F#: An Introductory Look

by Kody Myers

# Introduction

This paper is intended to serve as a broad introduction and overview of the F# programming language. It is my belief that one of the most important traits of a great software developer is to foster a positive attitude towards the pursuit of lifelong learning. Many of today’s developers are being trained up with imperative languages, but rarely have the time to explore other languages once they have settled into a career.

I’d wager that many developers are missing out on the many lessons that functional languages have to teach. Functional languages are a much different way of viewing and thinking about problems, and I believe there are many developers out there, myself included, who could benefit greatly from the alternate perspective.

So why did I choose F# in particular? Since the .NET Framework was released in February 2012, its user base of developers has grown into one of the biggest collections of like-minded developers to have ever come together. F# is in a unique position to leverage the tools and practices already adopted by this community in order to more easily explore and embrace the concepts of a functional language.

# History

"There are only two kinds of programming languages:   
those people always bitch about and those nobody uses.”   
  
- Bjarne Stroustrup

Since the formulation of lambda calculus in the 1930s by Alfonzo Church, the world has seen several functional programming languages come to prominence as a result. But why does the world need yet another functional language, when there are so many in existence already? Perhaps it’s best to examine that areas where existing languages fall short.

### Why No One Uses Functional Languages

The ideals behind the design of F# can be traced back to an influential article published in the August 1998 edition of the *ACM SIGPLAN Notices* newsletter by Philip Wadler, one of the principle designers of Haskell and a prominent figure in functional language design. Wadler’s article, “Why no one uses functional languages,” was widely received by the language design community as an eye-opening critique of existing functional programming languages. In his opening statement, Wadler admits that the article’s title is an exaggeration, but he later goes on to say that:

[…] there are a large number of factors that hinder the widespread adoption of functional languages. To be widely used, a language should support interlanguage working, possess extensive libraries, be highly portable, have a stable and easy to install implementation, come with debuggers and profilers, be accompanied by training courses, and have a good track record on previous projects. (Wadler, 1998)

At the time Wadler’s article was published, many programming language scholars held the belief that programmers were resistant to learning functional programming languages because they harbored a deep fear of change or were too entrenched with their thinking after spending such a large portion of their careers working with imperative programming languages. Wadler sought to challenge this idea by proposing that the functional programming languages of the time simply did not lend themselves to widespread adoption.

### The .NET Framework and a New Functional Language

Soon after the publication of Wadler’s article, Microsoft began investing heavily in the development of its .NET Framework, a large class library providing interoperability between several different programming languages. Several imperative programming languages had already been slated for inclusion in the framework, but Microsoft recognized the need for a functional programming language to round out the framework’s utility.

While working on the implementation of generics for the framework, Don Syme, a Principal Researcher at Microsoft Research in Cambridge, UK, found himself in the perfect position to act on the suggestions from the *Notices* article and began to develop a new functional programming language for the framework in an attempt to address Wadler’s grievances.

Syme and his team tried several different approaches were tried while trying to find a functional programming language that was a good fit for Microsoft’s new framework. Like many other programming endeavors, the team endured several false starts, failed attempts, and dead ends before finally arriving at the implementation that exists today. “We had a go doing Haskell for .NET,” Syme recounted in a 2009 interview. “We actually got a long way in doing that, but in the end there is quite a lot of dissonance between Haskell and .NET.” (Drobi, 2009)

Don Syme finally found the congruity he was searching for in the Objective Caml (OCaml) language, a dialect of Caml and descendant of Metalanguage (ML). Since he was unable to directly port the OCaml to the .NET Framework as OCaml.NET, Syme created F# as “a complete re-implementation of a Caml-like language,” and credits the OCaml design team for being the major inspiration for his new programming language. (Microsoft Corporation)

### F# Unleashed

Though it did not yet have an official specification, F# 1.0 made its official debut when it shipped with Microsoft Visual Studio 2005 in May 2005. F# 2.0 shipped with Microsoft Visual Studio 2010 in April 2010 with its first official specification following shortly thereafter in August 2010. F# 3.0 shipped with Microsoft Visual Studio 2012 in August 2012 with an updated specification finalized in November 2012.

The most current iteration of the language is F# 3.1, which shipped with Microsoft Visual Studio 2013 in October 2013. The 3.1 specification is still considered a “working draft” and has not yet been finalized at the time of this writing.

F# now exists as a first-class programming language in the .NET Framework rather than being treated as a utility language that exists as some subset of a predominant imperative language. F# enjoys the same library and IDE support as C# and Visual Basic.NET. It would seem that Don Syme and Microsoft have indeed managed to create a language that overcomes many of the hurdles outlined by Wadler’s article.

It is important to note that F# does not exist as a purely functional programming language. In fact, it is often labeled as a “multi-paradigm” programming language, since it mixes functional, imperative, and object-oriented programming constructs. (Podwysocki, 2008) The “functional” capabilities of the language take precedence, however, so F# is also labeled as a “functional-first” programming language. (F# Software Foundation)

# Design, Syntax, and Semantics

Given that F# is primarily a functional language, much of its syntax appears more like a mathematical proof than the sequences of instructions that are so prevalent among imperative languages. F# relies on declarative expressions rather than stepping through a series of state changes and modifying data. It’s easy to pick out the many traits of its OCaml and ML ancestry, but the more interesting functionality lies in its interoperability with the .NET Framework.

### Code Documentation

Comments lines in F# are prepended with **//**. They can also be prepended with **///** which designates them as XMLDoc comments which can contain additional internal markup tags such as **<summary>…</summary>**. Multiline comments are opened with **(\*** and closed with **\*)**.

// This is a single line comment.

/// <summary>This is a XMLDoc summary comment.</summary>

(\*  
This  
is  
a  
multiline  
comment.  
\*)

### Keywords

There are currently ninety-eight reserved words in the language, which are case-sensitive and lowercase. Of these ninety-eight reserved words, only sixty-four are currently considered active in the language. Eight of the keywords were carried over from OCaml but are not actually used. An additional twenty-six keywords are reserved for future implementation. (Microsoft Corporation)

### Active Keywords

abstract

and

as

assert

base

begin

class

default

delegate

do

done

downcast

downto

elif

else

end

exception

extern

false

finally

for

fun

function

global

if

in

inherit

inline

interface

internal

lazy

let

let!

match

member

module

mutable

namespace

new

not

null

of

open

or

override

private

public

rec

return

return!

select

static

struct

then

to

true

try

type

upcast

use

use!

val

void

when

while

with

yield

yield!

### Holdouts from OCaml

asr

land

lor

lsl

lsr

lxor

mod

sig

### Reserved for Future Use

atomic

break

checked

component

const

constraint

constructor

continue

eager

event

external

fixed

functor

include

method

mixin

object

parallel

process

protected

pure

sealed

tailcall

traitvirtual

volatile

### Variables

Unless explicitly declared otherwise, variables in F# are immutable, meaning that once set, they cannot be changed. Due to their immutable, invariable nature, F# programmers tend to prefer the term *value* to *variable* in both conversation and documentation of the language. The immutability of values in a programming language grants many of the same advantages that named constants, such as ***π*** and ***e***, grant in mathematics. Once defined, a programmer no longer needs to concern himself with the possibility of the value changing. (Syme, Granicz, & Cisternino, Expert F# 3.0, 2012, p. 9)

### Scoping

Typical of many other functional languages, F# is block scoped by means of indentation rather than delimiter tokens such as **{** … **}** or **begin** … **end**. Although code scoped with indentation is often considered cleaner because it lacks the visual clutter of opening and closing delimiters, it can be argued that this method of scoping lends itself to subtle logic errors caused by improper indentation.

### Data Types

F# supports all of the major data types that are supported in other .NET languages. This list includes primitives (e.g., characters, strings, integers, floating-point decimals, Booleans), arrays (both single-dimensional and multi-dimensional), tuples, user-defined structures, and a whole host of other .NET types such as DateTime and GUID in order to support a variety of software needs.

Some examples are as follows:

|  |  |
| --- | --- |
| Declare a string | **let hello = “Hello World!”** |
| Declare a floating-point decimal | **let pi = 3.1415926** |
| Declare a Boolean | **let isRunning = false** |
| Declare a list of integers | **let numList = [1 .. 10]** |
| Declare a tuple | **let vehicle = (2013, “Ford”, “Explorer”, true)** |
| Declare a GUID | **let userId = System.Guid.NewGuid()** |

Data types in F# are usually inferred, but it is possible to explicitly declare them using type annotations. For example, the line:

**let sum (x:float) (y:float) : float = x + y**

assigns the sum of the values x and y to the value sum with the express condition that x and y are of type float.

# Assignment and Expressions

The single most important keyword used in F# is its assignment operator, **let**, which is used to define data, computed values, and functions. A let statement uses the following form:

**let <identifier> = <expression>**

where **<identifier>** can be a value name or a function name with optional parameters and **<expression>** can be a constant value, computed value, object, or even another let statement.

|  |  |
| --- | --- |
| Assign the value 42 to the x | **let x = 42** |
| Declare a function named “plusOne” that returns x + 1 | **let plusOne x = x + 1** |

## Order of Operations

Below is an image straight from the 3.1 working draft specification which details the operator precedence in F#:



(Syme, The F# 3.1 Language Specification, 2013, p. 35)

# Control Structures

F# contains many of the same looping and selection structures as most imperative languages, but they aren’t very prevalent in most F# code listings since pattern matching and recursion are favored in F#’s “functional first” design philosophy. Functional programming is also known as “programming without side-effects.” In contrast, imperative programming is known as “programming with side-effects.” Though the goal of F# is to use as many functional constructs as possible in order to minimize side effects, there are limits to what a programmer can do with functions alone, which necessitates the availability of imperative constructs.

Some simple examples are provided below.

### Simple For Loop

**let countToNum x =  
 for i = 1 to x do  
 printf “%d\n” i  
 printf “Done!\n”**

### Simple While Loop

**let loopUntilSaturday() =  
 while (DateTime.Now.DayOfWeek <> DayOfWeek.Saturday) do  
 printf "Still working!\n"  
 printf "Saturday at last!\n"**

(Syme, Granicz, & Cisternino, Expert F# 3.0, 2012, p. 51)

# Support for Object-Oriented Programming

One of the main requirements for a functional language to be fully integrated with the .NET Framework was support for object-oriented programming (OOP). A fully OOP language stands on three fundamental pillars: inheritance, polymorphism, and encapsulation – all of which are present in F#. In line with F#’s “functional-first” design philosophy, objected-oriented constructs play a supporting role for functional designs, rather than composing the entire program. In other words:

Programming in F# tends to be less “object-oriented” than in other languages, since functional programming with values, functions, lists, tuples and other shaped data is enough to solve many programming problems. Objects are used as a means to an end rather than as the dominant paradigm. (Syme, Granicz, & Cisternino, Expert F# 3.0, 2012, p. 111)

Like other .NET languages, objects in F# are implemented with classes that contain constructors with optional parameters, public or private members, and public or private functions. Classes in F# follow the same dot-notation used throughout the rest of the .NET Framework to access class members and functions.

# Exception Handling

F#’s exception handling capabilities give programmers a chance to cope with and recover from exceptional circumstances rather than crashing the program. The language provides a **try … with** construct with which programmers can use to wrap sensitive operations and execute additional code if and when those operations fails.

The following code attempts to divide a number by zero and prints the message, “oops” when the resulting DivideByZeroException is raised:

**try  
 let divByZero = 100 / 0  
with  
 :? System.DivideByZeroException -> printfn “oops”**

The print statement for the custom error message is preceded by a type test using the **:?** operator. The type test returns true when compared with System.DivideByZeroException, and the custom error message is then printed.

Exceptions can also be raised manually with the **raise** keyword, as seen in the following code which prints out the error message, “operation failed with result: whoah, can’t do that”:

**try  
 raise (System.InvalidOperationException (“whoah, can’t do that”))  
with  
 err -> printfn “operation failed with result: %s” err.Message**

It may at times be necessary at times to perform some sort of cleanup after an exception is raised, such as closing open files or disconnecting open database connections, in order to prevent resource allocation problems. F# provides a **try … finally** construct to serve this need. Below is a sample code listing that creates a file, begins writing items to it, but then raises an error before it is able to close the file normally. Fortunately, the logic in the finally block closes the file instead:

**let output = File.CreateText “todo.txt”  
  
try  
 output.WriteLine(“walk dog”)  
 raise (System.InvalidOperation(“something went wrong”))  
 output.WriteLine(“take out trash”)  
 output.Close()  
finally  
 output.Close()**

# Concurrency

From the .NET Framework, F# has access to many different concurrency constructs with which many .NET programmers are already experienced, namely **Thread**, **AutoResetEvent**, **BackgroundWorker**, and **IAsyncResult**. However, F# also introduces a new model for managing asynchronous IO and background tasks called “asynchronous workflows,” which are an application of F#’s computation expression syntax. (Wlaschin, 2012)

**let task1 = async { return 10+10 }  
let task2 = async { return 20+20 }  
Async.Run (Async.Parallel [ task1; task2 ])**

The above code (Syme, Introducing F# Asynchronous Workflows, 2007) is a very simple example of how two tasks can run in parallel in F# using the **async** keyword. Asynchronous workflows lie outside the scope of this introductory overview of F#, but Don Syme has written a great article (from which the above example came) on the power that these new constructs bring to the language which can be found in the References section of this document.

# Evaluation of F#

### Reliability

With strong typing, static binding, and rich exception handling features, F# is considered a highly reliable language. Like the other .NET languages, this reliability typically comes at the cost of performance, as they are most often used in writing business applications. A report that is quick, but wrong, is useless to a business. It is generally acceptable to sacrifice a bit of performance for correctness.

### Readability

F# can also be given high marks for readability, as it lacks the usual opening and closing markup that denotes the beginning and ending of a scope. It’s also devoid of any terminating characters such as the semicolon. This leaves nothing but the keywords and the values they operate on, so the programmer is able to concentrate on what the code is doing without being distracted by the markup surrounding it.

### Writability

No matter how easy it is to write any particular language, the success of a programming language depends largely on the tools that support it. In the case of F#, its writability is bolstered by the inclusion of the language in Microsoft’s Visual Studio IDE. Powerful features like Visual Studio’s IntelliSense help developers look up difficult-to-remember and seldom-used class and method names. Visual Studio also checks syntax as code is entered and provides nearly immediate feedback when syntax errors are made.

As F# has become more and more accepted, other IDEs have also emerged for developers who are uncomfortable with being locked into the Windows operating system in order to write F# programs. The first to emerge was WebSharper in 2012, which was eventually followed by additional development environments such as LINQPad, MonoDevelop, SharpDevelop, and Xamarin Studio.

### Cost

Shortly after the release of Visual Studio 2008, Microsoft individually packaged Visual Basic.NET, C#.NET, Managed C++, and its web development tools into separate, lightweight Express Editions that were targeted at hobbyists and enthusiasts. The Visual Studio Express Editions were stripped of several premium features, such as support for extensions, but were still an excellent choice for developers who wanted the power of Microsoft’s IDE, but could not afford its considerable price tag.

Unfortunately, F# never warranted the release of its own Express Edition, so until recently, the only way to develop F# code was to either purchase a full version of Visual Studio or use an alternate IDE. This changed in November 2014 when Microsoft announced Visual Studio Community Edition, which is a new free version with features comparable to Visual Studio Professional. Targeted at individual developers and small teams, Community Edition supports all .NET languages and allows the installation of extensions in contrast to the Express Editions.

# Modern Applications of F#

It’s difficult to say whether F# has truly earned its keep as a first-class language. Though it has enjoyed moderate adoption across many different organizations across different industries, it still seems to be supplemental to some other primary language such as C# or Visual Basic.NET. From many of the testimonials found on the F# Software Foundation website, F# seems to excel at problems that require the processing and analysis of very large datasets in order to extract meaning and make predictions, but these types of problems are usually just a subset of some greater functionality needed by a modern business application. (F# Software Foundation)

# The Future of F#

F# is still very much in active development, and there are several more features planned for the next version of the language, tentatively named F# 4.0. The fact that more features are upcoming is a good sign for the state of a language. As long as there are developers out there who care about the language enough to make improvements, then the language will continue to thrive. It doesn’t help that the language seems to be a good solution to a very narrow scope of problems, but the worst mark against it currently is that the language just hasn’t quite proven itself yet to most developers. That may change if enough developers venture outside of their comfort zones and take advantage of the power that F# has to offer.

# Conclusion

I believe that F# is a great introduction to functional languages for developers who are already experienced in working with the .NET Framework. The knowledge they bring with them from their prior experiences should help ease the transition into F#. Unfortunately, F# does not seem well suited as a teaching language due in part because it is not a purely functional language. With the imperative and OOP constructs thrown in the mix, it can be quite difficult for students to keep the concepts straight – especially when they’re still trying to get a good grasp on just the OOP aspect of programming. If the goal is simply to learn how to think about problems functionally, it may be best to stick with a purely functional language that does not have all the distractions and “syntactic sugar” that F# brings.

I believe that Don Syme has been successful at addressing most of the problems outlined by Philip Wadler’s infamous article, but it’s quite clear that F# never quite turned into the runaway success that Microsoft was hoping for. So the question we’re left with is, “Did Wadler miss something, or is F# just not up to the lofty standards that Wadler set forth?”

I’m not sure we’re really in the position to answer that question, currently. Maybe in fifty years, when we have that wonderful gift of perfect hindsight, we can pinpoint exactly what happened. Regardless of the community’s reception, I believe that F# has earned its place in the family tree of programming languages as a significant milestone – one that will have a positive impact on the development of even better programming languages in the future.

And in the true spirit of programming, we’ll be sure to complain about those languages, too.

# References

Drobi, S. (2009, March 27). *Don Syme Answering Questions on F#, C#, Haskell and Scala.* Retrieved from InfoQ: http://www.infoq.com/interviews/F-Sharp-Don-Syme

F# Software Foundation. (n.d.). *About F#.* Retrieved from F# Software Foundation: http://fsharp.org/about

F# Software Foundation. (n.d.). *Testimonials and Quotes.* Retrieved from F# Software Foundation: http://fsharp.org/testimonials

Microsoft Corporation. (n.d.). *F# Historical Acknowledgements.* Retrieved from Microsoft Research: http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/ack.aspx

Microsoft Corporation. (n.d.). *Keyword Reference (F#).* Retrieved from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/dd233249.aspx

Podwysocki, M. (2008, March 6). *Adventures in F# - F# 101 Part 4.* Retrieved from Matthew Podwysocki's Blog: http://weblogs.asp.net/podwysocki/adventures-in-f-f-101-part-4

Syme, D. (2007, October 10). *Introducing F# Asynchronous Workflows.* Retrieved from Don Syme's WebLog on F# and Related Topics: http://blogs.msdn.com/b/dsyme/archive/2007/10/11/introducing-f-asynchronous-workflows.aspx

Syme, D. (2013, June). *The F# 3.1 Language Specification.* Retrieved from F# Software Foundation: http://fsharp.org/specs/language-spec/3.1/FSharpSpec-3.1-working.docx

Syme, D., Granicz, A., & Cisternino, A. (2012). *Expert F# 3.0.* New York City, New York: Apress.

Wadler, P. (1998, August). Why no one uses functional languages. *ACM SIGPLAN Notices*, pp. 23 - 27.

Wlaschin, S. (2012, April 23). *Concurrency: The next major revolution in how we write software?* Retrieved from F# for Fun and Profit: http://fsharpforfunandprofit.com/posts/concurrency-intro/