F# Research Paper

CSCI 208 – Programming Language Design, Spring 2012

Yifan Ge, Bucknell University

Abstract

The purpose of this paper is to briefly present the background information and basic properties of F# programming language. There are 25 detailed descriptions and 6 examples code included in this article. The background information of F# will be presented in the introduction and other information will be in the following discussions.

Introduction – What is F#?

(When was it created? Who created it? What paradigm(s) is it in? What purpose of creating it? Who is hiring programmers for this language right now and what kind of jobs?)

In the three dominant programming paradigms used today: functional, imperative, and object-oriented programming, functional paradigm is the oldest of the three and is often tend to be used in settings that perform heavy calculation, abstract symbolic processing, or theorem proving [1]. However functional paradigm isn't suitable for general-purpose programming. This makes the imperative and object-oriented language become the dominant language for the last half century. As the powerful Functional Paradigm languages are developing through the years, many of them have gradually embrace aspects of the imperative and object-oriented paradigms, although remaining true to the functional paradigm yet incorporating features needed to easily write any kind of program. F# is a natural successor on this path. It is much more than just a Functional Paradigm language [2].

F# was invented by Dr. Don Syme at 2005 and is now the product of a small but highly dedicated team he heads at Microsoft Research (MSR) in Cambridge, England. To eliminate the drawbacks of traditional functional languages such OCaml and Haskell, F# is a general-purpose programming language for .NET, using a general-purpose runtime rather than a custom runtime as other functional languages. F# smoothly integrates all three major programming paradigms. This means that programmer can choose whichever paradigm works best to solve the problem. Furthermore, F# seamlessly integrates with the .NET framework base class library (BCL), which enables users to do everything that .NET allows [2]. 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". This means that F# uses similar syntax of ML or OCaml, but 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 [1]. The integrated capability of three paradigms

and .NET Framework makes F# unique and much more powerful than other functional languages and other languages uses .NET Framework.

Although it is still under research at MSR at Cambridge, England, F# is not just used for research and academia. It is used for a wide variety of real-world applications whose number is growing rapidly. There is a research record in "Foundations of F#" by Robert Pickering. In the report, Pickering mentioned three areas that F# is being used. First, in Microsoft, both in MSR and throughout the company as a whole, there is a strong presence of F#. According to Ralf Herbrich, from Microsoft Research, F# is mainly used for heavy computation tasks. Outside of Microsoft, F# usage is also rapidly growing. Chris Barwick, who runs hubFS, a popular web site dedicated to F#, said F# is a great scientific and mathematics computing platform and is cornerstone needed for advanced scientific computing. Finally, Jude O'Kelly, a software architect at Derivatives One, a company that sells financial modeling software, point out that Derivatives One used F# because it has the succinct mathematical syntax [2].

After these introductions about the importance of F#, the rest of this article will describe some detailed features with sample codes.

Discussions and Demonstrations

Is it compiled or interpreted?

F# can be both interpreted and compiled. F# has F# interactive, which allow users entering code in console or Visual Studio interactive window. User can compile and execute the code by entering two semicolons to terminate a line or several lines of input. If errors occur, the interpreter prints the error messages. In Visual Studio, users even can select the lines they want to interpret by highlight the lines and press ALT + ENTER. On the other hand, F# can be compiled and executed as well. Users can type the codes text in Visual Studio (without the double-semicolon ending) and press CTRL + F5 to compile and execute the whole file [3].

What types of token does it have?

In F# tokens can be identified by three basic types: keywords, symbols, and literals. Keywords include all the words that indicate the type declaration, iteration, special properties, and etc. Symbols include all the comparators, operations, and etc. Literals is the collection of all the numbers and strings [3].

What are the regular expressions for its tokens?

Expressions in F# can be described in the following categories [3]:

- Conditional Expressions (**if … then … else**)
  Expressions run different branches of code and also evaluates to a different value depending on the Boolean conditions.
- Match Expressions (**match**)
  Expressions provide branching control that is based on the comparison of an expression with a set of patterns.
- Loops Expressions

    - **for … to expression:**
      This expression is used to iterate in a loop over a range of values of a loop variable.
    - **for … in expression:**
      It's a looping construct that is used to iterate over the matches of a pattern in an enumerable collection.
- Object Expressions
  These expressions create new instances of a dynamically created, anonymous object type that is based on an existing base type, interface, or set of interfaces.
- Lazy Computations (**lazy**)
  This expression indicates the computations that are not evaluated immediately, but are evaluated when the result is actually needed.
- Computation Expressions
  These expressions provide a convenient syntax for writing computations that can be sequenced and combined using control flow constructs and bindings.
- Code Quotations
  This enables the users to generate the work with F# code expressions programmatically.

Is the format of F# free or fixed?

On Microsoft Developer Network website, under Code Formatting Guidelines (F#), it is stated that F# language is sensitive to line breaks and indentation, it is not just a readability issue, aesthetic issue, or coding standardization issue to format the code correctly. The code must be formatted correctly in order for it to compile correctly [3]. According to this statement, F# has a very restrict and fixed format of coding despite of type of compilers.

What starts a scope?

Unlike C or Java, F# uses indentation to define its scopes. Each level of nesting and scope are indicated by the indentation level, which is called context. Code in a multiline construct must be indented relative to the opening line of the construct. The column position sets a minimum column, referred to as an offside line, for subsequent lines of code that are in the same context. When the indentation level decreases, the compiler assumes that the context has ended and takes the code to the higher level context [3].

Is string a primitive type?

In F# string is a primitive type. String operators include + and ^. Both of them are used for concatenating strings. Because the string type in F# is actually a .NET Framework String type, all the String members are available. By using Chars property, the string can be return as an array of Unicode characters [3].

What are the primitive types? What implicit type coercion can happen? What explicit coercion can be done?

F# has following primitive types: bool, byte, sbyte, int16, int, uint32, int64, unint64, nativeint, unativeint, char, string, decimal, unit, void, float32(single), and float(double). F# is a strongly typed language. It does not perform implicit casts, not even safe conversions. It requires explicit casts to convert between

datatype. Each of the casting operators has the same name as the name of the destination type. They are shown as following: byte, sbyte, int16, uint16, int32(int), uint32, int64, nativeint, unativeint, float(double), float32(single), decimal, char, enum [3].

What are the ranges of the various primitive types?

F# specifies the size of each primitive variable explicitly in their type names. For example **int16** means that it is a 16-bit signed int. And **uint32** means that it is a 32-bit unsigned int. A regular **int** is regarded as a 32-bit signed int. The following Table 1 shows the detailed range of each primitive number type [3].

Table 1 Primitive number types and ranges [3]

Type	Valid Range
bool	**true** or **false**
byte	[0 , 255]
sbyte	[-128 , 127]
int16	[-32768 , 32767]
uint16	[0 , 65535]
int	[-2147483648 , 2147483647]
uint32	[0 , 4294967295]
int64	[-9223372036854775808 , 9223372036854775807]
uint64	[0 , 18446744073709551615]
float32, single	32 bits
float64, double	64 bits

What is the order or precedence for all the operators?

F# has a large collection of operators. The precedence and associativity are described in the following Table 2. The order is from the highest to lowest.

Table 2 Precedence and Associativity of F# Operators [3]

Operators (Highest to lowest)	Associativity
Prefix operators (+, -, %, %%, &, && , !, ~)	Left
** *op*	Right
* *op*, / *op*, % *op*	Left
- *op*, + *op* (binary)	Left
^ *op*	Right
< *op*, > *op*, =, |*op*, &*op*	Left
&, &&	Left
or, ||	Left

Does it use short circuit evaluation?

The Boolean AND and OR operators in F# perform short-circuit evaluation, which means they evaluate the expression on the right of the operator only when it is necessary to determine the overall result of the expression. The following code demonstrates this property [3].

```
let isPositive x =
    if x>0 then
        printfn "x is positive."
    else
        printfn "x is not positive.";
    printfn "F# is not using short-circuit evaluation.";
    false

```

The executed result the example is shown below [3].

```
val isPositive : int -> bool
val res : string = "F# is using short-circuit evaluation."
```

Does it have exception?

There are two categories of exceptions in F#: .NET exception types and F# exception types. An exception in F# is an object that encapsulates information about an error. The **try…with** expression and **try…finally** expression are available for exception handling. Users can throw an exception object by the raise function or generate a general F# exception with the failwith function. The following sample code is demonstration for using the failwith function and try…with expression [3].

```
let divideFailwith x y =
  if (y = 0) then failwith "Divisor cannot be zero."
  else
    x / y

let testDivideFailwith x y =
  try
     divideFailwith x y
  with
     | Failure(msg) -> printfn "%s" msg; 0

let result1 = testDivideFailwith 100 0
```

The executed result is shown below [3].

```
Divisor cannot be zero.

val result1 : int = 0
```

Can you do multidimensional arrays? How? Are there limits?

F# does support multidimensional arrays. But there is no syntax for writing a multidimensional array literal. To create a two-dimensional array, users can use either the operator **array2D** or the function *Array2D.init* to initialize arrays of two dimensions. However, for higher dimensional arrays, only functions can be used, up to four dimensions. The following code demonstrates how to use the operator array2D and the function Array2D.init to generate a two-dimensional array [3].

```
// Using array2D operator
let my2DArray = array2D [ [ 1; 0]; [0; 1] ]

// Using Array2D.init function
let arrayOfArrays = [| [| 1.0; 0.0 |]; [|0.0; 1.0 |] |]
let twoDimensionalArray = Array2D.init 2 2 (fun i j ->
arrayOfArrays.[i].[j])

// Edit values in the array
twoDimensionalArray.[0,1] <- 1.0
```

The type of two-dimensional array is written as <type>[,], and the type of a three-dimensional array is written as <type>[, ,], and so on [3].

Does your language use references?

F# has reference cells that are storage locations that enable the users to create mutable values with reference semantics. In order to use references, users need to use the ref operator before a value to create a new reference cell. Objects created in F# are using this type of operations. There are a few operation can be done on the references, which are shown in the following Table 3 [3].

Table 3 Operations on reference cells [3]

Operator, member, or field	Description	Type
! (dereference operator)	Returns the underlying value.	'a ref -> 'a
:= (assignment operator)	Changes the underlying value.	'a ref -> 'a -> unit
ref (operator)	Encapsulates a value into a new ref cell.	'a -> 'a ref
Value (property)	Gets or sets the underlying value.	unit -> 'a
contents (record field)	Gets or sets the underlying value.	'a

What kind of parameter passing does it do?

In F#, usually parameters are passed by values. But it also allows users to specify a parameter to pass by references. In order to do this, users need to use **byref** keyword. This means that any changes to the value are retained after the execution of the function. The following code are demonstration of using the **byref** keyword to accomplish pass by reference.

```
type Incrementor(z) =
    member this.Increment(i : int byref) =
       i <- i + z

let incrementor = new Incrementor(1)
let refInt = ref 10
incrementor.Increment(refInt)
printfn "%d" !refInt
```

And the printed result is 11, which means the parameter value changed after the function call [3].

Does it use infix, prefix, postfix, postfix, mixfix operators, function calls, some combo of them?
F# uses infix binary operator just like Java, such as +, -, *, /, **, ^, etc. It also uses prefix operators such as +, -, %, %%, &, &&, !, ~, etc. For all the function calls, F# uses prefixed in form of **function x y z …** .

Can it overload operators?
F# can overload arithmetic operators in a class or record type, and at the global level. The Syntax is as following:

```
// Overloading an operator as a class or record member.
static member (operator-symbols) (parameter-list) =
    method-body
// Overloading an operator at the global level.
let [inline] (operator-symbols) parameter-list =
    function-body
```

When overloading unary operators, such as + and -, user must use a title (~) in the operator-symbol to indicate that the operator is a unary operator and not a binary operator [3]. The following code demonstrates one way of using overloading operators to accomplish vector calculations.

```
open System

// Define a vector class and overloads the - and + operators for vector calculation.
type Vector (x: float, y: float) =
    // Define member data
    member this.x = x
    member this.y = y

    // Overload negate operator
    static member (~-) (v : Vector) =
        Vector (-1.0 * v.x, -1.0 * v.y)

    // Overload - operator
    static member (-) (v1 : Vector, v2 : Vector) =
        Vector (v1.x - v2.x, v1.y - v2.y)

    // Overload + operator
    static member (+) (v1 : Vector, v2 : Vector) =
        Vector (v1.x + v2.x, v1.y + v2.y)

    // Define Vecter ToString
    override this.ToString () =
        this.x.ToString() + ", " + this.y.ToString()

// Demonstration.
let v1 = Vector(1.0, 2.0)
let v2 = Vector(20.0, 10.0)

let v3 = v1 - v2
let v4 = v1 + v2

let v5 = - v2

printfn "Yifan Ge F# Phase-3."
printfn "v1 = [%s]" (v1.ToString())
printfn "v2 = [%s]" (v2.ToString())
printfn "v3 = v1 - v2 = [%s]" (v3.ToString())
printfn "v4 = v1 + v2 = [%s]" (v4.ToString())
printfn "v5 = -v2 = [%s]" (v5.ToString())


let name = Console.ReadLine()
```

How to create list?

F# allows users to create list both using rangers and generators. The following example code with comment demonstrates how to use both rangers and generators to create list in F#.

```
// Rangers have the constructs [start .. end] and [start .. step .. end].
Example:

// 1-10
let numbers = [1 .. 10]
printfn "%A" numbers // Print all elements

// 1-10 odd numbers
let oddNumbers = [1 .. 2 .. 10]
printfn "%A" oddNumbers

// a-h
let charList = ['a' .. 'h']
printfn "%A" charList

// Generators have the construct [for x in collection do ... yield expr],
// and they are much more flexible than ranges. Examples:

// squared 1-10
let numSquared =
    [for a in numbers do
        yield (a * a)]
printfn "%A" numSquared

// vectors with x = 1-3, y = 1-3
let vectors =
    [for a in 1 .. 3 do
     for b in 1 .. 3 do
        yield (a, b)]
printfn "%A" vectors

// numbers can be divided by 15 in 1-100
let fifteenNum =
    [for a in 1 .. 100 do
     if a%3 = 0 && a%5 = 0 then yield a]
printfn "%A" fifteenNum

// 'yield' keyword pushes a single value into a list.
// 'yield!' keyword pushes a collection of values into the list. Example:
let aStringList =
    [for a in 1 .. 5 do
        match a with
        | 3 -> yield! ["hello"; "world"]
        | _ -> yield a.ToString()]
printfn "%A" aStringList
```

How to use list comprehension to reverse a list?
Reversing a list uses F#'s functional paradigm property. By using a nested function, users can easily using one single function to accomplish reversing a list. The detailed example code is shown below.

```
//*********************************************************************************************
// List Comprehension Example: Reverse a List
//*********************************************************************************************

let reverse l =
    let rec loop acc = function
        // when l = [], yield acc
        | [] -> acc
        // when l \= [], add hd to acc, and recursively call reverse of the remainder.
        | hd :: t1 -> loop (hd :: acc) t1
    loop [] l
// 'acc' is an accumulating parameter which holds the new list as we generate it.
// 'loop' is a nested function to hide the implementation details of the function form
clients.

let revNum = reverse [1 .. 10]
printfn "%A" revNum
```

Is it dynamically or statically scoped?
First of all, a language is statically scoped if the body of a method is executed in the environment of the method definition. A language is dynamically scoped is the body of a method is executed in the environment of its execution. The following program is designed to determine and verify weather F# is statically or dynamically scoped. In this program, a global string variable called scopeType and a class called ScopeTest, which has a method Test that prints scopeType variable are defined. Finally, after another value is assigned to scopeType, Test is called. The printed result determines the scoping type.

```
let scopeType = "Statically "

type ScopeTest=
    static member Test() =
        System.Console.Write("F# is ")
        System.Console.Write(scopeType)
        System.Console.WriteLine(" scoped.")

let testScopeType =
    let scopeType = "Dynamically "
    ScopeTest.Test()
```

After the program is executed, the printed result was "F# is Statically scoped." Therefore, F# is statically scoped.

Does it support higher-order functions?
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 [1]. Typical measures of first-class status include the following:

- Can users bind an identifier to the value?
- Can users store the value in a data structure?
- Can users pass the value as an argument in a function call?
- Can users return the value as the value of a function call?

The last two measures are also the definition of higher-order functions [3]. Therefore, F# does support higher-order functions. The following examples designed to demonstrate its properties of higher-order functions.

This program demonstrates passing a function as an argument to another function. It defines a fold right function, an insert function, and an insert sort function. The fold right function takes an function as input and returns an output. In the insert sort case, we pass insert as a parameter to fold right function to accomplish the purpose of insert sort.

```
let rec foldr f z list =
    match list with
    | []        -> z
    | h::tail   -> f h (foldr f z tail)

let rec insert e list =
    match list with
    | []        -> [e]
    | h::tail   -> if e<h then e::h::tail
                   else h::(insert e tail)

let insertionSort l = foldr insert [] l

let l = insertionSort [1;9;7;2;6]

System.Console.WriteLine(sprintf "%A" l)

// The output is the sorted list: [1;2;6;7;9]
```

The follow program demonstrates returning a function as the result of another function. In the program, **checkFor** is defined to be a function that takes one argument, *item*, and returns a new function, which can check whether *item* exist in the pre-defined list [3].

```
let checkFor item =
    let functionToReturn = fun lst ->
                           List.exists (fun a -> a = item) lst
    functionToReturn
```

Conclusion

By adopting properties from three dominant programming paradigms, F# demonstrates itself as a unique and powerful language. With functional programming, F# is easier to test and parallelize and is also extensible in a ways where object-oriented code makes extending difficult. With object-oriented programming, F# is able to interoperate with other .NET languages and use the large library from .NET Framework. Typically, F# uses object for implementing elementary data types, grouping a set of elementary functions that are together used to perform some complicated operation, and also working with object oriented user interface frameworks [1]. With imperative programming constructs, F# is able to support many useful construct such as branching and loop constructs [3]. F# has established itself as the de facto .NET functional programming language because of the quality of its implementation and its superb integration with .NET [2].

References

[1] T. Petricek, "F# Language Overview," 3 11 2007. [Online]. Available: http://tomasp.net/articles/fsharp-i-introduction/article.pdf. [Accessed 5 5 2012].

[2] R. Pickering, Foundations of F#, New York: Springer-Verlag, 2007.

[3] "Microsoft Developer Network," Microsoft Corporation, [Online]. Available: msdn.microsoft.com/en-us/library. [Accessed 6 5 2012].