F# Cheat Sheet

Administrivia

F# is a strict, statically, and strongly typed, multi-paradigm, language where types are inferred. It supports first-order functions and currying.

Roughly,

$$\text{F\#} \approx \text{OCaml} + \text{C\#}$$

- Single-line comments begin with `//`.
- Multi-line comments are enclosed in `(* ··· *)`.
- Here's an example of explicit type annotations.

```
let x : int = 3
let first (x : 'a) (y: 'b) : 'a = x
```

- Being "strongly typed" means that F# does little to no coercions, casts, for you.

```
// 5 / 2.5 (* Crashes: 5 and 2.5 are different types *)
  float 5 / 2.5
≈ 5.0 / 2.5
≈ 2.0
```

F#'s conversion functions are named by the type they convert to; akin to C casts.
  - E.g., `int 23.1` and `int "23"` both yield the integer `23`.
  - `string` is then the traditional "to string" method.

Getting Started

The F# REPL and compiler are named `fsi/fsc` on Windows and `fsharpi/fsharpc` on Mac/Linux. (Running these in Emacs Shell stalls; use `ansi-term` instead!)

Ubuntu	`sudo apt install mono-complete fsharp`
Mac	`brew install mono`

Emacs Setup

```
(use-package fsharp)
(use-package ob-fsharp)
```

The `[<EntryPoint>]` is necessary for using `fsharpc`.

Example Source File

```
module CheatSheet

let myInt = 1972;;

[<EntryPoint>]
let main argv
  = printfn "%s" (string myInt)
    0
```

In a terminal, one runs `fsharpi CheatSheet.fs` to load this script, then `open CheatSheet;;` to have unqualified access to all contents; otherwise type in `CheatSheet.myInt;;` to access items. One may enter multiple lines in the REPL, then execute them by entering `;;`. Use `#quit;;` to leave the REPL.

Execute `fsharpc CheatSheet.fs; mono CheatSheet.exe` to compile the file then run it.

Functions

A function is declared with the `let` keyword —variables are functions of zero arguments. Function & varaible names *must* begin with a lowercase letter, and may use _ or `'`.

- Identifiers may have spaces and punctuation in them if they are enclosed in double-backticks; but no unicode or dashes in-general.

```
let ``this & that`` = 2
```

- Functions are like variables, but with arguments, so the same syntax applies.

```
(* A curried function *)
let f x y = x + y

(* Function application *)
let result = f 10 (2 * 6)

(* Partial application *)
let g x = f x 2
```

```
// Composition
let sum9 = f 4 >> f 5

// Threading: x |> f  ≈  f x
1 |> f 4 |> fun x -> 2 //// ⇒ 2
```

Recursive definitions are marked with the `rec` keyword.

```
let rec fact n
  = if n = 0
    then 1
    else n * fact (n - 1)
```

Here's an example of a higher-order function & multiple local functions & an infix operator & an anonymous function & the main method is parametricly polymorphic.

```
let try_add (bop : 'a -> 'a -> 'a) (test : 'a -> bool)
            (fallback : 'a) (x : 'a) (y : 'a)
  = (* (/@/) x y  =  x /@/ y *)
    let (/@/) x y = bop x y
    let wrap a = if test a then a else fallback
    wrap x /@/ wrap y

699 = try_add (+) (fun a -> a % 3 = 0) (666) (-1) 33
(* The anonymous function uses '=' as Boolean equality. *)

-2 = -2 % 3 (* /Remainder/ after dividing out 3s *)
```

Top level and nested functions are declared in the same way; the final expression in a definition is the return value.

We also have the η-rule: `(fun x -> f x) = f`.

F# has extension methods, like C#. That is, types are "open" —as in Ruby.

```
type System.String with
    member this.IsCool = this.StartsWith "J"

// Try it out.
true = "Jasim".IsCool
```

Booleans

Inequality is expressed with <>.

```
(* false, true, false, true, false, true, true, 1 *)
let x , y = true , false
in x = y, x || y, x && y, x >= y, 12 < 2, "abc" <= "abd"
, 1 <> 2, if x then 1 elif y then 2 else 3
```

Strings

F# strings are not arrays, or lists, of characters as in C or Haskell.

```
"string catenation" = "string " ^ "catenation"

Seq.toList "woah"  // ⇒ ['w'; 'o'; 'a'; 'h']

Printf.printf "%d %s" 1972 "taxi";;

let input = System.Console.ReadLine()
```

Records

Records: Products with named, rather than positional, components.

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

(* Construction *)
let jasim = {Name = "Jasim"; Work = "Farm"}

(* Pattern matching for deconstruction *)
let {Name = who; Work = where} = jasim
    // ⇒ who = "Jasim" && where = "Farm"
let {Name = woah} = jasim  // ⇒ woah = "Jasim"
let go {Name = qx; Work = qy} = qx.Length + 2

(* Or we can use dot notation -- usual projections *)
let go' p = p.Name ^ p.Work

(* Or using explicit casing *)
let go'' x =
  match x with
    | {Name = n} -> n
    | _ -> "Uknown"

(* "copy with update" *)
let qasim = {jasim with Name = "Qasim"}
```

Types are "open", as in Ruby.

```
type Person with
    member self.rank = self.Name.Length

qasim.rank // ⇒ 5
```

Variants and Pattern Matching

Sums, or "variants": A unified way to combine different types into a single type;

- Essentially each case denotes a "state" along with some relevant "data".
- Constructors must begin with a capital letter.
- We may parameterise using OCaml style, `'a`, or/and C# style, `<'a>`.

```
type 'a Expr = Undefined | Var of 'a | Const of int | Sum of Expr<'a> * 'a Expr

let that = Const 2 (* A value is one of the listed cases. *)
```

The tags allow us to *extract* components of a variant value as well as to case against values by inspecting their tags. This is *pattern matching*.

- `match···with···` let's us do case analysis; underscore matches anything.
- Patterns may be guarded using `when`.
- Abbreviation for functions defined by pattern matching: `function cs` ≈ `fun x -> match x with cs`

```
let rec eval = function
    | Undefined  as u           -> failwith "Evil"  (* Throw exception *)
    | Var x                     -> 0 + match x with "x" -> 999 | _ -> -1
    | Const n    when n <= 9    -> 9
    | Sum (l, r)                -> eval l + eval r
    | _                         -> 0 (* Default case *)

4   = eval that
-1  = (Var "nine" |> eval)
999 = eval (Var "x")
0   = eval (Const 10)

(* Type aliases can also be formed this way *)
type myints = int
let it : myints = 3
```

Note that we can give a pattern a name; above we mentioned `u`, but did not use it.

- Repeated & non-exhaustive patterns trigger a warning; e.g., remove the default case above.
- You can pattern match on numbers, characters, tuples, options, lists, and arrays.
  - E.g., `[| x ; y ; z|] -> y`.

Builtins: Options and Choice —these are known as `Maybe` and `Either` in Haskell.

```
type 'a Option = None | Some of 'a
type ('a, 'b) Choice = Choice1Of2 of 'a | Choice2Of2 of 'b
```

See here for a complete reference on pattern matching.

Tuples and Lists

Tuples: Parentheses are optional, comma is the main operator.

```
let mytuple  : int * string * float = (3, "three", 3.0)

(* Pattern matching & projection *)
let (woah0, woah1, woah2) = mytuple
let add_1and4 (w, x, y, z) = w + z
let that = fst ("that", false)

(* A singelton list of one tuple !!!!  *)
let zs = [ 1, "two", true ]

(* A List of pairs *)
['a',0 ; 'b',1 ; 'c', 2]

(* Lists:  type 'a list = [] | (::) of 'a * 'a list  *)
let xs = [1; 2; 3]
[1; 2; 3] = 1 :: 2 :: 3 :: [] (* Syntactic sugar *)

(* List catenation *)
[1;2;4;6] = [1;2] @ [4;6]
(* Pattern matching example; Only works on lists of length 3 *)
let go [x; y; z] = x + y + z
14 = go [2;5;7]
```

```
(* Crashes: Incomplete pattern matching *)
match [1; 2; 3] with
 | []     -> 1
 | [x; y] -> x
 // | (x :: ys) -> x
```

Here is `[0 ; 3 ; 6 ; 9 ; 12]` in a number of ways:

```
  [0..3..14]                                  (* Ranges, with a step    *)
≈ [for i in 0..14 do if i % 3 = 0 then yield i] (* Guarded comprehensions *)
≈ [for i in 0..4 -> 3 * i]                    (* Simple comprehensions  *)
≈  List.init 5 (fun i -> 3 * i)
    (* First 5 items of an "unfold" starting at 0 *)
```

Expected: concat, map, filter, sort, max, min, etc. `fold` starts from the left of the list, `foldBack` starts from the right. `reduce` does not need an inital accumulator.

```
zs |> List.reduce (+)  // ⇒ 9
(* Example of a simple "for loop". *)
[1..10] |> List.iter (printfn "value is %A")
```

Arrays use `[|···|]` syntax, and are efficient, but otherwise are treated the same as lists; Pattern matching & standard functions are nearly identical. E.g., `[| 1; 2 |]` is an array.

Lazy, and infinite, structures are obtained by 'sequences'.

Options

Option: Expressing whether a value is present or not.

```
(* type 'a option = None | Some of 'a *)

let divide x y = if y = 0 then None else Some (x / y)
None = divide 1 0

let getInt ox = match ox with None -> 0 | Some x -> x
2 = getInt (Some 2)
```

Side Effects —Unit Type

Operations whose use produces a side-effect return the `unit` type. This' akin to the role played by `void` in C. A *function* is a sequence of expressions; its *return value* is the value of the final expression —all other expressions are of unit type.

```
(* type unit = () *)
let ex : unit = ()

let myupdate (arr : 'a array) (e : 'a)
        (i : int) : unit
  = Array.set arr i e

let nums = [| 0; 1; 2|]
myupdate nums 33 1
33 = nums.[1]
```

```
let my_io () = printfn "Hello!"

let first x y
  = my_io ()
    let _ = y
    x

let res = first 1972 12
```

Printing & Integrating with C#

We may use the `%A` to generically print something.

```
// ⇒ 1 2.000000 true ni x [1; 4]
printfn "%i %f %b %s %c %A" 1 2.0 true "ni" 'x' [1; 4]
```

Let's use C#'s integer parsing and printing methods:

```
let x = System.Int32.Parse("3")
System.Console.WriteLine("hello " + string x)
```

Reads

- F# Meta-Tutorial
- Learn F# in ~60 minutes —`https://learnxinyminutes.com/`
- F# for Fun & for Profit! – EBook
  - Why use F#? —A series of posts
- Microsoft's .Net F# Guide
  - F# Language Reference
- Learn F# in One Video —Derek Banas' "Learn in One Video" Series
- Real World OCaml —F# shares much syntax with OCaml
- F# Wikibook