# An Introduction to F#, or: Writing Simple Code to Solve Complex Poblems


Marc Sigrist
Zurich F# Users Meetup (fsharp.ch)

Originally presented at SoftShake Conference 2013, Geneva

# F# History

| | | | |
|---|---|---|---|
| 1958 | Lisp | Pioneers functional programming, garbage collector, … | John McCarthy, Stanford University |
| 1973 | ML | Pioneers generics = parametric polymorphism | Robin Milner, University of Edinburgh |
| 1996 | OCaml | ML with object orientation | Xavier Leroy, INRIA, France |
| 2002 | .NET 1.0 | Without generics | Microsoft Corp. |
| 2005 | .NET 2.0 | With generics | Don Syme et. al., Microsoft Research, Cambridge/U.K., add generics to the Common Language Runtime and C# compiler |
| 2005 | **F# 1.0** | Preview release | Don Syme, Microsoft Research, Cambridge/U.K. |
| 2007 | | VS “major language” decision | New: async { } expressions, … |
| 2010 | **F# 2.0** | VS 2010 release | New: Units of measure e.g. 192.84<km/h>, … |
| 2012 | **F# 3.0** | VS 2012 release | New: Type providers, query expressions, auto properties with initializers, … |
| 2013 | **F# 3.1** | VS 2013 release | New: Named union fields, operators with tooltips and “go to definition” support in the editor, … |

# Some Features of F#

who let you write simple code to solve complex problems… with fewer bugs.

Two fundamental bug-reducing features:
Immutability and Non-nullability

**Immutability**

```
// “Variables” and fields are immutable by default:
let s = "Hello world"

// To allow a variable to be mutable, it has to be explicitly declared so:
let mutable s = "Hello world"

// Then, to modify the variable, one has to apply a distinctly recognizable operator:
s <- "Goodbye world"
```

**Non-nullability**

Null references were introduced in 1965 by Tony Hoare, a distinguished British computer scientist who invented the quicksort algorithm and implemented ALGOL.

In 2009, he gave a lecture titled “Null references: The Billion Dollar mistake”.

```
// In F#, references of types defined in F# cannot be null.
type Account(value:decimal) =
    member this.Value = value

let account: Account = null
```

The type 'Account' does not have 'null' as a proper value

```
// If something may or may not exist, one uses the Option<…> type:
let account: Account option = None
```

| Working with Functions in F# | |
|---|---|
| **Feature** | Lets you… |
| **Functions as values** | initialize functions and pass them as arguments like any other values |
| **Anonymous functions (lambdas)** | define ad-hoc functions locally where they are needed |
| **Currying and partial function application** | call a function with an incomplete set of arguments. The result is a new function who “already knows” the received arguments (from the generated closure) and expects only the remaining arguments in its own signature. |
| **Operator-supported function composition** | compose and combine functions from other functions |
| **Custom operators** | define your own symbolic unary or binary operators, e.g.<br>.+ $ \|^ --> etc.<br>- as members of a type,<br>- as library functions, or<br>- as local functions |
| **Tail call optimization** | write recursive functions without risking a stack overflow |

| Working with Types in F# | |
|---|---|
| **Feature** | Lets you… |
| **Type inference** | write strongly-typed generic code with very few type annotations.<br>Introduced in the 70es in ML by Robin Milner |
| **Syntactic type support for unions, tuples, records, lists, arrays, and sequences** | write code who immediately expresses the business/algorithmic problem, without distraction by plumbing |
| **Pattern matching** | decompose/analyze expressions and branch program flow at the same time (“switch statement on steroids”) |
| **Active patterns** | compose and decompose patterns with unlimited flexibility<br>(F# innovation as a built-in language feature) |
| **Units of Measure** | write type-safe numeric code (never add meters to hours…)<br>(F# innovation) |
| **Type providers** | access schema-based data sources<br>- in a type-safe way<br>- even if the data source “contains” thousands of different types and members<br>- simply by adding a reference to the type provider for the schema (no code generation or any other intermediate steps involved)<br>(F# innovation) |

| Working with Abstract Computations in F# (a.k.a. “monads”, “monoids”, etc.) | |
|---|---|
| **Feature** | Lets you… |
| **Sequence computation expressions** | write blocks of **seq { }**, **[ ]**, and **[\| \|]** code with iterators to<br>• create enumerables (lazily evaluated)<br>• create lists and arrays (eagerly evaluated)<br>(C# 2.0 in 2006: return enumerables from methods with yield … syntax) |
| **Query computation expressions** | Write blocks of **query { }** code with SQL keywords<br>(C# 3.0 in 2007: LINQ syntax) |
| **Async computation expressions** | write blocks of **async { }** code as if it were synchronous code<br>(C# 5.0 in 2012: async method syntax) |
| **User-defined computation expressions** | Define your own computation expression *builder* types so you can then write custom computation expressions, e.g. maybe { } , cont { }, … |
| **Unified approach and syntax** | Works the same for different kinds of built-in or user-defined abstract computations |

# Some Currently Available Type Providers

| Available From: | Type Providers: |
| --- | --- |
| Six type providers from Microsoft in Microsoft.FSharp.Data.TypeProviders | SqlDataConnection<br>SqlEntityConnection<br>WsdlService, ODataService<br>DbmlFile, EdmxFile |
| Ca. fourty additional type providers, mostly open source, such as: | Xml, Xaml, Json, Csv<br>Regex<br>AppSettings<br>PowerShell, Python, Matlab, R<br>Word, Excel<br>Hadoop<br>Azure<br>etc…. |

# Further information on F#

- F# Software Foundation (fsharp.org)
- F# on MSDN (fsharp.net)
- Zurich F# Users Meetup (fsharp.ch)