



LambdaConf 2019

# Designing function families and bundles

---

With F# Lambdas and behaviours  
parameterization

Alain Lompo

05/05/2019

User requirements are sinking sands: should we care or should we leave it to the business managers? F#'s behaviours parameterization and lambdas help shift paradigms from business to design: here the audience will learn how to efficiently deal with the issue and make their customers smile.

<b>I.</b>	<b>Introduction</b>	<b>3</b>
<b>II.</b>	<b>Keeping developers happy</b>	<b>3</b>
<b>III.</b>	<b>Why use functional programming</b>	<b>4</b>
<b>IV.</b>	<b>Comparing functional programming and object oriented programming</b>	<b>5</b>
<b>V.</b>	<b>Passing behaviors through interfaces</b>	<b>6</b>
<b>VI.</b>	<b>Core concepts to keep in mind</b>	<b>7</b>
	VI.1 Lambdas expressions	7
	VI.2 Functions signatures and definitions	8
<b>VII.</b>	<b>Lab0: Refreshing the F# core syntax</b>	<b>9</b>
<b>VIII.</b>	<b>Lab1: a glimpse into the power of functional programming with Fsharp</b>	<b>10</b>
<b>IX.</b>	<b>Lab2: Using functions as first class citizens</b>	<b>11</b>
<b>X.</b>	<b>Compositionality</b>	<b>12</b>
<b>XI.</b>	<b>Promoting immutability</b>	<b>13</b>
	XI.1 Cases of refactoring and unit testing	13
	XI.2 Writing non-blocking code using F#	13
	XI.3 Working with immutable values	14
	XI.4 Using immutable data structures	14
	XI.5 Lab 3 – Building the hyperbank core components	14
	XI.5.1 Task 3.1 – Complete relationships in the domain model	15
	XI.5.2 Task 3.2 –Refactoring the code using lambda expression	15
	XI.5.3 Task 3.3 –Refactoring the amount computing function using compositionality	15
	XI.5.4 Task 3.4 –Default enforcement of immutability	15
<b>XII.</b>	<b>Higher order functions</b>	<b>16</b>
	XII.1 Functional types and values	16
	XII.2 Introducing the discriminated union type	16
	XII.2.1 Thinking about problems using functional data structures	17
	XII.3 The purity of F#	17
	XII.4 Tuples, lists and functions	17
	XII.4.1 Value and function declarations	18
	XII.4.2 Value declarations and scope	18
	XII.4.3 Introducing tuple type	18
	XII.4.4 Lists and recursion	19
	XII.4.5 Passing a function as an argument in F#	19
	XII.5 Benefits of parameterized functions	20
	XII.6 Wrap up	21
<b>XIII.</b>	<b>Further diving into higher order functions</b>	<b>21</b>
	XIII.1 using tuples instead of out parameters	21
	XIII.1.1 using tuples compositionally	22
	XIII.2 Generic option type in F#	22
	XIII.3 Type inference in F#	23
	XIII.4 Writing generic functions	24
	XIII.5 Function values	25
	XIII.5.1 What is a function value?	25

XIII.5.2 Lambda functions	26
XIII.5.3 The function type	26
XIII.5.4 Function as an argument and return value	27
XIII.5.5 Functions of multiple arguments	27
XIII.5.6 Partial function application	29
XIII.5.7 Partial function application and currying	29
XIII.5.8 Summary	30
XIII.6 Generic higher order functions	31
XIII.6.1 Generic code in functional and object-oriented programming	31
XIII.6.2 Writing generic functions in F#	32
XIII.6.3 Syntax for writing high order functions	32
XIII.6.4 The F# pipelining operator	33
XIII.6.5 Working with tuples	34
XIII.6.6 Map operations	34
XIII.7 Working with the option type	35
XIII.8 F# library functions	35
XIII.8.1 Using the map function	36
XIII.8.2 Using the bind function	36
XIII.8.3 Working with functions	37
XIII.9 Type inference	37
XIII.9.1 Type inference for function calls in F#	37
XIII.9.2 Automatic generalization	38
XIII.9.3 Understanding type signatures of list functions	39
XIII.9.4 Implementing list functions	39
XIII.10 Common processing language	39
XIII.10.1 The bind operation for lists	39
XIII.11 Data-centric and behavior-centric programs	40
XIII.11.1 Using different data representations	40

## I. Introduction

Our starting point for this workshop is the fact that user requirements are sinking sand. Here are a few solid affirmations we can make about them:

- They always seem good enough at first
- But they will always change later
- The changes happen generally during the course of the implementation
- But sometimes they happen even later

People often blame the methodology for this, but it does not matter what methodology is being used, the problem will still be there. It is however true that depending on the methodology we may find out sooner or later that the requirements should be updated. We generally find about the problem earlier when using agile methods.

It is generally agreed that software maintenance costs will form around 75% of the total cost of ownership. Behavior's parameterization in development phase can help significantly reduce these costs.

Correctly designing our software using functional programming can greatly shrink or even completely remove the issue. In this workshop, the audience will learn how to design families and bundles of functions using functional programming, behaviors parameterization, and the expressive lambdas of F#.

## II. Keeping developers happy

Sometimes, user requirements are obviously not sufficiently detailed. At other times, the underlying complexity is hidden until the initial implementations. It is then a Mary-go-round between the DOING team and the business stakeholders. Yet, at other times the customer simply wants more or a better or finer implementation than what he or she initially considered: without a good design this could be a developer's nightmare. At best, it could demotivate some of them and make them lose interest in the project.

Actually developers feel generally happy when:

- The task has been successfully implemented, tested and set as "Ready for PROD" (at received the green checkmark)

- The task was not specified clearly enough so it is set in a “REQUIRES CLARIFICATIONS” or “BLOCKED” status.

In both cases the developer can move forward with new (and exciting) tasks and live happily ever after.

### III. Why use functional programming

Functional programming has a very good reputation in technical communities. It comes with a lot of interesting features. We mention here a few of the benefits of functional programming:

- It helps us finish our tasks on time and meet deadlines: this is of course due to the conciseness and expressivity of the language. We write much less lines of codes to express the same things in functional programming as in OOP or procedural programming.
- It helps us write correct (and better) code by: enforcing immutability programming patterns and avoiding state handling issues. NPE are also another aspect that can be avoided using functional programming. In functional programming, the focus is on the “intent” or the “what” and not the “how” as it is done in OOP or procedural programming.
- It helps handling more complexity: more complex use cases can be expressed using simpler code, thus allowing us to introduce more advanced and performant algorithms.
- It helps writing efficient and scalable code: with functional programming the same code that is executed in a sequential style can be configured to be executed in a parallel style by simply using the equivalent method. The functional programming languages also generally provide better abstractions for writing reactive and asynchronous code.

#### IV. Comparing functional programming and object oriented programming

Imperative programming was fundamentally sequential and procedural, which means that bigger and complex programs would always necessitate longer sequences of instructions.

Even though object oriented programming made this better it was still sequential in essence. Some aspects of object oriented programming such as event handling (listeners, observers) would give a bird view impression of something non – sequential, but at the core of every unit of processing we still had a sequential approach.

What has been missing in OOP was the ability to simply communicate intent instead of implementing that intent manually. It is the desire to be able to tell the machine what we wanted done and leave it to him instead of guiding him in every step and detail to that realization.

Another interesting aspect is the reduced amount of ceremony necessary before getting the job done.

In OOP everything is object, and to be able to get a behavior processed we have to go through a certain amount of ceremony.

For example, let's consider the following C# class:

```
namespace Financeslib
{
    public class FinancialService
    {
        public double computeInterest(double interestRate, double amount)
        {
            return amount * interestRate;
        }
        public double computeNewAmount(double interestRate, double amount)
        {
            return amount + computeInterest(interestRate, amount);
        }
    }
}
```

And let's look at a client code:

```
FinancialService financialService = new FinancialService();
double interest = financialService.computeInterest(interestRate,
initialAmount);
double newAmount = financialService.computeNewAmount(interestRate,
initialAmount);
```

Sometimes we simply need to use functionality without needing a whole class or an instance of a class.

In functional programming, functions become first class citizens.

## V. Passing behaviors through interfaces

It is possible, in OOP to be able to perform some sort of behavior parameterization. It happens when we define services that take interface type parameters.

For example, let's consider the following interface:

```
namespace Financeslib
{
    public interface IFinanciable
    {
        double computeInterest(double interestRate, double amount);
    }
}
```

Now we can improve the **computeNewAmount** method by passing it a parameter of type **IFinanciable** and call the **computeInterest** method there. Since the call is done on the interface, the behavior is parameterized, the real behavior is known only at runtime.

For example:

```
namespace Financeslib
{
    public class EnhancedFinancialService
    {
        public double computeNewAmount(double interestRate, double amount,
                                       IFinanciable interestCalculator)
        {
            return amount +
                interestCalculator.computeInterest(interestRate, amount);
        }
    }
}
```

## VI. Core concepts to keep in mind

There are two cardinal concepts we will use all along this workshop, and it will pay us to always keep them in mind, these are:

- Lambdas expressions
- Function signatures and definitions

### VI.1 Lambdas expressions

We have seen that it is possible to somehow behaviors through interfaces. However to be able to do that we need to:

- Create an interface
- Create a class that implements the interface (could be explicitly created in a separate class file, could be inline creation or could be an anonymous class)
- Define a method witch signature takes a parameter of type of the interface
- Call the method and pass it a custom behavior (that means an object that is an instance of a class that implements the interface)

What if we could find a way to avoid all that extra stuff and just pass the action that we want done?



The way to do that is to define functions as values, so that we could associate them (the functions, not the result of their execution) to variables and reuse them.

## VI.2 Functions signatures and definitions

In F# the, the concept of function is very similar to what is known in mathematics, and therefore its definition is also closer.

For example to define a mathematic function that produces the square of any value it has been given to, we could define a function such as:

$$F(x) = x^2;$$

The equivalent of such function could be defined as following in F#

```
let f x = x**2
```

As we can see the notation differs a little bit but it is globally the same.

Some of the key benefits of mathematical functions are:

- Mathematical functions always give the same output for the same input
- Mathematical functions have no side effects

These characteristics are emulated by functional programming languages such as F#

The functions that have these given characteristics are also called “pure functions”. Here are a few powerful features that are inherent to pure functions:

- They are parallelizable. Having 16 CPU I could take all integers from 1 to 16 and give each value to each CPU to execute in parallel and do the same with the next sequence of numbers. There is safety because there is no possible interaction between the results of each execution and there is no need for complex multithreading safety handling processes.
- Lazy evaluation are possible since the result of the calling of the function for a given input will still be the same at anytime

- Since the same input always give the same result, I need to evaluate the function for a given value only once. So the result can be cached
- Having a family or a bundle of pure functions, I can evaluate them in any order “without any side effect”: It will make no difference in the results.

We have seen above that we can use the `let` keyword to create a simple function such as:

```
let identity x = x
```

We can also define a function using what is generally known as a “lambda expression”. In that case the function can be anonymous as in the following example:

```
fun x y -> x + y
```

Lambdas expressions are often used when we have a short expression and we do not want to define a more formal function for that expression, it is very common for example with list operations:

```
[1..10] |> List.map (fun i -> i + 1)
```

## VII. Lab0: Refreshing the F# core syntax

This lab is meant for memory refreshment regarding the F# syntax. If you are not familiar with F# or have not used it for a while then you are a good candidate for this lab. If you use F# on a regular basis, you may still find some of the advanced concepts useful. So skip the first sections and check that you have a good understanding of topics such as tuples, pipelines, functions composition, lists and arrays definitions, sequences, recursion and the more.

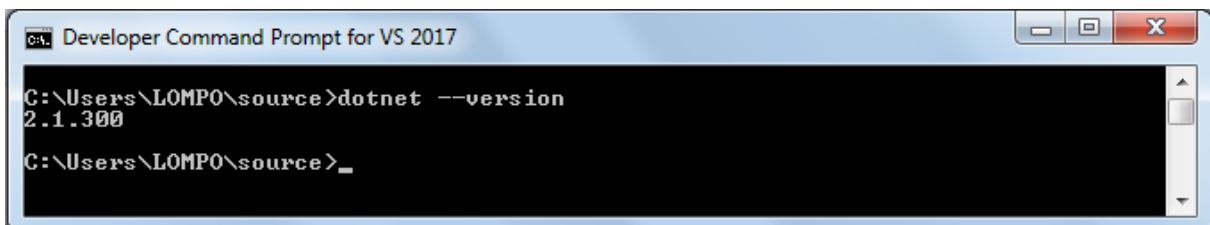
## VIII. Lab1: a glimpse into the power of functional programming with Fsharp

In this first lab you will experiment the power of functional programming with F# first hand. You will use the Fslab data science package for F# to sample what we can do with functional programming in F#.

First ensure that the .net sdk is correctly installed on your machine. Do this execute the following command:

```
dotnet -version
```

You should receive a response similar to:



```
Developer Command Prompt for VS 2017
C:\Users\LOMPO\source>dotnet --version
2.1.300
C:\Users\LOMPO\source>_
```

Now use the following commands to generate a project using the FsLab template:

```
Dotnet new -i FsLab.Templates
Dotnet new fslab-journal -lang F# -n Lab1
```

Navigate to the new Lab1 created folder and generate the html using the command: **build html**

Now use the **build run** command to show and watch the changes

The command should open a web page that displays a content similar to the following:

## Welcome to FsLab journal

FsLab journal is a simple Visual Studio template that makes it easy to do interactive data analysis using F# Interactive and produce HTML or PDF to document your research.

### Next steps

- To see how things work, run `build run` from the terminal to start the journal runner in the background (or hit **F5** in Visual Studio). Executing this project will turn this F# script into a report.
- To generate PDF from your experiments, you need to install `pdflatex` and have it accessible in the system `PATH` variable. Then you can run `build pdf` in the folder with this script (then check out `output` folder).

### Sample experiment

We start by referencing `Deedle` and `XPlot.GoogleCharts` libraries and then we load the contents of *this* file:

```
1: open Deedle
2: open System.IO
3: open XPlot.GoogleCharts
4:
5: let file = __SOURCE_DIRECTORY__ + "/Experiment1.fsx"
6: let contents = File.ReadAllText(file)
```

If it does not open directly try to open manually by navigating to the following url: <http://localhost:8901/Lab1.html>

Read through the web page and notice in particular how functional code snippet are written in F#. Notice also the `|>` used to pipe two or more instructions together.

Use the other options from the build command to generate various types of outputs:

- **build show**: generates html and shows it.
- **build latex**: generate LaTeX output
- **build pdf**: generate Latex and invokes pdflatex

## IX. Lab2: Using functions as first class citizens

One of the fundamental aspects of functional programming languages is the fact of treating functions as first class citizens that implies that:

- Functions can be bound to identifiers
- Functions can be stored in data structures such as lists
- Functions can be passed as arguments in a function call
- Functions can be returned from function calls.

The capacity to send a function as an argument to another function is at the core of common abstractions in functional programming languages such as filter operations, maps, and many more. For example a map operation is a high-order function that uses the computation shared by another function that iterates through a list, process each element and return the result of the processing in the form of a list.

In this lab you will learn to treat functions as first class citizen. You will learn to

- Create functions using lambdas expressions
- Store functions in a list
- Store functions in tuples
- Extract function stored in data structures and apply them
- Pass functions as parameters to other functions
- Returning functions as a result of calling another function
- Returning a composition of functions

## X. Compositionality

An important feature of declarative libraries is that we can use them in a compositional manner. You can move a part of a complex query into a separate function

```
let squareOddValuesAndAddOneComposition =  
    List.filter isOdd >> List.map (square >> addOne)  
  
let fog = fun f g ->  
    fun n -> f (g n)  
  
let expXSquare = fog (fun x -> exp x) (fun x -> x*x)
```

## XI. Promoting immutability

### XI.1 Cases of refactoring and unit testing

Immutability helps us (among other things) to understand what a program does. This is very helpful when refactoring the code. Another interesting functional refactoring is changing when some code actually executes. It may run when the program hits it for the first time, but it may as well execute when its result is actually needed. This way of evolving programs is very important in F# and immutability makes refactoring.

Another area where immutability helps a lot is when creating unit tests for functional programs. The only thing that a method can do in an immutable world is to return a result, so we only have to test whether a method returns the right result for specified arguments.

### XI.2 Writing non-blocking code using F#

It is not surprising to have long running operations in modern software. Http requests are often used to communicate with web services or load data from the internet. In such cases it is very difficult to know when the operation will end, and therefore when the processing is not correctly handled the application will be unresponsive.

Most of the classic techniques make it difficult to write code that performs I/O operations without blocking. F# simplifies it with its asynchronous

```
Let op =  
    async {  
        let req = HttpWebRequest.Create(https://www.allthetests.com)  
        let resp = req.AsyncGetResponse()  
        let stream = resp.GetResponseStream()  
        let reader = new StreamReader(stream)  
        let html = reader.AsyncReadToEnd()  
        Console.WriteLine(html)  
    }  
Async.Run(op)
```

### XI.3 Working with immutable values

In functional programming the notion of variable fades away in favor of the notion of value, thanks to immutability. Functional programming favors immutability and because of that once a “variable” is initialized its value does not change anymore.

### XI.4 Using immutable data structures

Just like variables that become “immutable values”, in functional programming, data structures also become immutable.

Immutable data structures are similar to immutable value bindings. In some programming languages such as C# we can make a class immutable by declaring the fields as read-only. However in F# all data structures are immutable by default. Creating mutable fields requires the use of a special keyword.

### XI.5 Lab 3 – Building the hyperbank core components

Starting with this lab we will build an online banking system called “Hperbank” and will implement and demonstrate the core concepts seen during this workshop.

We will be using a simple online banking domain model provided by “creately” web site:

- <https://creately.com/diagram/example/i12t8eep1/Domain%20Model%20Diagram%20of%20a%20Banking%20System>.

The model is available in the labs resource folder at a jpeg file (called **Online-banking-domain-model.jpg**).

### *XI.5.1 Task 3.1 – Complete relationships in the domain model*

In this first task you are asked to simply complete the relationships in the domain model as described in the domain model mentioned above. Notice that a small refactoring was made regarding two classes

- The ***Banking Software*** class is renamed ***CustomerSession***
- The ***Bank*** class is renamed ***BankAgency***

To complete the task:

- Open the Labs solution with your IDE and edit the source code in the **Hyperbank.Model** library class

### *XI.5.2 Task 3.2 –Refactoring the code using lambda expression*

In the next task your assignment is to simply refactoring the `computeInterest` and `computeNewAmount` functions using a lambda expression. The source code is found in the **Hyperbank.Service** class library.

### *XI.5.3 Task 3.3 –Refactoring the amount computing function using compositionality*

In this task, your assignment is to refactor the `computeNewAmount` function with the use of compositionality. Find a way to write the `computeNewAmount` as the composition of at least two functions.

### *XI.5.4 Task 3.4 –Default enforcement of immutability*

In this task you simply notice that immutability is the default feature of variables bound in F# using `let` and that we need to use a special keyword (*mutable*) in order to create a mutable variable. Inspect for example the ***SystemModel*** module to see how property setters are defined using mutable internal fields.



## XII. Higher order functions

### XII.1 Functional types and values

Some similar languages such as java and C# are statically typed languages: At compilation time, the type of every expression is known. With the help of static typing, the compiler can verify that it will use values consistently. That's how it will know to refuse using the `-` operator to subtract an Integer from a DateTime using the `-` operator.

Even though we generally do not write the types in F#, it is however a statically typed language, but it relies on type inference.

### XII.2 Introducing the discriminated union type

Almost as its name implies, discriminated unions allow us to create new types as a union (or all possible combinations) of some specified other types in exclusion of any other type.

For example let's imagine that we wish to create a sort of tuple where Boolean and integers are associated in all possible combinations: that's a case for discriminated union which can be defined as follow:

```
type BoolOrInt =  
    | B of Boolean  
    | I of int64
```

We can also use it to mix custom types and create new types similar to records

```
type CustomerFile =  
    | CustomerInfos of Customer  
    | Accounts of List<Account>  
    | Transactions of List<Transaction>  
    | AktiveSession of CustomerSession
```

### *XII.2.1 Thinking about problems using functional data structures*

Once you familiarize yourself enough with discriminated unions, you will discover that they can represent a great variety of programming problems.

### **XII.3 The purity of F#**

F# is considered by some as an impure functional language. The reason lies behind the pragmatic approach adopted by F#. F# finds its root and inspiration in O' Caml, Haskell and C# and since it is a “.Net language”, it is supposed to be able to completely interoperate with it as well as with external services and components.

Therefore, F# has to be able to access the object oriented features of .Net platform. Moreover, F# should be able to use such libraries, apis and frameworks as:

- Web development frameworks such as ASP.Net
- Windows user interface frameworks such as Winforms, WPF
- Graphics development frameworks such as DirectX, XNA,

From a global perspective F# is a complete functional programming language with the only difference with others that it is permissive regarding the use of immutable variables. Though it is its preferred way, it is still possible in F# to use mutable variables as we have seen above.

### **XII.4 Tuples, lists and functions**

The data structures available in F# contribute in a very important way to overall power and flexibility of the platform as a complete functional and general purpose programming language. Tuples, lists and functions are among the most cardinal types and structures used in F#

### *XII.4.1 Value and function declarations*

We can do value binding in F# using the `let` keyword. Value binding in F# is an awesome construct that is used for declaring local and global variables as well as functions.

### *XII.4.2 Value declarations and scope*

As we already know, the `let` keyword can be used for declaring immutable values. We haven't yet talked about a scope of the value, but it's easier to do that with a concrete example:

```
let num = 42 #1
printfn "%d" num
let msg = "Answer: " + (num.ToString())
printfn "%s" msg
```

### *XII.4.3 Introducing tuple type*

It is possible in F# to create some tuples. Tuples are very simple type which can group values of different types together. The following example shows how to create a value (called `tuple1`), which contains two values grouped together:

```
let tuple1 = ("Hello world!", 42)
val tuple1: string * int
```

To create a tuple we just write a comma separated list of values enclosed in parentheses. But let's look at the code in more detail - on the first line, we create a tuple and assign it to a `tuple1` value. Type inference takes place here and matching type will be automatically determined.

#### *XII.4.4 Lists and recursion*

The functional list is defined via recursion: a list is either an empty or it is composed from an element and a list.



Every rectangle represents a cell which contains a value and a reference to the rest of the list. Last cons cell contains a special value representing an empty list.

#### *XII.4.5 Passing a function as an argument in F#*

F# supports passing functions as arguments to other functions.

In F# functions are just another type. Just like the type of a tuple is constructed from other types, so is the type of a function defined in terms of the types of its arguments and its return type.

```
> let rec aggregateList (f:int -> int -> int) init list =  
match list with  
| [] -> init #2  
| hd::tl ->  
let rem = aggregateList f init tl #3  
f rem hd #3  
;;  
val aggregateList : (int -> int -> int) -> int -> int list -> int #4  
> let add a b = a + b #A  
let mul a b = a * b #A  
;;  
val add : int -> int -> int #B  
val mul : int -> int -> int #B  
> aggregateList add 0 [ 1 .. 5 ];; #C  
val it : int = 15 #C  
> aggregateList mul 1 [ 1 .. 5 ];; #C  
val it : int = 120 #C
```

Function taking two numbers  
And returning a number

List to be processed

$(\text{int} \rightarrow \text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int list} \rightarrow \text{int}$

Initial value of type int

Returns the aggregated value

### XII.5 Benefits of parameterized functions

Let's consider the following example:

```
> aggregateList max (-1) [ 4; 1; 5; 2; 8; 3 ];;  
val it : int = 8
```

This function could be generalized and that would make it more useful. The actual version does not state anywhere that the aggregated value is an integer. It is only specified in the type annotation for the *f* parameter. It specifies that the *f* function returns an integer. By removing the return type annotation we would make this code more general: this is the F# concept of generalization.

### XII.6 Lab 4 - Injecting parameterized behaviors

In the previous labs we have used some simple interest computing formulas. Knowing the generated interest allows us to compute the new value of the balance. Our interest computing formula is fixed, but what if the bank would introduce a dynamic coefficient that would depend on the volume of

transactions generated by a customer? Actually the last discussion with the PO suggested that this type of changes is included in the roadmap.

In order to make our code flexible and able to adapt to the change efficiently, we will refactor our code.

So your task in this step is to define a high order function called `calculateNewBalance` that takes a `calculateInterest` function as argument and produces the equivalent result as the function used in the last exercise.

## XII.7 Wrap up

Here are a few key concepts we have seen in this part:

- ✓ We can do value and function declaration and bindings via the `let` keyword
- ✓ Strictly (mathematically) speaking, an immutable value is just a function with zero arguments
- ✓ F# provides some simple immutable data structures such as
  - The tuples
  - The lists
  - The parameterization of a function by another function. This is called higher order function, which simply means that a function can be parameterized by another function passed to it as argument.

## XIII. Further diving into higher order functions

### XIII.1 using tuples instead of out parameters

F# automatically exposes .NET methods with "out" parameters methods that return a tuple. So even if your F# code is calling into .Net code which has no concept of tuples it will still look like normal functional code.

### *XIII.1.1 using tuples compositionally*

The key concern when thinking about what kind of a tuple should be returned from a function is compositionality. How do you expect the tuple to be used? What other functions might use a tuple of the same type? Is this consistent with similar situations in the rest of the program?

### **XIII.2 Generic option type in F#**

Just as in C#, we declare the type with a type parameter and then use that as the type of the value stored in the **Some** alternative:

```
type Option<'T> =  
| Some of 'T  
| None
```

The syntax for declaring generic type is similar to that used in C#—we write type parameters in angle brackets. Unlike in C#, we have to use special names for type parameters so the name of the type parameter always starts with an apostrophe.

When creating an instance of generic class in C# or a value of generic type in F#, the type parameter is "replaced" by the actual type used when creating the value. In C#, you have to specify the type explicitly when calling the constructor, but in F# the type argument is usually inferred by the compiler. Let's look at an example:

```
> Some("Hi there!");;  
  
val it : Option<string> = "Hi there!"
```

### XIII.3 Type inference in F#

In F#, we can often write large snippets of code without explicitly specifying any types, because the type inference mechanism is more sophisticated. When creating values, we use the `let` keyword and, in fact, we haven't yet seen any example where we would need to specify the type explicitly with a `let` binding. The following snippet shows some examples that you'd probably expect to work.

```
let num = 123 #A
let tup = (123, "Hello world") #B
let opt = Some(10) #C
let input = printfn "Calculating..." #D
            if (num = 0) then None
            else Some(num.ToString())
```

Out of these bindings, only the last example is particularly interesting or surprising. As we already know, everything in F# is an expression, so type inference has to work with any F# expressions (meaning any F# code, because everything is an expression). In this case, we have code that first prints something to screen and then returns an option type using a conditional expression. Note that whitespace is significant in F#'s lightweight syntax, so the `if` expression should start at the same offset as the `printfn` call.

```
let a = null #1
let (a:System.Random) = null #2
let a = (null:System.Random) #2
```

In the first case (#1), F# behaves differently to C#. Instead of reporting an error, F# automatically chooses the most general reference type, which is in this case .NET type `System.Object` abbreviated as `obj` in F#. The next two examples (#2) show two different ways for adding a type annotation. In general, you can place type annotation around any block of F# code if you need to. Now, let's look at one more interesting case:

```
> let n = None
val n : 'a option
```



Interestingly, this doesn't cause an error and instead F# creates a generic value. This construct doesn't have a C# equivalent; it's a value with only partially specified type. Instead of a concrete type (such as `int` or `obj`), F# uses a type parameter (and you can see that F# automatically names type parameters using letters starting with "a"). The type is fully specified later when using the value. We can for example compare the value with different types of option values without getting an error:

```
> Some("testing...") = n;;  
val it : bool = false
```

```
> Some(123) = n;;  
val it : bool = false
```

### XIII.4 Writing generic functions

Most of the functions or methods that work with generic types are higher order, which means that they take another function as an argument. This is a very important topic, but we can already write a generic function without straying into higher order territory. We'll create a function that takes an option type and returns the contained value when it contains a value. If the option doesn't contain a value, the function throws an exception. We can start by looking at the C# version:

```
T ReadValue<T> (Option<T> opt) { #A  
    T v;  
    if (opt.MatchSome(out v)) return v;  
    else throw new InvalidOperationException();  
}
```

As you can see, we have created a generic method with a single type parameter. The type parameter is used in the method signature as a return value and also as a parameter to the generic `Option<T>` type. Inside the body, we use it once again to declare a local variable of this type.

### XIII.5 Function values

Working with collections of data is probably the best way of showing why using functions as values are important. Having said that, it's far from the only scenario where this concept is useful. Let's start by looking at an example of imperative code that selects even numbers from the given collection and returns them in another collection:

```
var nums = new int[] {4,9,1,8,6};  
var evens = new List<int>( ); #A  
foreach(var n in nums) #A  
if (n%2 == 0) #B  
    evens.Add(n); #A  
return evens;
```

In functional programming languages, the existence of functions is motivated by mathematical notion of a function. This is in many ways different to the way that programmers with an imperative background intuitively think about functions. In imperative programming, a function is a routine that takes some arguments, executes some code and returns the result. However a function in this sense can do anything. Most importantly, it can use and modify global state, so the result of calling the same function with exactly the same arguments can differ. The most obvious example of this is probably a pseudo-random number generator - it wouldn't be very random if it always returned the same result!

In math, a function is more a relation between the arguments and the result. This means that a mathematical function always returns the same result given the same arguments.

Clearly, this is the way our predicate from the previous example works. It always returns the same result for the same argument (true for even numbers and false for odd ones). Most of the functions we'll write will behave like this, but we'll see some interesting and useful exceptions from this rule at the end of the next section. You might like to think about what a mathematical pseudo-random number generator function would have to look like...

For those who come from an object-oriented background, there is one more way to look at functions. You can think of a function value as an object implementing a really simple interface with just a single method. Using this understanding, the predicate from the previous example corresponds to the following interface:

### *XIII.5.2 Lambda functions*

In F#, lambda functions create exactly the same function as the usual declaration using a let binding. In C#, there is no built-in concept of a function, so we work either with methods or with delegates. When you write a lambda function it is converted into a delegate or an expression tree, but you cannot use lambda function in C# to declare an ordinary method. Vitrally, delegates can be also used like any other value in C#, so you can pass them as arguments to other methods, which in turn means we can use them to write higher order functions in C#. Let's start by looking at a short F# interactive session, then write similar code in C#.

### *XIII.5.3 The function type*

We've seen that the type of function values in F# is written using the arrow symbol. This is in many ways similar to the way tuples are constructed. Earlier, we've seen that a tuple type can be constructed from other simpler types using a type constructor with an asterisk (e.g. `int * string`). The function type is constructed in a similar way, but using the function type constructor (e.g. `int -> string`). Of course, there is no value constructor for functions. In some sense, a function is a relation that specifies return value for every possible input, so instead of specifying enormous number of all combinations of this relation, we specify code that calculates the result using lambda functions.

In C#, you can see this similarity as well. If we use our generic Tuple type and Func delegate, we can write the examples from previous examples as `Tuple<int, string>` and `Func<int, string>`. Instead of using built-in types as we can in F#, we have similar constructs implemented as ordinary C# types using generics. However, there is a very important difference between the F# function

type and C# Func (or Action) delegate. The difference is that the type of an ordinary F# function is exactly the same as the type of an equivalent function written as a lambda function. In C#, lambda functions are converted into delegates and a delegate isn't the same thing as method. The distinction is subtle but important: we'll see it more clearly when we consider functions with multiple parameters in F#. Before that, let's look how we can use function value as an argument or a return value.

#### *XIII.5.4 Function as an argument and return value*

Lambda functions are the easiest way to write a function that is just used as an argument to another function. The next code snippet provides a simple example. The function at the start of the listing takes a number and a function as arguments and calls the function twice, using the result of the first call as an argument for the second.

```
> let twice n (f: int -> int) = f(f(n))  
val twice :  
    int -> (int -> int)  
    -> int  
> twice 2 (fun n -> n * n)  
val it : int = 16
```

#### *XIII.5.5 Functions of multiple arguments*

First of all, let's quickly review what options we have when writing a function. In F#, we can use tuples when writing functions with multiple arguments. Let's look at an example of a function that adds two integers written in this style. I'll use the lambda function syntax, but you could get exactly the same results using simple let-binding in F# as well.

```
> let add = fun (a, b) -> a + b;;  
val add : int * int -> int
```

As you can see by looking at the type signature, the function takes a single argument which is a tuple of the form `(int * int)` and the return type is `int`. This corresponds to the C# lambda function written in this form:

```
Func<int, int, int> add =(a, b) => a + b
```

The `Func<int, int, int>` delegate represents a method which has two arguments of type **int** and returns an **int**, so this is very similar to the F# version written using tuples. You can see this similarity when calling the functions as well:

```
let n = add(39, 44) #A
```

```
var n = add(39, 44) #B
```

```
#A F#: Calling a function with type int * int ->
int
```

```
#B C#: Calling a Func<int, int, int> delegate
```

The syntax is exactly the same to call an F# function with a tuple as an argument as it is to call a C# Func delegate. Now, let's write the same code in F# using the more idiomatic F# style for writing functions with multiple arguments:

```
> let add = fun a b -> a + b;;
val add : int -> int -> int
```

This is the same signature we saw earlier when we were returning a function. We can read it as `int -> (int -> int)`. That would be a function that takes the first argument for the addition and returns a function. The result is then a function taking the second argument.

We can rewrite the code in this way using two lambda functions, nesting one inside the other.

### *XIII.5.6 Partial function application*

To show a situation where this new understanding of functions is useful, let's turn our attention back to lists. Imagine that we have a list of numbers and we want to add 10 to every number in the list. In F# this can be written using the `List.map` function; in C# we would use the **Select** method from LINQ:

```
list.Select(n => n + 10) #A
List.map (fun n -> n + 10) list #B
#A C# version
#B F# version
```

That's pretty brief already, but we can be even more concise if we already have the `add` function from the previous examples. The function that `List.map` expects as a first argument is of type `int -> int`; that is a function taking an integer as an argument and returning another integer. The technique that we can use is called partial function application:

```
> let add a b = a + b;;
val add : int -> int -> int #1
> let addTen = add 10;;
val addTen : int -> int #2
> List.map (addTen) [ 1 .. 10 ];; #3
val it : int list = [11; 12; 13; 14; 15; 16; 17; 18; 19; 20]
> List.map (add 10) [ 1 .. 10 ];; #4
val it : int list = [11; 12; 13; 14; 15; 16; 17; 18; 19; 20]
```

The `add` function has a type `int -> int -> int`. Since we now know that it actually means that the function takes an integer and returns a function, we can simply create a function `addTen` (#2) that adds 10 to a given argument just by calling `add` with only the first argument. We can then use this function as an argument to the `List.map` function.

### *XIII.5.7 Partial function application and currying*

A term that you can sometimes hear when using the partial function application is currying. This refers to converting a function that takes multiple arguments (as a tuple) into a function that takes the first argument and returns a

function taking the next argument and so on. So, for example the function of type `int -> int -> int` is a curried form of a function that has a type `(int * int) -> int`. Partial function application is then the use of a curried function without specifying all the arguments.

### *XIII.5.8 Summary*

In this section we've been talking about values—the fact that the discussion went into a lot of detail about functions just highlights the fact that in F# functions are values! We've seen several ways for creating different values and corresponding composed types. We started by looking at tuples, which gave us a way to store multiple values as one. Next, we've seen discriminated unions that allow us to represent values consisting of various alternatives. When declaring discriminated union, we specify what are the options and a value can then be one of the declared options. We also looked at generic types that are similar to generic classes in C#. We've used them to declare types that can be used for carrying different values, which makes the code more general and reusable.

As well as looking at the theory behind these types, we've looked at some of the common uses of them in F#. We've seen that multiple values (tuples) are useful for returning multiple results from a single function, and how this can be more appealing than using C# "out" parameters. A particularly interesting alternative value (discriminated union) is the option type, which can represent values that can be undefined. This is a very useful alternative to using null values, as the language forces the calling code to write a case that handles the "undefined" case when we use pattern matching.

Finally, we looked at the function type in F# and its equivalent in C# - the Func delegate. We've seen how functions can be created using lambda function syntax and how they can be used as arguments as well as return values from another function or a method.

In one last twist to function values, we've also seen a very useful technique called partial function application.

In this section we've seen only the basic ways for working with values. This is because many of the operations aren't usually written directly and are instead use higher order functions. Working with values in this way is the main topic for

the next section. Using higher order functions, we'll be able to hide the logic for working with the value in a function and specify just the most important part of the operation using a function value given as an argument.

### **XIII.6 Generic higher order functions**

Higher order functions are a way for writing generic functional code, meaning that the same code can be reused for many similar but distinct purposes. This is a key of modern programming, because it allows us to write fewer lines of code by factoring out the common part of the computation.

#### ***XIII.6.1 Generic code in functional and object-oriented programming***

When writing generic code, we usually want to perform some operation on the value that we obtain, but since the code should be generic, we don't want to restrict the type of the value too much: we want to allow further extension of the code.

The elementary (but not always the best) solution to this problem using object-oriented programming is to declare an interface. The actual value given to a method will have all operations required by the interface, so it will be possible to perform needed operations on the value. A trivial example in C# might look like this:

```
interface ITestAndFormat {  
    bool Test();  
    string Format();  
}  
  
void CondPrint(ITestAndFormat tf) {  
    if (tf.Test()) Console.WriteLine(tf.Format());  
}
```

In functional programming, the approach is usually to work with generic methods that use type parameters and can work with any type. However, we



don't know what operations can be performed on the value, since the type parameter can be substituted by any actual type. As a result, functional languages use a different method for specifying operations - they pass functions for working with the value as additional arguments. The functional version of the previous example in C# would look like this:

```
void CondPrint<T>(T value, Func<T, bool> test, Func<T, string> format) {  
    if (test(value)) Console.WriteLine(format(value));  
}
```

For a small number of functions this is a very efficient method, because we don't need to declare the interface in advance. However, for more complicated processing functions, we can still use interfaces as we'll see later. Also calling the function is easier, because we can implement the operations using lambda functions.

### *XIII.6.2 Writing generic functions in F#*

Through generic the specification of the data type of elements in classes or methods can be delayed until the moment where it is actually used in the program.

```
let makeList a b = [a; b]  
let function1 (x: 'a) (y: 'a) = printfn "%A %A" x y  
let function2<'T> x y = printfn "%A, %A" x y
```

### *XIII.6.3 Syntax for writing high order functions*

Previously, we were discussing whether it is better to pass multiple pieces of data to a function as separate arguments (for example add 2 3) or as a tuple (for example add (2, 3)). When writing higher order functions, we'll use the first

style, because this makes it easier to use lambda functions as arguments. It also supports the pipelining operator, which we'll see shortly.

#### *XIII.6.4 The F# pipelining operator*

The pipelining operator (`|>`) allows us to write the first argument for a function on the left side; that is, before the function name itself. This is useful if we want to invoke a several processing functions on some value in sequence and we want to write the value that's being processed first. Let's look at an example, showing how to reverse a list in F# and then take its first element:

```
List.hd(List.rev [1 .. 5])
```

This isn't very elegant, because the operations are written in opposite order then in which they are performed and the value that is being processed is on the right side, surrounded by several braces. Using extension methods in C#, we'd write:

```
list.Reverse().Head();
```

In F#, we can get the same result by using the pipelining operator:

```
[1 .. 5] |> List.rev |> List.hd
```

Even though, this may look tricky, the operator is in fact very simple. It has two arguments - the second one (on the right side) is a function and the first one (on the left side) is a value. The operator gives the value as an argument to the function and returns the result.

In some senses, pipelining is similar to calling methods using dot-notation on an object, but it isn't limited to intrinsic methods of an object. This is similar to extension methods, so when we write a C# alternative of an F# function that's usually used with the pipelining operator, we'll implement it as an extension method.

### *XIII.6.5 Working with tuples*

We have used tuples before. However, we haven't looked at how we can work with them using higher order functions. Tuples are really simple, so you can often use them directly, but in some cases the code isn't as concise as it could be. Tuples are a good starting point for exploring higher order functions because they're so simple. The principles we'll see here are applicable to other types, too. As a simple usage example, we could use tuples to increment the population like this:

```
let (name, population) = oldNuremberg
let newNuremberg = (name, population + 13195)
```

This is very clear, but a bit longwinded. The first line deconstructs the tuple and the second one performs a calculation with the second element and then builds a new tuple.

Ideally, we'd like to say that we want to perform a calculation on the second element deconstructing and re-constructing the tuple. First let's quickly look at the code we want to be able to write, in both F# and C#, and then we'll implement the methods which make it all work. This is what we're aiming for:

```
let newPrague = oldPrague |> mapSecond ((+) 13195) #A
var newPrague = oldPrague.MapSecond(n => n + 13195); #B

#A F# version
#B C# version
```

This version removes all the additional code to re-construct the tuple and specifies the core idea - that is, we want to add some number to the second element from the tuple. The idea that we want to perform calculation on the second element is expressed by using the `mapSecond` function in F#.

### *XIII.6.6 Map operations*

I used the term `map` in the name of the functions above. A `map` (also called a `projection`) is a very common operation and we'll see that we can use it with many data types. In general, it takes a function as an argument and applies this

function to one or sometimes more values that are stored in the data type. The result is then wrapped in a data type with the same structure and returned as a result of the map operation. The structure isn't changed, because the operation we specify doesn't tell us what to do with the composed value. It specifies only what to do with the component of the value and without knowing anything else the projection has to keep the original structure. This description may not be fully clear now, because it largely depends on the intuitive sense that you'll get after more similar operations later in this section.

### XIII.7 Working with the option type

One of the most important alternative values in F# is the option type. To recap what we've seen in the previous sections, it gives us a safe way to represent the fact that value may be missing. This safety means that we have to explicitly pattern match on option whenever we want to perform some operation with the actual value. In this section, we'll learn about two useful functions for working with the option type.

### XIII.8 F# library functions

The functions we saw earlier for working with tuples aren't part of the F# library, because they are extremely simple and using tuples explicitly is usually easy enough. However, the functions we'll see in this section for working with the option type are part of the standard F# library.

First of all, let's quickly look at an example that demonstrates why we need higher order operations for working with the option type. We'll use the `readInput` function, which reads user input from the console and returns a value of type `option<int>`. When the user enters a valid number, it returns `Some(n)`; otherwise it returns `None`.

### *XIII.8.1 Using the map function*

I'll first introduce both of the operations and first show you how to use them from F#, where they are already available in the F# library. Later we'll also look at their implementation and how we can use them from C#. As we've already seen, the best way to understand what a function does in F# is often to understand its type signature. Let's first look at `Option.map`:

```
> Option.map;;  
val it : (('a -> 'b) -> 'a option -> 'b option) = (...)
```

I said that map operations usually apply a given function to values carried by the data type and wrap the result in the same structure. For the option type, this

means that when the value is `Some`, the function given as the first argument (`'a -> 'b`) will be applied to a value carried by the second argument (`'a option`) and the result of type `'b` will be wrapped inside an option type, so the overall result has type `'b option`. When the original option type doesn't carry a value, the map function will simply return `None`.

We can use this function instead of the nested match. When reading the second input, we want to 'map' the carried value to a new value by adding the first number:

### *XIII.8.2 Using the bind function*

As a next step, we'd like to eliminate the outer pattern matching. Doing this using `Option.map` isn't possible, because this function always turns input value `None` into output value `None` and input value `Some` into output `Some` carrying another value.

```
let readAndAdd2() =  
    readInput() |> Option.bind (fun num -> #1  
    readInput() |> Option.map ((+) num) ) #2
```

After reading the first input, we pass it to the bind operation (#1), which executes the given lambda function only when the input contains a value. Inside this lambda function, we read the second input and project it into a result value (#2). The operation used for projection just adds the first input to the value. In this listing, we've written it using the plus operator and partial application instead of specifying the lambda function explicitly. Let's now analyze how it works in some more detail.

### *XIII.8.3 Working with functions*

The high orders functions we talked about are similar in structure: they have two parameters which are a value to process and a function that implements the behavior that is used to process the value.

In functional programming the value can also be a function, so that in that case we have a high order function that takes two functions as arguments.

### *XIII.9 Type inference*

When calling a generic method we can specify the types of arguments explicitly:

```
var dt = Option.Some<DateTime>(DateTime.Now);  
dt.Map<DateTime, int>(d => d.Year);
```

Generally type inference allows the types to be automatically deduced; like:

```
dt.Map(d => d.Year)
```

#### *XIII.9.1 Type inference for function calls in F#*

We rarely use specific type information in F#, even though it is possible. In fact when the compiler cannot infer all the information and there is a need for the programmer to aid further, add type annotation can be used at the specific location where it is needed.

Let's demonstrate this using an example:

```
> Option.map (fun v -> v.Year) (Some(DateTime.Now));;  
error FS0072: Lookup on object of indeterminate type.  
> Option.map (fun (v:DateTime) -> v.Year) (Some(DateTime.Now));;  
val it : int option = Some(2008)
```

In the first case the compiler does not know the type of the value `v` until it reaches the second argument, therefore it does know nothing about the `Year` property.

By adding the type annotation we can fix this. In case we include the pipeline operator in we might not even need to use the type annotation.

### *XIII.9.2 Automatic generalization*

Through the use of automatic generalization we can implement high order functions without needing to specify the types at all.

Automatic generalization, which is used when inferring the type of a function declaration. I'll explain how this process works using an implementation of the `Option bind` function as an example:

```
let bind func value = #1  
match value with #2  
| None -> None #3  
| Some(a) -> func(a) #4
```

### *XIII.9.3 Understanding type signatures of list functions*

It is possible to deduce what a high order list function (such as those used for filtering or projecting) does just by understanding their type signatures.

### *XIII.9.4 Implementing list functions*

Let inspect the RemoveAllMultiples function below. The function takes two lists A and B and removes from B every multiple of the elements of A.

```
let IsPrimeMultipleTest n x =  
    x = n || x % n <> 0  
let rec RemoveAllMultiples listn listx =  
    match listn with  
    | head :: tail -> RemoveAllMultiples tail (List.filter  
(IsPrimeMultipleTest head) listx)  
    | [] -> listx  
let GetPrimesUpTo n =  
    let max = int (sqrt (float n))  
    RemoveAllMultiples [ 2 .. max ] [ 1 .. n ]  
printfn "Primes Up To %d:\n %A" 100 (GetPrimesUpTo 100)
```

## **XIII.10 Common processing language**

We've seen a few recurring patterns over the course of this section, such as an operation called "map" which was available for both option values and lists. Actually, we also used it when we were working with tuples and implemented the mapFirst and mapSecond functions.

### *XIII.10.1 The bind operation for lists*

The bind operation is an extremely important functional operation.

```
Option.bind : ('a -> 'b option) -> 'a option -> 'b option  
List.bind : ('a -> 'b list) -> 'a list -> 'b list
```



The function `List.bind` is available in the F# library under a different name, so let's try to figure out what it does, just using the type signature. The input is a list and for each element, it can obtain a list with values of some other type. A list of this type is also returned as a result from the bind operation.

### **XIII.11 Data-centric and behavior-centric programs**

Most of functional programs are data centric, but some applications are however rather focused on behavior. Let's imagine for example that we are developing an application that does batch processing of images. It would be using filters are therefore will have a list of filters as data structure. A filter is from a functional programming point of view, just another function.

#### ***XIII.11.1 Using different data representations***

In functional programming we often use multiple data structures to represent the same program data. Transformations will help navigate between the various data representations.

### **XIV Parameterizing behavior's patterns**