F Sharp

By Kyle Hunter, Ian Martin, Aaron Ronzo, and Matt Johnson

Overview

- Introduction
- History/Purpose
- Main Features
- Code Examples
- Live Coding Session
- Conclusion

History

- Two separate projects being worked on
  - A team at Microsoft Research @ Cambridge wanted a metalanguage for the .NET platform
  - Don Syme working on implementing generics for .NET
- Eventually these two projects were combined to create F#
  - First release: 2005
  - Version 2.0: 2010
  - Version 3.0: 2012

History

- Version 2.0
  - Removal of deprecated functionality
  - Async API improved for performance & stability
  - Reduce size of library
  - Improved support for F# compiler on other OSes
- Version 3.0
  - Units of measure type (SI units)
  - Type Providers (generate types based on structured data)
  - Query Expressions (LINQ – SQL-like queries)
  - Parameter help and improved Intellisense in the IDE

Introduction

- Part of the .NET Framework
  - Easily integrate with other .NET languages (C#, C++, Visual Basic)
- Variant of ML (MetaLanguage)
  - Largely compatible with OCaml (#light)
- Multi-paradigm programming language
  - Primarily functional

Purpose

- To combine multiple programming paradigms into one language
- To provide a functional language for the .NET platform
- Less overhead for scientists and mathematicians

Features

- Programming Paradigms
  - Functional
  - Imperative
    - Control flow, I/O, Mutable Data, Exception Handling
  - Object Oriented
    - Data encapsulation, inheritance, polymorphism, type extensions
- Qualities
  - Strongly typed
    - With type inference
  - Immutable w/support for mutable data
  - Eager evaluation w/support for lazy evaluation
  - Easy (but not automatic) parallelism

Features

- Functional Programming
  - Functions are values too
  - Currying
  - Function compositions and pipelining
  - Type inference
  - Pattern matching
  - Lamba expressions/anonymous functions
- Tuples
- Records

Basics

Creating a function

```
let addOne (x : int) = x + 1

let addOne x = x + 1

val addOne : int -> int
```

Creating a list

```
let list = [1..10]
```

Mapping Data

```
let data = [1..10]

let square x = x * x

let result = List.map square data

printfn "%A" result
```

Attach item to list

```
let names = ["Kyle", "Aaron",
             "Matt"]

let fullNames = "Ian" :: names
```

Mutable Data

```
let mutable x = 5
val mutable x : int
x <- 10
```

```
let names = [| "Kyle"; "Aaaron";
               "Ian"; "Matt" |]
names.[1] <- "Aaron"
```

```
let x = ref "Hello"
val x : string ref

x //returns ref instance
!x //returns x.contents
x := "Goodbye"
```

Imperative & OO

```
let mutable res = 2

for n = 1 to 10 do
    res <- res * n
    printfn "%d" res
```

```
type Player(n : int) = class
    let mutable health = n

    member x.printHealth() =
        printfn "Health: %d" health

    member x.hitByGoblin(damage) =
        health <- health - damage
end

let kyle = new Player(300)
let aaron = new Player(300)

aaron.printHealth()
aaron.hitByGoblin(100)
aaron.printHealth()
```

Pipeline Operator & Function Comp.

Pipeline Operator

```
let square x = x * x
let add x y = x + y
let toString x = x.ToString()

let complexFunc x =
    toString (add 5 (square x))

let complexFunc2 x =
    x |> square |> add 5
    |> toString
```

Function Composition

```
let f x = x + 5
let g x = x * x
let fog = f << g //x^2 + 5
let gof = f >> g //(x+5) * (x+5)
```

Lamba Expressions & Currying

```
let complexFunc =
        2 |>
        ( fun x-> x * x ) |>
        ( fun x-> x + 5 ) |>
        ( fun x -> x.ToString() )
```

```
let multiply' (x, y) = x * y
let multiply x y = x * y

let double' x = multiply (2, x)
let double = multiply 2
```

Factorial (3 Examples)

(1)

```
let rec factorial n =
    if n = 0 then
        1
    else
        n * factorial (n - 1)
```

(2)

```
let rec factorial n =
    match n with
    | 0 -> 1
    | _ -> n * factorial (n - 1)
```

(3)

```
let factorial n =
    [1..n] |> List.fold (*) 1
```

Tuples & Generics

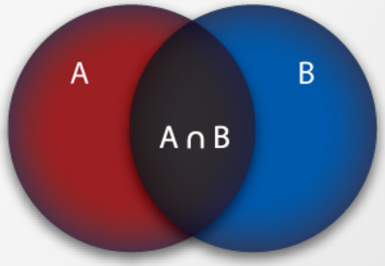
```
let swap (a, b) = (b, a)
val swap : 'a * 'b -> 'b * 'a
```

```
let divrem x y =
    match y with
    | 0 -> None
    | _ -> Some(x / y, x % y)
```

Sets

```
let x = Set.ofSeq [ 1..30 ]
let y = Set.ofSeq [ 5..15 ]
let z = Set.ofSeq [ 31..35 ]

Set.iter (fun x -> printf "%d " x) (Set.intersect x y)
Set.iter (fun x -> printf "%d " x) (Set.union x z)
printf "%A" (Set.isSubset y x)
```



Records

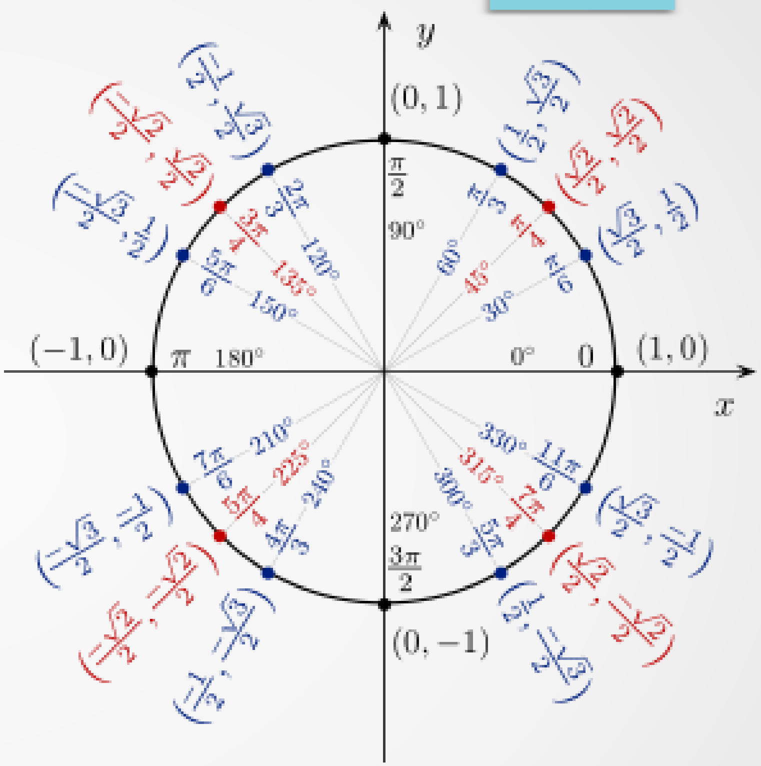
```
type circle = {
    XOrigin : float;
    YOrigin : float;
    Radius : float;
}

let getDiameter circle =
    circle.Radius * 2.0

let getPoints circle (rot : float) =
    (circle.XOrigin + circle.Radius * cos rot,
     circle.YOrigin + circle.Radius * sin rot)

let bigCircle = { XOrigin = 0.0; YOrigin = 0.0; Radius = 50.0 }

printf "%f " (getDiameter bigCircle)
printf "%A" (getPoints bigCircle 3.14)
```



Eager & Lazy Evaluation

```
let eagerDivision x =
    let oneOverX = 1.0 / x
    if x = 0.0 then
        printfn "Tried to divide by zero"
    else
        printfn "One over x is: %f" oneOverX

let lazyDivision x =
    let oneOverX = lazy 1.0 / x
    if x = 0.0 then
        printfn "Tried to divide by zero"
    else
        printfn "One over x is: %f" oneOverX
```

Asynchronous

```
let rec fib x =
    match x with
    | 1 -> 1
    | 2 -> 1
    | _ -> fib(x-1) + fib(x-2)

let fibRange s f =
    [s..f] |> List.map (fun x -> async { return fib x } )
    |> Async.Parallel
    |> Async.RunSynchronously

printf "%A" (fibRange 10 20)
```

Bitwise Functions

```
let divideByTwoFloor x = x >>> 1
let multiplyByTwo x = x <<< 1
let twosComplement x = ~~~x

let divValues = [1..20]
     |> List.map (fun x -> divideByTwoFloor x)
let multValues = [1..20]
     |> List.map (fun x -> multiplyByTwo x)
let twosComplements = [1..20]
     |> List.map(fun x -> twosComplement x)
```

Conclusion

- Targets .NET platform
  - Access to large array of .NET resources/libraries
  - High integration with other .NET languages
- Multi-paradigm (Functional, Imperative, OO)
- Supports both immutable and mutable data
- Strongly typed with type inference
- Defaults to eager evaluation, has lazy keyword
- Easy to parallelize (but not automatic)
- Has tuples and records but also create your own types using OO

Sources

- Set theory intersection image
  - http://en.wikipedia.org/wiki/Intersection_(set_theory)
- Unit circle image
  - http://en.wikipedia.org/wiki/Unit_circle
- Weak/Lazy Evaluation Example
  - http://stackoverflow.com/questions/6683830/f-lazy-evaluation-vs-non-lazy
- Factorial example w/matching
  - http://en.wikipedia.org/wiki/F_Sharp_(programming_language)