F# Cheat Sheet

F# Cheat Sheet

This sheet glances over some of the common syntax of the F# language. It is designed to be kept close for those times when you need to jog your memory on something like loops or object expressions. Everything is designed to be used with the #light syntax directive. If you type any of these lines directly into the interactive command shell, be sure to follow them up with two semicolons ";;".
If you have any comments, corrections or suggested additions please send them to chance@a6systems.com.

1. **Comments**

There are a few different kinds of comments in F#. Comment blocks, which are placed between `(*` and `*)` markers.

Line by line comments which follow `//` until the end of a line and xml doc comments which follow `///` and allow the programmer to place comments in xml tags that can be used to generate xml documents.

2. **Strings**

In F# Code the type string is equivalent to System.String

```
let s = "This is a string"

let hello = "Hello"+" World"
```

Preserve all characters

```
let share = @"\\share"
```

Use escape characters

```
let shareln = "\\\\share\n"
```

3 **Numbers**

type is int16 = System.Int16

```
let int16num = 10s
```

type is int32 = System.Int32

```
let int32num = 10
```

type is int64 = System.Int64

```
let int64num = 10L
```

type is float32, single or System.Single

```
let float32num = 10.0f
```

type is float, double or System.Double

```
let floatnum = 10.0
```

convert to int64

```
let int64frm32 = int64 int32num
```

Other conversion functions:

```
float float32 int int16
```

4 **Tuples**

Construction

```
let x = (1,"Hello")
```

Deconstruction

```
let a,b = x
```

Reconstruction and value reuse

```
let y = (x,(a,b))
```

Reconstruction into a 3 tuple (triple)

```
let z = (x,y,a)
```

Partial deconstruction triple

```
let ((a',b'),y',a'') = z
```

5 **Lists, Arrays, Seqs : Generation**

Creates the list [0 ; 2 ; 4]

```
 let lsinit = List.init 3
           (fun i -> i * 2)
```

Creates same list as above

```
let lsgen = [ 0 ; 2 ; 4]
```

Creates the list [0;2;4;6;8]

```
let lsgen2 = [0 .. 2 .. 8]
```

Can also do above one increment at a time to get [0;1;2;3;4;5;6;7;8]

```
let lsgen2' = [0..8]
```

Creates a list [0.0; 0.5; 1.0; 1.5]

```
let lsgen3 =
   [for i in 0..3 -> 0.5 * float i]
```

Put other steps into a generator

```
let lsgen3' =
     [for i in 0..3 ->
          printf "Adding %d\n" i
          0.5 * float i]
```

Place -1 at the head of a list

```
let inserted = -1 :: lsgen2'
```

Concatenation

```
let concat = lsgen2 @ lsgen2'
```

Create an array [|0 ; 2 ; 4|]

```
let arinit = Array.init 3
               (fun i -> i * 2)
```

Create same array as above

```
let argen = [| 0 ; 2 ; 4|]
```

Create the array [|0;2;4;6;8|]

```
let argen2 = [|0 .. 2 .. 8|]
```

Same as above one increment at a time to get [|0;1;2;3;4;5;6;7;8|]

```
let argen2' = [|0..8|]
```

Create an array [0.0; 0.5; 1.0; 1.5]

```
 let argen3 =
   [|for i in 0..3 -> 0.5 * float i|]
```

Put other computation steps into the generator

```
let argen3' =
     [|for i in 0..3 ->
          printf "Adding %d\n" i
          0.5 * float i|]
```

Creating a seq -- remember these are lazy

```
let s =
   seq { for i in 0 .. 10 do yield i }
```

Illustrate laziness – consume the seq below and note the difference from the generated array.

```
let s2 =
    seq { for i in 0 .. 10 do
             printf "Adding %d\n" i
             yield i }
```

6 **Lists, Arrays, Seqs : Consuming**

"left" fold starts from the left of the list, the "right" fold does the opposite

```
List.fold_left
      (fun state a -> state + 1 ) 0
      [ for i in 0 .. 9 -> true]
```

Reduce doesn't require the starter argument

```
List.reduce_left
     (fun accum a -> accum + a )
     [0..9]
```

Square all of the elements in a list

```
List.map (fun x -> x * x) [1..10]
```

Prints all the items of a list

```
 List.iter
   (fun x -> printf "%d" x) [1..10]
```

Same examples for arrays

```
Array.fold_left
      (fun state a -> state + 1 ) 0
      [| for i in 0 .. 9 -> true|]

Array.reduce_left
      (fun accum a -> accum + a )
      [|0..9|]
```

Squares all the elements in the array

```
Array.map
    (fun x -> x * x) [| 1 .. 10 |]
```

Prints all the items of an array

```
Array.iter
   (fun x -> printf "%d" x)
   [|1..10|]
```

Access all elements of an array from 2 on

```
let arr = [|for i in 0..3 -> i|]
arr.[2..]
```

Access elements between 2 and 4 (inclusive)

```
let arr = [|for i in 0..3 -> i|]
arr.[2..4]
```

Access all elements of an array up to 4

```
let arr = [|for i in 0..3 -> i|]
arr.[..4]
```

Seq also has iter, fold, map and reduce

```
Seq.reduce
    (fun accum a -> accum + a)
    (seq { for i in 0 .. 9 do
           yield i })
```

7 **Arrays: Manipulating**

Array elements can be updated

```
let arrayone = [|0..8|]
  arrayone.[0] <- 9
```

8 **Composition Operators**

the |> operator is very helpful for chaining arguments and functions together

```
let piped = [0..2] |> List.sum
```

the >> operator is very helpful for composing functions

```
open System
let composedWriter =
     string >>
     Console.WriteLine
```

9 **Functions as values**

Create a function of 3 arguments

```
let add x y z = x + y + z
```

Currying example

```
let addWithFour= add 4
```

Apply remaining arguments

```
addWithFour 2 10
```

Take a function as an argument

```
let runFuncTenTimes f a =
   [ for 0..9 -> f a]
```

Return a list of functions as arguments

```
let listOfPrintActions =
    [ for 0 .. 10 ->
          printf "%s\n"]
```

Apply those functions iteratively

```
listOfPrintActions
|> List.iteri  (fun i a -> a i)
```

Anonymous function (applied to 2)

```
(fun x -> x * x) 2
```

Anonymous function (applied to tuple,which is deconstructed inside)

```
let arg = (3,2)
(fun (x,y) -> x * y) arg
```

10 **Union Types**

Discriminated Union

```
 type option<'a> =
    | Some of 'a
    | None
```

Augmented Discriminated Union

```
type BinTree<'a> =
   | Node of
     BinTree<'a> * 'a *
     BinTree<'a>
   | Leaf
  with member self.Depth() =
       match self with
       | Leaf -> 0
       | Node(l,_,r) -> 1 +
                 l.Depth() +
                 r.Depth()
```

11 **Types: Records**

```
type Person = {name:string;age:int}

let paul = {name="Paul";age=35}

let paulstwin =
   {paul with name="jim"}

do printf "Name %s, Age %d"
        paul.name paul.age
```

Augmenting Records

```
type Person = {name:string;age:int}
        with member o.Tupilize() =
             (o.name,o.age)
```

12 **Types: OOP**

Classes

```
type BaseClass()=
   let mutable myIntValue=1
   member o.Number
       with get() = myIntValue
       and  set v = myIntValue<-v
   abstract member
            InheritNum:unit->int
   default o.InheritNum() =
              o.Number + 1
```

Subclass

```
type MyClass() =
  inherit BaseClass()
  let someval = “SomeVal”
  let mutable myIntValue = 1
  member self.SomeMethod(x,y) =
            g x y
  static member StaticMethod(x,y)=
            f x y
  member override o.InheritNum() =
            base.InheritNum()+
            myIntValue
```

Interface

```
type MyAbsFoo =
     abstract Foo:unit->string

type MyFooClass() =
     let mutable myfoo =”Foo”
     member o.MyFoo
            with get () = myfoo
            and   set v = myfoo<-v
     interface MyAbsFoo with
       member o.Foo() = myfoo
     end
```

Object Expressions

```
let foo =
  {new MyAbsFoo with
      member o.Foo()=”Bar”}
```

Augmenting Existing Objects (*note: augmented members only available when augmenting module is opened*)

```
open System.Xml
type XmlDocument() =
     member o.GetInnerXml() =
              self.InnerXml
```

Static Upcasting

```
let strAsObj =
  let str = “Hello”
  str :> obj
```

Dynamic Downcasting

```
let objSub (o:’a when ‘a:>object) =
  o :?> SomeSubType
```

13 **Pattern Matching**

Basic

```
let f (x:option<int>) =
  match x with
  | None -> ()
  | Some(i) -> printf “%d” i
```

As a function definition

```
let f = function
  | None -> ()
  | Some(i) -> printf “%d” i
```

With when operation

```
let f = function
   | None -> ()
   | Some(i) when i=0 -> ()
   | Some(i) when i>0 ->printf“%d”i
```

Common matches on a literal

```
let f x =
  match x with
   | 0 | 1 as y -> f y
   | i -> printf “%d” i
```

Wildcard

```
let f = function
   | 0 | 1 as y -> printf “Nothing”
   | _ -> printf “Something”
```

14 **Exceptions**

```
try
  obj.SomeOp()
with | ex ->
         printf “%s\n” ex.Message
```

With (exception) type test

```
try
  obj.SomeOp()
with
  | :? ArgumentException as ex ->
         printf “Bad Argument:\n”
  | exn -> printf “%s\n” exn.Message
```

Add block that runs whether exception is thrown or not

```
try
  obj.SomeOp()
finally
  obj.Close()
```

Raise an exception in code

-Shorthand

```
let f x =
    if not x.Valid then
      invalid_arg “f:x is not valid”
    else x.Process()
```

-Full

```
let f x =
    if not x.SupportsProcess() then
      raise
       (InvalidOperationException
         (“x must support process”))
    else x.Process()
```

Create your own

```
exception InvalidProcess of string

try
 raise InvalidProcess(“Raising Exn”)
with
 | InvalidProcess(str) ->
      printf “%s\n” str
```

15 **Loops**

```
for i in 0..10 do
  printf “%d” i
done
```

Over an IEnumerable

```
for x in xs do
  printf “%s\n”(x.ToString())
done
```

While

```
let mutable mutVal = 0
while mutVal<10 do
    mutVal <- mutVal + 1
done
```

16 **Async Computations**

(Note: *FSharp.PowerPack.dll* should be referenced in your project – as of the CTP - to get the augmented async methods available in existing IO operations)

Basic computation that returns Async<int> that will yield 1 when executed

```
let basic = async { return 1 }
```

Composing expressions and applying to arguments

```
let compound num =
          async {
                      let! anum = basic
                      return num + anum }
```

Returning existing expressions

```
let composedReturn =
        async { return! compound 2}
```

Creating Primitives with existing Begin/End Async Calls

```
let asyncCall args =
   Async.BuildPrimitive
       ((fun (callback,asyncState) ->
          myService.BeginMethod(args,
                                callback,
                                asyncState)),
          myService.EndMethod)
```

Make your own primitive from scratch

```
let asyncPrimitive args =
        Async.Primitive (fun (con,exn) ->
             let result = runSomething args
             if good result then con result
            else exn result)
```

Other primitives

```
Async.Parallel
Async.Primitive
Async.Catch
```

Making sure I/O threads don’t block

(Note the *MethodAsync* convention in “Expert F#” seems to have changed to *AsyncMethod*)

```
let asyncRead file (numBytes:int)=
  async {
       let inStr = File.OpenRead(file)
       let! data = inStr.AsyncRead numBytes
       return processData(data) }
```

Execution Methods (apply the async computation as an argument to these)

```
Async.Run
Async.Spawn
Async.SpawnFuture
Async.SpawnThenPostBack
```

17 **Active Patterns**

Basic

```
let (|Xml|) doc = doc.InnerXml

let getXml = function
  | Xml(xml) -> xml
```

Multiple Patterns

```
let (|Xml|NoXml|) doc =
      if doc.InnerXml=”” then NoXml
      else Xml(doc.InnerXml)

let getXml = function
      | Xml(xml) -> Some(xml)
      | NoXml ->  None
```

Partial Pattern

```
let (|Xml|_|) doc =
      if doc.InnerXml=”” then None
      else Some(doc.InnerXml)

let getXml = function
      | Xml(xml) -> Some(xml)   //Xml Matched
      | _ -> None              // Xml did not match
```

18 **Compiler Directives and Interop with other .NET Languages**

Make indentation significant in parsing (i.e. turn on light syntax)

```
#light
```

Reference a DLL from another .NET library (interactive F# scripts only – in compiled code use normal interface for reference additions)

```
#r @“.\src\bin\mylib.dll”
```

Include a directory in the reference search (also in interactive scripts only)

```
#I @“[dir path]”
```

For a C# class Foo in a dll with a method ToString(), invoke just as you would an F# class.

```
let foo = Foo()
let s = foo.ToString()
```

To have code run only in when working with the compiled version

```
#if COMPILED
...code
#endif
```

For example, when writing a windowed application that you test in script, but eventually compile to run

```
let window =
   Window(Title=”My Window”)
#if COMPILED
[<STAThread>]
do
  let app = Application in
  app.Run(window) |> ignore
#endif
… later in script (.fsx) file …
window.Show()
```

Version 1.01
You can always get the most recent updates to this cheat sheet from
http://a6systems.com/fsharpcheatsheet.pdf

A6 Systems, LLC is an Austin, TX based company that provides consulting services and F# QuickStart training.