

F# Tutorial—An Introduction to Functional Language in .NET

Daniel Kruze

November 11, 2022

The mid-2000s were something of a wasteland for high-level computer languages. Major companies rising and dissolving and tech innovations de jure seemed to demand from developers at the time that a new language be made every month; with a lack of regulating bodies, these languages all competed fiercely. Every app, website, embedded system, and database, among other things, was written using a different language from a different company and maintained by a different team. What could developers do to keep all their projects and interests organized? Suffer? For a while, yes, but after some time an answer seemed to have been reached.

Microsoft, the leading tech authority at the time and one that remains incredibly important today, had already innovated the interoperability field with their **C#** language, that was easy for people of many backgrounds to learn, highly portable, and chock-full of features for API integration. After its release, though, it was found to be lacking in some ways, and was in fact *very* limiting due to its insistence on being object-oriented making it uncooperative with some other procedural languages and older apps. The solution for *this* problem came in 2005, with the **F#** language—a functional extension of the C# concept that served any paradigm, any place, any time.

Developed just prior to 2005 by Don Syme and the Microsoft Research Team, F# was created to tie up the loose ends left by Microsoft's previous attempts to tie up loose ends, like C#, .NET, and Visual Studio. It was made with a hefty but incredibly strong compiler and it operates within .NET and Visual Studio just like C#, but the apps it exports can do a *lot* more than C# ever could. In fact, it encompasses *all* the features of C# in terms of object-orientation and is strong enough to be interoperable with existing C# apps. This sets it apart from other functional languages like Haskell or LISP



Don Syme, BDFL of F# and the originator of the language's design

in that it can be purely object-oriented if need be, or the object-orientation can be fluid with intense mathematical expressions and multithreading.

It also includes a lot of features that one may expect from conventional, imperative languages, just to make things easier for newcomers and to make logic a little less contrived. These include while and for loops, arrays, hash tables, and primitive data types that incorporate strings, among other things.



The current logo of F#, used by its maintenance team

Of course, being a functional language, F# is designed to be based around *expressions* rather than declarations and executions, making it read more like pure math than like code. There is far less emphasis on type declaration, even though typing is static and very strong, but types are also inferred by the compiler, making it optional to declare types. Typing and variables are secondary to *functions*, which understandably make up the bulk of the code and the declarations.

You can think of F# or any functional language as a language that treats

functions as data, where the functions themselves don't actually affect the runtime state. This is very different from an imperative language like, say, **C**, which uses functions as executable code during runtime to manipulate existing, static data, rather than by using function data to map static data.

It's confusing at first, but there are some similarities to other paradigms as well that may help you wrap your head around it. These can make F# more applicable to command-line environments or environments with UIs, which are typically where functional languages are considered overcomplicated. For example, in F#, like in some scripting languages, there's no need to specify an entry point, with code simply executing top to bottom if one isn't specified. Additionally, there is support for anonymous functions, asynchronous programming, and API integration that might make

F# a familiar tool for scripting language users and a useful tool in an environment where it wouldn't ordinarily be considered.

All of this is very high-level though, and it might not be clear without seeing examples. If someone simply described to you the difference between a hammer and a drill, it still might not make much sense why either one is better than the other for any given task without having looked at or used them. Let's look at some examples to see how some of these features are actually relevant or usable in any way.

Beginner Lesson: Typing and Functions in F#

Being that data takes the form of expressions in F#, and functions are the primary vehicles of expressions, declarations for functions and, indeed, variables have some important properties that might not be intuitive.

1. Declarations are constant by default and considered static by the compiler
2. Variables must be declared with the **mutable** keyword
3. Polymorphism is not allowed for functions, because declarations are static
4. Declaration types can be either explicit or implicit
5. **All** declarations use the **let** keyword

For example, a function can be defined as follows:

```
let [name] [parameters] : [return type] = [body]
```

Where the return type specification is optional and can be implicitly determined by the compiler. A function to calculate a hypotenuse could then be:

```
let hypotenuse a b : float = let c = (a*a) + (b*b)  
                               sqrt c
```

To invoke this function, you will typically declare a new constant and set it equal to the name of the function, followed by the parameters to pass, delineated by whitespace (Ruby uses this approach as well:)

```
let pythagoras = hypotenuse 2.0 3.0  
printfn "Hypotenuse of triangle with sides 2 and 3: %g" pythagoras
```

Where the second line simply prints the result to the command line (that result being ~3.6) You may notice that the constant **c** in the above function is not typed during

declaration, but the function has a return type specified—this is a common approach in F#. Also, note that the integrated **sqrt** function only works on floats.

This might seem make it seem like some foresight is necessary to create simple declarations and functions in F#, but its more intuitive (and lazy, in some ways, you'll get that joke later) than it may appear. This might be more evident when looking at **classes** in F#, the basis for objects, which are incredibly useful for C# interoperability as well as large-scale, multi-program applications (F# is commonly used in GPU drivers, as a matter of fact.) This is because **classes** can be written in a large variety of ways, allowing quite a lot of lenience.

Classes are actually *not* defined with the **let** keyword, rather with the **type** keyword, making them similar in a way to **templates** in C++, a sort of class. Properties and methods can both be defined at the start of a class with **let** keywords, making them private and in need of getters and setters. Below the **let** statements are the **do** statements, which specify code to run when the class is instantiated—these are **not** constructors. Below those **do** statements are **members**, which are public features of a class and can be properties or methods. A simple example is below:

```
type ExampleClass(_x : int, _y : int) =
    let mutable x = _x
    let mutable y = _y
    do printfn "Class initialized with (%d, %d)" x y
    member this.X with get() = x and set(value) = x <- value
    member this.Y with get() = y and set(value) = y <- value
    new() = ExampleClass(0, 0)
```

Where this class can be instantiated with:

```
let ExampleObject = new ExampleClass(2, 3)
```

While this may seem like daunting syntax, it's actually very compact, making getting and setting with integrated methods incredibly easy and removing the need for **private** and **public** keywords. For using methods that you have defined, dot notation is applied as you expect.

However, one thing may seem confusing: what is the *actual constructor* here? This can be confusing because the definition of the **class** specifies parameters (with

types,) yet a constructor seems to be defined in the **members** section and called during instantiation. This is because the members section is also used for alternate constructors, similarly to how C++ and Java can have **classes** with multiple constructors depending on the situation. When the class is instantiated here, the constructor in the **members** section (method **new()**) is called, that in turn creates an object of the given **class** whose properties are given by the parameters defined in the definition of the class. This is called a **primary constructor** and is the simplest and most common way to define a constructor in F#. In fact, while the **new()** method is good practice, it isn't necessary, and the **class** could be instantiated as:

```
let ExampleObject = ExampleClass(2, 3)
```

Where the **new()** method simply acts as a default for when a **class** is instantiated without given parameters.

For methods that *aren't* constructors, note that unless you are specifically making a **static** method (which is a little beyond the current scope,) you should define methods in the format:

```
member this.[name]() = [body]
```

Where properties accessed within the method are specified by the name of the getter and setter.

ASSIGNMENT: To make this less confusing, try writing a class of your own—it is the best way to practice, after all. Try making a class for a rectangle that uses primary construction like the above example, and may or may not specify an alternate constructor in its **members** section. It should have a length and a width that are private, and methods for both the perimeter calculation and the area calculation. Then, instantiate the class and make sure all four of these features are executed and printed to the command line. Keep in mind that while the example uses tabs for readability, indentation in F# is done using *spaces* and not tabs. An example is provided below for a square:

```
Program.fs X
D: > F#Builds > MyFSharpApp > Program.fs > {} Program > squareClass > new
1 type squareClass(x : int, y : int) = //define a class with two properties, set during construction
2   let mutable length = _x //length is a private property set by parameter 1 in the constructor
3   let mutable width = _y //length is a private property set by parameter 1 in the constructor
4   do printfn "Class initialized with length %d and width %d" length width //print the length and width at instantiation
5   member this.Length with get() = length and set(value) = length <- value //getter/setter for length
6   member this.Width with get() = width and set(value) = width <- value //getter/setter for width
7   member this.perimeter() = //define a method of a class instance
8     let result = (this.Length * 4) //let a variable hold the integer 4 * length
9     printfn "Perimeter of square is %d" result //print 4 * length
10  member this.area() = //define a method of a class instance
11    let arestult = this.Length * this.Width //let a variable hold the integer width * length
12    printfn "Area of square is %d" arestult //print width * length
13  new() = squareClass(0, 0) //new() method serves as alternate constructor for default values
14
15 let blockHead = new squareClass(2, 2) //instantiate the class as an object using the new() method, with length and width both 2
16 blockHead.perimeter() //call the perimeter method for this instance
17 blockHead.area() //call the area method for this instance
18
```

Below is a screenshot of the expected output:

```
Program.fs X
D: > F#Builds > MyFSharpApp > Program.fs > {} Program > squareClass > new
1 type squareClass(x : int, y : int) = //define a class with two properties, set during construction
2   let mutable length = _x //length is a private property set by parameter 1 in the constructor
3   let mutable width = _y //length is a private property set by parameter 1 in the constructor
4   do printfn "Class initialized with length %d and width %d" length width //print the length and width at instantiation
5   member this.Length with get() = length and set(value) = length <- value //getter/setter for length
6   member this.Width with get() = width and set(value) = width <- value //getter/setter for width
7   member this.perimeter() = //define a method of a class instance
8     let result = (this.Length * 4) //let a variable hold the integer 4 * length
9     printfn "Perimeter of square is %d" result //print 4 * length
10  member this.area() = //define a method of a class instance
11    let arestult = this.Length * this.Width //let a variable hold the integer width * length
12    printfn "Area of square is %d" arestult //print width * length
13  new() = squareClass(0, 0) //new() method serves as alternate constructor for default values
14
15 let blockHead = new squareClass(2, 2) //instantiate the class as an object using the new() method, with length and width both 2
16 blockHead.perimeter() //call the perimeter method for this instance
17 blockHead.area() //call the area method for this instance
18
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS D:\f#builds\myfsharpapp>
PS D:\f#builds\myfsharpapp>
PS D:\f#builds\myfsharpapp>
PS D:\f#builds\myfsharpapp> dotnet run
Class initialized with length 2 and width 2
Perimeter of square is 8
Area of square is 4
PS D:\f#builds\myfsharpapp>
PS D:\f#builds\myfsharpapp>
PS D:\f#builds\myfsharpapp>
```

Once you have made this class, you should have a good handle on how classes, simple functions, and general declarations work. F# doesn't treat data the way most imperative languages would, so before you can go any further, **it's *crucial* that you understand how the F# compiler evaluates and types declarations statically.**

Note that this tutorial assumes you have an existing RTE for F# on your machine, or on a virtual machine. There are no given instructions on how to set the language up in this guide.

Intermediate Lesson: Composition and Pipelining

Of course, things can certainly get more complicated than that. As a matter of fact, functions can be used in many more ways than just as expressions for data manipulation—that's right, they can behave *exactly like* mathematical functions, in more ways than one. Let's see some ways we can condense difficult to read code into something more manageable.

First though, a quick aside that may be beneficial later: conditional logic and looping logic function in F# much the same as they do in Ruby, as alluded to earlier. Should you need an **if/then** block, they follow this syntax:

```
if [expr]
  then [body]
elif [expr]
  then [body]
else
  [body]
```

Where expressions are Boolean. **Loops** can follow this syntax:

```
for [iterator] to [bound] do
  [body]
for [element] in [collection/sequence] do
  [body]
While [Boolean condition] do
  [body]
```

All of these constructs can be very helpful in certain expressions, but will definitely be less common than you may be used to. If you need to evaluate a piecewise function, calculate a factorial, or process a small set of data, though, these will be helpful notes to remember.

Now then, some work with functions. **Composition** is a concept you're likely already familiar with; it's when a function takes another function's result as its argument. The same concept can be done very simply in F# using a composition operator, but with a few syntactical details that are worth noting:

```
let [name] = [existing function] [parameters] >> [existing function]
```

Of course, this can be done for more than 2 functions, but things simply get more and more complicated as factors increase. For now, let's consider an example where we have only two string functions to compose:

```
let function1 x = "Hi, my name is " + x
```

```
let function2 y = y + " Cornelius"
```

We can use composition to make a third function that concatenates a full name for us, and then bind a constant declaration to that function's invocation:

```
let function3 = function2 >> function1
```

```
let name = function3 "Yukon"
```

```
printfn "%s" name
```

Which will yield the full name "Yukon Cornelius." Obviously this is a trivial, silly example, but composition can be very powerful for factorizing otherwise complicated expressions, making them much easier to read and debug. Note that in a composed function, you specify the nested function first and provide a parameter for the nested-most function, only when calling the composite function. **Execution order of a composite function determines the order in which nested functions should be written.**

In addition to **composition**, there is also a concept for functions called **pipelining**, which performs a similar operation in a more concise fashion. The operator is different and there's no need for a third function, but the order in which nested functions are specified remains the same. You will specify a parameter, and then pass it to a nested function, and then pass that function to another function ad infinitum:

```
let [constant] = [parameter] |> [function] |> [function] ...
```

Using the above example functions, we can print out our full name using **pipelining**:

```
Let name = "Yukon" |> function2 |> function1
```

printfn “%s” name

Which will also yield “Yukon Cornelius.” This is clearly simpler than composition being that it requires fewer declarations, but it can be a nuisance to have to specify so many functions by name on one line.

Where is better to use one over the other? The differences can be minute, but they aren’t actually doing the same thing—remember this. **Composition** uses its operator to return a *function* from two functions, making it illogical to pass a composite function a parameter. **Pipelining**, however, uses its operator to return a *value* from a value and a function. Use the former when you explicitly want a function that can be called, but use the other when you explicitly want a result from a series of functions. In analogical terms, think of **composition** as shorthand for *nested* functions, and **pipelining** as shorthand for *chained* functions. If that analogy doesn’t help, I recommend you study Javascript at your earliest convenience.

However, seeing these operators for functions brings to mind another feature of functions that may be more helpful and efficient than either previous option: **recursion**. **Recursion**, scientifically, is the concept of reducing a set of data or expressions into a more manageable, processable size. This most commonly manifests in the idea of “a function that calls itself,” which is fundamentally a way of condensing a set of operations into one operation that repeats—**recursion!**

This exists in F# and can be performed just as you may expect, by making a function that includes a call to itself in its body alongside a condition wherein successive calls are stopped. However, the **rec** keyword is also necessary in the declaration:

let rec [name] [parameters] : [return type] = [recursive body]

Where the return type is, as expected, an optional specification. Note that the body of any function declared recursively should be recursive, or else the compiler will start turning yellow on you.

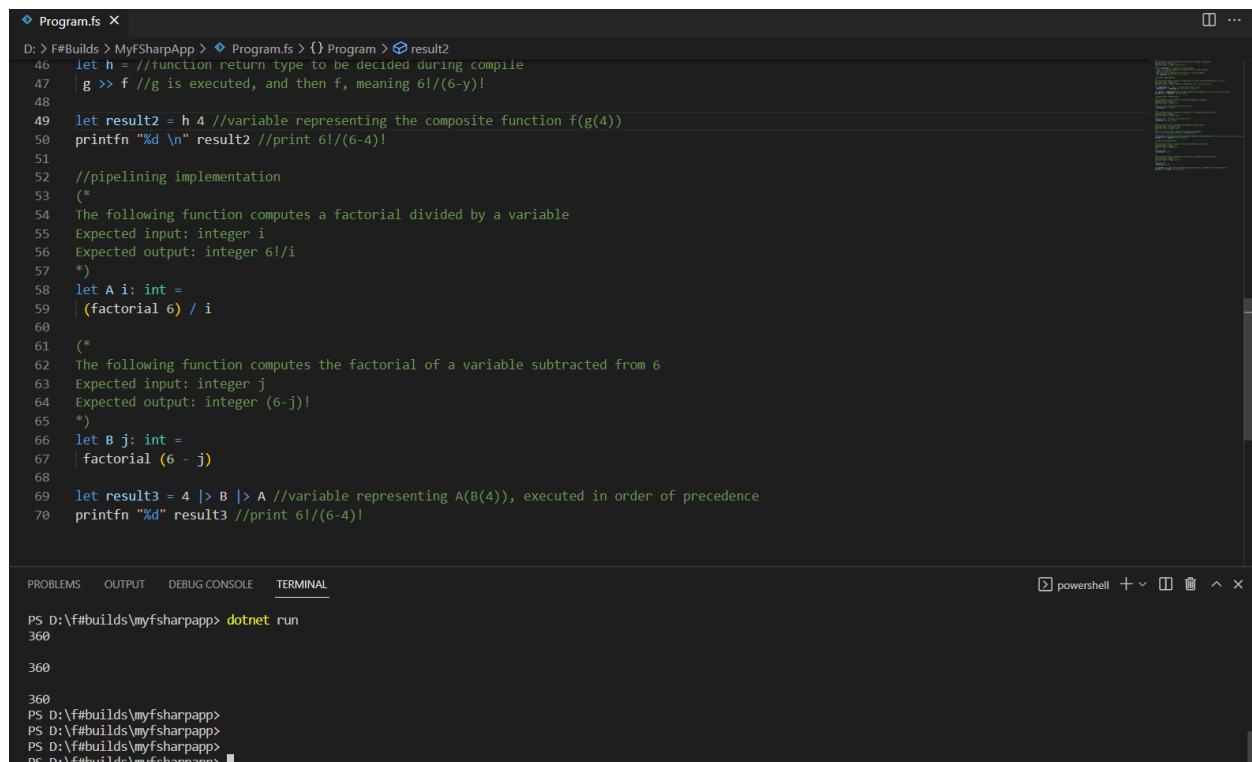
ASSIGNMENT: To practice with these concepts (all of which are essential to processing in F#,) it would be a good idea to tackle a complicated expression that *needs* to be broken down and factorized. Consider the formula for a permutation:

$$nPr = n!/(n-r)!$$

Where n is the amount of members in a set, and r is the amount of elements contained in a single permutation of that set. Try to calculate the permutations for 4 elements in a set of 6 in three different ways: **brute-force**, **composite**, and **pipelined**, all of which should print out the same result (which is 360.) Additionally, since there is no integrated function for a factorial calculation, make a **recursive** function that can calculate the factorial of an int. Possible solutions are exemplified below:

```
D:\> F#\Builds > MyFSharpApp > Program.fs > {} Program > result2
1  (*
2  The following function computes the factorial of a number, recursively
3  Expected input: integer e
4  Expected output: integer factorial of e
5  *)
6  let rec factorial e = //define recursive function
7  if e < 1 //if we are computing a factorial for 0 or fewer elements
8  then 1 //factorial is 1
9  else //if we are computing a factorial for 1 or more elements
10 | e * factorial (e - 1) //e! = e * (e-1)!
11
12 //standard implementation
13 (*
14 The following function computes a permutation on a set, using the formula  $nCr = n!/(n-r)!$ 
15 Expected input: integers r, n
16 Expected output: integer number of permutations size r on set of size n
17 *)
18 let permutation n r : int = //function will return an int
19 | (factorial n) / (factorial (n - r)) //nCr = n!/(n-r)!
20
21 let result1 = 6 4 //variable representing permutations of size 4 on a set of size 6
22 printfn "%d \n" result1 //print 6!/(6-4)!
23
24 //compositional implementation
25 (*
26 The following function computes a factorial divided by a variable
27 Expected input: integer x
28 Expected output: integer 6!/x
29 *)
30 let f x : int = //function will return an int
31 | (factorial 6) / x //6!/x
32
33 (*
34 The following function computes the factorial of a variable subtracted from 6
35 Expected input: integer y
36 Expected output: integer (6-y)!
37 *)
38 let g y : int = //function will return an int
39 | factorial (6 - y) //(6-y)!
40
41 (*
42 The following function represents the composite function f(g())
43 Expected input: functions f and g
44 Expected output: integer f(g())
45 *)
46 let h = //function return type to be decided during compile
47 | g >> f //g is executed, and then f, meaning 6!/(6-y)!
48
49 let result2 = h 4 //variable representing the composite function f(g(4))
50 printfn "%d \n" result2 //print 6!/(6-4)!
51
52 //pipelining implementation
53 (*
54 The following function computes a factorial divided by a variable
55 Expected input: integer i
56 Expected output: integer 6!/i
57 *)
58 let A i : int =
59 | (factorial 6) / i
60
61 (*
62 The following function computes the factorial of a variable subtracted from 6
63 Expected input: integer j
64 Expected output: integer (6-j)!
65 *)
66 let B j : int =
67 | factorial (6 - j)
68
69 let result3 = 4 |> B |> A //variable representing A(B(4)), executed in order of precedence
70 printfn "%d" result3 //print 6!/(6-4)!
```

Below is a screenshot of the expected output:



```
Program.fs x
D: > F#Builds > MyFSharpApp > Program.fs > { } Program > result2
46 let h = //function return type to be decided during compile
47 |> f //g is executed, and then f, meaning 6!/(6-y)!
48
49 let result2 = h 4 //variable representing the composite function f(g(4))
50 printfn "%d \n" result2 //print 6!/(6-4)!
51
52 //pipelining implementation
53 (*
54 The following function computes a factorial divided by a variable
55 Expected input: integer i
56 Expected output: integer 6!/i
57 *)
58 let A i: int =
59 |> (factorial 6) / i
60
61 (*
62 The following function computes the factorial of a variable subtracted from 6
63 Expected input: integer j
64 Expected output: integer (6-j)!
65 *)
66 let B j: int =
67 |> factorial (6 - j)
68
69 let result3 = 4 |> B |> A //variable representing A(B(4)), executed in order of precedence
70 printfn "%d" result3 //print 6!/(6-4)!
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS D:\f#builds\myfsharpapp> dotnet run

360

360

360

PS D:\f#builds\myfsharpapp>

PS D:\f#builds\myfsharpapp>

PS D:\f#builds\myfsharpapp>

PS D:\f#builds\myfsharpapp>

With this under your belt, you should be able to see how functions are both data in the form of expressions *and* manipulable, factorizable data *structures*. **Functions in F# are constructs that can be broken down into constituent pieces, and doing so often provides advantages for processing and debugging.**

Advanced Lesson 1: Delegates and Generics

Everything in the previous two lessons, however, has been pretty mathematical. Not everything you do in F# will be explicit processing of data that you have personally defined, and sometimes it won't even be *your code*. Interoperability is a big part of F#, after all, and there are a few important topics to get a handle on if you'll ever be using F# in the field.

The most specific of those would be **delegation**, a topic almost entirely endemic to the .NET environment being that only C# and F# ever encourage programmers to employ it. **Delegation** is when a separate type is declared that specifies an interaction

between integral types, and can create objects that stand in for existing functions or methods. It is a very roundabout process involving quite a few keywords, and it's used by interop programmers to specify functions for APIs that expect .NET apps to provide them, typically because the API doesn't have access to local or private features of a given program.

It isn't something that would come up in most contexts, but the context that F# exists in demands its existence. The steps and syntax involved begin with a **type** definition, as follows:

type [name] = delegate of [type/type expression] -> [type]

Where the first type is the type of a parameter for a function, and the second type is that function's return type. Obviously, delegates are intended for functions, or even groups of functions that all take the same parameters and return the same type. So, if you want to access a delegate, you need to first make an object of that **type**, just like you would for a **class** (being that they are both types.) It follows this syntax:

let [name] = [delegate type]([function])

You can think of this like the delegate is a **class**, and you are instantiating it with a name for an instance that is passed a given function as a parameter. Just make sure that the function you pass has the same parameter and return type as the delegate type! Once this object has been instantiated, you will need to create a separate method that can actually invoke it; you see, **delegates** actually *aren't* like classes because they aren't entirely user defined. The **delegate** keyword we used in the type definition actually references a .NET library object, and that object has its own specific method for actually executing a function specified by a delegate. This "middleman" function that you need to actually *use* your delegate follows this syntax:

**let [name] (dlg : [delegate type]) ([parameter] : [parameter type]) =
 dlg.Invoke([parameter])**

Where the parameter type is the same type that would be passed to the function specified by your delegate object.

With all of these ducks in a row, you can finally use your **delegate** to perform a proxy operation, although it will likely be an API doing it, not you. To manually invoke a **delegate**, use this instruction:

let [name] = [middleman function] [delegate object] [function]

Where the name you declare will be a variable holding the output of the function you have been **delegating**. Finally!

You can also **delegate** methods from classes if you want, which may actually be a more common implementation of the concept, being that **delegation** is generally reserved for protected data. To do this, you have two methods: the first is to follow the above steps in the same fashion, except to use dot notation in your middleman function to specify that the function you are **delegating** belongs to an instance of a **class**. However, you can also access methods *without* an instance by changing the format of the middleman function. It would become the following:

Let [name] : [delegate type] = new [delegate type]([class].[method])

And from there, the final step for invocation is the same.

More generally, though, there is another concept that is integral to functional languages beyond just F# alone, and it is incredibly powerful for interoperability much in the same way as **delegation**, if more universal because it applies more to *human* users. This would be **generics**, which are types of declarations that use non-specific data types, so that the execution depends on the type of the argument passed. Imagine you have made a function that adds two things together, but you have no way of knowing if a user is going to call that function by passing it an integer, a float, or a string. You *could* work some magic with exception handling to parse user input to your advantage, or you could force the user to input the type you want—both of those are verbose and crude ways of writing code, and are frankly *beneath* a compiler as powerful as F#'s.

You should instead write a **generic** in one of many ways, the most common of which is to use apostrophe operators in function definitions:

let [name]<'a> [parameters] = [body]

Where you patently *do not* want a return type, because that defeats the all-purpose nature of a **generic**. So much so, in fact, that the compiler will prevent you from doing so! In order to invoke this function, you need to specify in your call to it what kind of data type you are passing to it:

let [name]<[type]> = [function] [parameters]

Where parameters are of a uniform type. Of course, if you are used to the Java or C++ approach to **generics**, you may use '**T**' instead of '**a**' and it ought to work.

This can go for any kind of definition so long as you specify with the apostrophe operator that you are expecting a type to be assigned during runtime. As such, types can *also* be made generic, which allows for some intense manipulation of certain structures. Take, for example, a **generic class**:

```
type [name]<'a> ([parameters] : 'a) =  
[body of class]
```

Where the body of the class is business as usual. However, alternate constructors can cause *serious* trouble for **generic classes**, and it's recommended they be avoided. Instead, when you instantiate your **class**, simply specify the type of the parameters you intend to pass your **class**, just like you would a generic function. This does mean that, typically, you will have to pass parameters of all the same type to your **class**—this can be limiting. Often times, though, you won't need to make a generic class, and it would be more important for the methods within to be generic. For instance **members** of a **class**, you can define a **generic member** as:

```
member this.[name]<'a>([parameters])
```

Where again, a return type should not be specified. Be warned though, because for any **member** or, really, *any* function defined **generically**, avoid using mathematical operations on integral types with parameters of **generic** type. For example, don't make a function to double a number that specifies a parameter of type '**a**' multiplied by an int 2. This will not be possible for the compiler to typecast, even if you pass the **generic** an integer.

You may have noticed, also, that since types can be **generic** that **delegates** can also be **generic**. This is true, albeit an incredibly uncommon occurrence, and one that requires some...manipulation. For the sake of comprehension, the syntax is as follows:

```
let [name]<'a> = delegate of <['a expression]> -> <'a>
```

Note that when you create an object for invocation, you need to make sure that your **delegate** is specifying a **generic** function, and that you maintain all type definitions as '**a**'.

Hopefully, you never have to do that, because it can be a nightmare to troubleshoot. Instead, let's quickly see an example of a **generic** that could come up anywhere:

```
let cube<'a> x = x * x * x
printfn "%A" (cube<float> 3.0)
printfn "%A" (cube<int> 3)
```

Note that the results of functions can be printed using aliases just like constants or variables, as long as the call to the function is in parentheses. This function can accept a float or an int, and the result it prints will depend on the parameters passed to it...right? Actually, this will print out "27.0" twice in a row, despite the fact that the second call to it passes it an integer. Why is this? Because **generics** are constant, and once you pass them a specific type for the first time, they come to expect that type in all subsequent calls.

This is because they are **lazy evaluated**, meaning that the compiler doesn't necessarily evaluate the type of the function every time it executes, it simply evaluates it once and then assumes the type of future calls in order to save CPU space. As you can imagine, this can cause a *lot* of issues, and in the next (and final) lesson, we will examine **lazy expressions**, among other things.

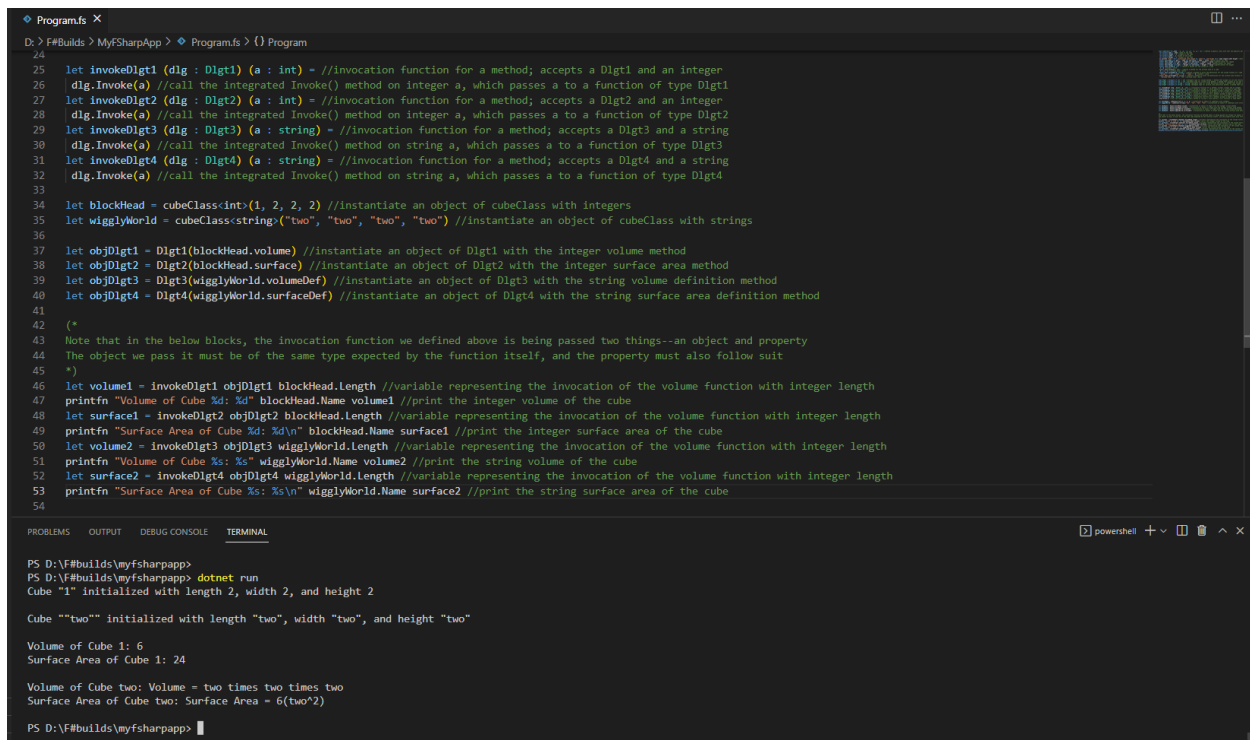
ASSIGNMENT: Before that point, though, you should practice these dense concepts you just looked at. Take the assignment you made before for lesson 1—the rectangle—and make it into a rectangular prism. This means it will have a height in addition to its width and length, and instead of a perimeter and area it should have a volume and a surface area. Make this class a **generic**, and call the methods for volume and surface area using **delegates**, either as static members or instance members. To test your class, instantiate it with two different data types, making sure there is a way to tell them apart when you print their dimensions and methods. Below is an example of a cube, which is a kind of rectangular prism, that can accept ints and floats, as well as additional methods to accept strings. You don't need to make extra methods to account for data type discrepancies like that, but the below example does, just to illustrate the utility of **generics**:


```

Program.fs x
D:\> F#Builds > MyFSharpApp > Program.fs { } Program
1 type cubeClass<'a>('name : 'a, _x : 'a, _y : 'a, _z : 'a) = //define a generic class with four non-specific parameters
2   let mutable name = _name //name of the cube
3   let mutable length = _x //length of the cube
4   let mutable width = _y //width of the cube
5   let mutable height = _z //height of the cube
6   do printfn "Cube \"%A\" initialized with length %A, width %A, and height %A\n" name length width height //when class is instantiated, print its properties
7   member this.Name with get() = name and set(value) = name <- value //getter/setter for name
8   member this.Length with get() = length and set(value) = length <- value //getter/setter for length
9   member this.Width with get() = width and set(value) = width <- value //getter/setter for width
10  member this.Height with get() = height and set(value) = height <- value //getter/setter for height
11  member this.volume(b: int) = //define a method for the volume of a cube
12    3 * b //volume = l*w*h
13  member this.surface(c: int) = //define a method for the surface area of a cube
14    6 * (c * c) //surface area = 6*w^2
15  member this.volumeDef(d: string) = //define a method to print the definition for the volume formula of a cube
16    "Volume = " + d + " times " + d + " times " + d //volume = l*w*h
17  member this.surfaceDef(e: string) = //define a method to print the definition for the surface area formula of a cube
18    "Surface Area = 6(" + e + " ^2)" //surface area = 6*w^2
19
20 type Digt1 = delegate of int -> int //delegate type for volume method, a method that accepts an int and returns an int
21 type Digt2 = delegate of int -> int //delegate type for surface area method, a method that accepts an int and returns an int
22 type Digt3 = delegate of string -> string //delegate type for volume definition method, a method that accepts a string and returns a string
23 type Digt4 = delegate of string -> string //delegate type for surface area definition method, a method that accepts a string and returns a string
24
25 let invokeDigt1 (dlg : Digt1) (a : int) = //invocation function for a method; accepts a Digt1 and an Integer
26   dlg.Invoke(a) //call the integrated Invoke() method on Integer a, which passes a to a function of type Digt1
27 let invokeDigt2 (dlg : Digt2) (a : int) = //invocation function for a method; accepts a Digt2 and an Integer
28   dlg.Invoke(a) //call the integrated Invoke() method on Integer a, which passes a to a function of type Digt2
29 let invokeDigt3 (dlg : Digt3) (a : string) = //invocation function for a method; accepts a Digt3 and a string
30   dlg.Invoke(a) //call the integrated Invoke() method on string a, which passes a to a function of type Digt3
31 let invokeDigt4 (dlg : Digt4) (a : string) = //invocation function for a method; accepts a Digt4 and a string
32   dlg.Invoke(a) //call the integrated Invoke() method on string a, which passes a to a function of type Digt4
33
34 let blockHead = cubeClass<int>(1, 2, 2, 2) //instantiate an object of cubeClass with integers
35 let wigglyWorld = cubeClass<string>("two", "two", "two", "two") //instantiate an object of cubeClass with strings
36
37 let objDigt1 = Digt1(blockHead.volume) //instantiate an object of Digt1 with the integer volume method
38 let objDigt2 = Digt2(blockHead.surface) //instantiate an object of Digt2 with the integer surface area method
39 let objDigt3 = Digt3(wigglyWorld.volumeDef) //instantiate an object of Digt3 with the string volume definition method
40 let objDigt4 = Digt4(wigglyWorld.surfaceDef) //instantiate an object of Digt4 with the string surface area definition method
41
42 (*
43 Note that in the below blocks, the invocation function we defined above is being passed two things--an object and property
44 The object we pass it must be of the same type expected by the function itself, and the property must also follow suit
45 *)
46 let volume1 = invokeDigt1 objDigt1 blockHead.Length //variable representing the invocation of the volume function with integer length
47 printfn "Volume of Cube %d: %d" blockHead.Name volume1 //print the integer volume of the cube
48 let surface1 = invokeDigt2 objDigt2 blockHead.Length //variable representing the invocation of the volume function with integer length
49 printfn "Surface Area of Cube %d: %d\n" blockHead.Name surface1 //print the integer surface area of the cube
50 let volume2 = invokeDigt3 objDigt3 wigglyWorld.Length //variable representing the invocation of the volume function with integer length
51 printfn "Volume of Cube %s: %s" wigglyWorld.Name volume2 //print the string volume of the cube
52 let surface2 = invokeDigt4 objDigt4 wigglyWorld.Length //variable representing the invocation of the volume function with integer length
53 printfn "Surface Area of Cube %s: %s\n" wigglyWorld.Name surface2 //print the string surface area of the cube
54
55

```

Below is a screenshot of the expected output:

The image shows a screenshot of the Visual Studio Code editor with a file named 'Program.fs'. The code defines two classes, 'cubeClass' and 'wigglyWorld', and several delegate functions. It then creates instances of these classes and uses the delegate functions to calculate volume and surface area. The bottom panel shows the 'TERMINAL' output, which matches the code's logic, showing calculations for 'Cube 1' and 'Cube two'.

With this under your belt, you are prepared to use interoperable F# to make basic applications in the .NET environment. **Delegates can be used to simulate pointers to functions that APIs can use, and generics allow function and type declarations that account for many data types.**

In conjunction with what you've learned prior to this assignment, you should be able to understand how F# works on the small-scale and where it would be implemented—and how to implement it. One more lesson will follow, to describe some of the behind-the-scenes of F#, and describe the motivations and mechanisms behind its CPU-friendly design.

Advanced Lesson 2: Exception Handling with Computation

In F#, like in any functional language, there are often runtime-computations going on in the background of expression execution known as **computation expressions**. These expressions are types of monads, or wrappers that define operators for

combining return values into unique types. Obviously the concept of a monad is much deeper than that, but for a **computation expression** you can think of it just like that, being that these expressions are used to wrap up pieces of existing, functional code into concise wrappers that execute independently of the user runtime.

There are, naturally, many kinds of computation expression, and their implementations can seem very nebulous and conceptual at first glance. Why would subsection of code need to be recomputed into a type with an operator, when it could just have its execution flow controlled by decisional logic? That would be the answer for many imperative or procedural languages, where logic is processed on a single thread, top-to-bottom, so that a user can use that logic to perform tasks one at a time. Functional languages do not often exist on this level however, and for F#'s case, it's actually *strikingly* often that multiple threads of execution might need to happen simultaneously within a .NET app. What if you have an app that displays UI, but also needs to query a server for updates and then install them? Do you simply stop displaying the UI? Do you make the user wait for the install? Or, what if you use an F# app to drive a GPU, which essentially *necessitates* multithreading in order to efficiently allocate video memory? Are you going to ask users to install 128GB of RAM on their computer in order to run Minesweeper?

One example of this sort of packaging using **computation expressions** is in **asynchronous expressions**, a group of expressions designated by the keyword **async** that can specify operations through multiple servers or .NET apps at once without the execution of one package affecting the speed or execution of another. The general syntax for an **async** expression is as follows:

let [name] (parameters) = async {[body]}

Where parameters can be passed as constants or even collections, allowing the **async** block to be run for each member in the collection. A common example is contained in this image, courtesy of the official Microsoft documentation, for an **async** program that uses the **Async** library and a couple external applications to print the names of a few specific websites, all at once:

```

open System.Net
open Microsoft.FSharp.Control.WebExtensions

let urlList = [ "Microsoft.com", "http://www.microsoft.com/"
                "MSDN", "http://msdn.microsoft.com/"
                "Bing", "http://www.bing.com"
              ]

let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length name
        with
            | ex -> printfn "%s" (ex.Message);
    }

let runAll() =
    urlList
    |> Seq.map fetchAsync
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore

runAll()

```

Note that **pipelining** is used to call library functions here, after passing the **async** block a parameter (which is, in this case, a collection.)

This is pretty abstract though, and beyond the scope of this tutorial, obviously. Using libraries this dexterously is something you do as a professional who is familiar with an environment—as a beginner, just looking to master the syntax and the concepts, we can look at a much simpler **computation expression** that can help optimize execution at every level.

Lazy expressions, as promised in the previous lesson, are an important concept in F# and come up pretty frequently, the advanced idea of the **computation expression** notwithstanding. By default, many expressions in F# are **lazy evaluated** by the compiler, which again means that aspects of an expression, or sometimes an entire expression, are only evaluated a maximum of *one time*, and future use of that expression relies on a sort of precedent set by the first evaluation. This is why expressions are constant by default, and if you remember, you need that **mutable**

keyword to make expressions variable. This is also why the whole language may seem a little obtuse and strict at first, and why it's considered "strongly-typed:" as soon as an expression is compiled and evaluated, it's never going to have its type evaluated again unless the compiler is explicitly told to do so.

So, how *do* we tell the compiler to do so? Using **lazy expressions** explicitly, we can specify that an expression is *not* to be evaluated until exactly when it is actually needed for execution. We can specify it by simply using this syntax:

```
let [name] [parameters] = lazy ([body])
```

And we can make it *actually* be evaluated for the one time it's *going* to be evaluated using the **Force()** method specified by the **lazy** object we are invoking here. For example:

```
let function x = lazy (3 * x)  
let result = function 5  
printfn "%d" (result.Force())
```

Note that **function** here is not *actually* returning an integer, because the expression isn't being fully evaluated when the compiler comes across it. Instead, it's an un-evaluated expression of type **Lazy<'a>** that will be typecast to an **int** when the **.Force()** method encounters it. Note that this is a **generic**.

This kind of expression can be used to save a lot of performance on large programs with dense, mathematical operations, but it can *really* shine when used for **exception handling**. If you can prevent an expression from being evaluated until you need it, you could factorize an expression into many parts and evaluate each part conditionally without either ending the program due to an error, or using a **try/catch** block (which F# does support.) The former is obviously a big problem, but the latter poses the question of why we would use this approach *over* a traditional **try/catch** block if we have access to it. This is because a **try/with** block, as it appears in F#, requires both its own, unique syntax and a reliance on specific compiler errors springing flags that can be predicted. You could always make a **class** representing your own, custom **exception**, but at that point you have done exponentially more work than simply using a **lazy** keyword with an **if/else** block.

Let's look at an example to see how simple this kind of exception handling can be. The classic example in F# is division by zero, which incidentally *does* cause a specific compiler error. The below example, adapted from an example by the once-in-a-generation minds at Stack Overflow, demonstrates two functions that both attempt to divide by zero, but only *one* utilizes a **lazy expression**:

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

In the first function, we can see that there is an **if/else** block that checks if a divisor is zero or not and intends to print out a message saying so, rather than trying to actually perform that operation and hitting a snag in the compiler. This will never occur, however, as the compiler evaluates **oneOverX** as soon as it sees it, in order to determine its type. It will see that there is division by zero when $x = 0$, and the program will fail to run altogether before the **if/else** is ever reached. The second function, however, specifies that **oneOverX** is a **lazy expression**, so the compiler will see it and immediately ignore its actual value, typing it as **Lazy<'a>** and moving on with compilation. Therefore, the **if/else** block is actually checked before the entire code fails to run, and should $x = 0$ it will simply display the given warning, and never encounter the call to the **.Force()** method that would perform the division by zero.

Hopefully, this gives you some perspective on why it might be useful to understand not only **lazy expressions** but **computation expressions** in general. Being able to execute code independently of the runtime environment is the kind of unique

advantage that only a functional language could easily supply, although it's no secret that many imperative languages now support asynchronous programming in some cases.

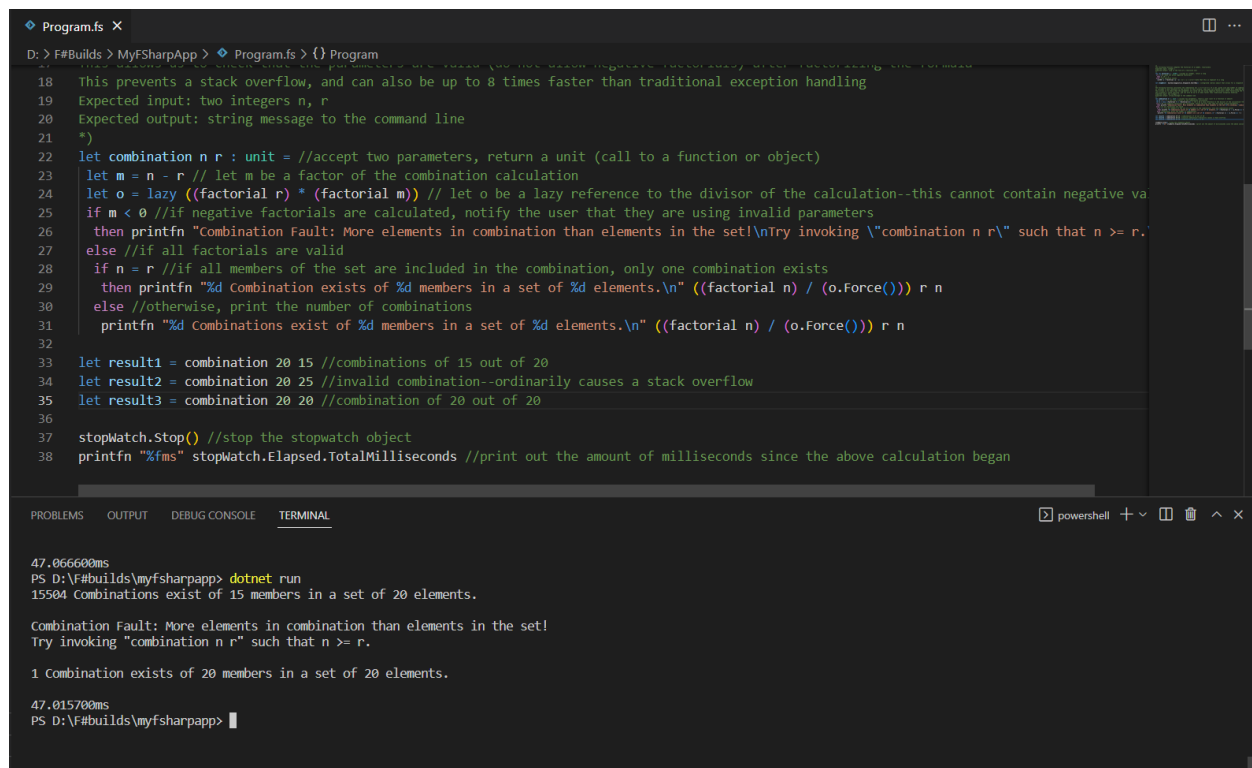
ASSIGNMENT: Using what we know, then, we can rework a previous assignment to be safer and more efficient than it was previously. Take assignment 2, then, where you created a permutation calculation; rework the **brute-force** implementation to calculate a combination, not a permutation, in the form:

$$nCr = n! / r!(n-r)!$$

Where n and r are the same as they were previously. Instead of rewriting this implementation to use **composition** and **pipelining**, use **lazy expressions** to prevent your calculation from operating on combinations for more elements than there are members in the set. This shouldn't throw an error or end the program, and should simply skip evaluation and move on, which means that your program should output three for three cases: $n > r$, $n = r$, and $n < r$. This will also mean reworking your factorial function to prevent it from calculating negative factorials. Below, an example is shown using large numbers to emphasize the processing speed saved by **lazy expressions**:

```
Program.fs X
D:\> F#Builds > MySharpApp > Program.fs > {} Program
1 (*
2 The following function computes the factorial of a number, recursively
3 Expected input: integer x
4 Expected output: long in the form of a recursive call
5 *)
6 let rec factorial x : int64 = //accept an integer, return a long
7   if x = 0 //does not allow negative factorials
8   then 1 //0! = 1
9   else //for positive values
10    (int64) x * factorial (x - 1) //x! = x * (x-1)!--note that this is typecast to a long
11
12 let stopwatch = System.Diagnostics.Stopwatch.StartNew() //integrated library object that allows for a stopwatch in milliseconds
13
14 (*
15 The following function calculates the combination nCr = n!/(r!(n-r)!) of a set, given size and number of combination elements
16 It includes a lazy reference to the divisor of this formula, that does not execute until needed by a separate declaration
17 This allows us to check that the parameters are valid (do not allow negative factorials) after factorizing the formula
18 This prevents a stack overflow, and can also be up to 8 times faster than traditional exception handling
19 Expected input: two integers n, r
20 Expected output: string message to the command line
21 *)
22 let combination n r : unit = //accept two parameters, return a unit (call to a function or object)
23   let m = n - r // let m be a factor of the combination calculation
24   let o = lazy ((factorial n) * (factorial m)) // let o be a lazy reference to the divisor of the calculation--this cannot contain negative values
25   if m < 0 //if negative factorials are calculated, notify the user that they are using invalid parameters
26   then printfn "Combination Fault: More elements in combination than elements in the set!\nTry invoking \"combination n r\" such that n >= r.\n"
27   else //if all factorials are valid
28     if n = r //if all members of the set are included in the combination, only one combination exists
29     then printfn "%d Combination exists of %d members in a set of %d elements.\n" ((factorial n) / (o.Force())) r n
30     else //otherwise, print the number of combinations
31     printfn "%d Combinations exist of %d members in a set of %d elements.\n" ((factorial n) / (o.Force())) r n
32
33 let result1 = combination 20 15 //combinations of 15 out of 20
34 let result2 = combination 20 25 //invalid combination--ordinarily causes a stack overflow
35 let result3 = combination 20 20 //combination of 20 out of 20
36
37 stopwatch.Stop() //stop the stopwatch object
38 printfn "%fs" stopwatch.Elapsed.TotalMilliseconds //print out the amount of milliseconds since the above calculation began
```

Below is a screenshot of the expected output:



The screenshot shows a Visual Studio Code editor with a file named `Program.fs` open. The code defines a function `combination n r` that calculates combinations using lazy evaluation and a `stopwatch` to measure execution time. The code includes comments explaining the logic and the purpose of the `stopwatch`. The terminal output shows the results of running the program, including the total runtime and the output of the `combination` function for various inputs.

```
Program.fs
D:\F#Builds> MyFSharpApp > Program.fs {} Program
18 This prevents a stack overflow, and can also be up to 8 times faster than traditional exception handling
19 Expected input: two integers n, r
20 Expected output: string message to the command line
21 *)
22 let combination n r : unit = //accept two parameters, return a unit (call to a function or object)
23     let m = n - r // let m be a factor of the combination calculation
24     let o = lazy ((factorial r) * (factorial m)) // let o be a lazy reference to the divisor of the calculation--this cannot contain negative va
25     if m < 0 //if negative factorials are calculated, notify the user that they are using invalid parameters
26     then printfn "Combination Fault: More elements in combination than elements in the set!\nTry invoking \"combination n r\" such that n >= r."
27     else //if all factorials are valid
28         if n = r //if all members of the set are included in the combination, only one combination exists
29         then printfn "%d Combination exists of %d members in a set of %d elements.\n" ((factorial n) / (o.Force())) r n
30         else //otherwise, print the number of combinations
31             printfn "%d Combinations exist of %d members in a set of %d elements.\n" ((factorial n) / (o.Force())) r n
32
33 let result1 = combination 20 15 //combinations of 15 out of 20
34 let result2 = combination 20 25 //invalid combination--ordinarily causes a stack overflow
35 let result3 = combination 20 20 //combination of 20 out of 20
36
37 stopwatch.Stop() //stop the stopwatch object
38 printfn "%fms" stopwatch.Elapsed.TotalMilliseconds //print out the amount of milliseconds since the above calculation began

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
47.066600ms
PS D:\F#builds\myfsharpapp> dotnet run
15504 Combinations exist of 15 members in a set of 20 elements.

Combination Fault: More elements in combination than elements in the set!
Try invoking "combination n r" such that n >= r.

1 Combination exists of 20 members in a set of 20 elements.

47.015700ms
PS D:\F#builds\myfsharpapp>
```

Note also that this example utilizes the **stopwatch** object from an F# library to display the total runtime. This is not necessary in your implementation. All that is necessary is that you understand how **computation expressions can package code for efficiency, and make existing code safer and more optimal in both local and server environments.**

And, with that, your cursory overview of F# is complete. These lessons have been designed to familiarize you with the typical use cases of F# and the motivation behind its existence. Did it succeed in its goal of being interoperable and efficient among a sea of more highly-specified languages? It seems so, because nearly every unique or dense feature in the entire environment exists to facilitate interoperability, from the advanced computations at the top all the way down to brass tacks, like expression evaluation and data declaration.

As a recent user of this language, I personally found myself struggling at first to understand its approach to parameterization and declaration—I used my knowledge of

C# to try and navigate some of the uncharted territory. A lot of the object-oriented concepts and computation functions are also implemented in C#, and everything that wasn't was and still is well-documented online. The concepts aren't honestly the hard part, or at least they weren't for me, because I personally have a significant background in math. What was difficult was the bizarre syntax and surprisingly unhelpful compiler, which I was constantly at odds with. Really, an unfortunate amount of syntactical details can be solved by simply trial-and-error-ing your way around, flipping keywords and operators around until the compiler stops complaining. This is effective because of how much implicit work the compiler does, making this a double-edged sword. It's a surprisingly easy language to learn for anyone with a background in C# or Java (or even Javascript, for that matter,) but it has just as many quirks and nuisances as any other language.

Because of its relative ease, however, I think anyone with a strong stomach for finicky compilers and scripting languages could latch onto this quickly, making it a good choice for a first functional language. There are plenty of ways to ease into more difficult concepts like multithreading or delegation, which in turn gives a new user a good knowledge base to dive into APIs and interoperability—two inordinately useful concepts on the job market. But, is F# a good *representative* for functional languages? Perhaps not, because its integration with .NET and C# might give it some unique quirks that people out of the loop find esoteric, and other functional languages simply don't tolerate. Multi-purpose **this** keywords? Asynchronous threads? Implicit typing? Manual entry points? Most functional languages don't try to solve problems to which such features are germane, but F#

commits itself to including a whole bunch of features for two reasons. The first reason is the one I explained earlier, at the start of this tutorial—competing

```
fizzBuzz :: Integer -> String
fizzBuzz x
  | x `mod` 15 == 0 = "Fizz Buzz"
  | x `mod` 5  == 0 = "Buzz"
  | x `mod` 3  == 0 = "Fizz"
  | otherwise      = show n
```

A small snippet of the Haskell language, demonstrating an implementation of the “Fizzbuzz” division game. Note the barebones approach to traditional, imperative logic operators

coding languages and apps need simple, interoperable languages to keep them organized. The second reason, though, only occurred to me after I personally had to write some apps using the language, it being that F# is a functional language that works *like* an imperative language *if you want it to*. So, a learner can keep the safety blanket on for as long as they need to, yet still get the experience of a functional language's approach to data.

Overall, I definitely *do* recommend F# as a functional language, especially for beginners with the sphere. It's easy and meshes well with the most common IDE on the market, and everything it makes is highly portable. Plus, it includes a lot more features than most would expect from a functional language that might make you feel more at home if you're already very experienced with other languages, whatever they may be. It has some drawbacks in that it might hand-hold a little *too* much, therefore stifling the ability to make expressive mathematical programs, but that concern is pretty niche in reality. F#'s ease-of-access and long list of features makes it an undeniable powerhouse of functional programming.

REFERENCES

1. "F# Tutorial." *Tutorialspoint*, Tutorials Point,
<https://www.tutorialspoint.com/fsharp/index.htm>
2. "F# Docs - Get Started, Tutorials, Reference." *F# Docs - Get Started, Tutorials, Reference.* | *Microsoft Learn*, Microsoft, <https://www.learn.microsoft.com/en-us/dotnet/fsharp/>
3. "F# Software Foundation." *F# Software Foundation*, Microsoft, <https://fsharp.org/>
4. "F# Snippets." *F# Snippets*, <http://fssnip.net/>
5. "F# For Fun and Profit." *F# For Fun and Profit*, F# Software Foundation,
<https://fsharpforfunandprofit.com/>
6. kvb. "F# Lazy Evaluation vs Non-Lazy." *Stack Overflow*, Stack Exchange, 5 Aug. 2014, <https://stackoverflow.com/questions/6683830/f-lazy-evaluation-vs-non-lazy>