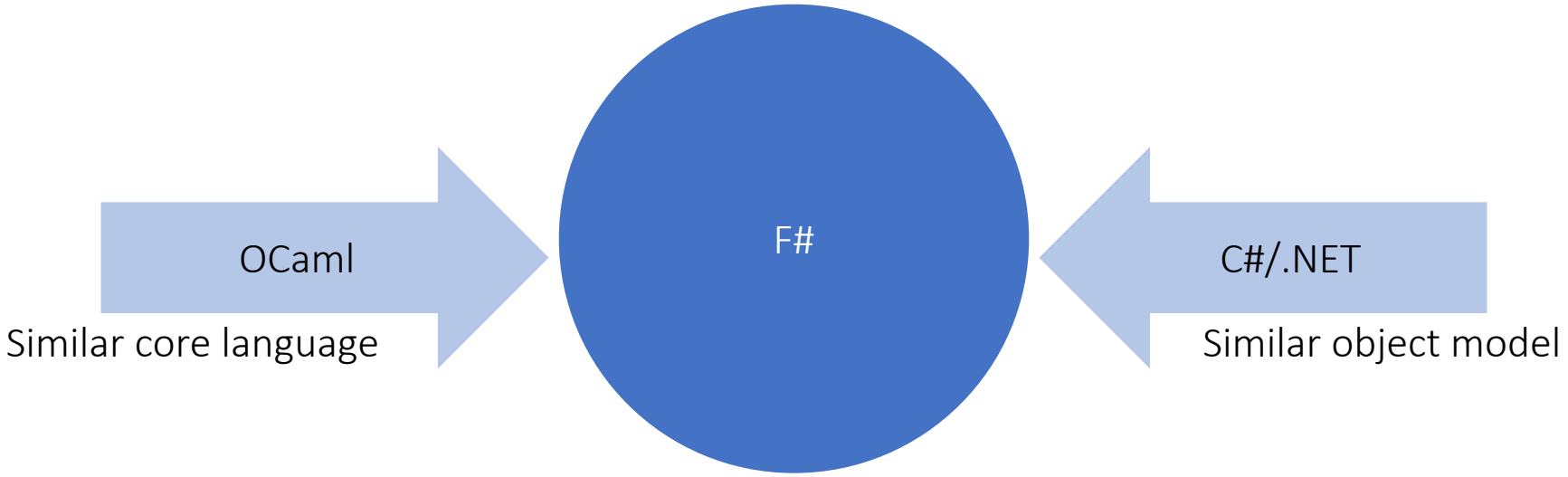
What’s Next?


Functional Basics
with F#

F# Syntax Cheat Sheets

- http://dungpa.github.io/fsharp-cheatsheet/
- http://www.samskivert.com/code/fsharp/fsharp-cheat-sheet.pdf
- https://msdn.microsoft.com/en-us/library/dd233181.aspx
- http://en.wikibooks.org/wiki/F_Sharp_Programming

History of F#



Imperative vs. Functional

What does “functional” even mean?

- Preferring immutability
  - Avoid state changes, side effects, and mutable data as much as possible.
- Using data in → data out transformations
  - Try modeling your problem as a mapping of inputs to outputs.
  - Everything is an expression! Too much |> ignore is often an anti-pattern
- Treating functions as the unit of work, not objects
- Looking at problems recursively
  - Think of ways to model a problem as successively smaller chunks of the same problem

Functional basics – Immutability

```
var x = 1;
```

≠

```
let x = 1

x++

let y = x+1
```

```
let mutable x = 1
x<-2
```

Declarative Style

```
var vipCustomers = new
List<Customer>();
foreach (var customer in customers)
{
   if (customer.IsVip)
      vipCustomers.Add(customer);
}
```

```
var vipCustomers = customers.Where(c
=> c.IsVip);
```

Functions

```
int Add(int x, int y)
{
   return x + y;
}
```

Func<int,int,int>

In Out

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

int -> int -> int

In Out

no parens

commas

Pipeline Operator

```
let filter (condition: int -> bool) (items: int list) = // ...
```

```
let filteredNumbers = filter (fun n -> n > 1) numbers
```

```
let filteredNumbers = numbers |> filter (fun n -> n > 1)
```

```
let filteredNumbers = numbers
                      |> filter (fun n -> n > 1)
                      |> filter (fun n -> n < 3)
```

Curryir

```
//normal version
let addTwoParameters x y =
  x + y

//explicitly curried version
let addTwoParameters x  =     // only one parameter!
  let subFunction y =
    x + y                 // new function with one param
  subFunction             // return the subfunction

// now use it step by step
let x = 6
let y = 99
let intermediateFn = addTwoParameters x  // return fn with
                                         // x "baked in"
let result  = intermediateFn y

// normal version
let result  = addTwoParameters x y
```

```
val printTwoParameters : int -> (int -> unit)
```

Partial Application

```
let sum a b = a + b
```

```
let result = sum 1 2
```

Returns int = 3

```
let result = sum 1
```

Returns int -> int

```
let addOne = sum 1
```

Returns int -> int

```
let result = addOne 2
```

Returns int = 3

```
let result = addOne 3
```

Returns int = 4

Composition

```
let addOne a = a + 1
```

```
let addTwo a = a + 2
```

```
let addThree = addOne >> addTwo
```

```
let result = addThree 1
```

Returns int = 4

Functional basics: Higher-order functions

```
[1..10]
   |> List.filter (fun x -> x % 2 = 0)
   |> List.map (fun x -> x + 3)
   |> List.sum
```

```
[|1.0;2.;3.;4.;5.;6.;7.;8.;9.;10.|]
   |> Array.filter (fun x -> x % 2. = 0.)
   |> Array.map (fun x -> x + 3.)
   |> Array.sum
```

```
let plus_3 x = x + 3
let list_plus_3 = List.map plus_3
let filtered = List.filter (fun x -> x % 2
= 0)

[1..10]
  |> filtered
  |> list_plus_3
  |> List.sum
```

```
let sumEvensPlusThree =
    Array.filter (fun x -> x % 2 = 0)
    >> Array.map (fun x -> x + 3)
    >> Array.sum

sumEvensPlusThree [|1..10|]
```

```
[|1..10|] |>
sumEvensPlusThree
```

Work with Higher Order Functions

- What is the sum of the numbers 1 to 100, each squared?
- What about the sum of just the even numbers?
- Write a function that takes any list of floats and a function as an input.
  - Add 10.25 to each element
  - Divide each element by 4
  - Finally act on the list with the function you sent in.

Higher-order functions: Answer

```
1   let numbers = [1..100]
2   numbers
3       |> List.map (fun x -> x*x)
4       |> List.sum
5
6   numbers
7       |> List.filter (fun x -> x%2=0)
8       |> List.map (fun x -> x*x)
9       |> List.sum
10
11  let floats = [1.0..10.0]
12
13  let changefloats mylist func =
14      mylist
15          |> List.map (fun x -> x+10.25)
16          |> List.map (fun x -> x/4.)
17          |> List.map func
18
19  changefloats floats (fun x -> x*x)
```

The Iterator and Disposable patterns in F#

- F# provides the use keyword as an equivalent of C#'s using statement keyword (not to be confused with C#'s using directive keyword, whose F# equivalent is open)
- In F#, seq is provided as a shorthand for IEnumerable
- Your preference for collections should be (in descending order): list, array, seq

What is polymorphism?

- Subtype polymorphism: when a data type is related to another by substitutability
- Parametric polymorphism: when code is written without mention to any specific type (e.g., list of X type, array of X type)
- Ad hoc polymorphism: when a function can be applied to arguments of different types
  - Overloading (built-in and/or custom)
  - Haskell: type classes
  - F# specific feature: statically resolved type parameters

Continuation Passing Style (a.k.a. Callbacks)

- “Hey, when you’re done doing that…”
- Explicitly pass the next thing to do
- Provides a method of composition of functions that can alter control flow
- More common than you may realize (we’ll come back to this…)
- Very common in **Javascript** as well

Blocking I/O and You – the reason for Async

- The operating system schedules sequential operations to run in a **thread**
- If code requires external I/O, the thread running that code will block until it is complete
- This is bad

Example of a blocking operation

```
// Synchronously - use blocking I/O calls
// to stall out threads and warm up your
// server room
let DownloadPageSync (url:Uri) =
    let request = WebRequest.Create(url)
    let response = request.GetResponse()
    use responseStream = response.GetResponseStream()
    use streamReader = new StreamReader(responseStream)
    let data = streamReader.ReadToEnd()
    let file = Path.GetTempFileName()
    use outputStream = new StreamWriter(file)
    outputStream.WriteLine(data)
    file
```

Your Web Server, Running Blocking I/O

Continuation Passing Style (a.k.a. Callbacks)

```
// Asynchronously, the messy way, using
// explicit Continuation Passing Style (CPS)
// a.k.a. "Callback Hell"
let DownloadPageAsyncBad (url:Uri) =
    let request = WebRequest.Create(url)
    request
        .GetResponseAsync()
        .ContinueWith(
            // Callback #1
            fun (responseTask:Task<WebResponse>) ->
                let response = responseTask.Result
                use responseStream = response.GetResponseStream()
                use streamReader = new StreamReader(responseStream)
                streamReader
                    .ReadToEndAsync()
                    .ContinueWith(
                        // Callback #2
                        fun (dataTask:Task<string>) ->
                            let data = dataTask.Result
                            let file = Path.GetTempFileName()
                            use outputStream = new StreamWriter(file)
                            outputStream
                                .WriteLineAsync(data)
                                .ContinueWith(
                                    // Callback #3
                                    fun (writeTask:Task) ->
                                        file
                                )
                    ).Unwrap()
        ).Unwrap()
```

Continuation Pas Style (a.k.a Callback Hell)

F# Async to the Rescue!

```
// Luckily, there's a better way!
let DownloadPageAsync (url:Uri) =
    async {
        let request = WebRequest.Create(url)
        let! response = request.GetResponseAsync() |> Async.AwaitTask
        use responseStream = response.GetResponseStream()
        use streamReader = new StreamReader(responseStream)
        let! data = streamReader.ReadToEndAsync() |> Async.AwaitTask
        let file = Path.GetTempFileName()
        use outputStream = new StreamWriter(file)
        do! outputStream.WriteLineAsync(data) |> Async.AwaitTask
        return file
    }
```


WHAT SORCERY
IS THIS?

Let’s start from the beginning…

```fsharp
// As basic an F# program as you can think of...
let run () =
    let x = 5
    let y = 10
    let z = x + y
    printfn "Step1:Z = %i (which is %i + %i)" z x y
// Bear with me, there's a point.
```

Let’s start from the beginning…

```fsharp
// The F# compiler is an overprotective parent, and hides a lot
// of the nasty details away from you ('sugaring' for syntactical sugar)
// Unsugared, it really looks like this...
let run () =
    let x = 5 in
        let y = 10 in
            let z = x + y in
                printfn "Step2:Z = %i (which is %i + %i)" z x y

// Let's pretty it up a bit
let run_formatted_nicer () =
    let x = 5 in
    let y = 10 in
    let z = x + y in
    printfn "Step2:Z = %i (which is %i + %i)" z x y
```

Let’s start from the beginning…

```
// Those 'let expr in otherexpr' lines look a lot like funcs...
let run () =
    // let x = y in body => y |> (fun x -> body)
    5 |> (fun x ->
        10 |> (fun y ->
            x + y |> (fun z ->
                printfn "Step3:Z = %i (which is %i + %i)" z x y)))

// Again, let's pretty it up a bit
let run_formatted_nicer () =
    5 |> (fun x ->
    10 |> (fun y ->
    x + y |> (fun z ->
    printfn "Step3:Z = %i (which is %i + %i)" z x y)))
```

Let’s start from the beginning…

```
// Simplify the above with a helper method to pack the pipe (lol)
let pipe (expression, lambda) =
    expression |> lambda

let run () =
    pipe(5, fun x ->
    pipe(10, fun y ->
    pipe(x + y, fun z ->
    printfn "Step4:Z = %i (which is %i + %i)" z x y)))
```

Hey, these look like callbacks!

Remember …

```
// Luckily, there's a better way!
let DownloadPageAsync (url:Uri) =
    async {
        let request = WebRequest.Create(url)
        let! response = request.GetResponseAsync() |> Async.AwaitTask
        use responseStream = response.GetResponseStream()
        use streamReader = new StreamReader(responseStream)
        let! data = streamReader.ReadToEndAsync() |> Async.AwaitTask
        let file = Path.GetTempFileName()
        use outputStream = new StreamWriter(file)
        do! outputStream.WriteLineAsync(data) |> Async.AwaitTask
        return file
    }
```

Additional libraries of interest

- FsCheck: https://github.com/fsharp/FsCheck
- Canopy: http://lefthandedgoat.github.io/canopy/
- FAKE: http://fsharp.github.io/FAKE/
- Paket: http://fsprojects.github.io/Paket/
- Type Providers:
  - Powershell: http://fsprojects.github.io/FSharp.Management/PowerShellProvider.html
  - FSharp.Data: http://fsharp.github.io/FSharp.Data/
  - FSharp.Configuration: http://fsprojects.github.io/FSharp.Configuration/

Questions?

Prof. Dr. Boris Düdder
University of Copenhagen
Copenhagen
Denmark

Email: boris.d@di.ku.dk
Wechat: BorisDuedder
Mobile: +45 93565748

Information:

diku.dk
ebcc.eu
blockchainschool.eu

Co-funded by the Erasmus+ Programme of the European Union

BLOCKNET

Project: BlockChain Network Online Education for interdisciplinary European Competence Transfer
Project No: 2018-1-LT01-KA203-047044