Introduction to the F# Programming Language

What is F# and Why Should I Learn It?

F# is a functional programming language built on .NET. Just like C# and VB.NET, F# can take advantage of core libraries such as Windows Presentation Foundation, Windows Communication Foundation, Visual Studio Tools for Office, etc. With F# you can even write XBox game using XNA.

But just because you can write code in a new language does not mean you should. So why use F#? Because being a functional language, F# makes writing some classes of programs much easier than its imperative cousins like C#. Parallel programming and language-oriented programming are two such domains that can be expressed easily in F#.

If you have ever written a .NET application and found yourself fighting against the language to get your idea expressed, then perhaps F# is what you have been looking for.

Language Basics

`#light` (OCaml Compatibility)

F# has its roots in a programming language called OCaml and has the ability to cross-compile OCaml code, meaning that it can compile simple OCaml programs unmodified. This ability however, means that F# requires some unsavory syntax by default. `#light` (pronounced "hash light") is a compiler directive that simplifies the syntax of the language.

It is highly recommended to keep `#light` on since most F# code snippets you find will either declare it, or assume that it has been declared.

let square x = x * x (Type Inference)

This defines a function called `square` which squares a number `x`. Consider for a moment the equivalent C# code.

```
public static int square( int n )
{
  return x * x;
}
```

Whereas C# requires you to specify type information as well as what the function actually returns, the F# compiler figures it out for the programmer. This is referred to as type inference.

From the function signature F# knows that square takes a single parameter named `x` and that the function would return `x * x`. The last thing evaluated in a function body is the "return value", so no need for a "return" keyword. Since many primitive types support `(*)` such as `byte`, `uint64`, `double`, etc. F# defaults to `int`, a signed 32-bit integer.

Now consider the following code which provides a "type annotation" for one of the parameters, that is telling the compiler the type to expect. Since x is stated to be of type `string`, and `(+)` is only defined for taking two strings, then the parameter `y` must also be a `string`. ANd the result of `x + y` is the concatenation of both strings.

```
> let concat (x: string) y = x + y;;
val concat : string -> string -> string

> concat "Hello, " "World!";;
val it : string = "Hello, World!"
```

let numbers = [1 .. 10] (F# Lists)

The next line declares a list of numbers one through ten. If you had typed `[|1 .. 10|]` that would have created a .NET array of integers. But an F# list is an immutable linked list, which is the backbone of functional programming.

```
// Define a list
let vowels = ['e'; 'i'; 'o'; 'u']

// Attach item to front (cons)
let cons = 'a' :: vowels

// Concat two lists
let sometimes = vowels @ ['y']
```

let squares = List.map square numbers (First-Order Functions)

Now we have a list of integers (`numbers`) and a function (`square`), we want to create a new list where each item is the result of calling our function. In other words, mapping our function to each item in the list.

Fortunately, `List.map` does just that. Consider another example:

```
> List.map (fun x -> x % 2 == 0) [1 .. 10];;
val it : bool list
= [false; true; false; true; false; true; false; true; false; true]
```

The code `(fun x -> x % 2 == 0)` defines an anonymous function, called a lambda expression, that has a parameter `x` and the function returns the result of "x % 2 = 0", which is whether or not `x` is even.

Now notice what we just did — we passed a function as a parameter to another function. You simply cannot do that in C#. But in F# it allowed us to be very expressive and succint in our code. Passing around functions as values is known as "first-order functions" and is a hallmark of functional programming.

printfn "N^2 = %A" squares

Function `printf` is a simple and type-safe way to print text to the console window. To get a better feel for how `printf` works, consider this example which prints an integer, floating-point number, and a string:

```
> prinfn "%d %f= %s" 5 0.75 (5.0 * 0.75).ToString());;
5 * 0.7500000 = 3.75
val it : unit = ()
```

The `%d`, `%f`, and `%s` are holes for integers, floats, strings. `%A` may also be used to print anything else.

Console.ReadKey(true) (.NET Interop)

Since F# is built on top of .NET, you can call any .NET library from F# — from Regular Expressions to WinForms. Namespace can be opened by the `open` keyword and brings its types into scope, similar to the `using` keyword of C#.

Modules

Though F# can declare the standard .NET classes you are familiar with, it also has the notion of a module, which a collection of values, functions, and types. In contrast to a namespace, which can only contain types.

This is how we are able to access `List.map`. In the F# library (`FSharp.Core.dll`) there is a module named `List` and on that is a function named `map`.

Modules serve as a way of encapsulating code when you are rapid prototyping without needing to spend the time to design a strict object-oriented type hierarchy. To declare your own module, you can use the `module` keyword. In the example, we will associate a mutable variable (explained later) with the module, which will serve as a global variable.

```
module ProgramSettings =
  let version = "1.0.0.0"
  let debugMode = ref false

module MyProgram =
  do printfn "Version %s" ProgramSettings.version

  // You can also open modules like you can a namespace, which
  // brings all their functions and values into scope
  open ProgramSettings
  debugMode := true
```

Tuples

A tuple (pronounced "two-pull") is an ordered collection of values treated like an atomic unit. Traditionally if you wantd to pass around a group of semi-related values you would need to create a struct or class, perhaps rely on "out" parameters. A tuple allows you to keep things organized by grouping related values together without introducing a new type.

To define a type, simply enclose the group of values in parentheses separate the individual components by commas.

```
> let tuple = (1, false, "text");;
val tuple : int * bool * string

> let getNumberInfo (x: int) = (x, x.ToString(), x * x);;
val getNumberInfo : int -> int * string * int

> getNumberInfo 42;;
val it : int * string * int = (42, "42", 1764)
```

Functions can even take tuples as arguments.

```
> let printBlogInfo (owner, title, url)
    = printfn "%s's blog [%s] is online at '%s'"
      owner title url;;
val printBlogInfo : string * string * string -> unit
```

Function Currying

A novel feature F# provides is the ability to provide a subset of a function's parameters, and bind a new value to the partial application. This is known as function currying. For example, consider a function which takes three integers and returns their sum. We can curry that function, by fixing the first parameter to be 10, resulting in a function that only takes two integers and returns their sum plus 10.

```
> let addThree x y z = x + y + z;;
val addThress : int -> int -> int -> int

> let addTwo x y = addThree 10 x y;;
val addTwo : int -> int -> int

> addTwo 1 1;;
val it : int = 12
```

Pattern Matching

Pattern matching is like a powerful switch statement, allowing you to branch control flow. You can do more than just comparing a value against a constant however, pattern matching allows you to also capture new values.

```
let printGreeting (emp: MicrosoftEmployee) =
  match emp with
    | BillGates     -> printfn "Hello, Bill"
    | SteveBallmer  -> printfn "Hello, Steve"
    | Worker(name)
    | Lead(name, _) -> printfn "Hello, %s" name
```

Pattern matching can also match against the "structure" of the data. So we can match against list elements joined together.

```
let listLength aList =
  match aList with
```

```
    | [] -> 0
    | a :: [] -> 1
    | a :: b :: [] -> 2
    | a :: b :: c :: [] -> 3
    | _ -> failwith "List is too big!"
```

At the end of the pattern match we had a wildcard `_`, which matches anything. So if variable `aList` had more than three items, the last pattern clause would execute and throw an exception. Pattern matching also allows you to execute an arbitrary expression to determine if the pattern is matched. (If that expression evaluates to false, then the clause is not matched.)

```
let isOdd x =
  match x with
    | _ when x % 2 = 0 -> false
    | _ when x % 2 = 1 -> true
```

You can even pattern match using a dynamic type test.

```
let getType (x : obj) =
  match x with
    | :? string    -> "x is a string"
    | :? int       -> "x is an int"
    | :? Exception -> "x is an exception"
```

Option Values (`Microsoft.FSharp.Core.Option<_>`)

It is difficult to find nulls in F# code since values are always initialized. However, there are times when a `null` value means more than an uninitialized variable. Sometimes it means the absence of a value. (Option values are similar to nullable types in C#.)

F# has an "option type" to represent two states: `Some` and `None`. In the following record type `Person`, the middle initial fields may or may not have a value.

```
type Person = { First : string; MI : string option; Last : string }
let billg = { First = "Bill"; MI = Some("H"); Last = "Gates" }
```

Lazy Values (`Microsoft.FSharp.Core.Lazy<_>`)

Lazy initialization is a term used to describe something which is computed as needed (and not right when it is declared). F# has lazy values, such as the following example where `x` is an integer, but as part of its evaluation prints "Computed" to the screen.

```
> let x = lazy (printfn "Computed."; 42);;
val x : Lazy<int>

> let listOfX = [x; x; x];;
val listOfX : Lazy<int> list

> x.Force();;
Computed.
val it : int = 42
```

You see that the value of `x` was computed when we called `Force` and the value of 42 was returned. You can use lazy initialization to avoid computing values which are not needed or if they are computationally expensive. In addition they are useful when constructing recursive values. For example, consider a representation for a circular linked list:

```
type InfiniteList =
  | ListNode of int * InfiniteList

let rec circularList = ListNode(1, circularList)
```

The value `circularList` has a reference to itself (representing an infinite loop of `ListNode`s with value 1). Without lazy initialization declaring this type of value would be impossible. But behind the scenes the compiler uses lazy initialization to make this work.

Some Common Immutable Data Structures

Data structures are divided between *mutable* and *immutable*. Immutable data structures are sometimes called *persistent* or *functional*.

- *Tuple values and option values:* These are immutable and are basic elements of F# programmings.
- *Immutable linked lists of type 'a list:* These are cheap to access from the left end. They are inefficient for random access lookup because the list must traversed from the left for each lookup, i.e. random access lookup is *O(n)* where n is the number of elements in the collection.
- *Immutable sets based on balanced trees:* Implementation of mmutable sets is provided via the type `Set<'a>` in the F# library. These are cheap to add, access, and union, with *O(log(n))* access times, where n is the number of elments in the collection. Because the type is immutable, internal nodes can be shared between different sets.

- *Immutable maps based on balanced trees:* These are similar to immutable sets but associate keys with values (i.e., they are immutable dictionaries). One implementation of these is provided via the F# library type `Map<'key,'value>` in `Microsoft.FSharp.Collections`. As with sets, these have *O(log(n))* access times.

Function Values

Function values are a foundational building block of F# functional programming. For example, they are used for using function values to transform one list into another.

```
let http(url: string) =
  let req = System.Net.WebRequest.Create(url)
  let resp = req.GetResponse()
  let stream = resp.GetResponseStream()
  let reader = new StreamReader(stream)
  let html = reader.ReadToEnd()
  resp.Close()
  html

let sites = [ "http://www.live.com"; "http://www.google.com" ]
let fetch url = (url, http url)
List.map fetch sites
```

One of the primary uses of F# lists is as a general-purpose concrete data structure for storing ordered list input and ordered rsults. Input lists are often transformed to output lists using "aggregate" operations that transform, select, filter, and categorize elements of the list according to a range of criteria.

Using Anonymous Function Values

Function values are common in F#, so it is convenient to define them without giving them names.

```
let primes = [2; 3; 5; 7]
let primeCubes = List.map (fun n -> n * n * n) primes
```

The definition of `primeCubes` uses an *anonymous function value*. These are similar to function definitions but are unnamed and appear as an expression rather than as a `let` declaration. The `fun` is a keyword meaning "function", `n` represents the argument to the function, and `n * n * n` is the result of the function. The overall type of the anonymous function expression is `int -> int`.

```
let resultsOfFetch = List.map (fun url -> (url, http url)) sites
List.map (fun (_,p) -> String.length p) resultsOfFetch
```

The argument of the anonymous function is a tuple pattern. Using a tuple pattern automatically extract the second element from each tuple and gives it the name `p` within the body of the anonymous function.

Computing with Aggregate Operators

Functions such as `List.map` are called *aggregate operators*, and they are powerful constructs, especially when combined with the other features of F#.

```
let delimiters = [ ' '; '\n'; '\t'; '<'; '>'; '=' ]
let getWords s = String.split delimiters s
let getStats site =
  let url = "http://" + site
  let html = http url
  let hwords = html |> getWords
  let hrefs = html |> getWords |> List.filter (fun s -> s = "href")
  (site,html.Length, hwords.Length, hrefs.Length)
```

Here we use the function `getStats` with three web pages. It computes the length of the HTML for the given website, the number of words in the text of that HTML, and the approximate number of links on that page.

```
let sites = [ "www.live.com"; "www.google.com"; "search.yahoo.com" ]
sites |> List.map getStats
```

The previous code sample extensively uses the `|>` operator to *pipeline* operations. It is called the "forward pipe" operator, and perhaps the most important operator in F# programming. Its definition is deceptively simple:

```
let (|>) x f = fx
```

Here is how to use the operator to compute the cubes of three numbers:

```
[1;2;3] |> List.map (fun x -> x * x * x)
```

It is just as if you had written this:

```
List.map (fun x -> x * x * x) [1;2;3]
```

In a sense, `|>` is just "function application in reverse". However, using `|>` has distinct advantages:

- *Clarity:* When used in conjunction with operators such as `List.map`, the `|>` operator allows you to peform the data transformations and iterations in a forward-chaining, pipelined style.
- *Type inference:* Using the `|>` operator allows type information to tbe flowed from input object to the functions manipulating those objects. F# uses information from type inference to resolve some language constructs such as property accesses and method overloading. This relies on information being propagated left to right through the text of a program. In particular, typing information to the right of a position is not taken into account when resolving property access and overloads.

Recurring Aggregate Operator Design Pattern from the F# Library:

```
List.map    : ('a -> 'b) -> 'a list   -> 'b list
Array.map   : ('a -> 'b) -> 'a[]      -> 'b[]
Option.map  : ('a -> 'b) -> 'a option -> 'b option
Seq.map     : ('a -> 'b) -> #seq<'a>  -> seq<'b>
```

Types will aso define methods such as `Map` that provide a slightly more succint way of transforming data.

```
sites.Map(getStats)
```

Composing Functions with >>

Beside the process of *computing with functions*, there is an essential and powerful programming technique in F# to compute new function values from existing ones using compositional techniques.

```
let google = http "http://www.google.com"
google
  |> getWords
  |> List.filter (fun s -> s = "href")
  |> List.length
```

This code can be rewritten using function composition as follows:

```
let countLinks = getWords
  >> List.filter (fun s -> s = "href")
  >> List.length
google |> countLinks
```

The `countLinks` has been defined as the composition of three function values usnig the `>>` "forward composition" operator. This operator is defined in the F# Library as follows:

```
let (>>) f g x = g(f(x))
```

The expression `f >> g` gives a function value that first applies `f` to the `x` then applies `g`.

Note that `>>` is typically applied to only two arguments — those on either side of the binary operator, here named `f` and `g`. The final argument `x` is typically supplied at a later point. F# is good at optimizing basic constructions of pipelines and composition sequences from functions — for example, the function `countLinks` will become a single function calling the three functions in the pipeline in sequence.

Building Functions with Partial Application

Composing functions is just one way to compute interesting new functions. Another useful way is by using *partial application*.

```
let shift (dx,dy) (px,py) = (px + dx, py + dy)
let shiftRight = shift (1,0)
let shiftUp    = shift (0,1)
let shiftLeft  = shift (-1,0)
let shiftDown  = shift (0,-1)
```

The last four functions have been defined by calling `shift` with only one argument, in each case leaving a residue function that expects an additional argument.

```
shiftRight (10,10)
List.map (shift (2,2)) [(0,0); (1,0); (1,1); (0,1)]
```

In the second line, the function `shift` takes two pairs as arguments. The results of this partial application is a function that takes one remaining tuple parameter and returns the value shifted by two units in each direction. This resulting function can now be used in conjunction with `List.map`.

Using Local Functions

Partial application becomes very powerful when combined with additional local definitions.

```
open System.Drawing;;
let remap (r1: Rectangle) (r2: Rectangle) =
  let scalex = float r2.Width / float r1.Width
  let scaley = float r2.Height / float r1.Height
  let mapx x = r2.Left + truncate (float (x - r1.Left) * scalex)
  let mapy y = r2.Top + truncate (float (y - r1.Top) * scaley)
  let mapp (p: Point) = Point(mapx p.X, mapy p.Y)
  mapp
```

The function `remap` computes a new function value `mapp` that maps points in one rectangle to points in another. The type `Rectangle` is defined in the .NET library `System.Drawing.dll` and represents rectangles specified by integer coordinates. The computations on the interior of the transformation are performed in floating point to improve precision.

In the previous sample, `mapx`, `mapy`, and `mapp` are *local functions*, i.e. functions defined locally as part of the implementation of `remap`. Local functions can be context dependent, so they can be defined in terms of any values and parameters that happen to be in scope. Here `mapx` is defined in terms of `scalex`, `scaley`, `r1`, and `r2`.

Local and partially applied functions are, if necessary, implemented by taking the *closure* of the variables they depend upon and storing them away until needed. In optimized F# code, the F# compiler often avoids this and instead passes extra arguments to the function implementations.

Using Functions As Abstract Values

The function `remap` generates values of type `Point -> Point`. It can be thought of this type as a way of representing "the familty of transformations for the type `Point`". Many useful concepts can be modeled using function types and values.

- The type `unit -> unit` is used to model "actions", i.e. operations that run and perform some unspecified side effect.
- Type of the form `type -> type -> int` are used to model "comparison functions" over the type `type`. Type `type * type -> int` is also used for this purpose, where a tuple is accepted as the first argument. Many collection types will accept such a value as a configuration parameter.
- Type of the form `type -> unit` are used to model *callbacks*. Callbacks are often run in response to a system event, such as when a user clicks a user interface element.
- Type of the form `unit -> type` are used to model *delayed computations*, which are values that will, when required, produced a value of type *type*.

Delayed computations are related to lazy computations and sequence expressions.

- Types of the form `type -> unit` are used to model *sinks*, which are values that will, when required, consume a value of type *type*.
- Types of the form `type_1 -> type_2` are used to model *transformers*, which are functions that transform each element of a collection.
- Types of the form `type_1 -> type_2 -> type_3` are used to model *visitor accumulating functions*, which are functions that visit each element of a collection (type `type_1`) and accumulate a result (type `type_2`).

Iterating with Aggregate Operators

It is common to use data to drive control, and indeed in functional programming the distinction between data and control is often blurred: function values can be used as data, and data can influence control flow. An example is using a function such a `List.iter` to iterate over a list.

```
let sites = [ "http://www.live.com"
            ; "http://www.google.com"
            ; "http://search.yahoo.com"]
sites
  |> List.iter
      (fun site ->
        printfn "%s, length = %d" site (http site).Length)
```

Function `List.iter` simply calls the given (anonymous) function for each element in the input list.

Many additional aggregate iteration techniques are defined in the F# and .NET libraries, particularly by using values of type `seq<type>`.

Abstracting Control with Functions

Let us consider the common pattern of timing the executiong of an operation by using `System.DateTime.Now`.

```
open System
let start = DateTime.Now
http "http://www.newscientist.com"
let finish = DateTime.Now
let elapsed = finish - start
elapsed
```

Not the type `TimeSpan` has been inferred from the use of the overloaded operator in the expression `finish - start`. The presented technique can be wrapped up now as a function `time` that acts as a new control operator:

```
open System
let time f =
  let start = DateTime.Now
  let res = f()
  let finish = DateTime.Npw
  (res, finish - start)
```

This function runs the input function `f` but takes the time on either side of the call. It then returns both the result of the function and elapsed time. The inferred type is as follows:

```
time : (unit -> 'a) -> 'a * TimeSpan
```

Note that F# has automatically inferred a generic type for the function, a technique called *automatic generalization* that lies at the heart of F# programming.

An example of using the `time` function:

```
time (fun () -> http "http://www.newscientist.com")
```

Using .NET Methods As First-Class Functions

Existing .NET methods can be used as first-class functions, for example:

```
open System.IO
[@"C:\Program Files"; @"C:\Windows"]
  |> List.map Directory.GetDirectories
```

Note that sometimes extra type is needed to be added to indicate which "overload" of the methods is required.

```
let f = (Console.WriteLine : string -> unit)
```

Getting Started with Sequences

Many programming tasks require the iteration, aggregation, and transformation of data streamed from various sources. One important and general way to code these tasks is in terms of values of the .NET type `IEnumerable<type>`

(`System.Collections.Generic`), which is typically abbreviated to `seq<type>` in F# code. A `seq<type>` is simply a value that can be *iterated*, producing results of type *type* on demand. Sequences are used to wrap collections, computations, and data streams and are used to represent the results of database queries.

Using Range Expressions

Simple sequences can be generated using *range expressions*. For integer ranges, these take the form of `seq {n .. m}` for the integer expressions `n` and `m`.

```
seq {0 .. 2}
```

Range expressions using other numeric types such as `double` and `single` can be also specified:

```
seq {-100.0 .. 100.0}
```

Values of type `seq<'a>` are *lazy* in the sense that they compute and return the successive elements on demand. Therefore sequences representing very large range can be created, and the elements of the sequence are computed only if they are required by a subsequent computation. In other words, data structures containing one trillion elements are not created, but rather sequences that have the *potential* to yield this number of elements on demand.

```
seq {1I .. 10000000000000I}
```

The default incrment for range expression is always 1. A different increment can be used via range expressions of the form `seq { n .. skip .. m }`.

Iterating a Sequence

Sequences can be iterated using the `for ... in ... do` construct, as well as the `Seq.iter` aggregate operator.

```
let range = seq {0 .. 2 .. 6}
for i in range do
  printfn "i = %d" i
```

This construct forces the iteration of the entire `seq` so must be used with care when working with sequences that may yield a large number of elements.

Transforming Sequences with Aggregate Operators

Any value of type `seq<type>` can be iterated and transformed using functions in the `Microsoft.FSharp.Collections.Seq` module.

```
let range = seq {0 .. 10}
range |> Seq.map (fun i -> (i, i * i))
```

The following operators necessarily evalute all the elements of the input `seq` immediately:

- `Seq.iter`: This iterates all elements, applying a function to each one.
- `Seq.to_list`: This iterates all elements, building a new list.
- `Seq.to_array`: This iterates all elements, building a new array.

Most of the other operators in the `Seq` module return one or more `seq<type>` values and force the computation of elements in any input `seq<type>` values only on demand.

```
Seq.append : #seq<'a> -> #seq<'a> -> seq<'a>
Seq.concat : #seq< #seq<'a> >  -> seq<'a>
Seq.choose : ('a -> 'b option) -> #seq<'a> -> seq<'b>
Seq.delay  : (unit -> #seq<'a>) -> seq<'a>
Seq.empty  seq<'a>
Seq.iter   : ('a -> unit) -> #seq<'a> -> unit
Seq.filter : ('a -> bool) -> #seq<'a> -> seq<'a>
Seq.map       : ('a -> 'b) -> #seq<'a> -> seq<'b>
Seq.singleton : 'a -> seq<'a>
Seq.truncate  : int -> #seq<'a> -> seq<'a>
Seq.to_list   : #seq<'a> -> 'a list
Seq.of_list   : 'a list -> seq<'a>
Seq.to_array  : #seq<'a> -> 'a[]
Seq.of_array  : 'a[] -> seq<'a>
```

Types prefixed with `#`, such as `#seq<'a>`. This means the function will accept any type that is compatible with (i.e. a subtype of) `seq<'a>`. Here are some of these types:

- *Array types:* For example, `int[]` is compatible with 'seq.
- *F# list types:* For example, `int list` is compatible with `seq<int>`.
- *All other F# and .NET collection types:* For example, `SortedList<string>` (`System.Collections.Generic`) is compatible with `seq<string>`.

There are some types that are not directly type compatible with `seq<'a>` but can be converted into sequences on demand.

Using Lazy Sequences from External Sources

Sequences are used to represent the process of streaming data from an external source, such as from a database query or from a computer's file system. For example, the following recursive function constructs a `seq<string>` that represents the process of recursively reading the names of all the files under a given path. The return types of `Directory.GetFiles` and `Directory.GetDirectories` are `string[]`, and as noted earlier, this type is always compatbile with `seq<string>`.

```
open System.IO
let rec allFiles dir =
  Seq.append
    (dir |> Directory.GetFiles)
    (dir
      |> Directory.GetDirectories
      |> Seq.map allFiles
      |> Seq.concat)
```

The `allFiles` function shows many aspects of F# working seamlessly together:

- *Functions are values:* The function `allFiles` is recursive and is used as a first-class function values within its own definition.
- *Pipelining:* The pipelining operator `|>` provides a natural way to present the transformations applied to each subdirectory name.
- *Type inference:* Type inference computes all types in the obvious way, without any type annotations.
- *.NET interoperability:* The `System.IO.Directory` operations provide intuitive primitives, which can then be incorporated in powerful ways using succint F# programs.
- *Laziness where needed:* The function `Seq.map` applies the argument function lazily (on demand), which means subdirectories will not be read until really required.

Note that side effects such as reading and writing from an external store should not in general happen until the lazy sequence value is actually consumed.

Using Sequence Expressions

Aggregate operators are a powerful way of working with `seq<type>` values. However, F# also provides a convenient and compact syntax called *sequence*

expressions for specifying sequence values that could be built using operations such as `choose`, `map`, `filter`, and `concat`. Sequence expressions can also be used to specify the shapes of lists and arrays.

- They are a compact way of specifying interesting data and generative processes.
- They are used to specify database queries when using data access layers such as Microsoft's Language Integrated Queries (LINQ).
- They are one particular use of *workflows*, a more general concept that has several uses in F# programming.

Creating Sequence Expressions Using `for`

The simplest form of a sequence expression is as follows:

```
seq { for value in expr .. expr -> expr }
```

Here `->` should be read as "yield". This is a shorthand way of writing `Seq.map` over a range expression.

```
let squares = seq { for i in 0 .. 10 -> (i, i * i) }
```

The more complete form of this construct is the following:

```
seq { for pattern in seq -> expression }
```

The pattern allows to decompose the values yielded by the input enumerable.

```
seq { for (i, isquared) in squares -> (i, isquared, i * isquared) }
```

The input `seq` can be a `seq<type>` or any type supporting a `GetEnumerator` method. Note that some important types from the .NET libraries support this method without directly supporting the `seq` interface.

```
seq { for Some(nm) in [ Some("James"); None; Some("John") ]
        -> nm.Length }
```

Enriching Sequence Expressions with Additional Classes

A sequence expression generally always begins with `for ... in ...`, but additional constructs can be used.

- *A secondary iteration:* `for pattern in seq do seq-expr`
- *A filter:* `if expr then seq-expr`
- *A conditional:* `if expr then seq-expr else seq-expr`
- *A let binding:* `let pattern = expr in seq-expr`
- *A final yield:* `-> expr or yield expr`
- *A final yield of another sequence:* `->> expr or yield! expr`

Secondary iterations generate additional sequences, all of which are collected and concatenated together. Filters allow to skip elements that do not satisfy a given predicate.

```
let checkerboardCoordinates n =
  seq { for row in 1 .. n do
            for col in 1 .. n do
                if (row + col) % 2 = 0 then
                    yield (row,col) }
```

Using `let` clauses in sequence expressions allows to compute intermediary results.

```
let fileInfo dir =
  seq { for file in Directory.GetFiles(dir) do
            let creationTime = File.GetCreationTime(file)
            let lastAccessTime = File.GetLastAccessTime(file)
            yield (file,creationTime,lastAccessTime) }
```

The final yield of a sequence expression can also be another sequence, signified through the use of the `->>` symbol or the `yield!` keyword. Symbols `->` and `->>` can be used in compact sequence expressions that do not contain `let`, `if`, and other more advanced constructs. Note that multiple generators can be included in one sequence expression.

```
let rec allFiles dir =
  seq { for file in Directory.GetFiles(dir) -> file
        for subdir in Directory.GetDirectories dir
          ->> (allFiles subdir) }
```

Note that when using the `#light` syntax option, the `do` and `in` tokens can be omitted when immediately followed by a `->` or `->>` or when part of a sequence of `for` or `let` bindings.

Enriching Sequence Expressions to Specify Lists and Arrays

Range and Sequence expressions can be also used to build list and array values. The syntax is identical except the surrounding braces are replaced by the usual `[ ]` for lists and `[| |]` for arrays.

```
[ 1 .. 4 ]
[ for i in 0 .. 3 -> (i, i * i) ]
[| for i in 0 .. 3 -> (i, i * i) |]
```

F# lists and arrays are finite data structures built immediately rather than on demand, so care must be taken that length of the sequence is suitable.

Exploring Some Simple Type Definitions

F# is a typed language, and it is often necessary for the programmer to declare new "shapes" of types via type definitions and type abbreviations. F# also lets programmer define a range of sophisticated type definition related to object-oriented programming, however, these are often not required in basic functional programming.

Defining Type Abbreviations

Type abbrevations are the simplest type definitions:

```
type index = int
type flags = int64
type results = string * TimeSpan * int * int
```

Type abbreviations can be generic:

```
type StringMap<'a> = Microsoft.FSharp.Collections.Map<string,'a>
type Projectsion<'a,'b> = ('a -> 'b) * ('b -> 'a)
```

Type abbrevations are not concrete, because they alias an existing type. Type abbrevations are expanded during the process of compiling F# code to the format shared between multiple .NET languages. The difference can be detected by other .NET languages, and because of this, a number of restrictions apply to type abbreviations.

Defining Records

The simplest concrete type definitions are records.

```
type Person =
  { Name: string;
    DateOfBirth: System.DateTime; }
```

Record values can be constructed by using the record labels.

```
{ Name = "Bill"; DateOfBirth = new System.DateTime(1962,09,02) }
```

Record values can be constructed by using the following more explicit syntax.

```
{ new Person
  with Name = "Anna"
  and DateOfBirth = new System.DateTime(1968,07,23) }
```

Record values are often used to return results from functions.

```
type PageStats =
    { Site: string;
      Time: System.TimeSpan;
      Length: int;
      NumWords: int;
      NumHRefs: int }
```

This technique works well when returning a large number of heterogeneous results:

```
let stats site =
  let url = "http://" + site
  let html,t = time (fun () -> http url)
  let hwords = html |> getWords
  let hrefs = hWords |> List.filter (fun s -> s = "href")
  { Site=site; Time=t; Length=html.Length;
    NumWords=hwords.Length; NumHRefs=hrefs.Length }
```

Handling Non-Unique Records Field Names

```
type Person =
    { Name: string;
```

```
      DateOfBirth: System.DateTime; }

type Company =
    { Name: string;
      Address: string; }
```

When record names are non-unique, constructions of record values may need to use object expressions in order to indicate the name of the record type, thus disambiguating the construction.

```
type Dot = { X: int; Y: int }
type Point = { X: float; Y: float }
```

On lookup, record labels are accessed using the `.` notation in the same way as properties. One slight difference is that in the absence of further qualifying information, the type of the object being accessed is inferred from the record label.

Cloning Records

Records support a convenient syntax to clone all the values in the record, creating a new value, with some values replaced.

```
type Point3D = { X: float; Y: float; Z: float }
let p1 = { X = 3.0; Y = 4.0; Z = 5.0 }
let p2 = { p1 with Y = 0.0; Z = 0.0 }
```

The definition of `p2` is identical to this:

```
let p2 = { X = p1.X; Y = 0.0; Z = 0.0 }
```

This expression form does not mutate the values of a record, even if the fields of the original record are mutable.

Defining Discriminated Unions

The second kind of concrete type is a discriminated union.

```
type  Route = int
type  Make = string
type  Model = string
type  Transport =
    | Car of Make * Model
    | Bicycle
    | Bus of Route
```

Each alternative of a discriminated union is called a *discriminator*. Values can be built by using a discriminator much as if it were a function.

```
let nick = Car("BMW", "360")
let don = [ Bicycle; Bus 8 ]
let james = [ Car ("Ford","Fiesta"); Bicycle ]
```

Discriminators can be also used in pattern matching:

```
let averageSpeed (tr: Transport) =
  match tr with
    | Car _ -> 35
    | Bicycle -> 16
    | Bus _ -> 24
```

Discriminated unions can include recursive references (the same is true for records and other type definitions). This is used when representing structured languages via discriminated.

```
type Proposition =
    | True
    | And of Proposition * Proposition
    | Or of Proposition * Proposition
    | Not of Proposition
```

Recursive functions can be used to traverse such a type.

```
let rec eval (p: Proposition) =
    match p with
    | True -> true
    | And(p1,p2) -> eval p1 && eval p2
    | Or (p1,p2) -> eval p1 || eval p2
    | Not(p1) -> not (eval p1)
```

Discriminated unions are a powerful and important construct and are useful when modeling a finite, sealed set of choice. This makes them a perfect fit for many constructs that arise in applications and symbolic analysis libraries. They are, by design, non-extensible: subsequent modules cannot add new cases to a discriminated union.

Using Discriminated Unions As Records

Discriminated union types with only one data tag are an effective way to implement record-like types.

```
type Point3D = Vector3D of float * float * float
let origin = Vector3D(0.,0.,0.)
let unitX =  Vector3D(1.,0.,0.)
let unitY =  Vector3D(0.,1.,0.)
let unitZ =  Vector3D(0.,0.,1.)
```

They can be decomposed using succint patterns in the same way as tuple arguments.

```
let length (Vector3D(dx,dy,dz)) = sqrt(dx * dx + dy * dy + dz * dz)
```

This technique is most useful for record-like values where there is some natural order on the constituent elements of the value or where the elements have different types.

Defining Multiple Types Simultaneously

Multiple types can be declared together to give a mutually recursive collection of types, including record types, discriminated unions, and abbreviations. The type definitions must be separated by the keyword `and`.

```
type node =
    { Name : string;
      Links : link list }
and link =
    | Dangling
    | Link of node
```

Imperative Programming in F#

The functional programming paradigm is associated with "programming without side effects", called *pure* functional programming. In this paradigm, programs compute the results of a mathematical expression and do not cause any side effects, except perhaps reporting the result of the computation.

F# is not a "pure" functional language, for example, programmers can write programs that mutate data, perform I/O communications, start threads, and

raise exceptions. Furthermore, the F# type system does not enforce a strict distinction between expressions that perform these actions and expressions that do not.

Programming with side effects is called *imperative* programming. If your primary programming experience has been with an imperative language such as C, C#, or Java, you will initially find yourself using imperative constructs in F#. However, over time F# programmers generally learn how to perform many routine programming tasks within the side-effect-free subset of the language. F# programmers tend to use side effects in the following situations:

- Scripting and prototyping.
- Working with .NET library components that use side effects heavily, such as GUI libraries and I/O libraries.
- Initializing complex data structures.
- Using inherently imperative, efficient data structures such as hash tables and hash sets.
- Locally optimizing routines in a way that improves on the performance of the functional version of the routine.
- Working with very large data structures or in scenarios where the allocation of data structures must be minimized for performance reasons.

Some F# programmers do not use any imperative techniques at all except as part of the external wrapper for their programs. Programming with fewer side effects is attractive for many reasons. For example, eliminating unnecessary side effects nearly always reduces the complexity of the code, so it leads to fewer bugs. Another thing experienced functional programmers appreciate is that the programmer or compiler can easily adjust the order in which expressions are computed.

A lack of side effects also helps *reason* about the code: it is easier to visually check when two programs are equivalent, and it is easier to make quite radical adjustments to the code without introducing new bugs. Programs that are free from side effects can often be computed on demand where necessary, often by making very small, local changes to the code to introduce the use of delayed data structures. Finally, side effects such as mutation are difficult to use when data is accessed concurrently from multiple threads.

Imperative Looping and Iterating

Three looping constructs are available to help make writing iterative code with side effects simple:

- `for` loops.
- `while` loops.
- Sequence loops.

All three constructs are for writing imperative programs, indicated partly by the fact that in all cases the body of the loop must have a return type of `unit`. Note that `unit` is the F# type that corresponds to `void` in imperative languages such as C, and it has a single value `()`.

Using Mutable Records

The simplest mutable data structures in F# are mutable records. A record is mutable if one or more of its fields is labeled `mutable`. This means that record fields can be updated using the `<-` operator. Mutable fields are usually used for records that implement the internal state of objects.

```
type DiscreteEventCounter =
    { mutable Total: int;
      mutable Positive: int;
      Name : string }

let recordEvent (s: DiscreteEventCounter) isPositive =
    s.Total <- s.Total + 1
    if isPositive then s.Positive <- s.Positive + 1

let reportStatus (s: DiscreteEventCounter) =
    printfn "We have %d %s out of %d" s.Positive s.Name s.Total

let newCounter nm =
    { Total = 0;
      Positive = 0;
      Name = nm }

let longPageCounter = newCounter "long page(s)"
let fetch url =
    let page = http url
    recordEvent longPageCounter (page.Length > 10000)
    page
```

Every call to the function `fetch` mutatas the mutable record fields in the global variable `longPageCounter`.

```
fetch "http://www.smh.com.au" |> ignore
fetch "http://www.theage.com.au" |> ignore
reportStatus longPageCounter
```

Note that record types can also support members (e.g. properties and methods) and give implicit implementations of interfaces. This means it can be used as a way to implement object-oriented abstractions.

Mutable Reference Cells

One particularly useful mutable record is the general-purpose type of mutable reference cells, or *ref* cells for short. These often play much the same role as pointers in the other imperative programming languages.

The key type is `'a ref`, and its main operators are `ref`, `!`, and `:=`. The types of these operators are as follows:

```
val ref  : 'a -> 'a ref
val (:=) : 'a ref -> 'a -> unit
val (!)  : 'a ref -> 'a
```

These allocate a reference cell, read the cell, and mutate the cell, respectively. Both the `'a ref` type and its operations are defined in the F# library as a simple record data structure with a single mutable field.

```
type 'a ref = { mutable contents: 'a }
let (!) r = r.contents
let (:=) r v = r.contents <- v
let ref v = { contents = v }
```

The type `'a ref` is actually a synonym for a type `Microsoft.FSharp.Core.Ref<'a>` defined in this way.

Mutability of Data Structures

It is useful to know which data structures are mutable and which are not. If a data structure can be mutated, then this will typically be evident in the types of the operations can be performed on that structure. For example, if a data structure `Table<'Key,'Value>` has an operation like the following, then in practice updates to the data structure modify the data structure itself.

```
val add : Table<'Key,'Value> -> 'Key -> 'Value -> unit
```

The updats to the data structure are *destructive*, and no value is returned from the operation. The result is the type `unit`. The presence of `unit` as a return type is a sure sign that an operation performs some imperative side effects.

In contrast, operations on immutable data structures typically return a new instance of the data structure when an operation such as `add` is performed.

```
val add : 'Key -> 'Value -> Table<'Key,'Value> -> Table<'Key,'Value>
```

Immutable data structures are called *functional* or *persistent*, because the original table is not modified when adding an element. Well-crafted persistent data structures do not duplicate the actual memory used to store the data structure every time an addition is made. Instead, internal nodes can be shared between the old and new data structures. Most data structures in the .NET libraries are not persistent, though they can be used as persistent ones by accessing them in "read-only" mode and copying them where necessary.

Avoid Aliasing

Two values of type `'a ref` may refer to the same reference cell — this is called *aliasing*. Aliasing of immutable data structures is not a problem, however, aliasing of mutable data can lead to problems in understanding code. In general, it is a good practice to ensure that no two values currently in scope directly alias the same mutable data structures.

Hiding Mutable Data

Mutable data is often hidden behind an *encapsulation boundary*. For example, the following code shows how to hide a mutable reference within the inner closure of values referenced by a function value.

```
let generateStamp =
  let count = ref 0
  (fun () -> count := !count + 1; !count)
```

The line `let count = ref 0` is executed one, when the `generateStamp` function is defined.

This is a powerful technique for hiding and encapsulating mutable state without resorting to writing new type and class definitions. It is a good programming practice in polished code to ensure that all related items of mutable state are collected under some named data structure or other entity such as a function.

Using Mutable Locals

Mutable references must be explicitly dereferenced, but F# also supports mutable locals that are *implicitly* dereferenced. These must either be top-level definitions or be a local variable in a function.

```
let sum n m =
  let mutable res = 0
  for i = n to m do
    res <- res + i
  res
```

There are strong restrictions on the use of mutable locals. In particular, unlike mutable references, mutable locals are guaranteed to be stack-allocated values, which is important in some situations because the .NET garbage collector will not move stack values. As a result, mutable locals may not be used at all in any inner lambda expressions or other closure constructs, with the exception of top-level mutable values, which can be used anywhere, and mutable fields of records and objects, which are associated with the heap allocated objects themselves.

Exceptions and Controlling Them

When a routine encounters a problem, it may respond in several ways, such as by recovering internally, emitting a warning, returning a marker value or incomplete result, or throwing an exception.

```
let req = System.Net.WebRequest.Create("not a URL")
```

Similarly, the `GetResponse` method also used in the `http` function may raise a `System.Net.WebException` exception. The exceptions that may be raised by routines are typically recorded in the documentation for those routines in the documentation for those routines. Exception values may also be raised explicitly by F# code.

```
raise (System.InvalidException("not today thank you"))
```

Exceptions are commonly raised using the F# `failwith` function.

```
if false then 3 else failwith "hit the wall"
```

The types of some of the common functions used to raise exceptions:

```
val failwith : string -> 'a
val raise : #exn -> 'a
val failwithf : StringFormat<'a,'> -> 'a
val invalid_arg : string -> 'a
```

Note that the return types of all these are generic type variables, meaning that the functions never return "normally" and instead return by raising an exception. This means they can be used to form an expression of any particular type and indeed can be handy when drafting code.

Catching Exceptions

Exceptions can be caught using the `try ... with ...` language construct and `:?` type-test patterns, which filter any exception value caught by the `with` clause.

```
try
  raise (System.InvalidOperationException ("it's just not my day"))
with
  | :? System.InvalidOperationException -> printfn "caught!";;
```

Note that `try ... with ...` is just an expression, and it may return a result in more branches.

```
open System.IO
let http(url: string) =
  try
    let req = System.Net.WebRequest.Create(url)
    let resp = req.GetResponse()
    let stream = resp.GetResponseStream()
    let reader = new StreamReader(stream)
    let html = reader.ReadToEnd()
    html
  with
    | :? System.UriFormatException -> ""
    | :? System.Net.WebException -> ""
```

When an exception is thrown, a value is created that records information about the exception. It is this value that is being matched against the earlier type-test patterns. This value may also be bound directly and manipulated in the `with` clause of the `try ... with` constructs.

Using `try ... finally`

Exceptions may also be processed using the `try ... finally ...` construct. This guarantees to run the `finally` clause both when an exception is thrown and when the expression evalutates evaluates normally. This allows the programmer to ensure that resources are disposed after the completion of an operation.

```
let httpViaTryFinally(url: string) =
  let req = System.Net.WebRequest.Create(url)
  let resp = req.GetResponse()
  try
    let stream = resp.GetResponseStream()
    let reader = new StreamReader(stream)
    let html = reader.ReadToEnd()
    html
  finally
    resp.Close()
```

A shorter form to close and dispose of resource is to use a `use` binding instead of a `let` binding.

```
let httpViaUseBinding(url: string) =
  let req = System.Net.WebRequest.Create(url)
  use resp = req.GetResponse()
  let stream = resp.GetResponseStream()
  let reader = new StreamReader(stream)
  let html = reader.ReadToEnd()
  html
```

Defining New Exception Types

F# lets you define new kinds of exception objects that carry data in a conveniently accessible form.

```
exception BlockedURL of string
let http2 url =
  if url = "http://www.kaos.org"
  then raise(BlockedURL(url))
  else http url
```

The information from F# exception values can be extracted, using pattern matching:

```
try
  raise(BlockedURL("http://www.kaos.org"))
with
 | BlockedURL(url) -> printf "blocked! url = '%s'\n" url;;
```

Exception values are always subtypes of the F# type `exn`, an abbreviation for the .NET type `System.Exception`.

Having an Effect: Basic I/O

Imperative programming style and input/output are closely related topics.

Very Simple I/O: Reading and Writing Files

The .NET types `System.IO.File` and `System.IO.Directory` contain a number of simple functions to make working with files easily.

```
open System.IO
File.WriteAllLines("test.txt", [| "This is a test file.";
                                  "It is easy to read." |])
File.ReadAllLines("test.txt")
```

The results of `System.IO.File.ReadAllLines` can be also used as part of a list or sequence defined using a sequence expression.

```
[ for line in File.ReadAllLines("test.txt") do
     let words = line.Split [| ' ' |]
     if words.Length > 3 && words.[2] = "easy" then
        yield line ]
```

.NET I/O via Streams

The .NET namespace `System.IO` contains the primary .NET types for reading and writing bytes and text to or from data sources. The primary output constructs in this namespace:

- `System.IO.BinaryWriter`: Writes primitive data types as binary values. Create using `new BinaryWriter(stream)`. Create output streams using `File.Create(filename)`.
- `System.IO.StreamWriter`: Writes textual strings and characters to a stream. The text is encoded according to a particular Unicode encoding. Create by using `new StreamWriter(stream)` and its variants or by using `File.CreateText(filename)`.

- `System.IO.StringWriter`: Writes textual strings to a `StringBuilder`, which eventually can be used to generate a string.

These are the primary input constructs in the `System.IO` namespace:

- `System.IO.BinaryReader`: Reads primitive data types as binary values. When reading the binary data as a string, it interprets the bytes according to a particular Unicode encoding. Create using `new BinaryReader(stream)`.
- `System.IO.StreamReader`: Reads a stream as textual strings and characters. The bytes are decoded to strings according to a particular Unicode encoding. Create by using `new StreamReader(stream)` and its variants or by using `File.OpenText(filename)`.
- `System.IO.StringReader`: Reads a string as textual strings and characters.

Using `System.Console`

Some simple input/output routines are provided in the `System.Console` class.

```
System.Console.WriteLine("Hello World")
System.Console.Readline()
```

Working with null Values

The keyword `null` is used in imperative programming languages as a special, distinguished value of a type that represents an uninitialized value or some other kind of special condition. In general, `null` is not used in conjunction with types defined in F# code, though it is common to simulate `null` with a value of the `option` type.

```
let parents = [("Adam",None); ("Cain",Some("Adam","Eve"))]
```

However, reference types defined in other .NET languages do support `null`, and when using .NET APIs, one may have to explicitly pass `null` values to the API and also, where appropriate, test return values for `null`. The .NET Framework documentation specified when `null` may be returned from an API. It is recommended to test for this condition using `null` pattern tests.

```
match System.Environment.GetEnvironmentVariable("PATH") with
| null -> printf "the environment variable PATH is not defined\n"
| res -> printf "the environment variable PATH is set to %s\n" res
```

The following is a function that incorporates a pattern type test and a `null`-value test.

```
let switchOnType (a:obj) =
  match a with
  | null                      -> printf "null!"
  | :? System.Exception as e -> printf "An exception: %s!" e.Message
  | :? System.Int32 as i      -> printf "An integer: %d!" i
  | :? System.DateTime as d   -> printf "A date/time: %O!" d
  | _                         -> printf "Some other kind of object\n"
```

There are other important sources of `null` values. For example, the "semisafe" function `Array.zero_create` creates an array whose values are initially `null` or, in the case of values types, and array each of whose entities is the zero bit pattern. This function is included with F# primarily because there is really no other alternative technique to initialize and create the array values used as building blocks of larger, more sophisticated data structures.

Functional Programming with Side Effects

F# stems from a tradition in programming languages where the emphasis has been on declarative and functional approaches to programming where state is made explicit, largely by passing extra parameters. Many F# programmers use functional programming techniques first before turning to their imperative alternatives.

However, F# also integrates imperative and functional programming together in a powerful way, and F# is actually an extremely succint imperative programming language. Furthermore, in some cases, no good functional techniques exist to solve a problem, or those that do are too experimental for production to use. As a consequence using imperative constructs and libraries is common in F# in practice.

Regardless of this, the reader is encouraged to "think functionally", even about his or her imperative programming. In particular, it is always helpful to be aware of the potential side effects of the overall program and the particular characteristics of those side effects.

Consider Replacing Mutable Locals and Loops with Recursion

When imperative programmers begin to use F#, they frequently use mutable local variables or reference cells heavily as they translate code fragments from imperative-language implementations.

Consider the following (naive) implementation of factorization, transliterated from C.

```
let factorizeImperative n =
  let mutable primefactor1 = 1
  let mutable primefactor2 = n
  let mutable i = 2
  let mutable fin = false
  while (i < n && not fin) do
    if (n % i = 0) then
        primefactor1 <- i
        primefactor2 <- n / i
        fin <- true
    i <- i + 1

  if (primefactor1 = 1) then None
  else Some (primefactor1, primefactor2)
```

This code can be replaced by use of an inner recursive function.

```
let factorizeRecursive n =
  let rec find i =
    if i >= n then None
    elif (n % i = 0) then Some(i,n / i)
    else find (i+1)
  find 2
```

This second code is not only shorter, but it also uses no mutation, which makes it easier to reuse and maintain. It is also easy to see that the loop terminates (`i` increasing toward `n`) and to see the two exit conditions for the function (`i >= n` and `n % i = 0`). Note that state `i` has become an explicit parameter.

Separate Pure Computation from Side-Effecting Computations

Where possible, separate out as much of your computation as possible using side-effect-free functional programming. For example, injecting `printf` expressions throughout your code may make for a good debugging technique but, if not used widely, can lead to code that is difficult to understand and inherently imperative.

Separating Mutable Data Structures

A common technique of object-oriented programming is to ensure that mutable data structures are private, non-escaping, and, where possible, fully *separated*, which means there is no chance that distinct pieces of code can access each other's internal state in undesirable ways. Fully separated state can even be

used inside the implementation of what, to the outside world, appears to be a purely functional piece of code.

For example, where necessary, side effects can be used on private data structures allocated at the start of an algorithm and then discard these data structures before returning a result. The result is then effectively a side-effect-free function.

One example of separation from the F# library is the library's implementation of `List.map`, which uses mutation internally, but the writes occur on an internal, separated data structure that no other code can access. Thus, as far as callers are concerned, `List.map` is pure and functional.

```
open System.Collections.Generic

let divideIntoEquivalenceClasses keyf seq =
  // The dictionary to hold the equivalence classes
  let dict = new Dictionary<'key,ResizeArray<'a>>()
  // Build the groupings
  seq |> Seq.iter (fun v ->
    let key = keyf v
    let ok,prev = dict.TryGetValue(key)
    if ok then prev.Add(v)
    else let prev = new ResizeArray<'a>()
         dict.[key] <- prev
         prev.Add(v))
  // Return the sequence-of-sequences. Don't reveal the
  // internal collections: just reveal them as sequences
  dict |> Seq.map (fun group -> group.Key, Seq.readonly group.Value)
```

This uses the `Dictionary` and `ResizeArray` mutable data structures internally, but these mutable data structures are not revealed externally.

An example use:

```
divideIntoEquivalenceClasses (fun n -> n % 3) [ 0 .. 10 ]
```

Not All Side Effects Are Equal

It is often helpful to use the "weakest" set of side effects necessary to achieve the desired programming task and at least be aware when using "strong" side effects:

- *Weak side effects* are ones that are effectively benign given the assumptions making about the application. For example, writing to a log file is very useful and is essentially benign. Similarly, reading data from a stable,

unchanging file store on a local disk is effectively treating the disk as an extension of read-only memory. so reading these files is a weak form of side effect that will not be difficult to incorporate into programs.

- *Strong side effects* have a much more corrosive effect on the correctness and operational properties of the program. For example, blocking network I/O is a relatively strong side effect by any measure. Performing blocking network I/O in the middle of a library routine can have the effect of destroying the responsiveness of a GUI application, at least if the routine invoked by the GUI thread of an application. Any constructs that perform synchronization between threads are also a major source of strong side effects.

Whether a particular side effect is stronger or weaker depends very much on the application and whether the consequences of the side effect are sufficiently isolated and separated from other entities. Strong side effects can and should be used freely in the outer shell of an application or when scripting.

Avoid Combining Imperative Programming and Laziness

It is generally thought to be bad style to combine delayed computations (i.e. laziness) and side effects. But sometimes it is reasonable to set up a read from a file system as a lazy computation using sequences. However, it is relatively easy to make mistakes in this sort of programming.

```
open System.IO

let reader1, reader2 =
  let reader = new StreamReader(File.OpenRead("test.txt"))
  let firstReader() = reader.ReadLine()
  let secondReader() = reader.ReadLine()
  reader.Close()
  firstReader, secondReader

let firstLine = reader1()
let secondLine = reader2()
firstLine, secondLine
```

This code is wrong because the `StreamReader` object `reader` is used after the point indicated by the comment. The returned function values are then called, and they will try to read from the "captured" variable `reader`. Function values are just one example of *delayed computations*: other examples are lazy values, sequences, and any objects that perform computations on demand. Be careful not to build delayed objects such as `reader` that represent handles to transient,

disposable resources unless those objects are used in a way that respects the lifetime of that resource.

The previous code can be corrected to avoid using laziness in combination with a transient resource.

```
open System.IO

let line1, line2 =
  let reader = new StreamReader(File.OpenRead("test.txt"))
  let firstLine = reader.ReadLine()
  let secondLine = reader.ReadLine()
  reader.Close()
  firstLine, secondLine
```

Another technique is to use language and/or library constructs that tie the lifetime of an object to some larger object. For example, we might want to read from a file to generate a delayed sequence. The `Seq.generate_using` function is useful for opening disposable resources and using them to generate results.

```
let reader =
  Seq.generate_using
    (fun () -> new StreamReader(File.OpenRead("test.txt")))
    (fun reader ->
      if reader.EndOfStream
        then None
        else Some(reader.ReadLine()))
```

A `use` binding within a sequence expression can be used, which augments the sequence object with the code needed to clean up the resource when iteration is finished or terminates.

```
let reader =
  seq { use reader = new StreamReader(File.OpenRead("test.txt"))
        while not reader.EndOfStream do
          yield reader.ReadLine() }
```

Try to keep the code application pure and use both delayed computations laziness) and imperative programming (side effects) where appropriate but to be careful about using them together.