DZone Refcardz

CONTENTS INCLUDE:

Essential F#

By Chance Coble and Ted Neward

ABOUT F#

F# is Microsoft's most recent language for the .NET platform. Developers who learn to take advantage of this new language's functional and object-oriented features will find new productivity gains and new programming design approaches not easily expressed in "just" objects alone. While functional programming can lead to some mind-bending coding, the basics are quite straightforward and should take little time to master. Functional programming is characterised by concise coding style and explicit modelling of behavior. Because F# also offers a rich set of object oriented features, its integration with other .NET languages such as C# and VB is nearly seamless.

GETTING STARTED

F# is available from the Microsoft F# Developer Center (http://msdn.microsoft.com/en-us/fsharp/default.aspx) if it is not already installed in your version of Visual Studio. It runs on any version of .NET after 2.0.

F# Hello World looks like this:

```
#light
System.Console.WriteLine("This is one hello")
printfn "This is another hello"
```

Compile this file (hello.fs) with the command-line fsc.exe to produce a .NET assembly:

```
fsc hello.fs
```

This produces hello.exe, which can be executed in the usual fashion.

Hot Tip F# is a compiled .NET language, but can also be run as a script interactively. Fire up the F# interactive session (Under the View menu in Visual Studio or "fsi.exe" in your F# installation bin directory). Add two semi-colons ";;" to terminate your interactive expressions.

The "#light" syntax is an artifact of earlier F# versions and will be removed in a future version of the language. The "System.Console.WriteLine" call is the .NET Base Class Library at work. The "printfn" is an F# function that ultimately does the same thing: prints to the console.

LANGUAGE SYNTAX

F# is a combination of both functional and object-oriented programming styles. As such, to the C# or Visual Basic programmer, it can at times seem strikingly similar yet entirely foreign.

F# programs are written as composed expressions, as opposed to a series of imperative statements in C#/VB. Each expression can be named (via the "let" keyword), and referenced as such:

```
let files = System.IO.DirectoryInfo(@"C:\Users\Chance").
GetFiles()
```

This defines the "files" name to a value, in this case an array of FileInfo objects. The actual type of "files" needn't be specified, since the language will infer it from the value returned from the right-hand side.

Hot Tip Types in F# are inferred statically by the compiler. If you are using a file editor in Visual Studio, hold your mouse over any value to see its inferred type.

Type Inference

We refer to the type of a value being inferred because a compile-time process attempts to figure out the types of values on its own, and in the process verify the program is type safe. Type annotations (in the form v:t, meaning v of type t) are not as necessary in a language with type inference, even though the language is still strongly typed.

Generics, which will be familiar to both .NET and Java developers, have taken on an even greater role in F# because of type inference. Notice that for any expressions without type annotations the compiler can just assume generic types. For example, the function below can be genericized to have the following type signature, even though it does not contain any type annotations from the programmer.

```
let f x =
    let y = g x
    h y
```

```
let f (x:'a) : 'b =
    let y:'c = (g:'a->'c) x
    (h:'c->'b) y
```

The compiler does something similar to that labeling automatically. From the generic labeling, determinations can already be made about the types of these functions (assuming the expression is strongly typed). First, notice f and h must yield the same type ('b). Second, the argument f is applied to must be the same type to which g is applied ('a). Using small hints like these, the compiler is often able to completely determine the types in a program with little assistance from the programmer.

Most of type inference in F# can be boiled down to two rules. First, if a function is applied to a value then the compiler may assume that value is the type the function requires. Second, if a value is bound to the result of an expression, then that value is the type the expression yields.

There are a few times when these simple rules aren't quite enough and type annotations must be added to assist the compiler. For example, when arithmetic operators are used, F# is careful not to cast one numeric type to another without explicit instructions from the programmer. That way type inference does not become a burden to intensively numerical computing.

Declare/update mutable value	`let mutable x = 0`
	`x <- x + 1`

Declare/update ref value	`let x = ref 0`
	`x := !x + 1`

Another example is in method overloading. The Write method in System.Console for example has about 18 overloads. Type inference may determine the type that is passed to it, but cannot determine a value's type in the other direction. That information is required by the overloaded method to select the proper logic to dispatch for the method.

Type inference does not just aide concise notation in the face of static typing, it is also a helpful check on functional programs. When you write a piece of code, and intellisense indicates they all have the proper type, it is one more indication that gross errors were not introduced into the program. Using this tool, F# gets much of the concise notation usually only available in dynamic languages while still maintaining a fully static type system.

Basic Syntax

The "let" expression is at the core of F#'s functional syntax, and is used in a variety of ways: defining a function, defining a sequence, and so on. F# uses significant whitespace to mark block beginnings and endings.

Define any value	`let x = 2`
Define a function value	`let f a = a + x`

Define a recursive function	`open System.IO`
	`let rec printSubDirFiles dir =`
	`   let files = Directory.GetFiles dir`
	`   let dirs =  Directory.GetDirectories dir`
	`   printf "%s\n%A\n\n" dir files`
	`   Array.iter printSubDirFiles dirs`

Anonymous function	`fun x -> Console.WriteLine (x.ToString())`

The language also provides traditional imperative looping and iteration constructs, such as "if", "for" and "while". Note that "if/then" and "if/then/else" are slightly different from traditional O-O languages, in that they, like most F# expressions, must yield a value, and so both sides of the "if/then/else" must result in the same type of value.

if/then	`if x=10 then printf "Was 10\n"`
if/then/else	`if x=10 then "Was 10\n" else "Was not 10\n"`

For loop	`for x in xs do`
	`    printf "%s" x.ToString()`

While loop	`let ls =`
	`   System.Collections.Generic.List<int>()`
	`while (ls.Count<10) do`
	`  ls.Add(ls.Count)`

Like most .NET languages, F# also provides some mechanism for organizing code; in fact, F# provides two, modules and namespaces, the latter acting the same way as C#/VB namespaces. Modules provide not only lexical scoping, but also space for module-level values, such as constants, fields and functions.

Basic Code Organization: namespaces, types and modules	`namespace MyFSharpProg`
	`open System.Net`
	`type Foo () =`
	`  member x.GetRequest = WebRequest.`
	`Create`
	`module Main = begin`
	` // values and functions here`
	`end`

While most identifiers in F# are immutable, in accordance with traditional functional programming principles, F# permits the definition and modification of values using the "mutable" keyword, or by taking the reference of the value in question by preceding the value with "ref". Assignment to mutable values is done using the left-hand arrow operator ("<-"). Assignment to "ref" values is done with the ":=" operator. Obtaining the value of a "ref" value uses the "!" operator.

Declare/update mutable value	`let mutable x = 0`
	`x <- x + 1`

Declare/update ref value	`let x = ref 0`
	`x := !x + 1`

Function Composition

Because programs are built as composed expressions, the sequence of program logic is typically defined through function composition. Arguments to a function are evaluated prior to the function body, making programming by composition intuitive for programmers coming from a C#/VB/Java background.

Operators such as >> (for function chaining) and |> (for value piping) allow programmers to chain composed functions together in the same order in which they will be evaluated.

Composing functions to sum 5 largest array values	`  let sumLargestFive:int array->int =`
	`    Array.sort`
	`    >> Array.rev`
	`    >> fun a -> Array.sub a 0 5`
	`    >> Array.sum`

Composing functions starting with a value (piping)	`let tabAndWrite (s:string) =`
	`    s`
	`    |> (+) "\t"`
	`    |> Console.WriteLine`

Functional Types and Data Structures

F# also defines tuples and records, which are simple data-centric constructs useful in situations where full-blown objects would be overkill.

Discriminated unions are similar in concept to enumerated types from C# and Visual Basic, representing a bound set of values; however, discriminated unions can incorporate multiple kinds of types, including tuples and collections. Functional code will make heavy use of all three (tuples, records, and discriminated unions).

Collections

Data rich programming always involves dealing with (often large) collections of information from the filesystem, database, network or other sources. Three tools in F# make this considerably easier than other programming paradigms and languages.

tuples	`let (name,value) = ("Two",2)`

record types	`type Person = {name:string;age:int}`
	`let p = { name="Bob";age=20 }`

discriminated unions	`type WebTree =`
	`   | Page of string * WebTree`
	`   | Address of string`

Collection generators provide an easy way to create sets of data without involving a loop. Generators are available for lists, arrays, or IEnumerables (called "seq" in F#), using the [start.. finish] style syntax. In the case of arrays, the [start..finish] syntax can be used on an existing array index to "slice" the array into a new one with the given range.

Pattern matching	`let rec listFromWebTree wt =`
	`  match wt with`
	`    | Page(url,subTree) ->`
	`           url :  listFromWebTree subTree`
	`    | Address(url) -> [ url ]`

Pattern matching on lists (naming the head and tail with "::")	`let rec containsZero ls = function`
	`  | first::rest -> if first=0 then true`
	`              else containsZero rest`
	`  | [] -> false`

Pattern matching with constants	`let startsWithZero ls = function`
	`  | 0::rest -> true`
	`  | ls -> false`

Pattern matching with when and _	`let startsWithZero ls = function`
	`  | first::rest when first=0 -> true`
	`  | _ -> false`

Sequence expressions result in an IEnumerable that can be consumed by any other language on the .NET platform, and are ideal for lazy collection generation or evaluation. They often use a pattern of the form "seq { for x in col do … yield x done }" to transform collections into their evaluated result sets. Finally, higher order functions (e.g. map, fold, reduce and sum) allow programmers to ditch boiler plate code around collection processing and just pass the body of the iterator to a function. The body is often passed as an anonymous function.

List defined in 2 different ways	`[ 1..10] =`
	`   List.map`
	`        (fun x -> x / 2)`
	`         [for x in 1..10 -> x*2]`

Array defined in 2 different ways	`[|2..2..20|] =`
	`   Array.map ((*) 2)`
	`      [|for x in 1..10 -> x|]`

Array slicing (0 to 9 and 10 to end)	`[|1..20|].[..9]`
	
	`[|1..20|].[10..]`

Sequence Expression (piped into iterator)	`seq`
	`{`
	`  let dirs =`
	`    System.IO.Directory.GetFiles @"C:\"`
	`  for x in dirs do`
	`     yield x }`
	`|> Seq.iter (printf "%s\t")`

Higher order multiply by 10, sum and print	`[1..10]`
	`|> List.map (fun x -> x * 10)`
	`|> List.reduce (fun s x -> s + x)`
	`|> printfn "%d"`

Pattern Matching

Another core construct in the language is pattern-matching, given by the "match" keyword and a series of expressions to match against; with this construct, combined with recursion, F# is able to create stack-centric, thread-friendly versions of traditional looping code:

```
let rec factorial n =
  match n with
  | 0 -> 1
  | v -> v * factorial(v-1)
```

Each case is demarcated by a new vertical "pipe" character, and the result by the right-hand side of the "->". Note that the match clause can introduce new local bindings ("v" in the example), which will be populated for use in the expression evaluation. Pattern matching can also include "guard" expressions, given by "when"clauses. More forms are given in the F# spec.

Pattern matching	`let rec listFromWebTree wt =`
	`  match wt with`
	`    | Page(url,subTree) ->`
	`           url ::  listFromWebTree subTree`
	`    | Address(url) -> [ url ]`

Pattern matching on lists (naming the head and tail with "::")	`let rec containsZero ls = function`
	`  | first::rest -> if first=0 then true`
	`              else containsZero rest`
	`  | [] -> false`

Pattern matching with constants	`let startsWithZero ls = function`
	`  | 0::rest -> true`
	`  | ls -> false`

Pattern matching with when and _	`let startsWithZero ls = function`
	`  | first::rest when first=0 -> true`
	`  | _ -> false`

More Pattern matching with _	`let startsWithZero ls = function`
	`  | 0::_ -> true`
	`  | _ -> false`

Exceptions

F# allows developers to trap exceptions thrown by methods that are called, as well as generate exceptions when desired. There are shortcut functions (e.g. "failwith") defined that provide a concise way to generate exceptions with custom messages.

Using the try/with syntax, developers can use the pattern matching syntax to check the type of the exception and invoke the appropriate logic.

Try catch with pattern match on type of exception	`open System.IO`
	`try`
	`  File.ReadAllText(@"C:\dir\myfile.txt")`
	`with`
	`| :? DirectoryNotFoundException as ex`
	`     -> "Dir does not exist"`
	`| ex -> ex.Message`

always run code after exception block	`Try`
	`  File.ReadAllText(@"C:\dir\myfile.txt")`
	`finally`
	`  printf "Always run this\n"`

Function to throw FailureException	`failwith "Operation Failed"`

OOP IN F#

F# is a full object-oriented language in the .NET ecosystem, meaning it knows not only how to use but also how to define new class types with the full range of O-O features: fields, methods, properties, events, interfaces, inheritance, and so on.

Defining a simple class type:

```
type Person(fn:string,ln:string,a:int) =
    member p.FirstName = fn
    member p.LastName = ln
    member p.Age = a
    override p.ToString() =
        String.Format("{0} {1})",
            p.FirstName, p.LastName)
```

The constructor is in the type declaration line, and that type is intrinsically immutable (that is, the properties FirstName, LastName and Age all have get access but not set). This is in line with traditional functional principles. To create mutable members, the F# "mutable" keyword must appear on the member to be mutable, and the explicit "get" and "set" members for each property must be established.

Types in F# can inherit from base classes or interfaces, using the "override" keyword to override inherited members:

```
type Student(fn:string,ln:string,a:int)=
    inherit Person(fn, ln, a)
    override s.ToString() = ...
```

F# can also create object expressions, which are anonymously-defined types that inherit from an existing class or interface; in many cases, this will be much quicker and easier than creating a new type.

Classes and Object Expression

Class definition	`type AccessCounter() =`
	`    let mutable i = 0`
	`    member me.Access () = i <- i + 1`
	`    member me.Count`
	`    with get () = i`
	`         set v  = i <- v`

Multiple constructors	`type Person`
	`   (fn:string,ln:string,age:int) =`
	`    new (age:int) =`
	`    Person("John","Doe",age)`
	`    member p.FirstName = fn`
	`    member p.LastName = ln`
	`    member p.Age = age`
	`    override p.ToString() =`
	`     String.Format("{0} {1})",`
	`            p.FirstName, p.LastName)`

Abstract methods for an interface and its construction	`type Shape =`
	`   abstract Area: unit -> float`
	`type Rectangle(l,w) =`
	`  interface Shape with`
	`    member me.Area () = l * w`

Abstract methods for an abstract class and its construction	`[<AbstractClass>]`
	`type Shape(nm:string) =`
	`   member me.Name = nm`
	`   abstract Area : unit -> float`
	`type Rectangle(l,w) =`
	`   inherit Shape("Rectangle")`
	`   override me.Area() = l * w`

Augmenting Record types with methods	`type PersonRec =`
	`  {fn:string;ln:string;age:int}`
	`  with`
	`    member`
	`me.AgePlusOne = me.age + 1`

Augmenting Discriminated Unions with methods	`type PersonKinds =`
	`  | Male of Person`
	`  | Female of Person`
	`  with`
	`    member me.AgePlusOne =`
	`      match me with`
	`       | Male p -> p.Age + 1`
	`       | Female p -> p.Age + 1`

Instantiating the type and applying the method	`(Male (Person("John","Doe",24)))`
	`  .AgePlusOne`

Augmenting existing types with methods	`type Person`
	`    with`
	`       member x.AgePlusOne = x.Age + 1`

Object expression	`let dirty =`
	` { new System.IDisposable with`
	`     member me.Dispose() = () }  // clean up`

F# EXPRESSIONS

Below is a table of F# expressions for reference. Some of these can be typed into the interactive shell. Feel free to fire up the interactive shell and type these in yourself.

Type Notation

value	`x : int`
function that is applied to an int, and yields an int	`f:(int -> int)`
function in a function	`map:('a->'b)->'a list->'b list`
generics	`f:'a -> 'b`
annotate an argument	`let toStr (x:int) = x.ToString()`

F# Interactive Commands

Reference a dll	`#r`	`#r "System.Windows.Forms";;`
Load an F# code file	`#load`	`#load "Module.fs";;`
Add a directory to the search path	`#I`	`#I @"c:\lib";;`
Refer to the last yielded result from an expression	`it`	`printf "%O\n" it`

Time the evaluation of an expression	`#time`	`>#time;;`
		`> [1..10000];;`
		`Real: 00:00:00.028, CPU: 00:00:00.015, GC gen0: 1, gen1: 0, gen2: 0`

Units of Measure

Declare a Unit	`type [<Measure>] seconds`
	`type [<Measure>] meters`

Manipulate unit values	`let velocity`
	`      (d:float<meters>)`
	`      (t:float<seconds>) =`
	`         d / t`

Break the unit	`type [<Measure>] inches`
	
	`velocity 5.0<inches>`
	`         10.0<seconds>`

Async Combinators

Used to create and manipulate async expressions.

Build an async expression using the continuations for continue, cancel and exception	`Async.Primitive (con,can,exn)`

Build an async expression using the begin and end methods provided	`Async.BuildPrimitive`
	`   (b,e)`

Yield an async expression that, when executed, runs the sequence of async expressions in parallel and yields an array of results	`aExprs`
	`|> Async.Parallel`
	`|> printf "%A\n"`

Fork the expression	`let! child = Async.StartChild aExpr`

Fork the expression and yield a threading task that executes the operation.	`let! tsk =`
	`   Async.StartChildAsTask aExpr`

Manipulate the current thread	`sync.SwitchToGuiThread Async.SwitchToNewThread Async.SwitchToThreadPool`

Async Expression Execution

Run the computation of type Async<T> and yield the T	`Async.RunSynchronously aExpr`

Provide three continuations for continue, exception and cancellation to be evaluated when the expression yields a result	`Async.RunWithContinuations`
	
	`   (con,exc,can,aExpr)`

Fire and Forget in the thread pool	`Async.Start`
Yield a Threading Task that executes the computation	`Async.StartAsTask`

Apply a function to every element of a sequence in parallel	`let aMap f xs =`
	`  Seq.map`
	`    (fun x -> async {return f x})`
	`    xs`
	`  |> Async.Parallel`

Active Patterns

Notice that the Active Pattern creates a function view on an object oriented architecture. The point of this approach is to marry functional languages on top of existing object oriented architectures and class libraries. The combination of views on the objects that result in functional types, and pattern matching can make functional programming on an object oriented framework more readable and clear. The approach also encourages object oriented extensions where they make sense to existing functional architectures while minimizing clutter.

Defining Patterns	`let (|Xml|NoXml|)`
	`      (doc:XmlDocument) =`
	`  if doc.InnerXml = ""`
	`  then NoXml`
	`  else Xml(doc.InnerXml)`

Matching Active Patterns	`match xml with`
	`| NoXml -> printf "Doc was empty!"`
	`| Xml(xml) -> printf "%O" xml`

Partial (Incomplete) Active Patterns that read a set of file contents and yield different types, depending on the specific views of the object	`open System.IO`
	`let (|NoFiles|_|) (fs:FileInfo []) =`
	`  if fs.Length=0 then Some ()`
	`  else None`
	`let (|TooManyFiles|_|) n`
	`           (fs:FileInfo []) =`
	`  if fs.Length > n then Some ()`
	`  else None`
	`let (|FilesContents|_|)`
	`      maxReadSize (fs:FileInfo [])`
	`    :byte array array option=`
	`  let buff =`
	`     Array.create maxReadSize 0uy`
	`  fs`
	`  |> Array.map`
	`      (fun file ->`
	`           let len =`
	`               file`
	`               .OpenRead()`
	`               .Read`
	
	`(buff,0,maxReadSize)`
	`               (len,buff))`
	`  |> Array.map`
	`      (fun (len,contents) ->`
	`             Array.init`
	`               len`
	`          (fun i -> contents.[i]))`
	`  |> Some`

Pattern matching piece to consume the active pattern setup above	`let processFiles =`
	`  match files with`
	`  | TooManyFiles 50 () -> ">>oh no!"`
	`  | NoFiles () -> "0 Oh no!"`
	`  | FilesContents 1000 fs ->`
	`      "Got "`
	`       + fs.Length.ToString()`
	`       + " files"`

Error Messages

Included below are some common error messages with F#. The areas that tend to trip up people beginning F# are usually exacerbated by misunderstanding the error messages. But a clear understanding of these error messages makes your F# programming a cinch. If one of these messages pops up when you are trying to compile, or drop some code in the interactive session then use the reference below to clarify the problem. The first column of the table describes the problem, the second column is the error message reference, and the third column gives examples for recreating the error.

F#'s type system does not automatically cast numbers	`This expression has type`
	`         float`
	`but is here used with type`
	`         int`
		`let add x y = x + y`
		`let isum = 1 + 1`
		`let fsum = 1.0 + 1.0`

Values should be eventually typed through inference, or annotation (i.e. generic types can't be instantiated)	`Value restriction. The value 'x' has been inferred to have generic type.`	`let rec f x = f x`
		
		`//Alternatively`
		
		`let id x = x`
		`let arr =`
		`   Array.create`
		`        10`
		`        id`

Different type expectations: Expression is typed as 'b, but 'a was put in the code	`This expression has type 'a but here is used with type 'b`	`let printStr = printf "%s"`
		
		`printStr 10`

Incomplete Pattern Match Warning (unhandled patterns possible)	`Incomplete pattern matches on this expression`	`match x with`
		`| 1 -> "Was one"`
		`| 2 -> "Was two"`
		
		`// The wildcard _ can`
		`// fix this`

Overloaded functions often require type annotations to select the correct method call	`The method 'Write' is overloaded.`	`open System`
		`let print s =`
		`  Console.Write s`

F# RESOURCES

F# - Microsoft Research

http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/

F# - Downloads

http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/release.aspx

Microsoft F# Developer Center

http://msdn.microsoft.com/en-us/fsharp/default.aspx

F# Samples

http://www.codeplex.com/fsharpsamples

ABOUT THE AUTHORS

Chance Coble has been doing functional programming for nearly a decade, begining with Haskell years ago. His interests in programming are primarily in machine intelligence and pattern recognition. He has implemented applications in a number of contexts over the last 10 years including enterprise financial platforms, scientific applications in biology, virtual/augmented reality and most recently biometrics.

Blog: http://leibnizdream.wordpress.com/

Ted Neward is a software architect, consultant, author, and presenter who has consulted for such companies as Intuit and Pacific Bell, and UC Davis. He is the author of "Server-Based Java Programming" (Manning, 2000), and coauthor of "C# in a Nutshell" (O'Reilly, 2002) and "SSCLI Essentials" (O'Reilly, 2003). Ted was a member of the JSR 175 Expert Group. He now frequently speaks on the conference circuit and to user groups all over the world. He continues to develop and teach courses on Java and .NET.

Blog: http://blogs.tedneward.com/
Website: http://www.tedneward.com/

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-81-3
ISBN-10: 1-934238-81-3
50795
9 781934 238813
$7.95

Version 1.0