

The F# Handout

F# - A typed functional programming language for .NET – ML language family – Core compatible with OCaml – C# and Haskell influenced – First-class .NET citizen – C# and F# call each other directly – Access to .NET Framework APIs: WPF, DirectX, etc – Libraries developed in F# may be used from other .NET languages

<pre>printfn "Hello World"</pre>	<pre>do printfn "Hello again"</pre>
<pre>open System.Windows.Forms let form = new Form() form.Visible <- true form.TopMost <- true form.Text <- "Hello WinForms!"</pre>	<pre>open System.Windows.Forms let form = new Form(Visible=true, TopMost=true, Text="Hello WinForms!")</pre>
<pre>let one = 1 // let binds let two = 2 // values let three = one + two // and functions let twist x = 10 - x printfn "res = %d" (twist (three+4))</pre>	<pre>// let rec binds recursive functions let rec factorial n = if n=0 then 1 // if... then... else n * factorial (n-1) // else ...</pre>
<pre>let rec highestCommonFactor a b = if a=0 then b elif a<b then highestCommonFactor a (b-a) else highestCommonFactor (a-b) b</pre>	<pre>// 2 args are space separated // use parenthesis only if needed // note if... then... elif... then... else</pre>
<pre>let tupleA = (1,2,3) let tupleB = (1,"fred",3.1415) let swap (a,b) = (b,a) // tuple function</pre>	<pre>let rec hcFactor (a,b) = // arg is tuple if a = 0 then b else if a<b then hcFactor (a,b-a) else hcf2 (a-b,a)</pre>
<pre>let b1 = true let b2 = false let b3 = not b1 && (b2 false)</pre>	<pre>let s1 = "hello" let s2 = "world" let greet = s1 + " " + s2 let greet2 = String.concat " " [s1;s2] sprintf "%s, %d" greet greet2.Length</pre>
<pre>// functional immutable lists let emptyList = [] let threeIntList = [1;2;3] let threeIntListToo = 1 :: [2;3] // cons let list = [4;5;6] @ [1;2;3] // append let firstTen = [1 .. 10]</pre>	<pre>let rec sumList xs = match xs with [] -> 0 y::ys -> y + sumList ys let y = sumList [1;2;3]</pre>
<pre>let arr = Array.create 4 "hello" arr.[1] <- "world" arr.[3] <- "Don Syme"</pre>	<pre>let length = arr.Length let length2 = Array.length arr let firstTwo = arr.[0..1] //slice syntax let firstTwo2 = Array.sub arr 0 2</pre>
<pre>let add2Function x = x + 2 let add2Lambda = fun x -> x + 2 let aboveFive x = x > 4</pre>	<pre>let r1 = List.map add2Function [1;2;3] let r2 = List.map add2Lambda [1;2;3] let r3 = List.map (fun x -> x+2) [1;2;3]</pre>
<pre>let pipeline1 = [1;2;3] > List.map (fun x -> x+4) let pipeline2 = [1;2;3] > List.map add2Lambda > List.filter (fun x -> x>4)</pre>	<pre>//composition pipelines let processor = List.map (fun x -> x+4) >> List.filter (fun x -> x>5) let processor2 = List.map add2Lambda >> List.filter aboveFive</pre>
<pre>type Card = { Name : string; Phone : string; Ok : bool } let cardA = { Name = "Erik Meijer" ; Phone = "555-1212" ; Ok = false } let cardB = { cardA with Phone = "588-2300"; Ok = true }</pre>	<pre>let string_of_card c = c.Name + " Phone: " + c.Phone + (if not c.Ok then "(not OK)" else "") // longer syntax in case of name conflict let card = { new Card with Name = "Erik Meijer" and Phone = "555-1212" and Ok = false }</pre>
<pre>//interface types contain functions type IPeekPoke = abstract member Peek: unit -> int abstract member Poke: int -> unit //Peek takes no args returns int //Poke takes int arg returns nothing</pre>	<pre>let Counter(initialState) = let state = ref initialState { new IPeekPoke with member x.Poke(n) = state := state.Value + n member x.Peek() = state.Value }</pre>
<pre>// ref keyword creates heap pointer. Use := to assign new value and ! to dereference</pre>	

<pre> type WidgetClass(initialState:int) = let mutable state = initialState member x.Poke(n) = state <- state + n member x.Peek() = state member x.HasBeenPoked = (state <> 0) </pre>	<pre> type WodgetClass() = let mutable state = 0 interface IPeekPoke with member x.Poke(n) = state <- state + n member x.Peek() = state member x.HasBeenPoked = (state <> 0) </pre>
<pre> // mutable keyword creates variable whose value can be changed with <- //curry example let add x y = x + y let addTen x = add 10 x let eleven = addTen 1 </pre>	
<pre> type MSEmployee = // discriminated BillGates // union example SteveBalmer Worker of string Lead of string * MSEmployee list let myBoss = Lead("Yasir", [Worker("Chris"); Worker("Matteo")]) </pre>	<pre> let isOdd x = // wildcard example match x with _ when x % 2 = 0 -> false _ when x % 2 = 1 -> true let printGreeting (emp : MSEmployee) = match emp with BillGates -> printfn "Hi, Bill" SteveBalmer -> printfn "Hi, Steve" Worker(name) Lead(name, _) -> printfn "Hi, %s" name // extending union generated warning let getType (x : obj) = match x with :? string -> "x is a string" :? int -> "x is an int" :? Exception -> "x is an exception" </pre>
<pre> let listLength alist = match alist with [] -> 0 // [] is empty list a :: [] -> 1 // a :: b is like car/cdr a :: b :: [] -> 2 _ -> failwith "List is too big!" // sequences can specify infinite series // current value is stored and // sequence computes only next item let allInts = Seq.init_infinite (fun i -> i) </pre>	<pre> List.iter(fun i -> printfn "%d" i) [1 .. 10] Array.map (fun (i : int) -> i.ToString()) [1 .. 10] Seq.fold (fun acc i -> i + acc) 0 { 1 .. 10 } </pre>
<pre> // 'option type' represents two states: 'Some' and 'None'. null is never used type Person = { First : string; MI : string option; Last : string } let billg = { First = "Bill"; MI = Some("H"); Last = "Gates" } let steveb = { First = "Steve"; MI = None; Last = "Bulmer" } </pre>	
<pre> > let x = lazy (printfn "Computed"; 42);; val x : Lazy<int> > let listOfX = [x; x; x];; val listOfX : Lazy<int> list > x.Force();; Computed val it : int = 42 </pre>	<pre> // discriminated union with self reference type InfiniteList = ListNode of int * InfiniteList // infinite list of ones let rec circularList = ListNode(1, circularList) // compiler uses lazy initial. for both </pre>
<pre> // Robert Fischer's Code Challenge (http://enfranchisedmind.com) #light // hash-light simplifies the language syntax from that of OCaml printfn "Enter a number between -1 and 100:" let seed = (new System.Random()).Next 99 let rec evalGuess() = let guess = Int32.of_string(System.Console.ReadLine()) if guess=seed then printfn "Correct!" else if guess<seed then printfn "Too low! Try again:" else printfn "Too high! Try again:" evalGuess() evalGuess() </pre>	

F# for Visual Studio: Provides Codesense, TypeTips, IntelliSense, & MethodTips
Getting F#: Install the free Visual Studio 2008 Shell (Integrated mode) then F#. <http://tinyurl.com/5sxx67> and <http://research.microsoft.com/fsharp/release.aspx>
Tool Support: Command Line Compiler: fsc.exe – Interactive Debugging and REPL: Visual Studio and .NETDebugger – Parsing and Lexing: fslex.exe & fsyacc.exe – Profiling: Any .NET profiler, like CLR profiler – Spring.NET: <http://tinyurl.com/68rh62> – Nhibernate for .NET: <http://www.hibernate.org/343.html>
References: Chris Smith's Blog(!): <http://blogs.msdn.com/chrsmith/default.aspx> – HubFS: <http://cs.hubfs.net/> – Most content from Microsoft F# examples and Chris Smith's blog.