

F# IN ACTION

Amanda Laucher

 MANNING





MEAP Edition
Manning Early Access Program

Copyright 2009 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

Part 1 Expanding vocabulary and thought process

Chapter One: F# A multi-paradigm Language on the .NET platform

Chapter Two: F# basics

Chapter Three: Functional Programming

Part 2 Mixing your new tools with the old

Chapter Four: Multiparadigm Focus

Chapter Five: Concurrency in the real world

Chapter Six: Interop with C#

Chapter Seven: Working effectively with Data

Chapter Eight: Updating and working with legacy applications

Chapter Nine: F# in production

Part 3 F# by example

Chapter Ten: DSLs

Chapter Eleven: Services for your SOA

Chapter Twelve: Visualizations with F#

Chapter Thirteen: Rules Engines and F#

1

F# - A multi-paradigm Language on the .NET platform

This chapter covers
Introducing F#
Why you would want to use F#
Paradigms that F# is modeled from
F#'s place in today's real world solutions

F# is the newest fully supported language on the .NET platform. The language is built on the premise of being a more concise and expressive way of solving problems in the Microsoft space by subscribing to a multiparadigm approach. Although it is a statically typed language, the types are inferred by the compiler which leads to a less verbose syntax. Function composition is favored over extensive inheritance models so the code is more declarative and expressive, reading more like a natural language than imperative counterparts.

F# came out of a Microsoft Research project in which Don Syme and a team were looking to develop a more functional, ML style language on .NET. They borrowed heavily from other languages with proven track records, such as OCaml, and Haskell. Don't worry if these languages are not familiar to you. You don't need to understand them in order to pick up F#. We'll discuss some of the ideas when they become important or can help you to understand ideas in this language. F# is many developers first formal introduction to these concepts.

In this chapter we'll take a look at what F# is and why it is an important addition to the .NET ecosystem. We'll see what it means for a language to be multiparadigm, comparing some of the underlying concepts of the paradigms and then taking a look at what happens when they are used together in the same language. This is quite valuable in many situations, such as building concurrent applications, or reasoning about large amounts of data. While we'll discuss many situations where F# is a great fit, we'll also take a moment to discuss where other languages might be more suitable. The rest of the first section of the book will dig into the language syntax so that we can focus on building real world applications.

1.1 What is F#

F# is defined as a multi-paradigm language. This is a fancy way of saying that its constructs do not conform strictly to any one way of thinking. A paradigm is a way of solving a problem. It is a general set of guidelines to follow when writing software. Notice that it is not a strict set of unbreakable rules. Many languages have constructs that allow conformity to multiple paradigms (including C# and VB.NET!). Take for instance the idea of recursion, or a function that calls itself in its body. This is an idea from the functional paradigm that can be created in almost any language; however some languages better facilitate the use of recursion while some make it more difficult to do so.

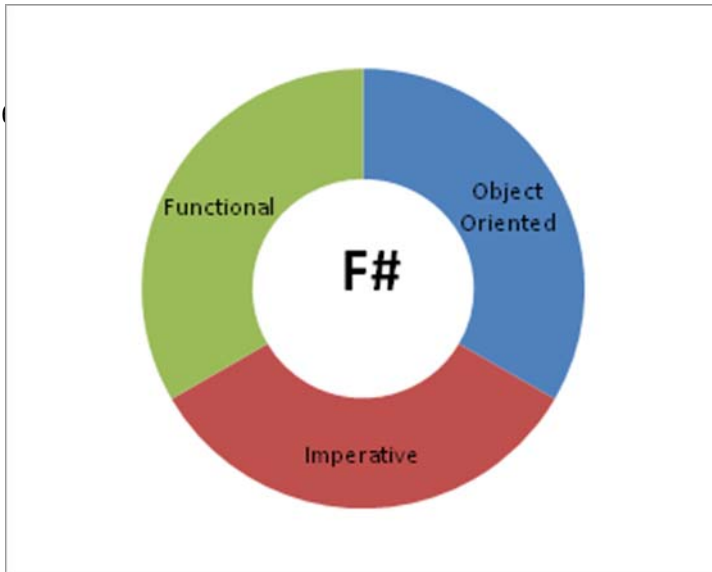


Figure 1.# F# Paradigms Modeled

The above figure shows the paradigms that compose F#. We'll discuss a bit about the paradigms that F# is modeled from.

1.1.1 Object Oriented and Imperative Paradigms

For the past 20 years, most mainstream development has been done in languages that support mostly imperative or object oriented concepts. Some of you will remember when the idea of object oriented development became mainstream. It was sold as the only right way to develop software. It took many years for line of business developers to jump on the bandwagon and to become good at this method. As you know, many developers are still not really doing it very well. It is hard to change the way you think about solving problems. It takes much diligence and practice to become good.

This type of programming is great for modeling the nouns of a system. You listen to a customer speak of a problem domain with an ear that picks out nouns, and then build the system around an inheritance model. We have the ability to create objects from classes which describe their data and behavior. OO is in fact a way of organizing code. It is a way of putting specific concepts together in a way that is easy to understand when thinking about these nouns, their data and behavior. It is hard to imagine the days before the 3 main tenets of object oriented programming: inheritance, encapsulation, and polymorphism.

There has since been a broad recognition of shortcomings which are addressed by the development of design patterns. Design patterns give a name to a development pattern so that we can talk about it with other developers. They are a tool for communication. Most good software developers and architects are familiar with the Gang of Four design patterns. Some of those patterns will be addressed later in this book with alternative ways to think about the problems that they address.

1.1.2 Functional and Declarative Paradigms

A paradigm which has had less attention from enterprises and line-of-business (LOB) developers is that of functional programming (FP). The FP developer hears problem domains

differently that the object oriented developer, picking up on the verbs, or actions of the system and **composing** these actions together to form a system. This composition and abstraction of common actions is the main idea behind FP.

Functional languages **compose** immutable constructs into function's inputs, process, and outputs to solve problems while avoiding any state manipulation. The term immutable means that the value cannot be changed. Consider the following C# code.

Listing 1.# Mutable Value

```
Person Josh = new Person();
Josh.City = "Sydney";
Josh.City = "London";
Josh.City = "Chicago";           #1
```

The value Josh.City is mutable and can be changed in an imperative language .

This cannot be done in a purely functional language. The fact that those values are mutable allows them to be updated. This is an oversimplified example of how code manages state. The value of City is different at different times of the codes execution.

Not being able to update those values might seem like an impossible way to develop software at first but if you think back to algebra class, we might consider the value of x in $x = 5 + 10$. The value s specified and then not changed. The value of x might depend on other values such as: $x = 3y + 2$. The value of x might also be determined in terms of a specific function such as: $f(x) = x^2 - 4x + 12$. In any of these cases, the value is set and not changed. The value of a function of x is different based on what the value of x is, but all of these values are in fact immutable.

This idea is valuable because anytime that this code is evaluated with the same inputs, the value is always going to be the same. This is a concept called **referential transparency**. Nothing can change the result of this evaluation. The value of the function is fixed, and therefore the definition of the expression can be replaced with its value and the application's behavior will not be changed. Think about an application with referential transparency in regards to testing. We are able to better reason about the behavior of the application and the program's correctness can be more easily proven.

Referential Transparency

Referential transparency is a flowery way of saying that an expression can be replaced by its value and will have the exact same meaning. This means that every time you evaluate an expression with the same input's, it will have the same outputs and side effects.

This is not the case in many imperative languages. Consider a function that determines its value based on a global mutable variable. The variable may change throughout the

cycle of the program, and since it is not passed into the function as a parameter, the function is not referentially transparent.

```
int x = 42;
void printx(){
    System.Console.WriteLine(this.x.ToString());
    x = 3;
```

The value of the expression has to be the same at all points during a programs execution. This is very common in functional programming as values, once set are not able to be changed, so even global values can be said to be referentially transparent.

Everything is an expression in a functional programming language, as it is in F#. There are no statements, which are pieces of code that do not return a value. Instead of returning a value, statements might declare an identifier (class Order {}), set a variable's value, or execute control flow (if/then/else/endif). An expression must have a return value, even if it is unit (). If a program contains statements, it is likely that it is not referentially transparent. If you replace the statement with its value, the program will behave differently.

Unit Type

The type unit is similar to void in C#. In F# it can be used as an input or the return type for a function. The syntax is empty parenthesis; (). You might think of it as an empty tuple or one of 0 arity. If a function is taking in unit as a parameter, it might look like this: let function1() = 42. This is the same result of passing in no values stored in a tuple. If a function returns unit, this means that there is no other meaningful value to be returned and it is likely that a side effect has occurred in the function. For instance a function that prints to the console would return unit.

If you ask several developers what it is about programming languages that makes them functional, most will have different answers or find it hard to give an answer in the first place. Most, however, will tell you that functions must be first class citizens in an FP language. This means that functions themselves have a type and a value. They can be used just as data, meaning they can be passed around between functions as parameters and return types as well as put into lists or other data constructs. Another way of saying this is that FP languages have **higher order functions**. We'll discuss functions types in the next chapter.

FP has recently been receiving so much press because of the slowdown of the increase of processor speed and increase of multiple processor machines. This is commonly referred to as the slowdown of **Moore's Law**. Writing code that maintains state by making objects mutable makes it really difficult to make code run on multiple processors and machines. Something can happen on one thread to change a value that was expected to be used on another thread. This makes it difficult to write software to work on multiple processors, using

all of the systems resources. While avoiding global mutable state in programs written in any language is typically good practice, FP and immutability is a natural solution to the problem. We'll discuss this more in section 1.2.4.

1.1.3 Multiple Paradigms

F# is not a pure functional programming language. It is also not an object oriented programming language. It is a multiple paradigm language, meaning it has the constructs and ideas behind OO and FP. Each paradigm has constructs that make problem abstraction simpler for software development as well as its own list of shortcomings. When combined, we get a language that is extremely powerful, and like anything else, care must be taken to use it correctly. Using some of the ideas from one paradigm can counteract the benefits of the other and this can be very dangerous. It is important to consider what parts of each paradigm will be used during the completion of a specific task. In order to do this we must examine many examples of mixing paradigms.

There are two techniques to a multiparadigm approach. First is the idea of using both paradigms in the same language or even the same file which as previously mentioned can be quite complex and requires much care. The other way to use this multiparadigm idea is to use libraries that are functional in conjunction with those that are object oriented. In this book we'll discuss how to properly use this complex language to create robust software using both techniques, and how to decide which is appropriate in certain situations. We'll look at different ways to arrange our code and ideas so that we are utilizing the best aspects of each paradigm and steer clear of the common pitfalls.

By this point you might be thinking that we already have multiple .NET languages available to us. Why would Microsoft not just add these concepts to one or both of the existing languages? Indeed this has been done in many cases. Some of the concepts we've discussed have recently been added to the other languages. The next section will look to answer this question.

1.2 Why do we need another language

Why do we need yet another language? This might be the first question that comes to mind when you hear that there will be another language available when installing Visual Studio 2010. As mentioned in the previous section, F# comes out of ideas that have been used in other programming languages, which are much older than .NET itself. For that matter, there is no shortage of computer languages. There are thousands of languages, many of which, most programmers have never even heard of. If you consider the fact that computer programming is a relatively young practice, it can be difficult to imagine why there are so many different languages, see figure below. This is because many developers think of a language as something that can be used universally to complete every programming task that may arise. If instead we think about each language as way to complete a certain set of tasks in a very clear way, it is easier to imagine. Let's think about a project for building a house.

//Insert figure here with timeline of many languages.

Figure 1.# Language Timeline

The figure shows a timeline of a small portion of languages. Each language has a specific purpose in the .NET ecosystem.

Imagine a project to build a house. A group of teams would be employed for this job. The job is broken down into several parts which are assigned to different groups. There will be a group that will put up the walls, a group will do the plumbing, one will do the roofing and so on. Each of these teams is very good at what they do. In fact, some of the teams will have skills that are transferable to different parts of the job. They could get the task done, but they may not do it in the best possible way and therefore they probably won't be the best team for the job.

If we transfer this analogy back to computer programming languages it is easier to see why we need many languages. Each language has a job that it is very good at. It may be able to do many jobs. It may be good for just one or two. As long as these languages can easily work together, we should use the correct language for the job.

This idea seems to go against what has become the normal way of thinking. About every 10 years we are introduced to a new language that is supposed to take over the development world. It tries to put one team up to the task of building an entire house. What makes this language any different? We already have tools that can help solve most any task that we come across. How do we know that it is time to look at another language for a job?

Are we always delivering the best possible solutions? Are they the fastest? Are they using the right amount of system resources? Do the users complain about seemingly simple problems, which end up being incredibly difficult to fix? Do you find yourself writing a lot of extra syntax around the code that actually solves the problem? These are all indications that you might be using the wrong language and are coming up with the wrong solutions. F# is not an attempt to solve every software development task. It is simply the best language for many tasks. This book will attempt to describe more about what those tasks are, and how exactly to accomplish them. We'll discuss ways of working with large amounts of data, creating our own domain specific language, rules engines, concurrent applications and many more. Although it would be impossible to detail every task that F# is best suited for, hopefully by the end you will be able to choose when F# is the right decision.

C# and VB are mostly imperative object oriented languages that have been the most prevalent in the Microsoft space for about the past 10 years. F#, however, represents a new way of thinking about and solving problems. It is a language that subscribes to multiple

paradigms of thinking including the previously mentioned object oriented and imperative as well as the less commercial functional paradigm. We'll discuss the ideas behind these paradigms in just a moment.

Learning F# will have value for development teams even if it is never used in a production environment; although I doubt that will be the case. F# forces developers to think differently. It opens up an entire new world of software development tools. In fact it works together with the other .NET languages to strengthen applications, as well as lends ideas to those languages for new design techniques. We'll discuss techniques for working effectively with other languages and describe resources that can be used to facilitate in designing C# applications in a more functional manner.

1.2.2 Interoperability

Think back to the example of building a house. Imagine if the roofing team couldn't communicate with the team that is putting up the walls, or if they couldn't communicate with the team laying the foundation. It would be terribly difficult for the final product to be something that could even be considered acceptable. The same can be said for using many languages to solve different tasks. If they can't work with the libraries built in other languages, it will be very difficult to have an acceptable application.

With all of the languages available today, it takes more than a wide variety of abstraction constructs to impress the commercial world. F# also has the backing of the .NET platform. Garbage collection, DLL versioning, extensive Base Class Library, and interoperability with a variety of other languages are a few of the niceties that come with being a fully supported .NET language. The F# team has paid specific attention to making the language work relatively easily with the other .NET languages, even though they have very different capabilities. We'll discuss the caveats of interoperability in Chapter 7.

1.2.3 Syntax and program structure

F# is a statically typed language. It uses type inference to determine that type of a value which makes the syntax more concise as it might appear in a dynamically typed language. Make no mistake, the type is set and checked at compile time. Other languages make it very verbose to use a static type compiler. F# tries to infer the type and you only need to specify when it is unclear. This often occurs when using values from other libraries. Values are as generic as possible. A function may be able to operate on many types of values. F# does not make you code the same function multiple times to accomplish this task. Below are some examples of F# assignments. It is not the time to go too deep into the syntax, but you will see a bit how the type inference works.

Listing 1.# Type Inference

```
let intList = [7;10;3] #1
let stringList = ["seven";"ten";"three"] #2
```

```

let anyList value = value::[] #3
let floatList (value:float) = value::[] #4

val intList : int list = [7; 10; 3] #5
val stringList : string list = ["seven"; "ten"; "three"]
val anyList : 'a -> 'a list #6
val floatList : float -> float list #7

```

#1 A list of integers defined
#2 A list of stings defined
#3 A function taking in a value and appending it to an empty list
#4 A function with a parameter specified to be a float, returning a list of floats
#5 Return type and value in FSI showing type inference of int list
#6 Automatic Generalization in #3, the compiler infers generic type 'a
#7 Type specification in #4 made the result a float list

NOTE COPY EDITOR, PUT CUEBALLS IN TEXT AND CODE.

Here I have created an identifier called `intList` and assigned the value `[7;10;3]`. Notice that I didn't tell the compiler what type the value will be. If you look at #5, you will see that the type is inferred to be an int list. The same is true in #2 where the identifier is inferred to be a string list. Notice #3. In this expression we take in a parameter, called `value`. The value is then added to a list. Nowhere in this statement is any type inferred. The value can result in any type of list. The compiler infers this value to be a generic list, or 'a list as seen in #6. There may be times when we don't want the value to be so generic. We may want to ensure that the type of the parameter to be passed in is a float. #7 shows the return type from #4 as a float list.

As you can see, F# has an extremely succinct syntax, much of which has been borrowed from a language called OCaml. It gets rid of a lot of the extra keystrokes that don't add any functionality to code but are necessary in other .NET languages. It uses white space to determine scope instead of brackets and semicolons which might take a little while to get used to, but it makes the code very easy to read. In most cases F# code can do what C# code is doing with much less code and even fewer files. This makes it very appealing to C# developers who are accustomed to typing large amounts of code in order to accomplish even trivial tasks. Consider the following example which calculates a very simple algorithm called the Fibonacci Sequence. This is an algorithm that is commonly used to show code in a functional language. It could be said to be the Hello World of FP. This is about where the value stops as the algorithm has been implemented already in most languages. Notice the C# example has several keystrokes that are not needed by the F# compiler. Again don't worry if all of the code in these early samples doesn't seem familiar.

Listing 1.# recursive Fibonacci function in C# and F#

C#

```
static int fib (int x)
{
    if (x < 2)
        return 1;
    return fib (x-1) + fib (x-2);
}
```

F#

```
let rec fib x = if x < 2 then 1 else fib (x-2) + fib(x-1)
```

The difference between the way the syntax for the Fibonacci algorithm looks in C# vs F#. Notice the additional characters needed in C#.

In general all lines of the F# file are evaluated top to bottom and left to right. The figure below shows for example if function A is defined after function B, then function B cannot refer to function A. Files in an F# project are evaluated top to bottom so the same is true. This leads to clean code organization with no possibility of circular references.

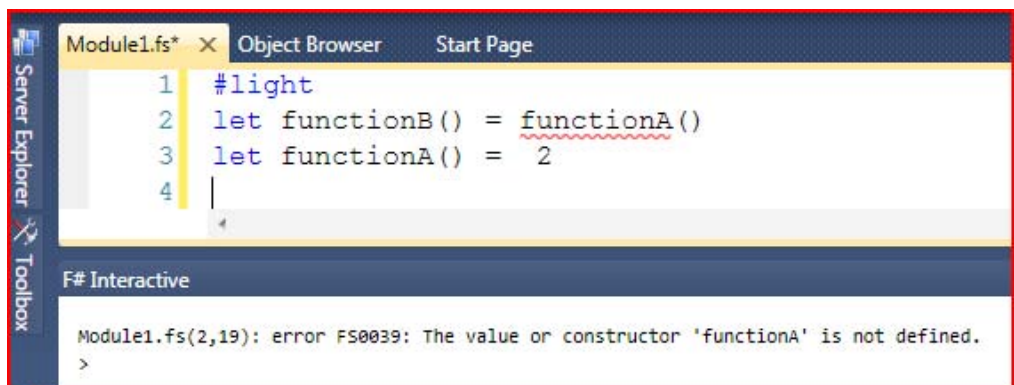


Figure 1.# A Visual Studio screenshot of error due to evaluation order

The screenshot shows Visual Studio compilation error and the error is identified in F# Interactive. Here we see that the order of the functions is important.

As mentioned earlier, F# favors functional composition over inheritance. With this idea comes additional types of functions. Partial application of parameters is something that is very difficult to accomplish in many of today's mainstream languages. This is a very powerful tool even if not immediately obvious. One of the first reasons that it becomes valuable is the order of declaration. Suppose you know the value of most of the parameters at one point in the application, and not one of the others. Also, you could want to assign identifiers to values with some parameters given in order to make the code more declarative. This concept will be described in more detail in the functional programming chapter within this section.

Polymorphism is one of the main tenets of object oriented development. Partial application and type inference change the way that polymorphism is handled in F#. It is not possible to have any 2 functions with the same name but with different parameters in a given scope. This also leads to cleaner more maintainable code. It can be difficult for developers who are used to organizing their code in an object oriented manner though. Polymorphism is instead accomplished with higher order functions and functions that can operate over multiple types, or generic functions.

Although some of these style semantics seem difficult to become accustomed to, the syntax and program structure are a very valuable asset in F# as they lead to different ways of thinking of a task. When you must adhere to these guidelines, you come up with very different solutions. They tend to lead to cleaner, easier to read and therefore more maintainable code. We'll get a better taste of that in later chapters as we look at larger examples. At this point let's talk about some of the benefits of F# with its multiparadigm approach.

1.3 Setting F# apart from other .NET languages

It has been said that VB.NET, C#, and indeed any other .NET language can accomplish any task. There are only a handful of situations where it is very valuable to choose one language over another. This distinction is much clearer with F#. In this section we will discuss the situations that might make you choose F# as the simplest solution.

1.3.1 Giving OO developers a toolset for concurrency

Consider for a moment a trip to the grocery store. After you have selected your many items, you go to the cashier who swipes each of your items and passes them to the conveyor belt. At the other end of the belt is typically somebody who will put the items into a sack. Each item must be scanned for its price and put into a grocery bag. This process goes much faster if two people doing the work. Items can be bagged while others are being scanned. The entire process is very slow if you must wait for the person scanning the items to then bag them. This is a process that consumers are very used to in a store, but when using a computer, we expect everything to be almost instant. We complain when we have to wait even just a few seconds. This is why it's extremely important to do multiple tasks at the same time.

Jay fields, author of "Refactoring Ruby" recently stated that "Adopting a language without a concurrency story is irresponsible, at this stage." This is true because the number of processors that are being put into computers. It has become apparent that in order to take advantage of system resources, developers will have to make conscious design decisions that allow for multi processor development.

Functional programming has traditionally been very good at handling concurrency. Pure functional languages have an advantage in that all constructs are immutable. This means that there is no chance of shared state and no worry about needing to lock memory. The fact that every evaluation can be completed lazily and results can be cached, after all, the evaluation of every expression will always have the same result (the grocery item, the price, and the bag are always the same) makes concurrency a side effect of other good practices.

In line of business application development with mutable state, concurrency has been a difficult task with the complications of shared memory access in more imperative based languages. Many languages have tried to allow for concurrent processes through the use of threading. Operating system processes are spawned and memory is locked until the main thread is rejoined later. Green threads are another type of threading involving spawned processes which are scheduled by the virtual machine but cannot span multiple processors. Neither option has been extremely successful as they are hard to write and even harder to test thoroughly.

In addition to the .NET thread pool and shared memory locking already available in all of the .NET languages, F# adds asynchronous workflows as well as Erlang style message passing to help deal with concurrency. These multiparadigm additions allow developers to more easily develop software which takes advantage of all of a machine's available resources while hiding some of the complexities of the current memory locking techniques.

Asynchronous workflows, seen in the example below are the main reason that many people start looking at F#. An Asynchronous workflow is an array of expressions. They allow for running multiple pieces of code in parallel easily, with extremely little syntax. There is no need to deal with memory locks at this level. The hard part has been done in the libraries behind the code. You can see in the sidebar the mention of monads. Asynchronous workflows are an implementation of a monad. We will learn more about creating our own monadic expressions like these in Chapter 4.

Listing 1.# Asynchronous Workflow Monad

A monad is the name for a method of chaining functions together in order to hide a functions complexities. Asynchronous workflows are an exmample of something that can be done with a monad in F#. The example below demonstrates how much simpler asynchronous code can be.

```
let identifier = [async{expression0}
                 async{expression1}
```

```

    async{expression2}]
    |> Async.Parallel
    |> Async.RunSynchronously

```

Concurrent programming is much simpler in F# than most other languages through the use of Asynchronous workflows. Each of the expressions above are executed in parallel. There is very little code needed to make them run in parallel instead of one at a time.

Erlang is an open source language that was originally developed by Ericsson, where it is still used for concurrent highly fault tolerant development for telephony products. It has been highly celebrated for its asynchronous message passing style of concurrency. This concept involves starting many processes that can only communicate with each other by passing around these asynchronous messages. A process waits until it receives a message and then processes it in its particular way. The same idea has been implemented in many languages and F# has implemented a similar method which involves mailboxes, messages and a messaging queue. This type of development is great for processing large amounts of data. We'll discuss message passing in F# in Chapter 6.

1.3.2 Lazy Evaluation allows us to evaluate as needed

Imagine writing an application that defines many long running operations that work with a great deal of data. If these functions are evaluated all at once, the system would slow to a halt and you wouldn't be able to produce results. It is nearly impossible to work with infinite amounts of data in this manner. It is valuable to evaluate these functions only when needed, and if they are actually needed. As mentioned earlier, all code is evaluated from top to bottom, so if a long running function is never needed, it could be a complete waste to evaluate it. Wrapping lazy around the function, as seen in the example below #1, allows this to be an on demand evaluation. When you need to actually run the computation, you would call the Value property on the identifier, #2. Line #3 shows the return value for longRunningFunction as a Lazy<int>. This means that the return is an integer type that is modified to not evaluate until it is forced.

Listing 1.# Using Lazy

```

let longRunningFunction = lazy(update allRecordsInDatabase) #1
let forced = longRunningFunction.Value #2
val longRunningFunction : System.Lazy<int> #3

```

#1 Wrap longRunningFunction in an lazy block.

#2 Force the longRunningFunction to evaluate

#3 The type of longRunningFunction is a modified int to show that it is lazy.

In purely functional applications, an identifier holds the same value at all times. One function cannot change the value of another. This means that it does not matter in what order values of functions are evaluated. They can be evaluated in whatever order is most

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=602>

convenient for the compiler. This means that we can use lazy evaluation instead of strict or eager evaluation which is most common in other .NET languages. As F# is not a purely functional language, it is also evaluated eagerly, however there is a keyword that allows us to tell the compiler that a function needs not be evaluated until it is specifically asked for. This can save on valuable processing power in cases where values never need to be evaluated. Some guidelines to consider when deciding whether or not to use the keyword, will be enumerated in chapter 6 about concurrency.

There are also some types that are specifically lazily evaluated. This allows for creating infinite and near infinite values. You probably wouldn't want to evaluate a list containing all of the digits of pi, as it would eat all of the systems memory. F# has types like lazy list and sequence which will only evaluate when necessary. You can perform operations on these types and call their given functions to work with specific subsets. This is referred to as short circuit evaluation.

Listing 1.# Infinite Sequence accessed lazily

```
let infiniteSeq = Seq.initInfinite (fun x -> x*2)           #1
let someOfInfiniteSeq = Seq.take 3 infiniteSeq             #2
val infiniteSeq : seq<int>                                #3
val someOfInfiniteSeq : seq<int> = seq [0; 2; 4]           #4
```

#1 Create an infinite list with an anonymous function

#2 Use the take function to evaluate for only 3 items in the sequence

#3 The type of infiniteSeq is a sequence of ints

#4 the value of someOfInfiniteSeq is a sequence containing 1,2 and 4.

In the above listing, #1 assigns the infiniteSeq identifier to an infinite sequence which is built with the (fun x -> x*2) function. You can see the type in line #3 is a seq<int>. This sequence is not actually generated until it is needed. Line #2 actually uses some of the values. You can see the value generated in line #4, a sequence containing the integer values of 0, 2, and 4. These are generated on demand, instead of all upfront. This saves the processing power that would be used to generate large or in this case infinite sequences. You can see how this idea is very powerful.

1.3.3 A rich type system allows us new capabilities

In addition to the already plush .NET type system, F# adds many new types that allow for functional development and easier handling of data. It seems to be a regular occurrence that a function needs to return multiple values.

In object oriented languages, special types needed to be defined to handle returning more than one result from a method. Tuples give the ability to concatenate values to create a new type, without having to explicitly define an object. Simply place a comma between values of any type and a new type is created defining a tuple of the multiple types. These

tuples can be used as parameters and return types to other functions. The following is an example of assigning a tuple containing 2 string values to the identifier "name".

Listing 1.# Tuple

```
let fName = "Amanda"           #1
let lName = "Laucher"         #2
let name = fName,lName        #3

val fName : string            #4
val lName : string           #5
val name : string * string    #6
```

#1, #2 Assign string values to the identifiers on the left of the equal sign

#3 Assign a tuple value containing values in #1 and #2

#4, #5 Type inference shows the results as strings

#6 Type inferences showing the string values separated by an asterisk to show the value is a tuple

The first line shows assigning the string value "Amanda" to the identifier fName. Similarly in line #2 we assign the value "Laucher" to the identifier lName. You can see the types were inferred to be sting in lines #4 and #5 which show the return value from #1 and #2. In line #3 we assign 2 string values to the identifier name. You can see the return type in line #6 is string * string. The * asterisk indicates that the value is a tuple, meaning that these two values only have value when used together. We'll discuss the implications of tuples in chapter 2.

It can be difficult to work with tuples of many values as the values can remain unnamed. If this becomes a problem, a record type can be created. A record is basically a tuple with named parts. A record can be accessed or created by the names given to the parts as seen in the next sample of code.

Listing 1.# Record

```
type Person = {Name : string; Age : int; HairColor : Color} #1
let author = {Name="Amanda"; HairColor = Blonde; Age = 28} #2
let authorAge = author.Age #3

type Person = #4
    {Name: string;
    Age: int;
    HairColor: Color;}
val author : Person = {Name = "Amanda"; #5
    Age = 28;
    HairColor = Blonde;}
val authorAge : int = 28 #6
```

#1 Define a record type called Person

#2 Create an instance of Person called author

#3 Assign one of the fields of a record to another identifier

#4 The type of a Person, FSI result from #1

#5 The type of author is inferred to be Person and the values are shown

#6 The return value for #3 is the integer value of 28

The first line #1 shows the creating of the record type Person, and #4 shows the return value of Person. The second is an instance of a person and #5 shows the value of the identifier author. When accessing one of the parts of the type, we use the name associated as shown in #3. You can see in #6 that the value of author.Age was returned.

Discriminated unions are sort of like enumerations. They allow for definition of types that are mutually exclusive. These types can have methods of their own. Look at the example below. Line #1 shows a discriminated union with different levels for customers. The Plus #2 customers have data associated with them, possibly the level as well as the number of years of loyalty. The keyword with (#3) is followed by the definition of a member, or method associated with this type #4.

Listing 1.# Discriminated union

```

type CustomerLevel =                                     #1
    | Business
    | Personal
    | Plus of CustomerLevel * int                         #2
with                                                       #3
    member x.calculateDiscount = getDiscount x          #4

```

#1 The definition of a discriminated union

#2 A Plus type carries two pieces of data, one of which is a CustomerLevel

#3 Keyword with defines additional functionality of the type

#4 Members define the functions of the discriminated union

Now that we've seen some of what sets F# apart from other .NET languages we can take a deeper look at where it fits into the ecosystem. We will discuss some of the places where F# is a natural fit.

1.4 F#'s place in today's real world solutions

With all of the other languages available today, it is important to know where F# fits in the .NET ecosystem. If a language has no defined purpose, then it is in fact useless. It is important to use the right tool for the job, and to do so, we must first evaluate what jobs are right for F#. As a general rule, you should use F# when using C# feels like a hard fit. If you feel that you have to do a lot of extra work to make the other languages solve the problem, F# may be worth looking at. This will probably seem extremely vague. In this section, and indeed the rest of the book, I will attempt to provide some examples of where FP seems to be a good fit. It will become more clear as you start to follow some of the examples in the book and especially as you understand the concepts in chapters 2 and 3.

1.4.1 Domain Specific Languages

A domain specific language is a programming language that is pointedly written to be used for a specific, limited set of tasks. DSLs give you the ability to create the language that works best for the task you need to complete. It has specific task oriented syntax with very limited expressivity. This can be achieved in many ways. F# provides some interesting tools for creating domain specific languages. It has a concise, flexible syntax that can be molded in ways that make it easier to create a comfortable syntax. This might mean that the language only makes sense to people who can speak the jargon of the domain, or task at hand. It also has some interesting meta-programming techniques as well as a lexer and a parser (fslex and fsparse). We'll go in depth with the concepts and discuss an example application in chapter 12.

1.4.2 Services

WCF, or Windows Communication Foundation is a part of the .NET 3.0 stack that allows for easier communication between service oriented systems. A Service Oriented Architecture approach has proven to be valuable in a distributed systems environment. Systems are able to interact with each other via asynchronous, untyped message-passing primitives.

If you consider each service has inputs and outputs but cannot have side effects, you might immediately think of functions. They have an input, perform some operation or service, and produce an output, while not affecting the requesting party's view of the data. This is a larger scale of a function which makes functional programming a natural selection. We'll discuss working with distributed systems and building services in chapter 13.

1.4.3 Reasoning about large amounts of data

As technology continues to grow, so does the amount of data that we need to process and store. We need to be able to reason about massive amounts of data and respond to information gained. This could mean dealing with scientific data like DNA sequences, or infinite sequences created by applying some function repeatedly to the previous item. This might also mean making trading decisions based on both historical and current market conditions, or simulating what might happen when making changes to a products pricing structure. Even seemingly simplistic types of data, like social networking site status messages can be collected and reasoned about it meaningful ways. Take for instance, Twitter. The website lists a group of terms that are the most commonly occurring during a given period of time. This requires analysis of all of the statuses being updated at the given time. The challenges presented are quite interesting. We'll spend some time in chapter 9 figuring out how to handle large quantities of data so that we can analyze in some meaningful way.

In chapter 14 we will use this information to create visualizations in F#. The information gathered using techniques in chapter 9 can be much more valuable with the ability to quickly view it and make some decision. Often the information is not valuable to managers and nontechnical people without a picture. We'll discuss DirectX as well as the capabilities of F#

when combined with WPF, Microsoft's Windows Presentation Foundation. WPF is a foundation for building graphical user interfaces with rich interactive experiences. It provides a rich set of tools that can be easily manipulated with underlying F# algorithms.

1.4.4 Rules Engines

Chapter 15 will discuss rules engines. Rules engines are systems that execute and apply business rules in order to make some decision. F# is a good tool for both creating rules and the execution engine.

The extremely powerful pattern matching available with F#, along with the simplicity of asynchronous workflows and lazy evaluation allows for easy description and evaluation of these rules. Let's quickly look at some code that could represent a rule.

Listing 1.# Pattern matching and a Discriminated Union.

```

type CustomerLevel =                                     #1
    | Business
    | Personal
    | Plus of CustomerLevel * int                       #2

let discount = function                                 #3
    | Business -> 10                                    #4
    | Personal -> 15
    | Plus(Business,yrs) when yrs > 10 -> 30           #5
    | Plus(Business,yrs) when yrs > 5 -> 25
    | Plus(Personal,yrs) when yrs > 10-> 30
    | Plus(_,yrs) -> yrs + 1                           #6

```

#1 We define a discriminated union type called CustomerLevel

#2 One of the types is named Plus, and carries 2 pieces of data with it. One is of type CustomerLevel (recursive) and the other an int.

#3 A pattern matching function called discount

#4 Will evaluate different CustomerLevels for a match and return what is after the arrow

#5 When the type has data with it, the function can also match on the data. Here we use a when clause to further specify a match

Some of the code in the above listing might not be immediately familiar, but don't worry. We will take a deeper look at F# syntax in the next section, before we get to the larger samples. The first line #1 defines a type called CustomerLevel. This is an example of a discriminated union, or a list of closely related, mutually exclusive types. Discriminated unions can often be used in the place of simple inheritance. You could define a level type and then the types Business, Personal, and Plus, which would be child types. Notice that the Plus type has some data associated with it #2. The Plus type contains a CustomerLevel as well as some integer

carried along with it. This might be that the customer is a Business level customer and the integer may be the number of years that the customer has done business with us, say 10. In this case our CustomerLevel would be (Business,10).

The next important thing to notice is #3, the definition of our discount function. Here discount is the identifier that will hold the value of the function that follows. This function is an example of a pattern match that matches the argument passed in with the lines below. This is similar to a switch statement in C#. #4 shows the first possible match. If the pattern is matched, then the return value of the function will be what is after the arrow (->). Each pattern is tested until a match is found and a return occurs.

The Plus type is evaluated for the first time in #5. The data associated with the Plus type is now matched. If the type is Business, and a number is present, the when clause will be evaluated. When clauses make the pattern matching construct even more powerful. If a match is found, the associated when clause will be evaluated, and only when it returns true will the value after the -> be returned.

#6 will be the default for the Plus type. No matter what the data associated with Plus is, this case will be matched. The "_" character is like a wildcard. Notice in this case, the return value is an expression. Pattern matches are used as a way of directing code. This means that any expression with the same return type as the rest of the function can be used here. This might have been a function call that returned an integer, just the same.

We will take a much deeper look at pattern matching in chapter 3, but this will give a hint to the power of this construct and hopefully an idea that they will be quite useful in a rules engine implementation.

1.5 F# is not always the answer you are looking for

Every programming language, or in fact class of languages has a group of zealots. These people follow the language as if it were a religion. If you ask them, their preferred language is the solution to every problem. It is hard for them to see the shortcomings or disadvantages of using the language in every situation. That is not the type of language that F# is. As I mentioned earlier, it is a tool that is very good at what it does, but does not intend to take over the development world. Functional programming or even multiple paradigm development is not a golden hammer. It is not well suited for every type of problem in software development. Industry leaders have put forth much energy trying to perfect the art of object oriented development. There are plenty of cases where OO works in a logical clean way to abstract certain problem domains. If you can map a problem with OO easily, there may not be a whole lot of benefit in the extra effort to convert to FP.

Graphical User Interfaces using Windows Forms might seem like an inherently OO problem. Microsoft has not yet created a UI designer for F#. This is not the problem that they are trying to resolve as it is not a place where F# can add a lot of additional value. FP is however perfect for the events that flow from an event driven UI. I tend to use C# for the graphics and F# for the events behind them.

Many industry experts have designed code generation tools that can be very valuable in software development. Most of these tools continue to work alongside F#. It can be easier to use these tools that generate code in other .NET languages and can interoperate with the F# that you write to pull the solution together. I imagine there will be plenty of F# generating tools in the future, but for now, I will use the C# gen tools.

Most simple crud apps can be built easily or generated from a tool that creates C# code. This is probably the best place for that work to be done, unless there are going to be operations performed on the data.

As F# matures, the domains for its use will become clearer. Some might argue that the listed areas are fine places to use F#. There is not a hard and fast rule. Use the tool that feels comfortable.

1.6 Summary

In this chapter we have seen a little bit about what F# is and why existing languages might not be able to solve problems in the same way. We've discussed the different paradigms that set the basic ideas behind the language. We learned some of the ideas behind functional programming, like immutable constructs, higher order functions, and the ability to determine when code should be evaluated called lazy evaluation. We have also looked at a new way of directing code with pattern matching.

We have begun to scratch the surface of what F# is and in what ways it can be useful. F# is great for dealing with large amounts of data as well as dealing with complex computations or working with algorithms. We discussed some of the types of examples that we will dig into in later chapters such as domain specific languages, services, data visualizations and rules engines. We have also looked at a couple of places that F# is not really going to be the right tool for the job such as in graphical user interface design.

In the next chapter and throughout the rest of section 1 we will dive into the how-to of F#. We'll look at the development environments and different ways to interact with the code. Next of course we'll discuss the obligatory hello world application and move on from there to syntax that we will be the essential building blocks for the rest of the book, and in fact the F# language.