

F# Language Overview

Tomáš Petříček (tomas@tomasp.net)
<http://tomasp.net/blog>

1 Introduction

This text is based on a short overview of the F# language that was included in my bachelor thesis, but I extended it to cover all the important F# aspects and topics. The goal of this article is to introduce all the features in a single (relatively short) text, which means that understanding of a few advanced topics discussed later in the text may require some additional knowledge or previous experience with functional programming.

Anyway, the article still tries to introduce all the interesting views on programming that the F# language provides and the goal is to show that these views are interesting, even though not all of them are fully explained as they would deserve. Of course, this text won't teach you everything about F#, but it tries to cover the main F# design goals and (hopefully) presents all the features that make F# interesting and worth learning. In this first part I will shortly introduce F# and the supported paradigms that will be discussed further in the text.

1.1 Introducing F#

In one of the papers about F#, the F# designers gave the following description: "*F# is a multi-paradigm .NET language explicitly designed to be an ML suited to the .NET environment. It is rooted in the Core ML design and in particular has a core language largely compatible with OCaml*". In other words this means that the syntax of the F# language is similar to ML or OCaml (don't worry if you don't know these languages, we'll look at some examples shortly), but the F# language targets .NET Framework, which means that it can natively work with other .NET components and also that it contains several language extensions to allow smooth integration with the .NET object system.

Another important aspect mentioned in this description is that F# is *multi-paradigm* language. This means that it tries to take the best from many programming languages from very different worlds. The first paradigm is *functional programming* (the languages that largely influenced the design of F# in this area are ML, OCaml and other), which has a very long tradition and is becoming more important lately for some very appealing properties, including the fact that functional code tends to be easier to test and parallelize and is also extensible in a ways where object oriented code makes extending difficult.

The second paradigm is widely adopted *object oriented* programming, which enables interoperability with other .NET languages. In F# it is often used for implementing elementary data types (meaning that the operations on the type are well known and change very rarely), for grouping a set of elementary functions that are together used to perform some complicated operation (i.e. implementing an interface) and also when working with object oriented user interface frameworks.

Finally, the third paradigm supported by F# is *language oriented programming* (the design of F# in this area is largely influenced by ML, Haskell and also by LINQ). In general, language oriented programming is focused on developing executors for some code which has a structure of a language (be it a declarative language like XML, or a fully powerful language like some subset of F#). In this overview, I will focus on two techniques provided by F# that allow you to give a different meaning to blocks of F# code. In a programming language theory, this is often called *internal domain specific languages*, because the code is written in the host language, but is specifically designed as a way for solving problems from some specific *domain*. An example of such language (and an associated executor) is a block of code that is written as a linear code, but is executed asynchronously (in F# this can be implemented using *computation expressions*), or a query that is written in F#, but is executed as a SQL code by some database server (this can be implemented using *F# quotations*).

1.2 Organization of the Text

In the rest of this article series we will look at all these three paradigms supported by F# starting with functional programming and basic F# types used when writing code in a functional way, continuing with object oriented programming and the support for .NET interoperability which is closely related to the OOP in F#. Lastly, we will look at the language oriented programming paradigm including some of the most important .NET and F# library functions that make it possible.

2 Functional Programming

As already mentioned, F# is a typed functional language, by which I mean that types of all values are determined during the compile-time. However, thanks to the use of a *type inference*, the types are explicitly specified in the code very rarely as we will see in the following examples. The type inference means that the compiler deduces the type from the code, so for example when calling a function that takes `int` as an argument and returns `string` as a result, the compiler can infer the type of the variable where the result is assigned (it has to be `string`) as well as the type of the variable that is given as an argument (it has to be `int`). Basic data types (aside from a standard set of primitive numeric and textual types that are present in any .NET language) available in F# are tuple, discriminated union, record, array, list, function and object. In the following quick overview, we will use the F# interactive, which is a tool that compiles and executes the entered text on the fly.

The F# interactive can be either used from Visual Studio or by running the `fsi.exe` from the F# installation directory. In the whole article series we will also use the F# lightweight syntax, which makes the code white-space sensitive, but simplifies many of the syntactical rules. To enable the lightweight syntax enter the following command in the FSI:

```
> #light;;
```

The double semicolon is used as the end of the FSI input and sends the entered text to the compiler. This allows us to enter multiple lines of code in a single command. With a few exceptions (mostly when showing a declaration of a complex type) all the code samples in this article are written as commands for FSI including the double semicolon and the result printed by the FSI. Longer code samples can be entered to FSI as well - just add the double semicolon to terminate the input.

2.1 F# Data Types Overview

Tuples

The first example demonstrates constructing and deconstructing a tuple type. Tuple is simple type that groups together two or more values of any (possibly different) types, for example `int` and `string`:

```
> let tuple = (42, "Hello world!");;  
val tuple : int * string  
  
> let (num, str) = tuple;;  
val num : int  
val str : string
```

As you can see, the compiler deduced a type of the expression that is present on the right side of the equals sign and the F# interactive printed the type, so we can review it. In this example the type of a first element in a tuple is `int` and the type of the second element is `string`. The asterisk denotes that the type is a tuple. Similarly, you can define a tuple with more than three elements, but the type changes with the number of elements in a tuple, which means that tuples can't be used for storing an unknown number of values. This can be done using lists or arrays, which will be discussed later.

The syntax used for deconstructing the value into variables `num` and `str` is in general called pattern matching and it is used very often in the F# language – the aim of pattern matching is to allow matching a value against a pattern that specifies different view of the data type – in case of tuple, one view is a single value (of type tuple) and the second view is a pair of two values (of different types). Pattern matching can be used with all standard F# types, most notably with tuples, discriminated unions and record types. In addition, F# also supports generalized pattern matching constructs called *active patterns*, which are discussed later in this overview.

Tuple types are very handy for returning multiple values from functions, because this removes the need to declare a new class or use references when writing a function that performs some simple operation resulting in more returned values (especially in places where C# uses `ref` and `out` parameters). In general, I would recommend using tuples when the function is either simple (like division with remainder), local (meaning that it will not be accessed from a different module or file) or it is likely to be used with pattern matching. For returning more complicated structures it is better to use record types which will be discussed shortly.

Discriminated Union

In the next sample we demonstrate working with the discriminated union type. This type is used for representing a data type that store one of several possible options (where the options are well known when writing the code). One common example of data type that can be represented using discriminated unions is an *abstract syntax tree* (i.e. an expression in some programming language):

```
> // Declaration of the 'Expr' type
type Expr =
  | Binary    of string * Expr * Expr
  | Variable  of string
  | Constant  of int;;
(...)
> // Create a value 'v' representing 'x + 10'
let v = Binary("+", Variable "x", Constant 10);;
val v : Expr
```

To work with the values of a discriminated union type, we can again use pattern matching. In this case we use the `match` language construct, which can be used for testing a value against several possible patterns – in case of the `Expr` type, the possible options are determined by all identifiers used when declaring the type (these are called *constructors*), namely `Binary`, `Variable` and `Constant`. The following example declares a function `eval`, which evaluates the given expression (assuming that `getVariableValue` is a function that returns a value of variable):

```
> let rec eval x =
  match x with
  | Binary(op, l, r) ->
    let (lv, rv) = (eval l, eval r)
    if (op = "+") then lv + rv
    elif (op = "-") then lv - rv
    else failwith "Unknow operator!"
  | Variable(var) ->
    getVariableValue var
  | Constant(n) ->
    n;;
val eval : Expr -> int
```

When declaring a function we can use the `let` keyword that is used for binding a value to a name. I don't use a term *variable* known from other programming languages for a reason that will be explained shortly. When writing a recursive function, we have to explicitly state this using the `rec` keyword as in the previous example.

Discriminated unions form a perfect complement to the typical object-oriented inheritance structure. In an OO hierarchy the base class declares all methods that are overridden in derived classes, meaning that it is easy to add new type of value (by adding a new inherited class), but adding a new operation requires adding method to all the classes. On the other side, a discriminated union defines all types of values in advance, which means that adding a new function to work with the type is easy, but adding a new type of value (new constructor to the discriminated union) requires modification of all existing functions. This suggests that discriminated unions are usually a better way for implementing a Visitor design pattern in F#.

Records

The next data type that we will look at is a record type. It can be viewed as a tuple with named members (in case of record these are called *labels*), which can be accessed using a dot-notation and as mentioned earlier it is good to use this type when it would be difficult to understand what the members in a tuple represent. One more difference between a record type and a tuple is that records have to be declared in advance using a type construct:

```
> // Declaration of a record type
type Product = { Name:string; Price:int };;

> // Constructing a value of the 'Product' type
let p = { Name="Test"; Price=42; };;
val p : Product

> p.Name;;
val it : string = "Test"

> // Creating a copy with different 'Name'
let p2 = { p with Name="Test2" };;
val p2 : Product
```

The last command uses an interesting construct - the `with` keyword. The record types are by default immutable, meaning that the value of the member can't be modified. Since the members are immutable you will often need to create a copy of the record value with one (or more) modified members. Doing this explicitly by listing all the members would be impractical, because it would make adding a new members very difficult, so F# supports the `with` keyword to do this.

F# records are in many ways similar to classes and they can be, indeed, viewed as simplified classes. Record types are by default immutable, which also means that F# use a structural comparison when comparing two values of a record type (instead of the default reference comparison used when working with classes) and if you need this behavior (e.g. for storing records as a keys in a dictionary) it is very practical to use them. Also, using a record instead of a class is a good idea in a functional code where you can use the `with` construct. Exposing a record type in a public interface of the module requires additional care and it is often useful to make the labels available as members, which makes it easier to modify implementation of the type later. This topic will be further discussed in the third part of this article series.

Lists

The types used for storing collections of values are list and array. F# list is a typical linked-list type known from many functional languages – it can be either an empty list (written as []) or a cell containing a value and a reference to the tail, which is itself a list (written as `value::tail`). It is also possible to write a list using a simplified syntax, which means that you can write [1; 2; 3] instead of `1::2::3::[]` (which is exactly the same list written just using the two basic list constructors). Array is a .NET compatible mutable array type, which is stored in a continuous memory location and is therefore very efficient – being a mutable type, array is often used in imperative programming style, which will be discussed later. The following example shows declaration of a list value and an implementation of a recursive function that adds together all elements in the list:

```
> let nums = [1; 2; 3; 4; 5];;
val nums : list<int>

> let rec sum list =
    match list with
    | h::tail -> (sum tail) + h
    | [] -> 0
val sum : list<int> -> int
```

Similarly as earlier we declared a recursive function using `let rec` and inside the body we used pattern matching to test whether the list is an empty list or a list cell. Note that list is a generic type, which means that it can store values of any F# type. The type in our example is `list<int>`, which means that the declared instance of list contains integers. Functions working with generic types can be restricted to some specific type - for example the `sum` function above requires a list of integers that can be added (this is inferred by the type inference, because the default type used with the `+` operator is `int`). Alternatively, the function can be generic as well, which means that it works with any lists - for example a function that returns the last element in the list doesn't depend on the type and so it can be generic. The signature of a generic function to return the last element would be `last : list<'a> -> 'a`.

An important feature when writing recursive functions in F# is the support for *tail-calls*. This means that when the last operation performed by the function is a call to a function (including a recursive call to itself), the runtime drops the current stack frame, because it isn't needed anymore - the value returned by the called function is a result of the caller. This minimizes a chance for getting a stack overflow exception. The `sum` function from the previous example can be written using an auxiliary function that uses a tail recursion as following:

```
> // 'acc' is usually called an 'accumulator' variable
let rec sumAux acc list =
    match list with
    | h::tail -> sumAux (acc + h) tail
    | [] -> acc
val sum : int -> list<int> -> int

> let sum list = sumAux 0 list
val sum : list<int> -> int
```

Functions

Finally, the type that gives name to the whole *functional programming* is a function. In F#, similarly to other functional languages, functions are first-class values, meaning that they can be used in a same way as any other types. They can be given as an argument to other functions or returned from a function as a result (a function that takes function as an argument or returns function as a result is called *high-order* function) and the function type can be used as a type argument to generic types - you can for example create a list of functions. The important aspect of working with functions in functional languages is the ability to create closures - creating a function that captures some values available in the current stack frame. The following example demonstrates a function that creates and returns a function for adding specified number to an initial integer:

```
> let createAdder n = (fun arg -> n + arg);;
val createAdder : int -> int -> int

> let add10 = createAdder 10;;
val add10 : int -> int

> add10 32;;
val it : int = 42
```

In the body of the `createAdder` function we use a `fun` keyword to create a new unnamed function (a function constructed in this way is called a *lambda function*). The type of `createAdder` (`int -> int -> int`) denotes that when the function is called with `int` as an argument, it produces a value of type function (which takes an integer as a parameter and produces an integer as a result). In fact, the previous example could be simplified, because any function taking more arguments is treated as a function that produces a function value when it is given the first argument, which means that the following code snippet has the same behavior. Also note that the types of the function `createAdder` declared earlier and the type of the function `add` are the same):

```
> let add a b = a + b;;
val add : int -> int -> int

> let add10 = add 10;;
val add10 : int -> int
```

When declaring the function value `add10` in this example, we used a function that expects two arguments with just one argument. The result is a function with a fixed value of the first argument which now expects only one (the second) argument. This aspect of working with functions is known as *currying*.

Many functions in the F# library are implemented as high-order functions and functions as an arguments are often used when writing a *generic* code, that is a code that can work with generic types (like `list<'a>`, which we discussed earlier). For example standard set of functions for manipulating with list values is demonstrated in the following example:

```
> let odds = List.filter (fun n -> n%2 <> 0) [1; 2; 3; 4; 5];;
val odds : list<int> = [1; 3; 5]

> let squares = List.map (fun n -> n * n) odds;;
val squares : list<int> = [1; 9; 25]
```

It is interesting to note that the functions that we used for manipulating with lists are generic (otherwise they wouldn't be very useful!). The signature of the `filter` function is `('a -> bool) -> list<'a> -> list<'a>`, which means that the function takes list of some type as a second argument and a function that returns a `true` or `false` for any value of that type, finally the result type is same as the type of the second argument. In our example we instantiate the generic function with a type argument `int`, because we're filtering a list of integers. The signatures of generic functions often tell a lot about the function behavior. When we look at the signature of the `map` function `('a -> 'b) -> list<'a> -> list<'b>` we can deduce that `map` calls the function given as a first argument on all the items in the list (given as a second argument) and returns a list containing the results.

In the last example we will look at the pipelining operator (`|>`) and we will also look at one example that demonstrates how currying makes writing the code easier - we will use the `add` function declared earlier:

```
> let nums = [1; 2; 3; 4; 5];;
val nums : list<int>

> let odds_plus_ten =
    nums
    |> List.filter (fun n-> n%2 <> 0)
    |> List.map (add 10)
val odds_plus_ten : list<int> = [11; 13; 15];;
```

Sequences of `filter` and `map` function calls are very common and writing it as a single expression would be quite difficult and not very readable. Luckily, the sequencing operator allows us to write the code as a single expression in a more readable order - as you can see in the previous example, the value on the left side of the `|>` operator is given as a last argument to the function call on the right side, which allows us to write the expression as sequence of ordinary calls, where the state (current list) is passed automatically to all functions. The line with `List.map` also demonstrates a very common use of currying. We want to add `10` to all numbers in the list, so we call the `add` function with a single argument, which produces a result of the type we needed - a function that takes an integer as an argument and returns an integer (produced by adding `10`) as the result.

2.2 Function Composition

One of the most interesting aspects of working with functions in functional programming languages is the possibility to use function composition operator. This means that you can very simply build a function that takes an argument, invokes a first function with this argument and passes the result to a second function. For example, you can compose a function `fst`, which takes a tuple (containing two elements) and returns the first element in the tuple with a function `uppercase`, which takes a string and returns it in an uppercase:

```
> (fst >> String.uppercase) ("Hello world", 123);;
val it : string = "HELLO WORLD"

> let data = [ ("Jim", 1); ("John", 2); ("Jane", 3) ];;
val data : (string * int) list

> data |> List.map (fst >> String.uppercase);;
val it : string list = ["JIM"; "JOHN"; "JANE"]
```


In the first command, we just compose the functions and call the returned function with a tuple as an argument, however the real advantage of this trick becomes more obvious in the third command, where we use the function composition operator (>>) to build a function that is given as an argument to a map function that we used earlier. The function composition allows us to build a function without explicitly using a lambda function (written using the fun keyword) and when this features are used reasonably it makes the code more compact and keeps it very readable.

2.3 Expressions and Variable Scoping

The F# language doesn't have a different notion of a statement and an expression, which means that every language construct is an expression with a known return type. If the construct performs only a *side effect* (for example printing to a screen or modifying a global mutable variable or a state of .NET object) and doesn't return any value then the type of the construct is unit, which is a type with only one possible value (written as "()"). The semicolon symbol (;) is used for sequencing multiple expressions, but the first expression in the sequence should have a unit as a result type. The following example demonstrates how the if construct can be used as an expression in F# (though in the optional F# lightweight syntax, which makes whitespace significant and which we used in the rest of this overview, the semicolon symbol can be omitted):

```
> let n = 1
   let res =
     if n = 1 then
       printfn "..n is one..";
       "one"
     else
       "something else";;
   ..n is one..
   val res : string = "one"
```

When this code executes it calls the true branch of the if expression, which first calls a side-effecting function, which prints a string and then returns a string ("one") as the result. The result is then assigned to the res value.

Unlike some languages that allow one variable name to appear only once in the entire function body (e.g. C#) or even treat all variables declared inside the body of a function as a variable with scope of the whole function (e.g. Visual Basic or JavaScript), the scope of F# values is determined by the let binding and it is allowed to hide a value by declaring a value with the same name. The following (slightly esoteric) example demonstrates this:

```
> let n = 21
   let f =
     if n < 10 then
       let n = n * 2
       (fun () -> print_int n)
     else
       let n = n / 2
       (fun () -> print_int n)
   let n = 0
   f ();;
42
val it : unit
```

In this example, the value `n` declared inside a branch of the `if` expression is captured by a function created using the `fun` keyword, which is returned from the `if` expression and bound to the value named `f`. When the `f` is invoked it indeed uses the value from the scope where it was created, which is `42`. In languages, where the variable named `n` would refer to a value stored globally, it would be rather problematic to write a code like this. Of course, writing a code similar to what I demonstrated in this example isn't a good idea, because it makes the code very difficult to read. There are however situations where hiding a value that is no longer needed in the code is practical.

3 Imperative and Object-Oriented Programming

In the third part of the F# Overview article series, we will look at language features that are mostly well known, because they are present in most of the currently used programming languages. Indeed, I'm talking about imperative programming, which is a common way for storing and manipulating application data and about object oriented programming which is used for structuring complex programs.

In general, F# tries to make using them together with the functional constructs described in the previous section as natural as possible, which yields several very powerful language constructs.

3.1 Imperative Programming and Mutable Values

Similarly as ML and OCaml, F# adopts an *eager evaluation* mechanism, which means that a code written using sequencing operator is executed in the same order in which it is written and expressions given as an arguments to a function are evaluated before calling the function (this mechanism is used in most imperative languages including C#, Java or Python). This makes it semantically reasonable to support imperative programming features in a functional language. As already mentioned, the F# value bindings are by default immutable, so to make a variable mutable the `mutable` keyword has to be used. Additionally F# supports a few imperative language constructs (like `for` and `while`), which are expressions of type `unit`:

```
> // Imperative factorial calculation
let n = 10
let mutable res = 1
for n = 2 to n do
    res <- res * n
// Return the result
res;;
val it : int = 3628800
```

The use of the eager evaluation and the ability to use mutable values makes it very easy to interoperate with other .NET languages (that rely on the use of mutable state), which is an important aspect of the F# language. In addition it is also possible to use the `mutable` keyword for creating a record type with a mutable field.

Arrays

As mentioned earlier, another type of value that can be mutated is .NET array. Arrays can be either created using `[| .. |]` expressions (in the following example we use it together with a range expression, which initializes an array with a range) or using functions from the `Array` module, for example `Array.create`. Similarly to the mutable variables introduced in the previous section, the value of an array element can be modified using the `<-` operator:

```
> let arr = [| 1 .. 10 |]
val arr : array<int>

> for i = 0 to 9 do
    arr.[i] <- 11 - arr.[i]
(...)

> arr;;
val it : array<int> = [| 10; 9; 8; 7; 6; 5; 4; 3; 2; 1 |]
```

3.2 .NET interoperability

The .NET BCL is built in an object oriented way, so the ability to work with existing classes is essential for the interoperability. Many (in fact almost all) of the classes are also mutable, so the eager evaluation and the support for side-effects are two key features when working with any .NET library. The following example demonstrates working with the mutable generic `ResizeArray<T>` type from the BCL (`ResizeArray` is an alias for a type `System.Collections.Generic.List` to avoid a confusion with the F# `list` type):

```
> let list = new ResizeArray<_>()
  list.Add("hello")
  list.Add("world")
  Seq.to_list list;;
val it : string list = ["hello"; "world"]
```

As you can see, we used the underscore symbol when creating an instance of the generic type, because the type inference algorithm in F# can deduce the type argument from the code (in this example it infers that the type argument is `string`, because the `Add` method is called with a `string` as an argument). After creating an instance we used `Add` method to modify the list and add two new items. Finally, we used a `Seq.to_list` function to convert the collection to the F# list.

As a fully compatible .NET language, F# also provides a way for declaring its own classes (called *object types* in F#), which are compiled into CLR classes or interfaces and therefore the types can be accessed from any other .NET language as well as used to extend classes written in other .NET languages. This is an important feature that allows accessing complex .NET libraries like Windows Forms or ASP.NET from F#.

3.3 Object Oriented Programming

Object Types

Object oriented constructs in F# are compatible with the OO support in .NET CLR, which implies that F# supports single implementation inheritance (a class can have one base class), multiple interface inheritance (a class can implement several interfaces and an interface can inherit from multiple interfaces), subtyping (an inherited class can be casted to the base class type) and dynamic type tests (it is possible to test whether a value is a value of an inherited class casted to a base type). Finally, all object types share a common base class which is called `obj` in F# and is an alias to the CLR type `System.Object`.

F# object types can have fields, constructors, methods and properties (a property is just a syntactic sugar for getter and setter methods). The following example introduces the F# syntax for object types:

```
type MyCell(n:int) =
  let mutable data = n + 1
  do printf "Creating MyCell(%d)" n

  member x.Data
    with get() = data
    and set(v) = data <- v

  member x.Print() =
    printf "Data: %d" data
```

```

override x.ToString() =
    sprintf "(Data: %d)" data

static member FromInt(n) =
    MyCell(n)

```

The object type `MyCell` has a mutable field called `data`, a property called `Data`, an instance method `Print`, a static method `FromInt` and the type also contains one overridden method called `ToString`, which is inherited from the `obj` type and returns a string representation of the object. Finally, the type has an implicit constructor. Implicit constructors are syntactical feature which allows us to place the constructor code directly inside the type declaration and to write the constructor arguments as part of the `type` construct. In our example, the constructor initializes the mutable field and prints a string as a side effect. F# also supports *explicit* constructors that have similar syntax as other members, but these are needed rarely.

In the previous example we implemented a *concrete object type* (a class), which means that it is possible to create an instance of the type and call its methods in the code. In the next example we will look at declaration of an interface (called *abstract object type* in F#). As you can see, it is similar to the declaration of a class:

```

type AnyCell =
    abstract Value : int with get, set
    abstract Print : unit -> unit

```

The interesting concept in the F# object oriented support is that it is not needed to explicitly specify whether the object type is abstract (interface), concrete (class) or partially implemented (class with abstract methods), because the F# compiler infers this automatically depending on the members of the type. Abstract object types (interfaces) can be implemented by a concrete object type (class) or by an object expression, which will be discussed shortly. When implementing an interface in an object type we use the `interface .. with` construct and define the members required by the interface. Note that the indentation is significant in the lightweight F# syntax, meaning that the members implementing the interface type have to be indented further:

```

type ImplementCell(n:int) =
    let mutable data = n + 1
    interface AnyCell with
        member x.Print() = printf "Data: %d" data
        member x.Value
            with get() = data
            and set(v) = data <- v

```

The type casts supported by F# are *upcast*, used for casting an object to a base type or to an implemented interface type (written as `o :> TargetType`), *downcast*, used for casting back from a base type (written as `o :?> TargetType`), which throws an exception when the value isn't a value of the specified type and finally, a *dynamic type test* (written as `o :? TargetType`), which tests whether a value can be casted to a specified type.

Object expressions

As already mentioned, abstract types can be also implemented by an object expression. This allows us to implement an abstract type without creating a concrete type and it is particularly useful when you need to return an implementation of a certain interface from a

function or build an implementation on the fly using functions already defined somewhere else in your program. The following example implements the `AnyCell` type:

```
> let newCell n =
    let data = ref n
    { new AnyCell with
      member x.Print() = printf "Data: %d" (!data)
      member x.Value
        with get() = !data
          and set(v) = data:=v };;
val newCell : int -> AnyCell
```

In this code we created a function that takes an initial value as an argument and returns a cell holding this value. In this example we use one more type of mutable values available in F#, called *reference cell*, which are similar to a mutable values, but more flexible (the F# compiler doesn't allow using an ordinary mutable value in this example). A mutable cell is created by a `ref` function taking an initial value. The value is accessed using a prefix `!` operator and can be modified using `:=` operator. When implementing the abstract type, we use a `new ... with` construct with members implementing the functionality required by the abstract type (an object expression can't add any members). In this example we need a reference cell to hold the value, so the cell is declared in a function and captured in a *closure*, which means that it will exist until the returned object will be garbage collected.

3.4 Adding Members to F# Types

Probably the most important advantage of using object types is that they hide an implementation of the type, which makes it possible to modify the implementation without breaking the existing code that uses them. On the other side, basic F# types like discriminated unions or records expose the implementation details, which can be problematic in some cases, especially when developing a larger application. Also, the dot-notation used with object types makes it very easy to discover operations supported by the type. To bridge this problem, F# allows adding members to both discriminated unions and record types:

```
> type Variant =
    | Num of int
    | Str of string with
      member x.Print() =
        match x with
        | Num(n) -> printf "Num %d" n
        | Str(s) -> printf "Str %s" s;;
(...)

> let v = Num 42
    v.Print();;
Num 42
```

In this example we declared a type called `Variant` which can contain either a number or a string value and added a member `Print` that can be invoked using dot-notation. Aside from adding members (both methods and properties) it is also possible to implement an abstract object type by a record or discriminated union using the `interface ... with` syntax mentioned earlier.

Rather than writing all code using *member* syntax, it is often more elegant to implement the functionality associated with an F# type in a function and then use *type augmentations* to make this functionality available as a member via dot-notation. This is a pattern used very often in the F# library implementation and I personally believe that it makes the code more readable. The following example re-implements the *Variant* type using this pattern:

```
type Variant =
    | Num of int
    | Str of string

let print x =
    match x with
    | Num(n) -> printf "Num %d" n
    | Str(s) -> printf "Str %s" s

type Variant with
    member x.Print() = print x
```

The construct `type ... with` is a type augmentation, which adds the member `Print` to a type declared earlier in the code. The type augmentation has to be included in a same compilation unit as the declared type - usually in a same file. It is also possible to attach *extension members* to a type declared in a different compilation unit - the main difference is that these members are just a syntactical sugar and are not a part of the original type, meaning that they can't access any implementation details of the type. The only reason for using extension members is that they make your function for working with the type available using the dot-notation, which can simplify the code a lot and it will be easier to find the function (for example it will be available in the Visual Studio IntelliSense). When declaring an extension member you use the same syntax as for type augmentations with the difference that the name of the type has to be fully qualified (e.g. `System.Collections.Generic.List<'a>`):

```
> type System.Collections.Generic.List<'a> with
    member x.ToList() = Seq.to_list x;
(...)

> let r = new ResizeArray<_>()
    r.Add(1)
    r.Add(2)
    r.ToList();;
val it : list<int> = [ 1; 2 ]
```

In this example we use extension members to add a `ToList` method to an existing .NET generic type. Note that when declaring the extension members we have to use the original type name and not the F# alias. You should also bear in mind that extension members are resolved by the F# compiler and so calling them from C# will not be easily possible. In general, extension members are not declared very often, but some parts of the F# library (for example the features for asynchronous and parallel programming) use them.

4 Language Oriented Programming

Defining precisely what the term *language oriented programming* means in context of the F# language would be difficult, so I will instead explain a few examples that will demonstrate how I understand it. In general, the goal of language oriented programming is to develop a *language* that would be suitable for some (more specific) class of tasks and use this language for solving these tasks. Of course, developing a real programming language is extremely complex problem, so there are several ways for making it easier. As the most elementary example, you can look at XML files (with certain schema) as language that are processed by your program and solve some specific problem (for example configuring the application). As a side note, I should mention that I'm not particularly happy with the term 'language' in this context, because the term can be used for describing a wide range of techniques from very trivial constructs to a complex object-oriented class libraries, but I have not seen any better term for the class of techniques that I'm going to talk about.

What I will focus on in this article is using languages inside F# - this means that the custom language will be always a subset of the F# language, but we will look at ways for giving it a different meaning than the standard F# code would have. In some articles you can also see the term *domain specific language*, which is quite close to what we're discussing here. The domain specific language is a language suitable for solving some class of problems and in general it can be either *external*, meaning that it is a separate language (e.g. a XML file) or an *internal*, meaning that it is written using a subset of the host language. Using this terminology, we will focus on writing *internal DSLs* in F#.

Since this term is not as widely used as functional or object oriented programming which we discussed in earlier parts of this document, let me very quickly introduce why I believe that this is an important topic. I think the main reason why language oriented development is appealing paradigm is that it allows very smooth cooperation of people working on the project - there are people who develop the language and those who use it. The language developers need to have advanced knowledge of the technology (F#) and also of the problem that their language is trying to solve (e.g. some mathematical processing), but they don't need to know about all the problems that are being solved using the language. On the other side, the users of the language need only basic F# knowledge and they can fully focus on solving the real problems.

4.1 Discriminated Union as Declarative Language

Probably the simplest example of domain-specific language that can be embedded in the F# code is a discriminated union, which can be used for writing declarative specifications of behavior or for example for representing and processing mathematical expressions:

```
> type Expr =
  | Binary of string * Expr * Expr
  | Var    of string
  | Const  of int;;
(...)

> let e = Binary("+", Const(2), Binary("*", Var("x"), Const(4)));;
val e : Expr
```


In this example we created a discriminated union and used it for building a value representing a mathematical expression. This is of course very primitive 'language', but when you implement functions for working with these values (for example differentiation or evaluation) you'll get a simple language for processing mathematical expressions inside F#. Another problem that could be solved using this technique includes for example configuration of some graphical user interface or definition of template for some simple data manipulation.

4.2 Active Patterns

A language feature that is closely related to discriminated unions is called *active patterns*. Active patterns can be used for providing different views on some data type, which allows us to hide the internal representation of the type and publish only these views. Active patterns are similar to discriminated unions, because they can provide several views on a single value (in the previous example we had a value that we could view either as `Binary`, `Var` or `Const`) and similarly as constructors of discriminated union, active patterns can be used in pattern matching constructs.

A typical example, where a type can be viewed using different views is a complex number, which can be either viewed in a Cartesian representation (real and imaginary part) or in a polar form (absolute value and phase). Once the module provides these two views for a complex number type, the internal representation of the type can be hidden, because all users of the type will work with the number using active patterns, which also makes it easy to change the implementation of the type as needed.

It is recommended to use active patterns in public library API instead of exposing the names of discriminated union constructors, because this makes it possible to change the internal representation without breaking the existing code. The second possible use of active patterns is extending the 'vocabulary' of a language built using discriminated union. In the following example we will implement an active pattern `Commutative` that allows us to decompose a value of type `Expr` into a call to commutative binary operator:

```
> let (|Commutative|_|) x =
    match x with
    | Binary(s, e1, e2) when (s = "+") || (s = "*") -> Some(s, e1, e2)
    | _ -> None;;
val ( |Commutative|_| ) : Expr -> (string * Expr * Expr) option
```

As you can see, the declaration of active pattern looks like a function declaration, but uses a strangely looking function name. In this case we use the `(|PatternName|_|)` syntax, which declares a pattern that can return a successful match or can fail. The pattern has a single argument (of type `Expr`) and returns an `option` type, which can be either `Some(...)` when the value matches the pattern or `None`. As we will show later, the patterns that can fail can be used in a `match` construct, where you can test a value against several different patterns.

As demonstrated in this example, active patterns can be used in a similar sense in which you can use discriminated unions to define a language for constructing the values. The key difference is that discriminated unions can be used for building the value (meaning that they will be used by all users of the language) and active patterns are used for decomposing the values and so they will be used in a code that interprets the language (written usually by the language designer) or by some pre-processing or optimizing code (written by advanced users of the language).

In the next example we will look at one advanced example of using the numerical language that we define earlier. We will implement a function that tests whether two expressions are equal using the commutativity rule, meaning that for example $10*(a+5)$ will be considered as equal to $(5+a)*10$:

```
> let rec equal e1 e2 =
    match e1, e2 with
    | Commutative(o1, l1, r1), Commutative(o2, l2, r2) ->
        (o1 = o2) && (equal l1 r2) && (equal r1 l2)
    | _ -> e1 = e2;;
val equal : Expr -> Expr -> bool

> let e1 = Binary("*", Binary("+", Const(10), Var("x")), Const(4));;
    let e2 = Binary("*", Const(4), Binary("+", Var("x"), Const(10)));;
    equal e1 e2;;
val it : bool = true
```

As you can see, implementing the `equal` function that uses the commutativity rule is much easier using the `Commutative` active pattern than it would be explicitly by testing if the value is a use of specific binary operator. Also, when we'll introduce a new commutative operator, we'll only need to modify the active pattern and the `equal` function will work correctly.

4.3 Sequence comprehensions

Before digging deeper into advanced language-oriented features of F#, I'll need to do a small digression and talk about *sequence comprehensions*. This is a language construct that allows us to generate sequences, lists and arrays of data in F# and as we will see later it can be generalized to allow solving several related problems. Anyway, let's first look at an example that filters an F# list:

```
> let people = [ ("Joe", 55); ("John", 32); ("Jane", 24); ("Jimmy", 42) ];;
val people : (string * int) list

> [ for (name, age) in people
    when age < 30
    -> name ];;
val it : string list = ["Jane"]
```

In this example we first declared a list with some data and then used a sequence expression, wrapped between square brackets `[` and `]`, to select only some elements from the list. The use of square brackets indicate that the result should be an F# list (you can also use `[| .. |]` to get an array or `seq { .. }` to get a sequence as I'll show later). The code inside the comprehension can contain most of the ordinary F# expressions, but in this example I used one extension, the `when .. ->` construct, which can be used for typical filtering and projection operations. The same code can be written like this:

```
> [ for (name, age) in people do
    if (age < 30) then
        yield name ];;
val it : string list = ["Jane"]
```

In this example, we used an ordinary `for .. do` loop (in the previous example the `do` keyword was missing and we used `if .. then` condition instead of `when`). Finally, returning a value from a sequence comprehension can be done using the `yield` construct. The point of this

example is to demonstrate that the code inside the comprehension is not limited to some specific set of expressions and can, in fact, contain very complex F# code. I will demonstrate the flexibility of sequence comprehensions in one more example - the code will generate all possible words (of specified length) that can be generated using the given alphabet:

```
> let rec generateWords letters start len =
    seq { for l in letters do
        let word = (start ^ l)
        if len = 1 then
            yield word
        if len > 1 then
            yield! generateWords letters word (len-1) }
val generateWords : #seq<string> -> string -> int -> seq<string>

> generateWords ["a"; "b"; "c"] "" 4;;
val it : seq<string> = seq ["aaaa"; "aaab"; "aaac"; "aaba"; ...]
```

This example introduces two interesting constructs. First of all, we're using `seq { .. }` expression to build the sequence, which is a lazy data structure, meaning that the code will be evaluated on demand. When you ask for the next element, it will continue evaluating until it reaches `yield` construct, which returns a word and then it will block again (until you ask for the next element). The second interesting fact is that the code is recursive - the `generateWord` function calls itself using `yield!` construct, which first computes the elements from the given sequence and then continues with evaluation of the remaining elements in the current comprehension.

4.4 F# Computation Expression

The next F# feature that we will look at can be viewed as a generalization of the sequence comprehensions. In general, it allows you to declare blocks similar to the `seq { .. }` block that execute the F# code in a slightly different way. In the case of `seq` this difference is that the code can return multiple values using `yield`.

In the next example we will implement a similar block called `maybe` that performs some computation and returns `Some(res)` when the computation succeeds, but it can also stop its execution when some operation fails and return `None` immediately, without executing the rest of the code inside the block. Let's first implement a simple function that can either return some value or can fail and return `None`:

```
let readNum () =
    let s = Console.ReadLine()
    let succ,v = Int32.TryParse(s)
    if (succ) then Some(v) else None
```

Now, we can write a code that reads two numbers from the console and adds them together, producing a value `Some(a+b)`. However, when a call to `readNum` fails, we want to return `None` immediately without executing the second call to `readNum`. This is exactly what the `maybe` block will do (I'll show the implementation of the block shortly):

```

let n =
  maybe { do printf "Enter a: "
            let! a = readNum()
            do printf "Enter b: "
            let! b = readNum()
            return a + b }
printf "Result is: %A" n

```

The code inside the block first calls `printf` and then uses a `let!` construct to call the `readNum` function. This operation is called *monadic bind* and the implementation of `maybe` block specifies the behavior of this operation. Similarly, it can also specify behavior of the `do` and `return` operation, but in this example the `let!` is the most interesting, because it tests whether the computed value is `None` and stops the execution in such case (otherwise it starts executing the rest of the block).

Before looking at the implementation of the `maybe` block, let's look at the type of the functions that we'll need to implement. Every block (usually called *computation expression* in F#) is implemented by a *monadic builder* which has the following members that define elementary operators:

```

// Signature of the builder for monad M
type MaybeBuilder with
  member Bind : M<'a> * ('a -> M<'b>) -> M<'b>
  member Return : 'a -> M<'a>
  member Delay : (unit -> M<'a>) -> M<'a>

```

We'll shortly discuss how the F# compiler uses these members to execute the computation expression, but let me first add a few short comments for those who are familiar with Haskell monads. The `Bind` and `Return` members specify the standard monadic operators (known from Haskell), meaning that `Bind` is used when we use the `let!` operator in the code and `Return` is called when the computation expression contains `return` and finally, the `Delay` member allows building monads that are executed lazily.

The computation expression block is just a syntactic extension that makes it possible to write a code that uses the monadic operations, but is similar to an ordinary F# code. This means that the code inside the computation expression is simply translated to calls to the basic monadic operation, which we looked at earlier. The following example should put some light on the problem, because it shows how the F# compiler translates the code written using the `maybe` block:

```

maybe.Delay(fun () ->
  printf "Enter a"
  maybe.Bind(readNum(), fun a ->
    printf "Enter b"
    maybe.Bind(readNum(), fun b ->
      maybe.Return(a + b)))

```

As we can see, the original code is split into single expressions and these are evaluated separately as arguments of the monadic operations. It is also important to note that the expression may not be evaluated, because this depends on the behavior of the monadic operation.

For example, let's analyze the third line, where a first call to the `Bind` operation occurs. The first argument will be evaluated asking for a user input and will produce either `None` or `Some(n)`. The second argument is a function that takes one argument (`a`) and executes the rest of the computation expression. As you can see, the `let` binding in the original code was translated to a call to the `Bind` operation which can perform some additional processing and change the semantics and then assign a value to the variable by calling the given function. Also note that the first argument of the `Bind` operation is a *monadic type* (in the signature presented above it was `M<'a>`), while the argument of the function given as a second argument is ordinary type (unwrapped `'a`). This means that the monadic type can hold some additional information - in our `maybe` monad, the additional information is a possibility of the failure of the operation.

Let's look at the implementation of the `maybe` monad now. The `Bind` operation will test if the first argument is `Some(n)` and then it will call the function given as a second argument with `n` as an argument. If the value of the first argument is `None` the `Bind` operation just returns `None`. The second key operation is `Result` which simply wraps an ordinary value into a monadic type - in our example it will take a value `a` (of type `'a`) and turn it into a value `Some(a)` (of type `M<'a>`):

```
type M<'a> = option<'a>

let bind d f =
    match d with
    | None -> None
    | Some(v) -> f v
let result v = Some(v)
let delay f = f()

type MaybeBuilder() =
    member x.Bind(v, f) = bind v f
    member x.Return(v) = result v
    member x.Delay(f) = delay f

let maybe = MaybeBuilder()
```

In this example we looked at computation expressions and implemented a simple *monadic builder* for representing computations that can fail. We implemented support only for basic language constructs (like `let` and `let!`), but in general the computation expression can allow using constructs like `if`, `try .. when` and other. For more information, please refer to [13]. Computation expressions are very powerful when you want to modify the behavior of the F# code, without changing the semantics of elementary expressions, for example by adding a possibility to fail (as we did in this example), or by executing the code asynchronously (as *asynchronous workflows* [14], which are part of the F# library do).

4.5 F# Meta-Programming and Reflection

The last approach to language oriented programming that I'll present in this overview is using *meta-programming* capabilities of the F# language and .NET runtime. In general the term 'meta-programming' means writing a program that treats code as data and manipulates with it in some way. In F# this technique can be used for translating a code written in F# to other languages or formats that can be executed in some other execution environment or it can be used for analysis of the F# code and for calculating some additional properties of this code.

The meta-programming capabilities of F# and .NET runtime can be viewed as a two separate and orthogonal parts. The .NET runtime provides a way for discovering all the types and top-level method definitions in a running program: this API is called *reflection*. F# *quotations* provide a second part of the full meta-programming support - they can be used for extracting an abstract syntax trees of members discovered using the .NET reflection mechanism (note that the F# *quotations* are a feature of the F# compiler and as such can't be produced by C# or VB compilers).

.NET and F# Reflection

The F# library also extends the .NET `System.Reflection` to give additional information about F# data types – for example we can use the F# reflection library to examine possible values of the `Expr` type (discriminated union) declared earlier:

```
> let exprTy = typeof<Expr>
    match Type.GetInfo(exprTy) with
    | SumType(opts) -> List.map fst opts
    | _ -> [];;
    val it : string list = ["Binary"; "Var"; "Const"]
```

An important part of the .NET reflection mechanism is the use of custom attributes, which can be used to annotate any program construct accessible via reflection with additional metadata. The following example demonstrates the syntax for attributes in F# by declaring `Documentation` attribute (simply by inheriting from the `System.Attribute` base class) and also demonstrates how a static method in a class can be annotated with the attribute:

```
type DocumentationAttribute(doc:string) =
    inherit System.Attribute()
    member x.Documentation = doc

type Demo =
    [<Documentation("Adds one to a given number")>]
    static member AddOne x = x + 1
```

Using the .NET `System.Reflection` library it is possible to examine members of the `Demo` type including reading of the associated attributes (which are stored in the compiled DLL and are available at run-time):

```
> let ty = typeof<Demo>
    let mi = ty.GetMethod("AddOne")
    let at = mi.GetCustomAttributes(typeof<DocumentationAttribute>, false)
    (at.[0] :?> DocumentationAttribute).Doc;;
    val it : string = "Adds one to a given number"
```

F# Quotations

F# quotations form the second part of the meta-programming mechanism, by allowing the capture of type-checked F# expressions as structured terms. There are two ways for capturing quotations – the first way is to use quotation literals and explicitly mark a piece of code as a quotation and the second way is to use `ReflectedDefinition` attribute, which instructs the compiler to store quotation data for a specified top-level member. The following example demonstrates a few simple quoted F# expressions – the quoted expressions are ordinary type-checked F# expressions wrapped between the Unicode symbols « and » (alternatively, it is also possible to use <@ and @>):

```

> « 1 + 1 »
val it : Expr<int>

> « (fun x -> x + 1) »
val it : Expr<int -> int>

```

Quotation processing is usually done on the raw representation of the quotations, which is represented by the non-generic `Expr` type (however the type information about the quoted expression is still available dynamically via the `Type` property). The following example implements a trivial evaluator for quotations. `GenericTopDefnApp` is an active pattern that matches with the use of a function given as a first argument (in this example a plus operator); the `Int32` pattern recognizes a constant of type `int`):

```

> let plusOp = « (+) »
  let rec eval x =
    match x with
    | GenericTopDefnApp plusOp.Raw (_, [l; r]) ->
      (eval l) + (eval r)
    | Int32(n) ->
      n
    | _ ->
      failwith "unknoww construct"
val eval : Expr -> int

> let tst = « (1+2) + (3+4) »
  eval tst.Raw
val it : int = 10

```

Quotation Templates and Splicing

When generating quotations programmatically, it is often useful to build a quotation by combining several elementary quotations into a one, more complex quotation. This can be done by creating a *quotation template*, which is a quotation that contains one or more *holes*. Holes are written using the underscore symbol and define a place, where another quotation can be filled in the template. In the following example, we will look at a template that contains two holes and can be used for generating a quotation that represents addition of two values:

```

> open Microsoft.FSharp.Quotations.Typed;;

> let addTemp1 = « _ + _ »;;
val addTemp1 : (Expr<int> -> Expr<int> -> Expr<int>)

> addTemp1 « 1 » « 2*3 »;;
val it : Expr<int> = « op_Addition (Int32 1)
                    (op_Multiply (Int32 2) (Int32 3)) »

```

In this example, we first open a module `Typed` where the quotation functionality is implemented and on the second line, we create a quotation template `addTemp1`. This template contains two holes and represents an addition of values that will be later filled in these holes. Note that the holes are typed, meaning that the values that can be filled in the template have to be quotations representing an expression of type `int`.

The F# quotations also provide mechanism for splicing values into the quotation tree, which is a useful mechanism for providing input data for programs that evaluate quotations. The operator for splicing values is the Unicode symbol (§) as demonstrated in the following example,

where we use it for embedding a value that represents a database table (the `|>` is a pipelining operator, which applies the argument on the left hand side to the function on the right hand side). This example is based on the FLINQ project, which allows writing database queries in F# and executing them as SQL queries on a database engine:

```
> « §db.Customers
    |> filter (fun x -> x.City = "London")
    |> map (fun x -> x.Name) »
val it : Expr<Seq<string>>
```

In the raw representation, the spliced value can be recognized using the `LiftedValue` pattern, which returns a value of type `obj`, which can contain any F# value. Spliced values can be also created using a `lift` function, which has a signature `'a -> Expr<'a>` and returns a quotation containing a single `LiftedValue` node. Together with quotation templates, the `lift` function can be used instead of the `§` operator mentioned earlier.

Quoting Top-Level Definitions

The second option for quoting F# code is by explicitly marking top-level definitions with an attribute that instructs the F# compiler to capture the quotation of the entire definition body. This option is sometimes called *non-intrusive meta-programming*, because it allows processing of the member body (e.g. translating it to some other language and executing it heterogeneously), but doesn't require any deep understanding of meta-programming from the user of the library. The following code gives a simple example:

```
[<ReflectedDefinition>]
let addOne x =
    x + 1
```

The quotation of a top-level definition (which can be either a function or a class member) annotated using the `ReflectedDefinition` attribute is then made available through the F# quotation library at runtime using the reflection mechanism described earlier, but the member is still available as a compiled code and can be executed.

When a quotation represents a use of a top-level definition it is possible to check if this top-level definition was annotated using the `ReflectedDefinition` attribute and so the quotation of the definition is accessible. This can be done using the `ResolveTopDefinition` function as demonstrated in the following example:

```
let expandedQuotation =
    match (« addOne »).Raw with
    | AnyTopDefnUse(td) ->
        match ResolveTopDefinition(td) with
        | Some(quot) -> quot
        | _ -> failwith "Quotation not available!"
    | _ ->
        failwith "Not a top-level definition use!"
```


Using Active Patterns with Quotations

As already mentioned, the programmatic access to F# quotation trees uses F# active patterns, which allow the internal representation of quotation trees to be hidden while still allowing the use of pattern matching as a convenient way to decompose and analyze F# quotation terms. Active-patterns can be also used when implementing a quotation processor, because they can be used to group similar cases together. In the following example we declare an active pattern that recognizes two binary operations:

```
let plusOp = « (+) »
let minusOp = « (-) »

let (|BinaryOp|_) x =
  match x with
  | GenericTopDefnApp plusOp.Raw (_, [l; r]) -> Some("+", l, r)
  | GenericTopDefnApp minusOp.Raw (_, [l; r]) -> Some("-", l, r)
  | _ -> None

let rec eval x =
  match x with
  | BinaryOp (op, l, r) ->
    if (op = "+") then (eval l) + (eval r)
    else (eval l) - (eval r)
  (* ... *)
```

In this example we declared `BinaryOp` active pattern, which can be used for matching a quotation that represents either addition or subtraction. In a code that processes quotations, grouping of related cases together by using active patterns is very useful, because you can define active patterns for all quotation types that your translator or analyzer can process, factor out all the code that recognizes all the supported cases and keep the translator or analyzer itself very simple.

5 References & Other F# Resources

There are many other places where you can find useful information about F#. First of all, there is an official F# web site [1] where you can find the language specification, documentation and other useful resources. There are also two books written about F# (one already published [2], second will be available soon [3]). Lastly, there are also a lot of community resources including an F# community web site with discussion board [4], wiki [5] and several blogs [6,7,8,9,10]. Finally, there are also some projects developed by the F# community that are available at CodePlex - the first one includes various F# code samples [11, 12] and the second is based on my thesis and tries to solve several web development problems [12].

- [1] Official F# homepage, <http://research.microsoft.com/fsharp/fsharp.aspx>
- [2] Expert F#, Apress 2007, Don Syme, Adam Granicz and Antonio Cisternino
- [3] Foundations of F#, Apress 2007, Robert Pickering
- [4] hubFS: THE place for F#, <http://cs.hubfs.net>
The F# community web site with blogs, forums, etc..
- [5] F# Wiki Homepage, <http://www.strangelights.com/fsharp/wiki>
F# Wiki started by Robert Pickering
- [6] Don Syme's WebLog on F# and Other Research Projects - Blog written by the F# language designer Don Syme, <http://blogs.msdn.com/dsyme>
- [7] Robert Pickering's Strange Blog, <http://strangelights.com/blog>
Blog of the "Foundations of F#" book author
- [8] Granville Barnett, <http://weblogs.asp.net/gbarnett>, Explorations in programming
- [9] Tomas Petricek, <http://tomaspetricek.net/blog>, My blog on F# and various other topics
- [10] F# News (<http://fsharpnews.blogspot.com>) and F#.NET Tutorials (<http://www.ffconsultancy.com/dotnet/fsharp>) by Jon Harrop
- [11] F# Samples, <http://www.codeplex.com/fsharpsamples>
Contains code that demonstrate various F# language features
- [12] F# WebTools, <http://www.codeplex.com/fswebtools>
Project that allows writing client/server Ajax web applications entirely in F#
- [13] Some Details on F# Computation Expressions, Don Syme
<http://blogs.msdn.com/dsyme/archive/2007/09/22/some-details-on-f-computation-expressions-aka-monadic-or-workflow-syntax.aspx>
- [14] Introducing F# Asynchronous Workflows, Don Syme
<http://blogs.msdn.com/dsyme/archive/2007/10/11/introducing-f-asynchronous-workflows.aspx>