Active Patterns

```
type Complex(x, y) =
    let mutable x = x
    let mutable y = y
    member this.XY with get() = x, y and set((x', y')) = x<-x'; y<-y'
    member this.RO with get() = sqrt (x * x + y * y), atan2 y x
                   and  set((r, o)) = x <- r * cos o; y <- r * sin o
    static member fromXY xy = new Complex(xy)
    static member fromRO ro = let c = new Complex(0., 0.) in c.RO <- ro; c
let (|XY|) (c:Complex) = c.XY
let (|RO|) (c:Complex) = c.RO

let add a b = match (a, b) with
               XY (x1,y1), XY (x2,y2) -> Complex.fromXY (x1+x2, y1+y2)
let mul a b = match (a, b) with
               RO (r1,o1), RO (r2,o2) -> Complex.fromRO (r1*r2, o1+o2)


let (|Sude|Liche|) n = if n&&&1 = 0 then Sude else Liche
match 3 with | Sude -> printfn "sude"
             | Liche -> printfn "liche"


let (|Nasobek5|_|) n = if n%5 = 0 then Some (n/5) else None
match 10 with | Nasobek5 n -> printfn "10=5*%d" n
              | _ -> printf "neni nasobek 5"
let (|NasobekK|_|) k n = if n%k = 0 then Some (n/k) else None
match 21 with | NasobekK 4 n -> printfn "21=4*%d" n
              | NasobekK 3 n -> printfn "21=3*%d" n
```

Quotations

```
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns

type Error = E of float

let rec errorEstAux t (env : Map<_,_>) =
  match t with
    | SpecificCall <@ (+) @> (tyargs, [xe; ye]) ->
        let x,E xerr = errorEstAux xe env
        let y,E yerr = errorEstAux ye env
        x+y, E (xerr + yerr)
    | SpecificCall <@ ( * ) @> (tyargs, [xe; ye]) ->
        let x,E xerr = errorEstAux xe env
        let y,E yerr = errorEstAux ye env
        x*y, E (xerr*abs(y) + yerr*abs(x) + xerr*yerr)
    | Let (v, vd, t) ->
        let vv = errorEstAux vd env
        errorEstAux t (env.Add(v.Name, vv))
    | Double x ->
        x, E 0.0
    | Var v ->
        env.[v.Name]
    | Call (None, MethodWithReflectedDefinition(Lambda(argn,body)), [argv]) ->
        errorEstAux (Expr.Let(argn,argv,body)) env
    | Call (None, MethodWithReflectedDefinition(Lambdas(argn,body)), argv) ->
        let lets = Seq.zip (Seq.concat argn) argv
                   |> Seq.fold (fun b (n,v) -> Expr.Let(n,v,b)) body
        errorEstAux lets env
    | _ -> failwithf "Unrecognized aux expression: %A" t

let rec errorEstRaw t =
  match t with
    | Lambda (x,t) ->
    fun xv -> errorEstAux t (Map.of_list [x.Name, xv])
    | _ -> failwithf "Unrecognized raw expression: %A" t
```

```
let errorEst (t:Expr<float -> float>) = errorEstRaw t.Raw

errorEst <@ fun x -> 4.0 + 4.0 @> (3., E 0.3) |> print_any
errorEst <@ fun x -> x + x * x @> (3., E 0.3) |> print_any
errorEst <@ fun x -> let y = x * x in y * y @> (3., E 0.3) |> print_any

[<ReflectedDefinition>]
let f x = x * x * x + 3. * x * x
errorEst <@ fun x -> f x @> (3., E 0.3) |> print_any


                    Reflexe v .NETu - XML serializace
                    ---------------------------------

open System.Xml
open System.Xml.Serialization

type Item =
  [<field: XmlAttribute("Name"); property: XmlIgnoreAttribute>]
  val mutable name : string
  [<field: XmlAttribute("Prize"); property: XmlIgnoreAttribute>]
  val mutable prize : int

  new(n, p) = { name = n; prize = p }
  new() = Item("", 0)

type Items =
  [<field: XmlArray("Items"); property: XmlIgnoreAttribute>]
  val mutable items : Item[]
  new(i) = { items = i }
  new() = Items([||])

let serialize (i : 'a) =
  let s = new XmlSerializer(typeof<'a>)
  s.Serialize(System.Console.Out, i)

  serialize (new Item("Ahoj", 5))
  serialize (new Items([|new Item("Ahoj", 5); new Item("Cau", 6)|]))

                              Reflexe v F#
                              ------------
open Microsoft.FSharp.Reflection

type ColAttribute(col:int) =
  inherit System.Attribute()
  member x.Col = col

type Item = { [<Col(1)>] p : int;
              [<Col(0)>] n : string }

type CSVSerializer<'csv>() =
  let csvT = typeof<'csv>
  let fields = FSharpType.GetRecordFields csvT
  let col (pi : System.Reflection.PropertyInfo) =
      match pi.GetCustomAttributes(typeof<ColAttribute>,false) with
        | [| (:? ColAttribute as a) |] -> a.Col
        | _ -> failwith "Missing Col attribute of %A" pi
  let perm = fields |> Array.map col
  let fields = let a = Array.zero_create (fields.Length)
               perm |> Array.iteri (fun i j -> a.[j] <- fields.[i] )
               a
```

```
  member this.Serialize what =
    let values = FSharpValue.GetRecordFields (box what)
    let values = let a = Array.zero_create (values.Length)
                 perm |> Array.iteri (fun i j -> a.[j] <- values.[i] )
                 a

    let dump a = printf "%A;" a
    values |> Array.iter dump
    //Nebo
//  fields |> Array.iter (fun fi -> FSharpValue.GetRecordField(what, fi)
//                                  |> dump)
    printf "\n"

  member this.Deserialize (line : string) =
    let values = line.Split [|';'|]
                 |> Array.mapi (fun i str -> match fields.[i].PropertyType with
                                 | ty when ty = typeof<int> -> int(str) |> box
                                 | ty when ty = typeof<string> -> str |> box
                                 | _ -> failwith "Bad type")
    let values = let a = Array.zero_create (values.Length)
                 perm |> Array.iteri (fun i j -> a.[i] <- values.[j] )
                 a
    FSharpValue.MakeRecord (csvT, values)

  let c = new CSVSerializer<Item>()
  c.Serialize { p = 42; n = "bagr" }
  c.Deserialize "bagr;42" |> print_any

FSharpType
member GetFunctionElements : typ:Type -> Type * Type
member GetRecordFields : typ:Type * ?bindingFlags:BindingFlags -> PropertyInfo[]
member GetTupleElements : typ:Type -> Type []
member GetUnionCases : typ:Type * ?bindingFlags:BindingFlags -> UnionCaseInfo []
member IsFunction : typ:Type -> bool
member IsModule : typ:Type -> bool
member IsRecord : typ:Type * ?bindingFlags:BindingFlags -> bool
member IsTuple : typ:Type -> bool
member IsUnion : typ:Type * ?bindingFlags:BindingFlags -> bool
member MakeFunctionType : domain:Type * range:Type -> Type
member MakeTupleType : typ:Type [] -> Type

FSharpValue
member GetRecordField : record:obj * info:PropertyInfo -> obj
member GetRecordFields : record:obj * ?bindingFlags:BindingFlags -> obj []
member GetTupleField : tuple:obj * index:int -> obj
member GetTupleFields : tuple:obj -> obj []
member GetUnionFields : obj:obj * typ:Type * ?bindingFlags:BindingFlags ->
                          UnionCaseInfo * obj []
member MakeFunction : typ:Type * impl:(obj -> obj) -> obj
member MakeRecord : typ:Type * values:obj [] * ?bindingFlags:BindingFlags -> obj
member MakeTuple : tupleElements:obj [] * typ:Type -> obj
member MakeUnion : unioncase:UnionCaseInfo * args:obj [] *
                           * ?bindingFlags:BindingFlags -> obj
```