F# Cheatsheet

Fork me on GitHub

This cheatsheet glances over some of the common syntax of F# 3.0. If you have any comments, corrections, or suggested additions, please open an issue or send a pull request to https://github.com/dungpa/fsharp-cheatsheet.

Comments

Block comments are placed between `(*` and `*)`. Line comments start from `//` and continue until the end of the line.

```
(* This is block comment *)

// And this is line comment
```

XML doc comments come after `///` allowing us to use XML tags to generate documentation.

```
/// The `let` keyword defines an (immutable) value
let result = 1 + 1 = 2
```

Strings

F# `string` type is an alias for `System.String` type.

```
/// Create a string using string concatenation
let hello = "Hello" + " World"
```

Use verbatim strings preceded by `@` symbol to avoid escaping control characters (except escaping `"` by `""`).

```
let verbatimXml = @"<book title=""Paradise Lost"">"
```

We don't even have to escape `"` with triple-quoted strings.

```
let tripleXml = """<book title="Paradise Lost">"""
```

Backslash strings indent string contents by stripping leading spaces.

```
let poem =
    "The lesser world was daubed\n\
     By a colorist of modest skill\n\
     A master limned you in the finest inks\n\
     And with a fresh-cut quill."
```

Basic Types and Literals

DOWNLOADS

F# Cheatsheet in PDF

REFERENCES

F# Language Reference

F# Programming Wikibook

F# tag on StackOverflow

Learn X in Y minutes: F# version

The Old F# Cheat Sheet

F# Quick Guides: Object Oriented Programming

Most numeric types have associated suffixes, e.g., `uy` for unsigned 8-bit integers and `L` for signed 64-bit integer.

```
let b, i, l = 86uy, 86, 86L

val b : byte = 86uy
val i : int = 86
val l : int64 = 86L
```

Other common examples are `F` or `f` for 32-bit floating-point numbers, `M` or `m` for decimals, and `I` for big integers.

```
let s, f, d, bi = 4.14F, 4.14, 0.7833M, 9999I

val s : float32 = 4.14f
val f : float = 4.14
val d : decimal = 0.7833M
val bi : System.Numerics.BigInteger = 9999
```

See Literals (MSDN) for complete reference.

Functions

The `let` keyword also defines named functions.

```
let negate x = x * -1
let square x = x * x
let print x = printfn "The number is: %d" x

let squareNegateThenPrint x =
    print (negate (square x))
```

Pipe and composition operators

Pipe operator `|>` is used to chain functions and arguments together. Double-backtick identifiers are handy to improve readability especially in unit testing:

```
let ``square, negate, then print`` x =
    x |> square |> negate |> print
```

This operator is essential in assisting the F# type checker by providing type information before use:

```
let sumOfLengths (xs : string []) =
    xs
    |> Array.map (fun s -> s.Length)
    |> Array.sum
```

Composition operator `>>` is used to compose functions:

```
let squareNegateThenPrint' =
```

```
    square >> negate >> print
```

Recursive functions

The `rec` keyword is used together with the `let` keyword to define a recursive function:

```
let rec fact x =
    if x < 1 then 1
    else x * fact (x - 1)
```

Mutually recursive functions (those functions which call each other) are indicated by `and` keyword:

```
let rec even x =
   if x = 0 then true
   else odd (x - 1)

and odd x =
   if x = 1 then true
   else even (x - 1)
```

Pattern Matching

Pattern matching is often facilitated through `match` keyword.

```
let rec fib n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | _ -> fib (n - 1) + fib (n - 2)
```

In order to match sophisticated inputs, one can use `when` to create filters or guards on patterns:

```
let sign x =
    match x with
    | 0 -> 0
    | x when x < 0 -> -1
    | x -> 1
```

Pattern matching can be done directly on arguments:

```
let fst' (x, _) = x
```

or implicitly via `function` keyword:

```
/// Similar to `fib`; using `function` for pattern matching
let rec fib' = function
    | 0 -> 0
    | 1 -> 1
```

```
    | n -> fib' (n - 1) + fib' (n - 2)
```

For more complete reference visit Pattern Matching (MSDN).

Collections

Lists

A list is an immutable collection of elements of the same type.

```
// Lists use square brackets and `;` delimiter
let list1 = [ "a"; "b" ]
// :: is prepending
let list2 = "c" :: list1
// @ is concat
let list3 = list1 @ list2

// Recursion on list using (::) operator
let rec sum list =
    match list with
    | [] -> 0
    | x :: xs -> x + sum xs
```

Arrays

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements.

```
// Arrays use square brackets with bar
let array1 = [| "a"; "b" |]
// Indexed access using dot
let first = array1.[0]
```

Sequences

A sequence is a logical series of elements of the same type. Individual sequence elements are computed only as required, so a sequence can provide better performance than a list in situations in which not all the elements are used.

```
// Sequences can use yield and contain subsequences
let seq1 =
    seq {
        // "yield" adds one element
        yield 1
        yield 2

        // "yield!" adds a whole subsequence
        yield! [5..10]
    }
```

Higher-order functions on collections

The same list `[ 1; 3; 5; 7; 9 ]` or array `[| 1; 3; 5; 7; 9 |]` can be generated in various ways.

- Using range operator `..`

```
let xs = [ 1..2..9 ]
```

- Using list or array comprehensions

```
let ys = [| for i in 0..4 -> 2 * i + 1 |]
```

- Using `init` function

```
let zs = List.init 5 (fun i -> 2 * i + 1)
```

Lists and arrays have comprehensive sets of higher-order functions for manipulation.

- `fold` starts from the left of the list (or array) and `foldBack` goes in the opposite direction

```
let xs' = Array.fold (fun str n ->
            sprintf "%s,%i" str n) "" [| 0..9 |]
```

- `reduce` doesn't require an initial accumulator

```
let last xs = List.reduce (fun acc x -> x) xs
```

- `map` transforms every element of the list (or array)

```
let ys' = Array.map (fun x -> x * x) [| 0..9 |]
```

- `iter`ate through a list and produce side effects

```
let _ = List.iter (printfn "%i") [ 0..9 ]
```

All these operations are also available for sequences. The added benefits of sequences are laziness and uniform treatment of all collections implementing `IEnumerable<'T>`.

```
let zs' =
    seq {
        for i in 0..9 do
            printfn "Adding %d" i
            yield i
    }
```

Tuples and Records

A tuple is a grouping of unnamed but ordered values, possibly of different types:

```
// Tuple construction
let x = (1, "Hello")

// Triple
let y = ("one", "two", "three")

// Tuple deconstruction / pattern
let (a', b') = x
```

The first and second elements of a tuple can be obtained using `fst`, `snd`, or pattern matching:

```
let c' = fst (1, 2)
let d' = snd (1, 2)

let print' tuple =
    match tuple with
    | (a, b) -> printfn "Pair %A %A" a b
```

Records represent simple aggregates of named values, optionally with members:

```
// Declare a record type
type Person = { Name : string; Age : int }

// Create a value via record expression
let paul = { Name = "Paul"; Age = 28 }

// 'Copy and update' record expression
let paulsTwin = { paul with Name = "Jim" }
```

Records can be augmented with properties and methods:

```
type Person with
    member x.Info = (x.Name, x.Age)
```

Records are essentially sealed classes with extra topping: default immutability, structural equality, and pattern matching support.

```
let isPaul person =
    match person with
    | { Name = "Paul" } -> true
    | _ -> false
```

Discriminated Unions

Discriminated unions (DU) provide support for values that can be one of a number of named cases, each possibly with different values and types.

```
type Tree<'T> =
```

```
    | Node of Tree<'T> * 'T * Tree<'T>
    | Leaf


let rec depth = function
    | Node(l, _, r) -> 1 + max (depth l) (depth r)
    | Leaf -> 0
```

F# Core has a few built-in discriminated unions for error handling, e.g., Option and Choice.

```
let optionPatternMatch input =
   match input with
    | Some i -> printfn "input is an int=%d" i
    | None -> printfn "input is missing"
```

Single-case discriminated unions are often used to create type-safe abstractions with pattern matching support:

```
type OrderId = Order of string

// Create a DU value
let orderId = Order "12"

// Use pattern matching to deconstruct single-case DU
let (Order id) = orderId
```

Exceptions

The `failwith` function throws an exception of type `Exception`.

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

Exception handling is done via `try/with` expressions.

```
let divide x y =
   try
       Some (x / y)
   with :? System.DivideByZeroException ->
       printfn "Division by zero!"
       None
```

The `try/finally` expression enables you to execute clean-up code even if a block of code throws an exception. Here's an example which also defines custom exceptions.

```
exception InnerError of string
exception OuterError of string

let handleErrors x y =
```

```
try
  try
     if x = y then raise (InnerError("inner"))
     else raise (OuterError("outer"))
  with InnerError(str) ->
       printfn "Error1 %s" str
finally
   printfn "Always print this."
```

Classes and Inheritance

This example is a basic class with (1) local let bindings, (2) properties, (3) methods, and (4) static members.

```
type Vector(x : float, y : float) =
    let mag = sqrt(x * x + y * y) // (1)
    member this.X = x // (2)
    member this.Y = y
    member this.Mag = mag
    member this.Scale(s) = // (3)
        Vector(x * s, y * s)
    static member (+) (a : Vector, b : Vector) = // (4)
        Vector(a.X + b.X, a.Y + b.Y)
```

Call a base class from a derived one.

```
type Animal() =
    member __.Rest() = ()

type Dog() =
    inherit Animal()
    member __.Run() =
        base.Rest()
```

Upcasting is denoted by `:>` operator.

```
let dog = Dog()
let animal = dog :> Animal
```

Dynamic downcasting (`:?>`) might throw an `InvalidCastException` if the cast doesn't succeed at runtime.

```
let shouldBeADog = animal :?> Dog
```

Interfaces and Object Expressions

Declare `IVector` interface and implement it in `Vector'`.

```
type IVector =
    abstract Scale : float -> IVector

type Vector'(x, y) =
    interface IVector with
```

```
        member __.Scale(s) =
            Vector'(x * s, y * s) :> IVector
    member __.X = x
    member __.Y = y
```

Another way of implementing interfaces is to use object expressions.

```
type ICustomer =
    abstract Name : string
    abstract Age : int

let createCustomer name age =
    { new ICustomer with
        member __.Name = name
        member __.Age = age }
```

Active Patterns

Complete active patterns:

```
let (|Even|Odd|) i =
    if i % 2 = 0 then Even else Odd

let testNumber i =
    match i with
    | Even -> printfn "%d is even" i
    | Odd -> printfn "%d is odd" i
```

Parameterized active patterns:

```
let (|DivisibleBy|_|) by n =
    if n % by = 0 then Some DivisibleBy else None

let fizzBuzz = function
    | DivisibleBy 3 & DivisibleBy 5 -> "FizzBuzz"
    | DivisibleBy 3 -> "Fizz"
    | DivisibleBy 5 -> "Buzz"
    | i -> string i
```

Partial active patterns share the syntax of parameterized patterns but their active recognizers accept only one argument.

Compiler Directives

Load another F# source file into FSI.

```
#load "../lib/StringParsing.fs"
```

Reference a .NET assembly (`/` symbol is recommended for Mono compatibility).

```
#r "../lib/FSharp.Markdown.dll"
```

Include a directory in assembly search paths.

```
#I "../lib"
#r "FSharp.Markdown.dll"
```

Other important directives are conditional execution in FSI (`INTERACTIVE`) and querying current directory (`__SOURCE_DIRECTORY__`).

```
#if INTERACTIVE
let path = __SOURCE_DIRECTORY__ + "../lib"
#else
let path = "../../../lib"
#endif
```