Adventures in F#

Matthew Podwysocki

Part 1 – Basic Functional Programming

In some previous posts, I pointed you to a bunch of links to help get me started. I'm still working on some more thorough examples, but this time around I'm going to walk you through the F# 101 much like I did with Spec#. What we first need to understand is the key differences between imperative and functional programming.

I've really enjoyed Robert Pickering's book Foundations of F# and Don Syme's Expert F# book as they both give a good overview of not only the language, but the uses. It's a lot for some stuck in the imperative world to think about functional programming, but these books do their job well.

So, today I'd like to cover the following topics:

- Hello World Sample
- #light
- Functions == Values basics

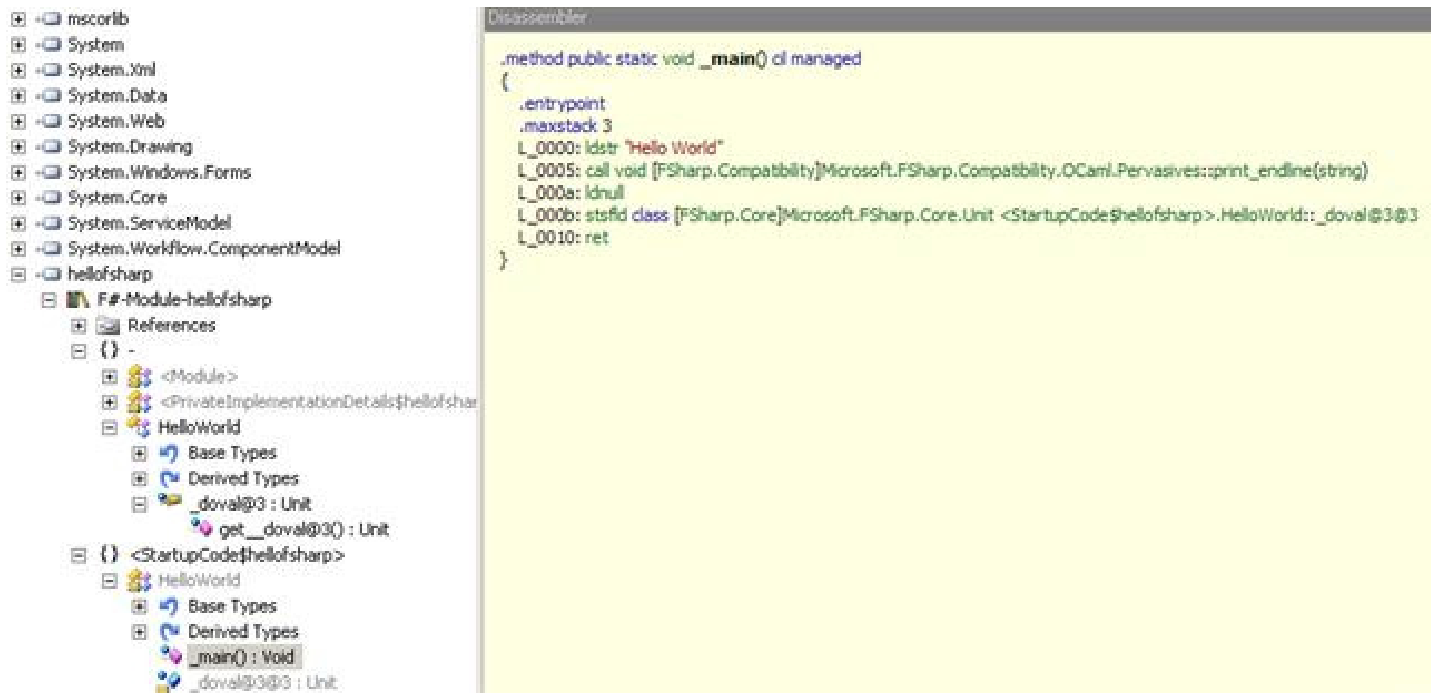
Hello World Example

As I like to do with most languages, I know it sounds trite, but Hello World, especially in .NET languages is interesting, especially how they translate in IL through Reflector. F# is especially interesting in this case as it is a functional language built on top of a imperative framework such as the CLI and C#.

So, let's walk through a simple example. First, of course you need to have F# installed which can be foundhere. Then go ahead and create a new F# project, which should be under the Other Project types. By default, it will not give you any classes as part of your project. So, go ahead and add an F# Source File. It will give you the extension of .fs by default. I went ahead and created a class called HelloWorld.fs. Get rid of the default text that is in there, and it is plenty, and just put in the following code:

```
#light

print_endline "Hello World"
```

As you can see from the simple example above, we've put in the #light directive. I'll cover what that means shortly. But anyways as you can notice it calls a default global function called print_endline. I always like to look at what the code compiles to in Reflector to IL, so let's take a look at that.



```
.method public static void _main() cil managed
{
   .entrypoint
   .maxstack 3
   L_0000: ldstr "Hello World"
   L_0005: call void [FSharp.Compatibility]Microsoft.FSharp.Compatibility.OCaml.Pervasives::print_endline(string)
   L_000a: ldnull
   L_000b: stsfld class [FSharp.Core]Microsoft.FSharp.Core.Unit <StartupCode$hellofsharp>.HelloWorld::_doval@3@3
   L_0010: ret
}
```

What's really interesting to see from the above, by default, it created the main function for us without us explicitly doing so. Also, it notes that we are calling an OCaml function. If you look through Reflector and the FSharp.Compatibility.dll, you'll notice it takes a few things from OCaml as its base.

Where to Turn

As always, if you have problems, the specification and the books help out well with this. If you should get lost, the information F# specification is very good at defining the language and can be found here.

Stay in the #light

From the above example and many other examples you will see from me, you'll notice the **#light** directive at the top of every source file. What this does is that it makes all whitespace significant. This directive allows you to omit such things as begin, end, in and the semi-colon. When defining functions and so on, you indent your code to make it fall under that scope implicitly by using that directive.

Identifiers, Keywords and Literals

In F#, it's important to realize that variables, aren't really variables, more like identifiers. In pure functional programming, it's fundamental to understand that once a "variable" is assigned, it is immutable. However, if using F# an imperative way of thinking, then it can be.

To make a simple assignment, just use the simple **let** keyword such as this:

```
#light

let message =  "Hello World"
printfn "%s" message
```

One of these identifiers can be either either just a simple value, or even a function, since these functions are values as well. I'll cover that more shortly in the next section.

```
#light

let rec fib n = if n < 2 then 1 else fib (n-2) + fib(n-1)
printfn "%i" (fib 10)
```

I think this example went a little too far with adding recursion, but you get the point that it's pretty easy to define a function much as you would with any old "variable". Also, you may note, I'm not returning anything from this function, but in fact I am returning the Fibonacci sequence that I asked for.

If you also note the keywords, there are a lot of direct mappings from C# to F#, but it's also interesting that it has a list of reserved keywords for future use such as mixin. Now, if we finally brought mixins to .NET, well, then I'm all for it.

Of the types that exist in F#, the only ones to note that are completely different are:

- bigint - Microsoft.FSharp.Math.BigInt
- bignum - Microsoft.FSharp.Math.BigNum

It's interesting to know that the .NET framework's BigInteger class is internal, yet these are publicly available. Both of these are rational large integers just in case you need that kind of thing.

Functions == Values?

Once again, like I said above, functions and values are one in the same in the pure functional programming stance. You noticed that in my functions, I'm not returning anything, nor is it imperative that you do so explicitly. This boggles many of OO mindsets, I'm sure and there are a lot of things in FP that an OO person may have problems with.

Let's walk through another simple function, but this time, we'll implement that main function and look at the results through Reflector.

```
#light

let rec fib n = if n < 2 then 1 else fib (n-2) + fib(n-1)
let print x = printfn "%i" x

let x1 = fib(10)
let x2 = fib(20)

let main() =
  print x1;
  print x2

main()
```

From the above sample, I created two functions, a recursive Fibonacci sequence function from above, and I also created a print function which calls the native **printfn** function which is a variation of the **printf** function with a newline at the end. For those who are curious about what it looks like in Reflector, let's take a look, and this time in C#. Instead of pasting a bunch of screenshots, it's much easier to paste it below the pieces we're interested in.

fib function:
```
public static int fib(int n)
{
   if (n < 2)
   {
      return 1;
   }
   return (fib(n - 2) + fib(n - 1));
}
```

print function:
```
public static void print(int x)
{
   Pervasives.printfn<FastFunc<int, Unit>>(new Format<FastFunc<int, Unit>, TextWriter, Unit, Unit>("%i")).Invoke(x);
}
```

main function:
```
public static void main()
{
   print(get_x1());
   print(get_x2());
}
```

So, as you can see, it's doing much like we think it would be, as it fit everything into our HelloWorld class. The more interesting pieces are how C# and F# functions differ. It's just interesting to see how FP fits on top of an imperative programming framework.

Next time, I'll cover more with functions such as Currying, Tuples, Recursion, Scopes and so on. Plus I need to talk about the interop story here which is strong between F# and C# as well. There's a lot to cover and quite frankly, it's making me stronger as well to go over this time and time again.

Conclusion

This is the first post in the series. As you can see, I have plenty to cover in the next installment. If you like the series and so on, subscribe and keep coming back. Until next time...

Part 2 – Currying and Tuples

I know it's been a little bit too long since I began this series from when I continued, but plenty of distractions came up along the way. I intend to go a little deeper today into what functional programming means and why you should care. As always, check out my previous post on the matter here.

Getting Sidetracked

Last night at the DC ALT.NET meeting, Craig Andera and I discussed Lisp, F# and functional programming as a whole. His language of the year was Lisp and I thought that was an interesting choice of the language to learn, but when you think about it, most good things in programming lead back to SmallTalk and Lisp... I had been hearing a bit about IronScheme, the IronLisp rewrite, so of course I had to check it out. Maybe this will be one of my future languages to learn, but I had enough exposure during my college years to last a little bit, so it's on the plan anyways.

Basic Introduction

So, just in case you missed the idea of functional programming and what it means, I have a few posts on where you should start here:

- My Adventures in F#
- Further Adventures in F#

But, if you really want the condensed version, Bart de Smet has a really nice introduction on functional programming in a series, although incomplete, which you can read here:

- Foundations of Functional Programming Part 0
- Foundations of Functional Programming Part 1

Now that I take it you have read that and caught up, let's continue onto today's part.

The Joys of Currying

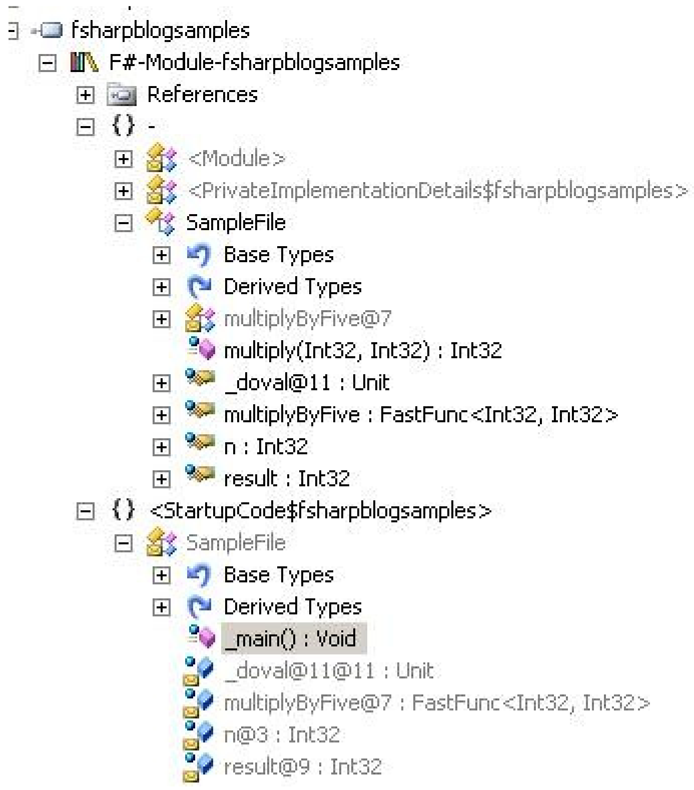
Where we left off last time was just dealing with the basics of functions and the fact that they are treated as values much like any other "variable". Now let's get into a popular topic among functional programming, a concept called Currying. The term was invented back in the earlier days of functional programming after the early logician Haskell Curry. He's so good in fact that he got not only a functional programming style named after him, but a language named after him with Haskell. If you listen to the DotNetRocks Episode 310 withSimon Peyton-Jones, you can learn more about that.

Anyhow, the basic idea of Currying is to take a function with multiple arguments and transform it into a function that passes in only one argument, with the other arguments specified in the Currying function itself. So you can think of it as a function that calls another function and returns a function. You can think of it also as being a partial function. So, let's actually get down to it and write some F# code to show how to do this:

```
#light

let n = 10

let multiply a b = a * b

let multiplyByFive = multiply 5

let result = multiplyByFive 4

printfn "result = %i" result
```

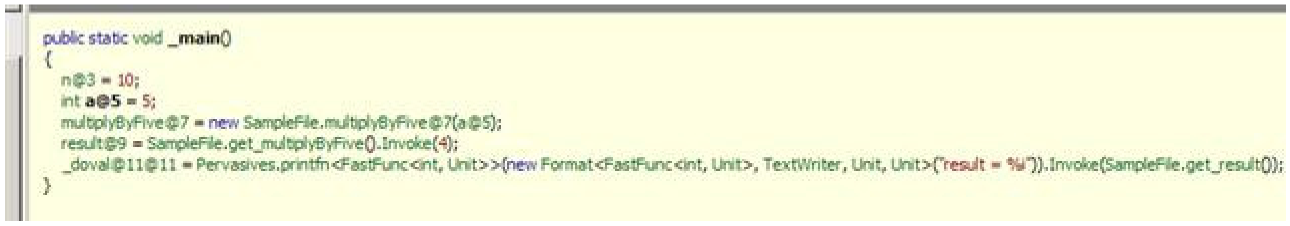
So, what I accomplished above is that I had a function called multiply that takes two arguments, a and b. Then I created a currying function called multiplyByFive which curries the multiply function by supplying a five value. Then I can call the multiplyByFive curried function with a 4 and sure enough the result is 20.

As always, I'm always curious on how it looks through .NET Reflector to see how F# does its magic. So, let's crack that open and take a look. First let's look at the tree of Reflector to see what it creates for us:



```
fsharpblogsamples
    F#-Module-fsharpblogsamples
        References
        { } -
            <Module>
            <PrivateImplementationDetails$fsharpblogsamples>
            SampleFile
                Base Types
                Derived Types
                multiplyByFive@7
                multiply(Int32, Int32) : Int32
                _doval@11 : Unit
                multiplyByFive : FastFunc<Int32, Int32>
                n : Int32
                result : Int32
        { } <StartupCode$fsharpblogsamples>
            SampleFile
                Base Types
                Derived Types
                _main() : Void
                _doval@11@11 : Unit
                multiplyByFive@7 : FastFunc<Int32, Int32>
                n@3 : Int32
                result@9 : Int32
```

Now as you can see, it created the multiply function. Then we also have a multiplyByFive FastFunc function. The functions get stored in the namespace free area and yet the main function gets placed elsewhere. Now, let's look at the function implementation itself.



```
public static void _main()
{
    n@3 = 10;
    int a@5 = 5;
    multiplyByFive@7 = new SampleFile.multiplyByFive@7(a@5);
    result@9 = SampleFile.get_multiplyByFive().Invoke(4);
    _doval@11@11 = Pervasives.printfn<FastFunc<int, Unit>>(new Format<FastFunc<int, Unit>, TextWriter, Unit, Unit>("result = %i")).Invoke(SampleFile.get_result());
}
```

From what you can see, it created a SampleFile class because I created a SampleFile.fs to create the curried functions.

For those of you who are C# buffs, well, you can also do the same in C# with the new anonymous function type, System.Func in .NET 3.5. Mads Torgersen, the C# Program Manager, noted on his blog quite a while ago asking the question, "Is C# Becoming a Functional Language?". This post was pretty great and it's unfortunate that he doesn't blog as much as he does. Remember kids, this isn't as pretty as F# because it's not what C# is really good at, but slowly but surely, functional programming paradigms are slowly seeping into C#.

```
namespace CSharpCurriedFunctions
{
    public static class CurriedFunctionExtensions
    {
        public static Func<A, Func<B, R>> Curry<A, B, R>(this Func<A, B, R> f)
        {
            return a => b => f(a, b);
        }

    }

    class Program
    {
        static void Main(string[] args)
        {
            Func<int, int, int> multiply = (a, b)=> a* b;
            Func<int, Func<int, int>> curriedMultiply = multiply.Curry();
            Func<int, int> multiplyByFive = curriedMultiply(5);

            Console.WriteLine("result = {0}", multiplyByFive(4));
        }
    }
}
```

If you're interested more in doing this with C#, I would suggest you head over to Dustin Campbell's blog and read the post about The Art of Currying. For those who want to learn more, well, Dustin should be at ALT.NET Open Spaces, Seattle and hopefully will submit an F# topic for us to get our heads around.

How useful is this in C#? Well, time will tell, but you could imagine having some search criteria for your repository using curried functions to "overload" with values. Anything is really possible, but as you can see, it's a lot cleaner in F# and that's where it really shines.

The Odd Tuple

When specifying arguments to any given function, you can specify them without parentheses much as you would in VB.NET should it be a subroutine. This is really helpful if you want to enable currying and such. But, you can also specify them as a tuple. For the math illiterate among you, that simply means a sequence of objects of a specified type. This you must put within parentheses. This makes you supply all arguments at once. A simple example of this would look like this:

```
#light

let add (a, b) = a + b

let result = add(3, 4)

printfn "result = %i" result
```

Pretty simple example and of course the result would be 7.

Where is the Returns Department?

As you may have noticed from the functions from above, I never really did specify any return type or return statement. Implicitly behind the scenes, F# does that for you. Basically when you define a function, you should indent with a tab and the last assignment becomes your return value. The return is implicit right after the indenting is ended. So, let's take a simplistic view of this:

```
#light

let addThings a b  =
  let c = a + b
  sprintf "result = %i" c

let result = addThings 3 7

print_string result
```

As you may notice, when using the light syntax (#light), you cannot indent, instead just use two spaces and that should suffice. Anyhow, you may note, I was doing a simple addition and then my final assignment becomes my return value, which is using the sprintf function, which to you C buffs out there, should look familiar.

Conclusion

I've got a lot more on this topic quite frankly and still just covering the basics of your first F# program. There is so much to learn on this, but these are the basic building blocks you need to make that leap into functional programming. It's becoming more interesting year by year as our languages such as Ruby, C# and so on continue to drive towards functional programming, so it's important to grok the fundamentals. Next time, I hope to cover some recursion stuff, functions, lists and even operators (the really fun stuff). Until next time...

Part 3 - Scope, Recursion and Anonymous Functions

Today we have another installment of the Adventures in F# - F# 101 series. This time we're going to cover more functional programming basics and hopefully cover some pretty interesting things along the way and compare them with normal imperative style programming. I believe that functional programming, mixed with imperative constructs is the natural evolution of the .NET framework, and indeed the future of it. Like I've said before in previous posts, the languages are starting to converge on a mix of it with C# 3.0 and beyond.

So, let's get caught up to where we are today and where we came from:

- Part 1 - Basic functional programming
- Part 2 - Currying and Tuples

On a side note, as you read these, Dustin Campbell also has a series that I've linked before, but a few have come out recently that you may want to pay attention to. What's great about the F# community, although small, is very willing to help each other out in forums, emails, chats and so on. Check out hubFS for that kind of community. You might even find some product team members there. Anyhow, before shiny things came along, let's get back to Dustin here and his "Why I Love F#" series:

- Option Types
- Pattern Matching
- Functions, Functions, Functions!

And the list goes on from there. Drink it up!

So, let's move onto the topics for today. Today we will cover the following areas:

- Scoping it Out
- Recursion, Recursion, Recursion
- Staying Anonymous

Scoping it Out

Scoping of variables is a common thing inside any programming language and really no different in F# at all. Remember that identifiers can be either values or functions themselves. The only thing to me that is interesting about F# is that you can redefine your identifiers. By default, your identifiers are immutable unless otherwise specified in functional programming languages. So, we're not going to change the value, but instead redefine. And what's more interesting, is that you can change the type on the fly as well with the redefine, so be careful!

```
#light


let message =

  let temp = "Foo"

  let temp = 1

  let temp = "Bar"

  temp
```

So, this one works and quite simply just by redefining what temp is. If we pop open Reflector, here's what it boils down to in C#:

```
public static void _main()
{
    message@3 = "Bar";
}
```

As you can see from there, it refactored that code out that we didn't need at all. It was a pretty stupid example, but interesting nonetheless.

Recursion, Recursion, Recursion

Recursion is one of those computer science lessons that everyone should have learned back in their early CS days. This was taught in line with pointers, reference, hex/octal math, linked lists and so on. I certainly hope they still teach that stuff nowadays with Java and .NET now the mainstays in universities. Anyhow, if you didn't have that luxury, it's basically where you have a function that calls itself in order to return an answer. Some of the classic examples of this are computing factorials, Fibonacci sequences and so on. Of course recursive functions can have their downsides including blowing out the call stack if the recursion gets too deep or gets into an infinite loop, so it's always a good thing to test for these sorts of things.

F# treats recursive functions just a bit differently than regular functions. In fact, in order to make a function act recursive, you must mark it as such using the **rec** keyword. This allows you to call that function within that function, else it will give a syntax error. So, let's step through a few examples.

Let's start out with the classic Fibonacci sequence. To do this in traditional imperative programming, it looks something like this in C# 2.0:

```
static int Fib(int i)

{

    if (i <= 1)

        return 1;


    return Fib(i - 1) + Fib(i - 2);

}
```

Ok, so that's old school to me. Instead, since I mentioned that C# is going more towards a functional programming style, let's go ahead and do that again using anonymous functions (System.Func) instead.

```
Func<int, int> fib = null;

fib = i => i > 1 ? fib(i - 1) + fib(i - 2) : 1;
```

Ah, much cleaner. But, let's take this to F# and use pattern matching instead of just the normal if/else control structure, just to be on the wild side.

```
#light


let rec fib n =

  match n with

  | 0 -> 1

  | 1 -> 1

  | n -> fib(n-2) + fib(n - 1)


let fib10 = fib 10

printfn "10th Fibonacci 10 %i" fib10
```

Ok, so let's throw a few more logs onto the fire with a couple more, one with recursing all files in a given directory (yes, I know you can do it with Directory.GetFiles, but this is more fun), and computing a factorial.

```
#light


open System.IO


let rec recurseDirectories x =

  let fileList = new ResizeArray<string>()

  fileList.AddRange(Directory.GetFiles(x))

  let subDirs = Directory.GetDirectories(x)

  subDirs |> Seq.iter(fun x -> fileList.AddRange(recurseDirectories x))
  fileList


let directories = recurseDirectories @"E:\Work"

directories.ForEach(fun x -> printfn "%s" x)


let rec fac n =

  match n with

  | 0 -> 1

  | n -> n * fac (n - 1)


let fac10 = fac 10

printfn "Factorial of 10 %i" fac10
```

In the recurseDirectories function, I was being a little tricky and using a mutable structure. The ResizeArray<T> is actually just a rename of the List<T> generic from the regular CLR. As you can note, I'm not using any control functions for doing so such as foreach loops, etc. Instead, I'm using the Seq collection inside of F# proper and I'm sure I'll get to that later. And yes, there are probably more ways of doing these, but these are pretty good nonetheless.

Staying Anonymous

As you saw from above, I used the keyword fun. What that is, is using an anonymous function. This is much like in C# using the System.Func structure but with different underlying base classes underneath. Some simple examples of creating anonymous functions are just that, for iterating over a block like above. We also can do something else like doing some simple examples such as below:

```
#light


let a = (fun b c -> b * c) 12 13

printfn "%i" a
```

We can also use tuples with this as well. This means that you must specify all parameters up front. You can only do pattern matching when using this style. If you need to supply multiple arguments to this, go ahead and use Currying, which we talked about last time. Below is a simple example of the above code:

```
#light


let a = (function (b, c) -> b * c) (12, 13)

let z = (function y -> function x -> y * x) 12 13

printfn "%i" z

printfn "%i" a
```

And this even can apply when using .NET classes such as the ResizeArray<T> such as this quick example:

```
#light


let stringList =

  let temp = new ResizeArray<string>();

  temp.AddRange([|"Hello"; " "; "Cleveland"|])

  temp


stringList.ForEach(fun x -> print_string x)
```

Wrapping It Up

Well, I walked through just a few more pieces for your F# Adventure. I have plenty more to cover including defining types, operators and so on that make DSLs quite useful in the .NET world. So, subscribe if you haven't already and stay tuned. Until next time...

Part 4 - History of F#, Operators and Lists

Time for another adventure in F#, covering the 101 level basics of the language and why I love it as much as I do. This time we're going to cover some topics such as custom operators, lists and so on.

As I want to stress in every installment of this series, the importance of functional programming and its influence on the .NET framework. Don Syme, the creator of F# was instrumental in bringing generics into the .NET framework. With such things as lambdas, object initializers, collection initializers, implicit initialization partially came from ideas from functional programming and F#.

Where We Are

Before we begin today, let's catch up to where we are today:

- Part 1 - Basic functional programming
- Part 2 - Currying and Tuples
- Part 3 - Scope, Recursion and Anonymous Functions

Drink up the goodness and let's continue. Today, I'd like to cover the following topics:

- A Brief History of F#
- Operators
- Lists

The list may seem small, but there is a lot to learn with all of this. But before we get technical again, let's step back and get a brief lesson of how F# came to be.

A Little History Lesson

As you may know, functional programming itself is one of the older programming paradigms that dates back to the 1950s. In fact, as with most things, it can be traced back to Lisp in the late 1950s. Like I've said before, most good things in programming nowadays can trace their history to Lisp, or Smalltalk.

Anyhow, back in 2002, Don Syme and a team at Microsoft Research wanted an Metalanguage (ML) approach to language design on the .NET platform. While this project was going on, Don was also working on implementing generics in the .NET platform as well. These two projects worked there way into becoming F# which made its first appearance in 2005.

As you may note from looking at the F# libraries such as FSharp.Compatibility.dll and the namespace Microsoft.FSharp.Compatibility.OCaml, it shares a common base with Objective Caml (OCaml). OCaml itself, another language in the ML family and been around since 1996. It also borrowed for Haskell as well in regards to sequences and workflows. But, in the end, as you may know, F# splits from those languages quite a bit because it allows for imperative and object oriented programming. That's the really interesting piece of this puzzle is to be able to interoperate with other .NET languages as if they were first class citizens.

One thing that I found particularly interesting about F# is that it is very portable, unlike other .NET languages. What I mean by that is that they provide you with the source so that you could use this with Mono, Shared Source CLI or any other ECMA 335 implementation. I've been able to fire up a Linux box with Mono and sure enough it compiles and runs without fail. Now onto the technical details...

Operators

Operators are an interesting and compelling feature in F#. Most modern languages support this feature except for say Java to overload operators. I think it's a compelling feature in the way of dealing with DSLs and such. But anyways, F# supports two types of operators, the prefix and infix types. The prefix operator type is an operator that takes one operand. In order to be prefix, it means that it precedes its operand such as (-x). The infix operator type takes two or more arguments. An example of this kind of operator is (x + y).

There are many operators in the F# libraries quite frankly and I'll list just a few of them:

Arithmetic Operators

Operator	Meaning
+	Unchecked addition
-	Unchecked subtraction
*	Unchecked multiplication

/	Division
%	Modulus
-	Unary Negation

Bitwise Operators

Operator	Meaning
&&&	Bitwise and
|||	Bitwise or
^^^	Bitwise Exclusive Or
~~	Bitwise Negation
<<<	Left shift
>>>	Right shift

Assignment and Others

Operator	Meaning
<-	Property Assignment
->	Lambda Operator
::	List Concatenation (cons)
|>	Forward Operator
[[ ]	Index Operator
:?	Pattern Match .NET Type

And many many more. It's a massive array of built-in operators...

Back to the subject at hand. As in C# and other .NET languages, the operators are overloaded. But unlike C# for example, both operands for the operator must be of the same type. But unlike C#, F# will allow you to not only define, but redefine operators as well. Below I'll show an example of using the common concat operator with strings and then redefine the division operator to do something really stupid and do multiplication instead. Aren't I crafty?

```
#light


let fooBar = "Foo" + "Bar"

print_string fooBar


let (/) a b = a * b

printfn "%i" (4 / 2)
```

So, as you can see it's rather powerful what we can do with existing operators. But, with F#, we can also define our own operators we wish to use. In fact we can use any of the below as our new operator.

! $ % & * / + - . < = > ? @ ^ | ~ as well as a :

As I did above, it should be no different for us to create our new operator. Let's walk through a simple operator that does raise to the power of two of the first operand and add the second.

```
#light


let ( *:/ ) a b = (a * a) + b

printfn "%i" (45 *:/ 42)
```

If you're curious like me what that actually looks like, let's look through .NET Reflector and see what it creates for us. I'll just copy and paste the results here:

```
public static int op_MultiplyColonDivide(int a, int b)
```

```
{
    return ((a * a) + b);
}
```

But we can also define operators for our custom types as well. Let's go ahead and define a simple type called Point and then define the addition and subtraction operators. This syntax should look rather familiar and to me, it looks a little Ruby-esque.

```
#light


type Point(dx:float, dy:float) =
  member x.DX = dx
  member x.DY = dy


  static member (+) (p1:Point, p2:Point) =
    Point(p1.DX + p2.DX, p1.DY + p2.DY)


  static member (-)(p1:Point, p2:Point) =
    Point(p1.DX - p2.DX, p1.DY - p2.DY)


  override x.ToString() =
    sprintf "DX=%f, DY=%f" x.DX x.DY


let p1 = new Point(5.0, 6.0)
let p2 = new Point(7.0, 8.0)
let p3 = p2 - p1
let p4 = p2 + p1


printfn "p3 = %s" (p3.ToString())
printfn "p4 = %s" (p4.ToString())
```

Lists

Lists are one of the most fundamental pieces that is built right into the F# language itself. The language itself around the use of lists is a bit verbose, but I'll cover the basics through the use of code. Lists in F# are immutable, meaning that once they are created, they cannot be altered. Think of them like a .NET System.Array in a way, but a little bit more powerful. Do not get them confused though with F# Arrays which are entirely different and will be talked about later. Let's show the basics here below about lists:

```
#light


let empty = [] (* Empty List *)
let oneItemInList = 1 :: [] (* One Item *)
let twoItemsInList = 1 :: 2 :: [] (* Two Items *)
List.iter(fun x -> printfn "%i" x) twoItemsInList (* Iterate Over List *)


let threeItems = [1 ; 2 ; 3] (* Shorthand notation *)
let listOfList = [[1 ; 2 ; 3]; [4 ; 5; 6]; [7 ; 8 ; 9]]
List.iter(fun x -> print_any x) listOfList (* Iterate over list of list *)
```

As you can see from the example above there are a couple ways of declaring lists, including the verbose way which includes the empty list, or the classic style of putting inside of square brackets. What's more interesting is to have lists of lists as well.

Another interesting aspect of F# lists is List Comprehensions. Using this syntax, we can specify ranges to use in our lists. If you specify a double period (..) in your list, it fills in the rest of the values. I have three samples below which includes 0 to 100, lower-case a to z and upper-case A to Z.

You can also specify steps in the numbers as well which is much like a for loop with a step in there. Note that characters do not support the stepping capability. To use the stepping capability, the format is (lower bound, step value, upper bound). Below I have a sample which pulls all even numbers from 100 to 0 and stepping -2 each time.

For loops can also be used in which you can transform each item along the way. To use this one, use this for keyword and the lower end to upper end and then the lambda to modify the value. Below I have an example where I double each number from 1 to 20.

Another example of a for loop is to create a loop in which you specify a when guard. You can then only print out values that match your criteria. The example I have below is to print all odd numbers using the formula when x % 2 <> 0.

```
#light


let zeroToOneHundred = [0 .. 100] (* 0 to 100 *)

let lowerCaseAToZ = ['a' .. 'z'] (* Lower-case a-z *)

let upperCaseAToZ = ['A' .. 'Z'] (* Upper-case a-z *)

let evensFromOneHundred = [100 .. -2 .. 0] (* 100 to 0 skipping every 2 *)

let doubledValues =

  { for x in 1 .. 20 -> x + x } (* Double each value from 1 to 20 *)


let odds n =

  { for x in 1 .. n when x % 2 <> 0 -> x } (* Print all odd numbers *)

print_any (odds 20)
```

And I've only just begun to touch the basics of lists. Dustin Campell also covered lists recently on his Why I Love F# Series.

Conclusion

We still have plenty to go before I'd say we were done with a 101 level exercise with F#. We still have yet to cover pattern matching, lazy evaluation, exception management and even using F# with imperative and object oriented code. So, subscribe and stick around. Until next time...

Part 5 - Pattern Matching

Time for another adventure in F#, covering the 101 level basics of the language and why I think it's useful and how it can even help your C# as well. This time, I want to spend a good deal of time on pattern matching and a few other topics.

Where We Are

Before we begin today, let's catch up to where we are today:

- Part 1 - Basic functional programming
- Part 2 - Currying and Tuples
- Part 3 - Scope, Recursion and Anonymous Functions
- Part 4 - History of F#, Operators and Lists

So, today, like I mentioned before, here are the topics I'm going to cover today:

- Pattern Matching
- Active Patterns

Pattern Matching

One of the interesting and more powerful features of the F# language is Pattern Matching. Don't be tempted to think of them as simple switch statements as they are much more powerful than that. Pattern matching is used to test whether the piece under test has a desired state, find relevant information, or substitute parts with other parts. Most functional languages such as Haskell, ML, and OCaml have them, and F# is no exception to its support.

At first glance, like I said, you might be tempted to think of them as the C# switch statement. But, this is more powerful as you can have ranges support, can case on .NET types and plenty of other things you cannot do in other .NET languages. Let's walk through a simple example of the old Fibonacci sequence that I've shown in the past. But this time, I'll walk through it several times as I make improvements each iteration by adding more features to support failover, etc.

In order to use pattern matching with explicit values, you can use the **match** and **with** keywords. This allows you to match a particular parameter with the following conditions.

```
#light


let rec fib n =

  match n with

  | 0 -> 1

  | 1 -> 1

  | x -> fib(n - 2) + fib(n - 1)


print_any (fib (-1))
```

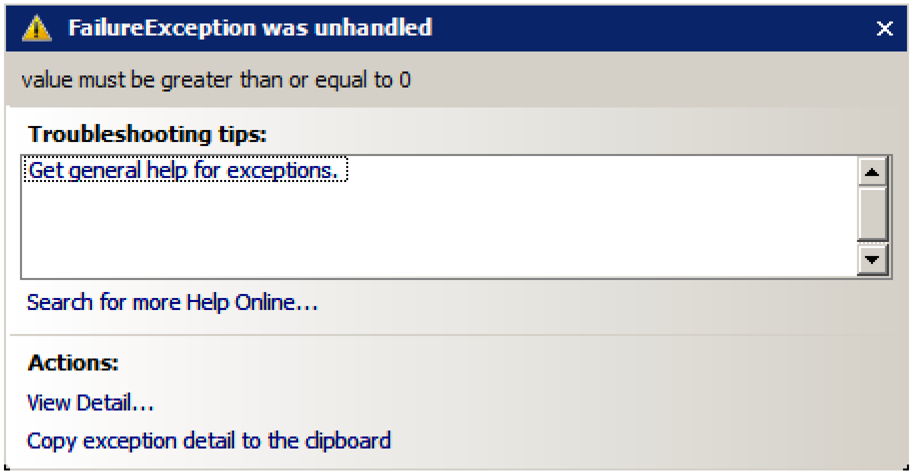
Ok, so what we have here is a simple one with pattern matching which checks for a 0, then a 1, and then the default case goes to do the Fibonacci sequence calculation. The other two would be considered escape clauses for the last calculation. But, we can do better than this. Let's try again by compacting it just a little.

```
#light


let rec fib n =

  match n with

  | 0 | 1 -> 1

  | x -> fib(n - 2) + fib(n - 1)


print_any (fib (-1))
```

Now the code is a bit more compact as we can combine two values on one line as you can see. But, we haven't hit a fail case if the value is less than 0. That's not a good thing. Luckily F# gives us a failure function to deal with something like this. We can use the when clause if we want to check for ranges. Also, we can specify that the condition is a failure with the failwith keyword. Let's now walk through the more complete scenario.

```
#light


let rec fib n =

  match n with

  | x when x < 0 -> failwith "value must be greater than or equal to 0"

  | 0 | 1 -> 1

  | x -> fib(n - 2) + fib(n - 1)


print_any (fib (-1))
```

And sure enough when you run the program through the debugger, you get the nice picture that tells you of the exact error. It fails with a FailureException and our message shows up as well.



FailureException was unhandled

value must be greater than or equal to 0

Troubleshooting tips:

Get general help for exceptions.

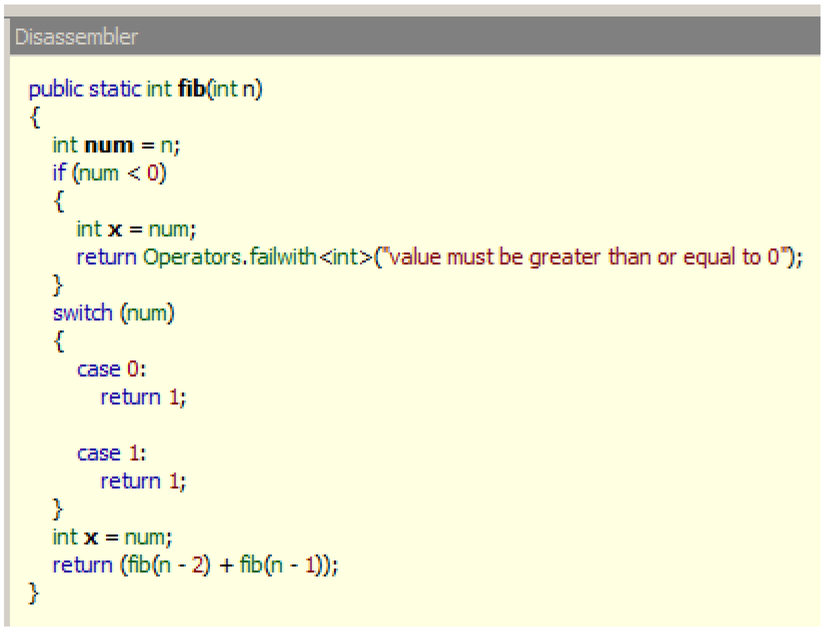
Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

What I always like to do in situations such as these is to look through .NET Reflector to look at how it might show up if it were rendered in C#.



Disassembler

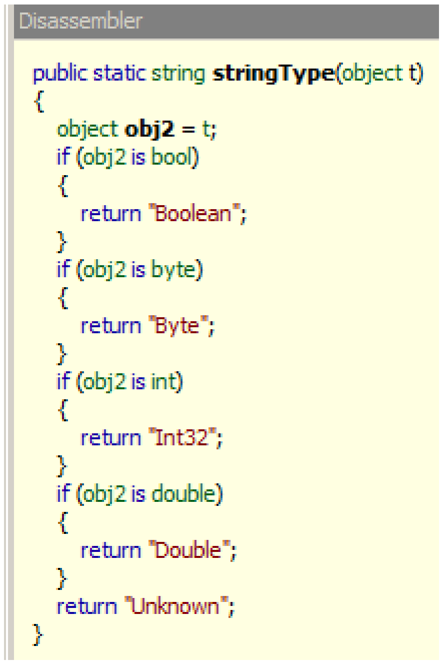
```
public static int fib(int n)
{
   int num = n;
   if (num < 0)
   {
      int x = num;
      return Operators.failwith<int>("value must be greater than or equal to 0");
   }
   switch (num)
   {
      case 0:
         return 1;

      case 1:
         return 1;
   }
   int x = num;
   return (fib(n - 2) + fib(n - 1));
}
```

Pretty simple as it uses the switch statements when it can, and then uses the failure outside of it. As you can see, the F# is much cleaner and more to the point than this stuff. Jacob Carpenter took it to a different level though in C# when he went through Generics and Lambda abuse to reproduce the same actions. You can read more about that here.

But, that's easy stuff that can be easily done by C#, albeit differently. Let's try for matching against .NET types instead, which is something C# cannot do via the switch statement. So, in a words, you can definitely tell that it's a bit more powerful than just a simple C# switch statement.

```
#light


let stringType (t : ob) =

  match t with

  | :? System.Boolean -> "Boolean"

  | :? System.Byte -> "Byte"

  | :? System.Int32 -> "Int32"
  | :? System.Double -> "Double"
  | _ -> "Unknown"


print_string (stringType (box 14uy))

print_string (stringType (box false))

print_string (stringType (box "Foo"))
```

What the above example does is check for the corresponding .NET types by using the :? operator especially reserved for this behavior. Also, note the use of the underscore **"_"** as a wildcard approach to this. Let's take a look at the results in .NET Reflector again to see what the results look like. Again, it's nothing spectacular just yet.

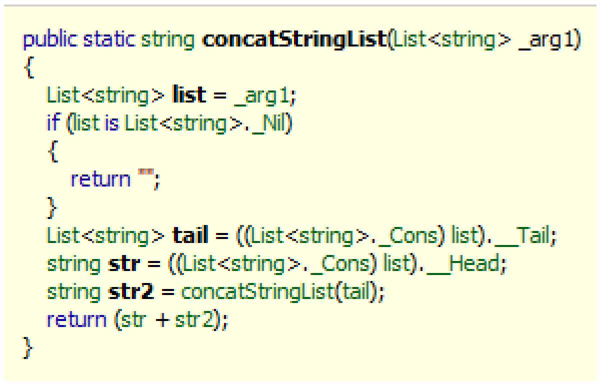


```
public static string stringType(object t)
{
    object obj2 = t;
    if (obj2 is bool)
    {
        return "Boolean";
    }
    if (obj2 is byte)
    {
        return "Byte";
    }
    if (obj2 is int)
    {
        return "Int32";
    }
    if (obj2 is double)
    {
        return "Double";
    }
    return "Unknown";
}
```

Let's look at yet another sample when calling a recursive function. This time, let's concatenate a few strings together from an F# list. This is a pretty simple example of using pattern matching to tell when to stop adding onto the existing string.

```
#light


let stringList = ["foo"; "bar"; "foobar"; "barfoo"]


let rec concatStringList = function

  | head :: tail -> head + concatStringList tail
  | [] -> ""


print_string (concatStringList stringList)
```

So, what we're doing is while we have a value to concatenate onto the tail, then add it to the list, else if it's an empty list, let's return a blank string. The keywords head and tail are actually built into the F# language as part of F# lists. The **head** is the head value on the list, in our case, a string. The **tail** represents the rest of the list. What's really cool is that ideas from functional programming can actually help your C# and imperative code thinking. We'll get into at a later time... Anyways, I'm always curious what that looks like in Reflector and how it gets translated into C#-ese.

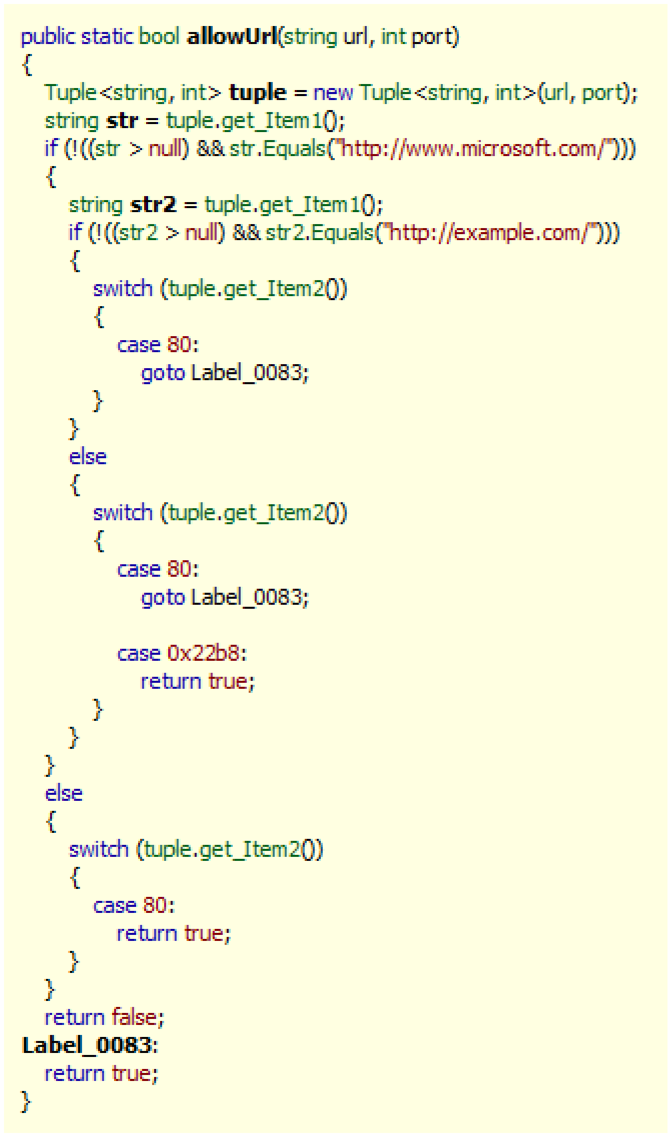


```
public static string concatStringList(List<string> _arg1)
{
    List<string> list = _arg1;
    if (list is List<string>._Nil)
    {
        return "";
    }
    List<string> tail = ((List<string>._Cons) list).__Tail;
    string str = ((List<string>._Cons) list).__Head;
    string str2 = concatStringList(tail);
    return (str + str2);
}
```

You can also match over two or more parameters as well. This is especially interesting if wanting to cover all variations of a particular combination without messy C# if statements. Let's take a look at matching over a tuple whether we allow a URL to be accessed or not:

```
#light


let allowUrl url port =

  match (url, port) with

  | "http://www.microsoft.com/", 80 -> true

  | "http://example.com/", 8888 -> true

  | _, 80 -> true

  | _ -> false


let allowMicrosoft = allowUrl "http://www.microsoft/" 80

printfn "%b" allowMicrosoft
```

What seems like simple code is actually quite squirrley if you want to translate this into C#. As you can see from the above code, if we pass in the Microsoft address and 80, we can return a true value, else look at the other conditions and if all else fails, then we return false. But, let's take a look at how it's translated into C#.



```
public static bool allowUrl(string url, int port)
{
    Tuple<string, int> tuple = new Tuple<string, int>(url, port);
    string str = tuple.get_Item1();
    if (!((str > null) && str.Equals("http://www.microsoft.com/")))
    {
        string str2 = tuple.get_Item1();
        if (!((str2 > null) && str2.Equals("http://example.com/")))
        {
            switch (tuple.get_Item2())
            {
                case 80:
                    goto Label_0083;
            }
        }
        else
        {
            switch (tuple.get_Item2())
            {
                case 80:
                    goto Label_0083;

                case 0x22b8:
                    return true;
            }
        }
    }
    else
    {
        switch (tuple.get_Item2())
        {
            case 80:
                return true;
        }
    }
    return false;
Label_0083:
    return true;
}
```

Definitely handled very elegantly in our F# code as it's one of the strengths. With C#, not as much as you can see from this pretty ugly code that it represents. Matching against enums is pretty easy as well with very little coding effort such as this:

```
#light


let calcRateByDay (day:System.DayOfWeek) =

  match day with

  | System.DayOfWeek.Monday -> 0.42

  | System.DayOfWeek.Tuesday -> 0.67

  | System.DayOfWeek.Wednesday -> 0.56

  | System.DayOfWeek.Thursday -> 0.34

  | System.DayOfWeek.Friday -> 0.78

  | System.DayOfWeek.Saturday -> 0.92

  | System.DayOfWeek.Sunday -> 0.18

  | _ -> failwith "Unexpected enum value"


print_any (calcRateByDay System.DayOfWeek.Monday)
```

So, just wrapping up this section for now, as you can notice, we can do all sorts of things with pattern matching and I've barely scratched the surface of it. I won't begin to cover things as pattern matching over unions just yet until I get to that section where I want to talk about them exclusively.

Active Patterns

Now that we understand how pattern matching works, let's take this to a different level. These active patterns allow you to create a union type of various .NET classes and match against them. Unfortunately, Robert Pickering couldn't fit the active patterns into his Foundations of F# book, so you can go ahead instead and read about them here. Also, Luis Diego Fallas posts about them here as well.

To give you a brief sample of what one looks like, let's look through a sample of whether we're looking through files or directories.

```
#light


let (|File| Directory|) (fileSysInfo : System.IO.FileSystemInfo) =

  match fileSysInfo with

  | :? System.IO.FileInfo as file -> File (file.Name)

  | :? System.IO.DirectoryInfo as dir -> Directory (dir.Name, { for x in dir.GetFileSystemInfos() -> x })

  | _ -> assert false
```

What this allows us to do work with Files as tree structures. You can also use these to walk XML documents and so on. Read the above samples to get more information about them.

Conclusion

Pattern Matching to me is one of the most fundamental pieces to F# and functional programming as a whole. What I've started you with are some basic samples, but these can be applied to binary trees, complex objects and so on. I hope to cover more scenarios soon, and until next time...

Part 6 - Lazy Evaluation

Time for another adventure in F#, covering some of the basics of functional programming and F# in particular. This is intended at looking not only at the language, but the implementation as it regards to C#.

Where We Are

Before we begin today, let's catch up to where we are today:

- Part 1 - Basic functional programming
- Part 2 - Currying and Tuples
- Part 3 - Scope, Recursion and Anonymous Functions
- Part 4 - History of F#, Operators and Lists
- Part 5 - Pattern Matching

So, today we'll be covering the topic of lazy evaluation, so, without further ado, let's get started.

Lazy Evaluation

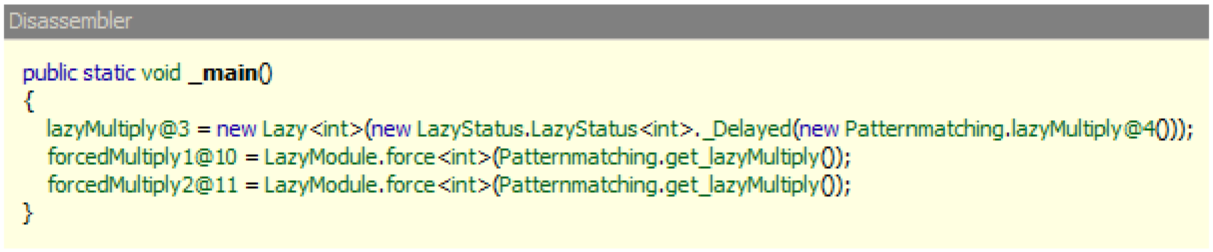
Another fascinating topic in the land of functional programming is lazy evaluation. This technique is basically delaying the execution of a given code block until it is needed. Imagine for a minute that you have a function that is basically an if-else function. This function would take a boolean condition, and if false, it will execute the second block of code. The last thing you'd want to do is evaluate the else statement if the if is true and the results could be disastrous. Instead, we can delay that result until we ultimately need it. But why is it useful? Well, think of infinite calculations, or infinite loops, the last thing you want to do is evaluate all of them, and instead do it lazily on command.

It's one of the more important features in languages such as Haskell where Simon Peyton Jones, the principal designer of the Glasgow Haskell Compiler (GHC), really likes this features and talks about it extensively onDotNetRocks Episode 310. By default, all function parameters computation are delayed by default. If you'd like to see more about Haskell's lazy evaluation of a Fibonacci sequence, check it out on the HaskellWiki.

Back to F# now. When we talk about a pure functional language, the compiler should be free to choose which order to evaluate the argument expressions. Instead, F# does allow for side effects in this way, such as printing to the console, writing to a file, etc, so being able to hold off on those kinds of items just aren't possible. F# by default does not perform lazy evaluation, instead has what is called eager evaluation. To take advantage of lazy loading, use the **lazy** keyword in the creation of your function and wrap the area to be lazy evaluated inside that block. You then use the **Lazy.force** function to evaluate the expression. Once that is called for the first time, the results is then cached for any subsequent call. Let's walk through a quick sample of how that works:

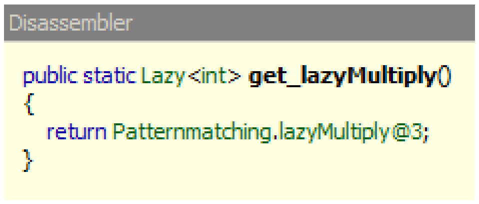
```
#light


let lazyMultiply =

  lazy

  (

    let multiply = 4 * 4

    print_string "This is a side effect"

    multiply

  )


let forcedMultiply1 = Lazy.force lazyMultiply
let forcedMultiply2 = Lazy.force lazyMultiply
```

What you'll notice from the above sample is that I intentionally put a side effect inside my lazy evaluated code. But, since the evaluation only happens once, then I should only see it in my console once, and sure enough, when you run it, it does. This is called memoization, which is a form of caching. This uses the Microsoft.FSharp.Control.Lazy class. But, I wonder how it's actually done through the magic of IL. Since I think C# is a little bit more readable, let's crack open .NET Reflector and take a look.



```
public static void _main()
{
    lazyMultiply@3 = new Lazy<int>(new LazyStatus.LazyStatus<int>._Delayed(new Patternmatching.lazyMultiply@4()));
    forcedMultiply1@10 = LazyModule.force<int>(Patternmatching.get_lazyMultiply());
    forcedMultiply2@11 = LazyModule.force<int>(Patternmatching.get_lazyMultiply());
}
```

What we notice is that it creates a new class for us called lazyMultiply. There is a lot of syntactic sugar behind the scenes, but as you can see, it's calling the same get_lazyMultiply function off my main class. It's nothing more than a simple property that does this:



```
public static Lazy<int> get_lazyMultiply()
{
    return Patternmatching.lazyMultiply@3;
}
```

If you'd like to dig more into the guts of how that works exactly, check out Jomo Fisher's blog post about the Lazy Keyword here. I'm definitely glad to see F# team members blogging about these sorts of things.

Lazy Collections?

F# also has the notion of lazy collections. What this means is that you can calculate the items in your collection upon demand. Some of the collections inside the F# library may also cache those results as well. We have two types that we're interested in, the LazyList class (full name: Microsoft.FSharp.Compatibility.FSharp.LazyList in FSharp.Compatibility.dll) and the Seq class (full name: Microsoft.FSharp.Collections.Seq in FSharp.Core.dll).

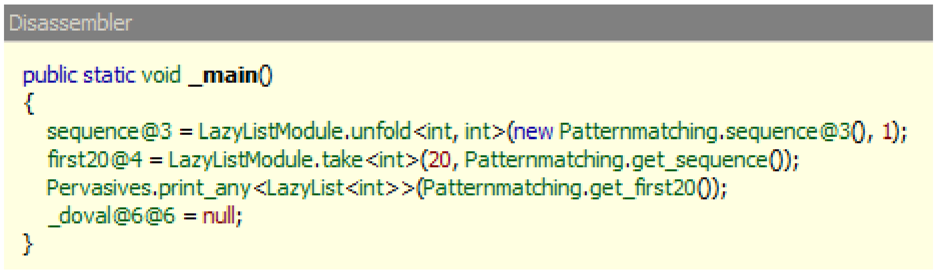
The LazyList is a list which caches the results of the computed results on the items inside the collection. In order to create a truly lazy list, you must use the unfold function. What this does is returns a list of values in a sequence starting with the given seed. The computation itself isn't computed until the first one is accessed. The rest of the elements are calculated by the residual and I'll show you that below:

```
#light


let sequence = LazyList.unfold (fun x -> Some(x , x + 1)) 1
let first20 = LazyList.take 20 sequence

print_any first20
```

What the above sample allows us to do is create a list of infinite numbers, starting at the seed of 1. But, the values aren't computed until the first is accessed, and then the results are cached. But, in order to access any value out of them, you need to take x number of values from them. The LazyList provides that capability to do that with the **take** function. What you'll also notice is that we can use the None or Some functions. The **Some**function, as shown above has its first value as the starting value, and the second value is what it will be like the next time it will be called. The **None** function, not shown above, represents the end of the list.

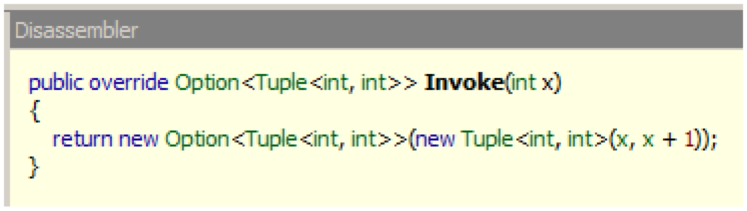
You may also use the Seq class to represent these lazy values. Think of the Seq as the magic collection class inside F# as it compatible with most, if not all arrays, and most collections as well.

But, what does that code look like as well? Very interesting question. Let's take a look at the main function first:



```
public static void _main()
{
    sequence@3 = LazyListModule.unfold<int, int>(new Patternmatching.sequence@3(), 1);
    first20@4 = LazyListModule.take<int>(20, Patternmatching.get_sequence());
    Pervasives.print_any<LazyList<int>>(Patternmatching.get_first20());
    _doval@6@6 = null;
}
```

Ok, so that looks a bit interesting. So, what does the sequence@3 really look like. What we see is that we're giving it a seed of 1 and then getting the first 20 from the list. Let's now take a look at the sequence@3. What's we're interesting in is not the actual constructor, but the Invoke method as you remember, it's a lazy list.

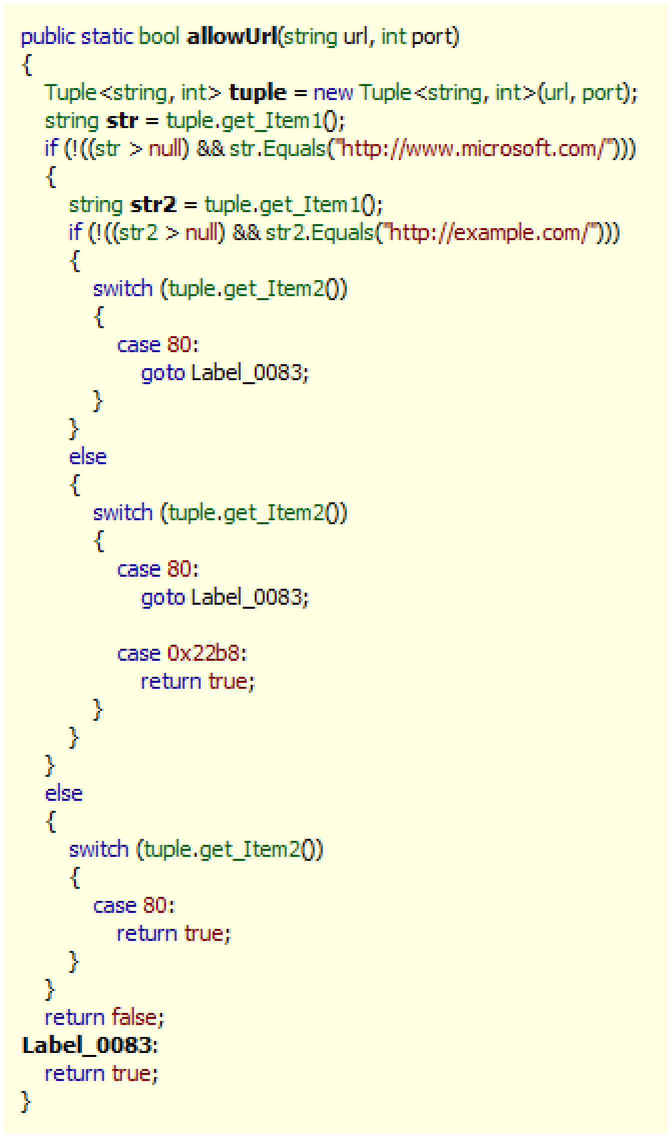


```
public override Option<Tuple<int, int>> Invoke(int x)
{
    return new Option<Tuple<int, int>>(new Tuple<int, int>(x, x + 1));
}
```

So, as you can see, it sure enough looks like we had in our code from above with a heck of a lot more angle brackets. I'm sure that's why some of the var stuff was introduced in C# 3.0 was to hide this ugliness.

Conclusion

Just to wrap things up here on a Friday, I hope you enjoyed looking at some of the possibilities of lazy loading. This could also help with Abstract Syntax Trees, loading of data from files, XML documents, etc. The real power is to take infinite data loops and take chunks of them at a time. And we really don't have to worry about the stack as well when we do this. I hope you've enjoyed the series so far and there are a few more things I'd want to cover before I consider this series done. Until next time...



```
public static bool allowUrl(string url, int port)
{
    Tuple<string, int> tuple = new Tuple<string, int>(url, port);
    string str = tuple.get_Item1();
    if (!((str > null) && str.Equals("http://www.microsoft.com/")))
    {
        string str2 = tuple.get_Item1();
        if (!((str2 > null) && str2.Equals("http://example.com/")))
        {
            switch (tuple.get_Item2())
            {
                case 80:
                    goto Label_0083;
            }
        }
        else
        {
            switch (tuple.get_Item2())
            {
                case 80:
                    goto Label_0083;

                case 0x22b8:
                    return true;
            }
        }
    }
    else
    {
        switch (tuple.get_Item2())
        {
            case 80:
                return true;
        }
    }
    return false;
Label_0083:
    return true;
}
```

Definitely handled very elegantly in our F# code as it's one of the strengths. With C#, not as much as you can see from this pretty ugly code that it represents. Matching against enums is pretty easy as well with very little coding effort such as this:

```
#light


let calcRateByDay (day:System.DayOfWeek) =
  match day with
  | System.DayOfWeek.Monday -> 0.42
  | System.DayOfWeek.Tuesday -> 0.67
  | System.DayOfWeek.Wednesday -> 0.56
  | System.DayOfWeek.Thursday -> 0.34
  | System.DayOfWeek.Friday -> 0.78
  | System.DayOfWeek.Saturday -> 0.92
  | System.DayOfWeek.Sunday -> 0.18
  | _ -> failwith "Unexpected enum value"

print_any (calcRateByDay System.DayOfWeek.Monday)
```

So, just wrapping up this section for now, as you can notice, we can do all sorts of things with pattern matching and I've barely scratched the surface of it. I won't begin to cover things as pattern matching over unions just yet until I get to that section where I want to talk about them exclusively.

Active Patterns

Now that we understand how pattern matching works, let's take this to a different level. These active patterns allow you to create a union type of various .NET classes and match against them. Unfortunately, Robert Pickering couldn't fit the active patterns into his Foundations of F# book, so you can go ahead instead and read about them here. Also, Luis Diego Fallas posts about them here as well.

To give you a brief sample of what one looks like, let's look through a sample of whether we're looking through files or directories.

```
#light

let (|File| Directory|) (fileSysInfo : System.IO.FileSystemInfo) =
  match fileSysInfo with
   | :? System.IO.FileInfo as file -> File (file.Name)
   | :? System.IO.DirectoryInfo as dir -> Directory (dir.Name, { for x in dir.GetFileSystemInfos() -> x })
   | _ -> assert false
```

What this allows us to do work with Files as tree structures. You can also use these to walk XML documents and so on. Read the above samples to get more information about them.

Conclusion

Pattern Matching to me is one of the most fundamental pieces to F# and functional programming as a whole. What I've started you with are some basic samples, but these can be applied to binary trees, complex objects and so on. I hope to cover more scenarios soon, and until next time...

Part 7 - Creating Types

Time for another adventure in F#, covering some of the basics of functional programming and F# in particular. This is intended at looking at the foundations of F# as well as how it relates to .NET and IL on the back end. I realize I need to spread more of the F# goodness around, so I'm hoping that I can work to bring it at least to the FringeDC user group. Their main meetings are every three months, so hopefully I'll get some time in to do that. Once I get that set up, I'll be sure to let everyone know.

Where We Are

Before we begin today, let's catch up to where we are today:

- Part 1 - Basic functional programming
- Part 2 - Currying and Tuples
- Part 3 - Scope, Recursion and Anonymous Functions
- Part 4 - History of F#, Operators and Lists
- Part 5 - Pattern Matching
- Part 6 - Lazy Evaluation

Today's topic will be covering creating custom types in F#. There is a lot to cover, so let's get started.

Types of Types?

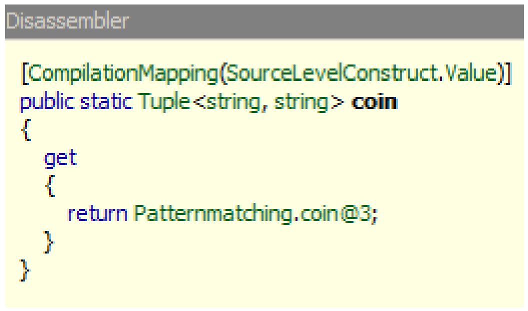
If you're familiar with OCaml, the type system in F# should look rather familiar. For a brief introduction to how OCaml works, check out Dr. Jon Harrop's OCaml for Scientists introduction here. We have several categories of types in base F#. The first category is the tuples or records, and the second type are unions.

Tuples

The most simple type in F# is a tuple. A tuple is a compound type composed of a fixed sequence of other types. Each value is separated by a comma and can be referred to by a single identifier. The values from the tuple can be retrieved by putting commas on the other side of the equals. Below is a quick sample of this concept:

```
#light

let coin = "heads", "tails"
let c1, _ = coin
let _, c2 = coin

print_string c1
print_string c2
```

What I did to look up the values was to use the _ to indicate I wasn't interested in the other values. Hence my c1 got me heads and c2 got me tails. As always, I'm always curious how it compiles down to C# and how it gets represented in IL, so let's take a look at the coin type that I created.



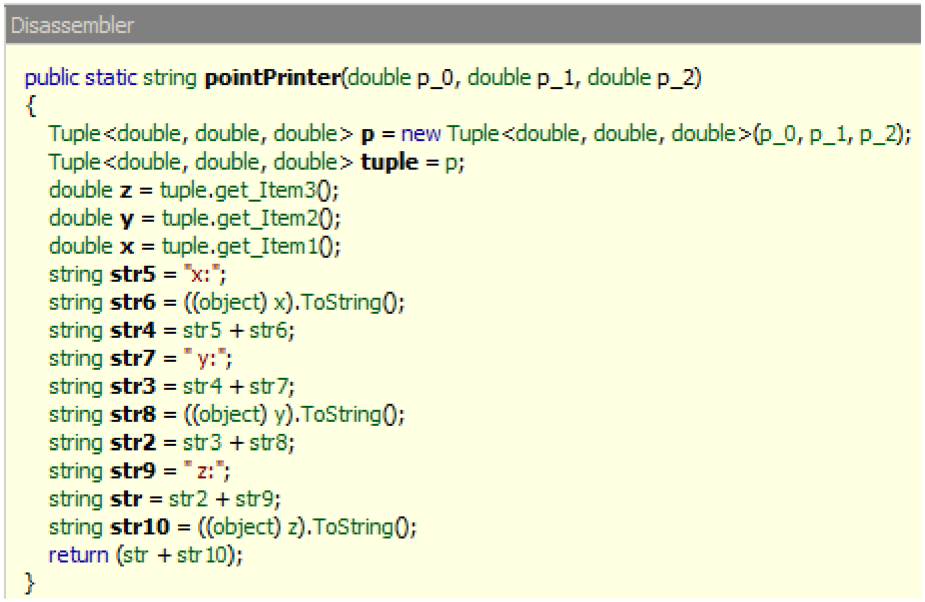
Disassembler

```
[CompilationMapping(SourceLevelConstruct.Value)]
public static Tuple<string, string> coin
{
   get
   {
      return Patternmatching.coin@3;
   }
}
```

Nothing really fancy, as we created a tuple of type string and string which held our "heads" and "tails" value. Like I said before, tuples are the simplest type in F#. In fact, as you saw above I didn't even need to use the type keyword to specify I was creating a new type. However, I can choose to use the type keyword so that I can create a type alias which can be very useful for method signature constraint. Below is a simple sample of doing that.

```
#light

type Point3D = double * double * double

let pointPrinter(p : Point3D) =
  let x, y, z = p in
  "x:" + x.ToString() + " y:" + y.ToString() + " z:" + z.ToString()

let pointString = pointPrinter(2.3, 5.3, 9.8)
print_string pointString
```

As you can see from above, I created a 3D point which contains an x, y and z axis. Then I constrained the pointPrinter function to take that as a constraint so that I can format it properly. Then of course I do all the great things I normally would do by printing it out to the console. But, if you want to take a look at what this pointPrinter function does behind the scenes, wonder no longer.


Disassembler

```
public static string pointPrinter(double p_0, double p_1, double p_2)
{
   Tuple<double, double, double> p = new Tuple<double, double, double>(p_0, p_1, p_2);
   Tuple<double, double, double> tuple = p;
   double z = tuple.get_Item3();
   double y = tuple.get_Item2();
   double x = tuple.get_Item1();
   string str5 = "x:";
   string str6 = ((object) x).ToString();
   string str4 = str5 + str6;
   string str7 = " y:";
   string str3 = str4 + str7;
   string str8 = ((object) y).ToString();
   string str2 = str3 + str8;
   string str9 = " z:";
   string str = str2 + str9;
   string str10 = ((object) z).ToString();
   return (str + str10);
}
```

Quite an interesting mouthful, I must admit, but that's what you get when you deal with immutable types such as strings.

Records

Record types are also a user defined data structure within F#. These are similar to tuples, but differ because each piece of data (field) inside a record must be named as well as the keyword type must be used. Think of this much as a .NET class with simple properties. What's interesting is that these fields that it creates for us aren't forced to be unique. Let's go ahead and show a simple example of a Person record that we may want to create.

```
#light

open System

type Person =
  {
    FirstName : string;
    MiddleName : string;
    LastName : string;
    DateOfBirth : DateTime;
  }

let person1 = { FirstName="Robert"; MiddleName="William"; LastName="Jones"; DateOfBirth = new DateTime(1960,
12, 12); }
let person2 =
  { new Person
    with FirstName = "William"
    and MiddleName = "Franklin"
    and LastName = "Smith"
    and DateOfBirth = new DateTime(1970, 1, 23)
  }



print_string person1.FirstName

print_string person2.FirstName
```

What I have done is created a Person record with a FirstName, MiddleName, LastName and a DateOfBirth field. Pretty simple. Now, we have two ways of creating an instance of this type. When I created person1, I just simply just set each field separated by a semicolon. On the other hand, since I'm not explicitly naming a type, there could be conflicts due to multiple types containing the same fields. Should I want to be more explicit, I can specify it using the longhand way with using the new operator and setting each property with the with keyword and followed by an and on each subsequent line to fill in all the other values. Such a conflict would look like this:

```
#light

type point2D = { X:double; Y:double }
type coordinate2D = { X:double; Y:double }

let myPoint = { X = 2.5; Y = 3.2; }
```

What the above sample will default to is that myPoint will be of type coordinate2D. Hover over and let Intellisense do its magic. Interesting, huh? That's why we should be explicit about our data structures should something like this arise.

Discriminated Unions

The next category of F# type that will be discussed is the discriminated union. This type is a data structure that can bring together values but have different meanings or structures. Only one of these types can be used at once, however. Think of it more of a type which is almost "case" like. Each piece of that is called a discriminator. A quick example of this would be something like distance where it could be measured in miles or kilometers, but not both at the same time. The structure of this data is the same, but have different meanings to the program.

Let's look at some quick examples of this:

```
#light

type Distance =
  | Kilometer of double
  | Mile of double

let distance1 = Mile 26.2
let distance2 = Kilometer 10.0

let convertDistanceToMile x =
  match x with
  | Mile x -> x
  | Kilometer x -> x * 1.609344

let convertDistanceToKilometer x =
  match x with
  | Mile x -> x * 0.621371192
  | Kilometer x -> x

let convertedDistance1 = convertDistanceToKilometer distance1
let convertedDistance2 = convertDistanceToMile distance2
```

As you can see from above, I created a Distance discriminated union that defines both miles and kilometers. Both discriminators are of type double. So, I wrote a function to convert from one to the other. It's just a simple pattern matching statement. If you've read my previous posts, you should be caught up to date with this.

I could also parameterize this data differently, meaning that each discriminator can hold a different value.

```
#light

type Carrier = string
type Route = int
type Make = string
type Model = string
type Year = int

type ModeOfTransport =
  | Airplane of Carrier * Route
  | Bus of Carrier * Route
  | Car of Make * Model

let mode1 = Car("Audi", "A4")
let mode2 = Airplane("United", 222)
```

As you can see, I type aliased a few things such as Carrier, Route, Make, Model and so on. After that, I defined my ModeOfTransport to be by bus, car or airplane. A pretty simple example.

I can also specify the type of arguments later, by specifying them as generics. One of them you may have already run across is the option type. To declare one, just use the <'a> or whichever letter you so choose. Let do a recursive example of a tree.

```
#light

type Tree<'t> =
  | TreeNode of 't Tree * 't Tree
  | TreeValue of 't

let tree =
  TreeNode(
    TreeNode(TreeValue 0, TreeValue 1),
    TreeNode(TreeValue 2, TreeValue 3))
```

From the above statement, I created a tree of nodes quite easily using this union type. These types can get complicated quickly, however when you get into language oriented programming, so we're just dipping our toes in at the moment.

.NET Types?

Yes, you can also create .NET types as well just as easily, although with a few major details. For example, the mutable types which is not by default in F#. I'll cover that in the next post.

Conclusion

As you can see, there are a few basic types in F#. They are mostly for holding analytical data, traversing them and so on. These are interesting and serve as the basis of what you will do in F# when you need to store and analyze data, especially custom types. In the next installment, I hope to cover some .NET specific topics such as creating .NET types. Until next time...

Part 8 - Mutables and Reference Cells

Time for another adventure in F#, covering some of the basics of functional programming and F# in particular. Today we'll manage to look more at regular .NET integration and .NET programming. With the previous efforts, we've looked more at functional programming and in turn F# specific things, but want to show that you can do anything normally in F# that you can in C#. To me, F# is the perfect all-purpose language because it can do a lot of the things C# can do, but in turn, F# can do things much more elegantly than C# can, such as Pattern Matching.

Where We Are

Before we begin today, let's catch up to where we are today:

- Part 1 - Basic functional programming
- Part 2 - Currying and Tuples
- Part 3 - Scope, Recursion and Anonymous Functions
- Part 4 - History of F#, Operators and Lists
- Part 5 - Pattern Matching
- Part 6 - Lazy Evaluation
- Part 7 - Creating Types

Today's topic will be covering imperative and object oriented programming in F#. There is a lot to cover, so let's get started. But there are a few administrative things to get out of the way first.

Learning F# ala Ted Neward?

Ted Neward recently announced on DotNetRocks Episode 332 that he's in the process of creating a class for F# for Pluralsight. That should be interesting to those who are interested in this series, as well as F# in general. Right now the community is rather small, so efforts like this should be rather rewarding I would hope. Ted's a pretty brilliant guy, so I'd imagine only the best. I'm hoping more details come out soon.

Pattern Matching in C#

Part of this series is intended to bring such concepts as Pattern Matching, Currying and other Functional Programming concepts to the C# developer. After all, the more C# language evolves, the more it seems to fall into the Functional Programming category. In previous posts, I showed how to relate currying to C# and it was less elegant than F# to say the least.

But, let's look at Pattern Matching. Bart De Smet has been posting recently on his blog about bringing the beauty of pattern matching to C#. So far it's been a good six posts into it and I urge you to go ahead and take a look at this series.

- Pattern Matching in C# - Part 0
- Pattern Matching in C# - Part 1
- Pattern Matching in C# - Part 2
- Pattern Matching in C# - Part 3
- Pattern Matching in C# - Part 4
- Pattern Matching in C# - Part 5
- Pattern Matching in C# - Part 6
- Pattern Matching in C# - Part 7

But when you read the series, it's all about getting into the low level and compiling expression trees to make the same simple beauty that is F#. Sure, it can be done in C#, but nowhere near as elegant. Performance is another issue that comes to mind with these.

Imperative Programming in F#

This section I'll lay out some of the basics of imperative style programming before I get into the full object oriented approach to programming. So, we'll cover just a few topics and then I'll feel comfortable moving onto the real parts of creating classes and such. We'll cover such things as void types and mutability in this section.

The unit Type

One of the first things I forgot to mention when describing F# functions and values in the unit type. Think of this as the void type in C# that you are used to. It's the type that doesn't accept or return a value. First, let's look at the typical C# program with the void type for Hello World.

```
    class Program

    {

        static void Main(string[] args)

        {

            Console.WriteLine("Hello World");

        }

    }
```

Now, let's go ahead and look at it from the F# perspective.

```
#light

let main() =
  printfn "Hello World"

main()
```

As you will see when you hover over our code is that it is a unit type. That in itself isn't all that interesting. But, what we'll run into is problems when functions return a value, but we're not all that interested in them. What happens? Well, F# will complain that your return value isn't compatible with the unit type, which is essentially true. So, how do you get around that? Let's walk through a simple unit test of a Stack written in F# and unit testing with the xUnit.net framework.

```
#light

#R @"D:\Tools\xunit-build-1252\xunit.dll"

open System
open System.Collections.Generic
open Xunit

type Stack<'t> = class
  val elements : LinkedList<'t>

  new() = { elements = new LinkedList<'t>() }

  member x.IsEmpty
    with get() = x.elements.Count = 0

  member x.Push element =
    x.elements.AddFirst(element:'t)

  member x.Top
    with get() =
      if x.elements.Count = 0 then
        raise (InvalidOperationException("cannot top an empty stack"))
      x.elements.First.Value

  member x.Pop() =
    let top = x.Top
    x.elements.RemoveFirst()
    top
end

[<Fact>]
let PopEmpty () =
  let stack = new Stack<string>()
  Assert.Throws<InvalidOperationException>(fun () -> stack.Pop() |> ignore )
```

The real interesting part you should pay attention to is the last line. As you can see, I am using the forward operator to indicate that I really don't care what the function returns, just that I'm interested in that it executes. This is most likely during such functions that have some sort of side effect to them. I could also use the ignore function instead of the forward operator such as this:

```
[<Fact>]
let PopEmpty () =
  let stack = new Stack<string>()
  Assert.Throws<InvalidOperationException>(fun () -> ignore(stack.Pop()) )
```

This is very helpful in these cases where we really don't care about the return value, instead want to mutate the state of our given object, such as removing a value from a collection and so on.

Mutables

As I said in many posts before, by default all "variables" by which I mean values in F# are immutable. This is a standard in functional programming and all in the ML family. You can easily redefine a value by using the let keyword, but not actually mutate its state. But, since F# is a multi-purpose language on the .NET platform, mutable state can be had. To take advantage of this, mark your value as mutable. Then to change the value, just use the <- operator to reassign the value. Below is a simple example of this:

```
#light

let mutable answer = 42
printfn "Answer is %i" answer
answer <- 8
printfn "Answer is %i" answer
```

A key difference from the reassignment is that you cannot change the value type. Whereas I can redefine answer by keep using the let keyword, I can only keep my answer in this above example of the int type.

This can also apply to record types as well where you can change the fields. In the last installment, we talked about record types. Well, by default there as well, the fields for the record type are immutable. But, as with before, that can be changed. I of course like to caution people that mutable state takes a lot of the value proposition away from the side effect free programming that you gain with F# by default. But, nevertheless, you can still do it as noted below:

```
#light

type Person = { FirstName : string; mutable LastName : string; mutable IsMarried : bool }

let friend = { FirstName = "Courtney"; LastName = "Cox"; IsMarried = false }
friend.LastName <- "Cox-Arquette"
friend.IsMarried <- true
```

What I was able to do was define a Person record and change a couple of fields while using the <- operator and defining the fields as mutable. Yes, I could have used some scientific calculation or something, but this was easy.

Reference Cells

The last thing I want to touch on in this post is reference cells. You can think of these much as pointers in other languages or reference types. These of course can be of any type. The idea behind using these is to make updating fields as easy as possible. As with mutable fields, you cannot change the type once it has been assigned. To use these, you need to remember three basic operators

- **ref** - Allocates a reference cell
- **:=** - Mutates a reference cell
- **!** - Reads the reference cell

Below is a quick example of mutation through reference cells:

```
#light

let x = ref 2
x := !x + 25
printfn "%i" !x
```

What the code example above lets me do is define a reference to the number 2. Then I can change that reference by reading the current x value and adding 25. Then I mutate the existing x value with the result.

Conclusion

This is just meant to be a brief overview to some imperative programming constructs that you might see in .NET, Java, C, C++ and so on. F# is a first class language all the way with constructs that support these things as well as your normal functional programming constructs. I hope we get to cover some of this at ALT.NET Open Spaces, Seattle at some point because I'm sure a lot of people will be interested. Until next time...

Part 9 - Control Flow

Taking a break from the Design by Contract stuff for just a bit while I step back into the F# and functional programming world. If you followed me at my old blog, you'll know I'm pretty passionate about functional programming and looking for new ways to solve problems and express data.

Where We Are

Before we begin today, let's catch up to where we are today:

- Part 1 - Basic functional programming
- Part 2 - Currying and Tuples
- Part 3 - Scope, Recursion and Anonymous Functions
- Part 4 - History of F#, Operators and Lists
- Part 5 - Pattern Matching
- Part 6 - Lazy Evaluation
- Part 7 - Creating Types
- Part 8 - Mutables and Reference Cells

Today's topic will be covering more imperative code dealing with control flow. But first, the requisite side material before I begin today's topic.

A Survey of .NET Languages And Paradigms

Joel Pobar just contributed an article to the latest MSDN Magazine (May 2008) called "Alphabet Soup: A Survey of .NET Languages And Paradigms". This article introduces not only the different languages that are supported in the .NET space, but the actual paradigms that they operate in. For example, you have C#, VB.NET, C++, F# and others in the static languages space and IronRuby, IronPython among others in the dynamic space. But what's more interesting is the way that each one tackles a particular problem. The article covers a little bit about functional programming and its uses as well as dynamic languages. Of course the mention is made that C# and VB.NET are slowly adopting more functional programming aspects over time. One thing I've lamented is the fact that VB.NET and C# are too similar for my tastes so I'm hoping for more true differentiation come the next spin. Instead, VB would be really interesting as a more dynamic language and not just one that many people just look down their noses at. Ok, enough of the sidetracking and let's get back to the subject at hand.

Control Flow

Since F# is a general purpose language in the .NET space, it supports all imperative ways of approaching problems. This of course includes control flow. F# takes a different approach than most functional programming languages in that the evaluation of a statement can happen in any order. Instead, in F#, we have a very succinct way of doing it in F# with the if, elif, else statements. Below is a quick example of that:

```
#light

let IsInBounds (x:int) (y:int) =
  if x < 0 then false
  elif x > 50 then false
  elif y < 0 then false
  elif y > 50 then false
  else true
```

What I was able to do is to check the bounds of the given integer inputs. Pretty simple example. As opposed to many imperative languages, when you are returning a value from the if, all subsequent elif or elses must also return values. This makes for balanced equations. Also, if you return a value from an if, then you are also forced to have an else which returns a value.

Although F# is using type inference to determine what my IsInBounds method returns, I cannot go ahead and return one type in an if and another different type in the elif or else. F# will complain violently, as it should because that's really not a good design of a function. Below is some code that will definitely throw an error.

```
#light

let IsInBounds (x:int) (y:int) =
  if x < 0 then "Foo"
  elif x > 50 then false
  elif y < 0 then false
  elif y > 50 then false
  else true
```

As I said before, the equations must be balanced. But of course if your if expression returns a unit (void type for those imperative folks), then you aren't forced to have and else statement. Pretty self explanatory there.

Let's move onto the for loops. The standard for loop is to start at a particular index value, check for the terminate condition and then increment or decrement the index. F# supports this of course in a pretty standard way, but by default, the index is incremented by 1. You must note though that the body of the for loop is a unit type (void once again) so, if you return a value, F# won't like it. Below is a simple for loop to iterate through all lowercase letters.

```
#light

let chars = [|'a'..'z'|]

let PrintChars (c:array<char>) =
  for index = 0 to chars.Length - 1 do
    print_any c.[index]

PrintChars chars
```

But, if I tried to return c from the for loop, F# will complain, but it will allow it to happen. It's just a friendly reminder that it's not going to do anything with that value you specified. I could also specify the for loop with a decrementer, so let's reverse our letters this time.

```
#light

let chars = [|'a'..'z'|]

let PrintChars (c:array<char>) =
  for index = chars.Length - 1 downto 0 do
    print_any c.[index]

PrintChars chars
```

F# also supports the while construct as well. This of course is the exact same as any imperative construct, but with the caveat of once again, the while loop should not return a value because it is of the unit type.

```
#light

let chars = ref ['a'..'z']

while (List.nonempty !chars) do
  print_any (List.hd !chars)
  chars := List.tl !chars
```

This time we're just printing out a char and then removing it from the list collection. Note that we're using the ref keyword and reference cells as we talked about before. Lastly, let's cover one last construct, the foreach statement. This is much like we have in most other languages, just the wording is a bit different. As always, the foreach statement has the unit type, so returning values is a warning.

```
#light

let nums = [0..99]

for n in nums do
  print_any n
```

Wrapping It Up

Just a quick walkthrough of just some of the imperative control statements allowed by F#. As you can see, it's not a huge leap here from one language to the next. I have a couple of upcoming talks on F#, so if you're in the Northern VA area on May 17th, come check it out at the NoVA Code Camp.