C# has borrowed a lot of features from ML and Haskell for example:

* C# 2.0 brought us parametric polymorphism (or "generics"). I've heard that Dom Syme, one of the creators of F#, was largely responsible for implementing generics in the .NET BCL.
* C# 2.0 also allows programmers to pass and returns functions as values for higher-order functions, and has limited support for anonymous delegates.
* C# 3.0 and 3.5 improved support anonymous functions for true closures.
* LINQ can be considered C#'s own flavor of list comprehensions.
* Anonymous types look like an approximation of ML records
* Type-inference is a given.
* I don't know about you, but C# [extension methods](http://msdn.microsoft.com/en-us/library/bb383977.aspx) look an awful lot like Haskell [type classes](http://en.wikipedia.org/wiki/Type_class).
* There's been a lot of talk about the "dynamic" keyword in C# 4.0. I'm not 100% sure of its implementation details, but I'm fairly sure its going to use [structural typing](http://en.wikipedia.org/wiki/Structural_type_system) rather than late binding to retain C#'s compile time safety. Structural typing is roughly equivalent to "duck typing for static languages", its a feature that Haskell and ML hackers have been enjoying for years.

This isn't to say that C# is a functional programming language. Its still missing important features such as pattern matching, tail-call optimization, and list and tuple literals. Additionally, idiomatic C# is fundamentally imperative with a heavy dependence on mutable state.

I wouldn't necessarily consider some of those features mentioned above as exclusive to functional programming languages, but its pretty clear that the C# developers have taken a lot of inspiration from functional programming languages in the past few years.

There being no rigourous definition of "OO Language", "Functional Language", "Procedural Language", one can make arguments that any language fits mostly any classification; one can write procedural Java, object oriented C and functional C++. I typically use a classification based around what the main semantic features support, along with common development practice. A good way of looking at this is to examine builtin and popular frameworks, and see what style they use.

Functional languages are mostly defined as those with first class function primitives, with development styles that use these to reduce complexity with idioms like "map". One other common feature is pattern matching, but I don't see this as exclusively functional. "Pure" functional languages also have no side effects, but that's not mandatory (see how fuzzy these concepts are? :).

So, what's C#? Well, it has first class function style primitives, with delegates (and has gained better syntactic support for the style with anonymous delegates and lambdas). Does this make it functional? Perhaps, if one writes in a functional style. Does the Framework use this style? No, not really.

As such, I wouldn't class C# as functional in general discussion - it is, at best, multi-paradigm, with some functional flavor.

Well, delegates and closures allow it to operate in a largely functional way... for example:

var sum = data.Sum(x=>x.SomeProp);

etc

You can write most higher-order functions using lambdas / delegates. The type inference isn't *quite* the same as pure functional languages such as F#, but C# generic-type-inference is still pretty good (especially in C# 3.0).

This is especially true in .NET 3.5 and C# 3.0, where LINQ takes a highly-functional approach to many of the problems. But you can still use the functional aspects of C# with .NET 2.0 and C# 2.0. It is just easier with C# 3.0 and lambdas ;-p

Actually, C# is a *pragmatic* programming language. It aims to make it possible to use a number of paradigms, without punishing you hideously if you want to do something different.

These are the main points whow makes c# functional 1-Lamba expresions 2-Extension methods 3-Type inferende 4-Object and collection initializators 5-Closures 6-Anonymous types 7-Linq

Programming Paradigm: is way or style of writing programs

Unstructured Programming Languages:

1. Entire application is written in single unit

2. Difficult to code, modify, debug and test

3. Follows linear approach (strategy)

4. Emphasize on how to solve problems

5. No data types (Un-typed)

6. No data abstraction (global scope)

7. Very high coupling and very low cohesion

8. Less flexibility, extensibility and maintainability

9. Not suitable for complex applications

10. Ex. BASIC

Structured Programming Languages: (POP)

1. Entire application is divided into smaller units called as procedures

2. Quite easy to code, modify, debug and test

3. Follows top down approach (strategy)

4. Emphasize on procedure to solve problems

5. Data types (Typed) - OPPs features

6. Less data abstraction (local and global scope)

7. More coupling less cohesion

8. Quite flexibility, extensibility and maintainability

9. Suitable for moderately complex applications

10. Ex. C

Object Oriented Programming Languages: (OOP)

1. Entire application is divided into smaller units called as entities or objects

2. Easy to code, modify, debug and test

3. Follows bottom up approach (strategy)

4. Emphasize on how to manage data

5. Data types ( Typed ) + Abstract Data types + OPPs Features

6. More data abstraction (local, class, object, package, application scope etc)

7. Less coupling more cohesion

8. More flexibility, extensibility and maintainability

9. Suitable for any complex applications

10. Ex. C++, C#, Java, Action Script

Object Based Programming Languages:

1. Entire application is divided into smaller units called as prototypes

2. Objects are created by using prototypes

3. Most of the OOPs concepts not supported

4. Weakly typed or Strongly typed

5. Ex. JavaScript