Řešení domácích úkolů -- cenzor

```
let cenzor (bug : string) (text : string) =
  let kmp = Array.zero_create bug.Length
  let rec next chr i = match i with | i when bug.[i]=chr -> i+1
                                    | i -> if i=0 then 0 else next chr kmp.[i]
  for i = 2 to bug.Length-1 do kmp.[i] <- next bug.[i-1] kmp.[i-1]

  let traceback gs cs =
    let rec drop n = function | [] -> []               | xs when n=0 -> xs
                              | x::xs -> drop (n-1) xs
    let rec shift n xs = function | [] -> xs, []       | ys when n=0 -> xs, ys
                                  | y::ys -> shift (n-1) (y::xs) ys
    gs |> drop (bug.Length-1) |> shift (bug.Length-1) cs |> fun (a,b) -> b, a
  let rec step s gs = function
     | []    -> gs
     | c::cs -> let s' = next c s
                if s' < bug.Length then step s' (c::gs) cs
                else let gs, cs = traceback gs cs in step 0 gs cs

  text |> List.of_seq |> step 0 [] |> List.rev
```

Řešení domácích úkolů -- median

```
let rec kth k xs =
  let kth_bysort k xs = xs |> List.sort compare |> fun xs -> List.nth xs k
  let rec quint_med acc = function
     | a::b::c::d::e::rest -> quint_med (kth_bysort 2 [a;b;c;d;e] :: acc) rest
     | _ -> acc

  match xs with
   | [_] | [_;_] | [_;_;_] | [_;_;_;_] -> kth_bysort k xs
   | xs -> let qm = quint_med [] xs
           let qmmed = kth (List.length qm / 2) qm
           let left, right = xs |> List.partition (fun x -> x < qmmed)
           let left_len = List.length left
           if k < left_len then kth k left else kth (k-left_len) right
```

Řešení domácích úkolů -- med2

```
type ArraySlice<'a>(a : 'a [], l, r) =
  member this.Length = max 0 (r-l)
  member this.Item with get i = a.[l+i]
  member this.Slice(l' , r') = new ArraySlice<'a>(a, l + l', l + r')

let kth k xs ys =
  let rec kth k (xs : ArraySlice<'a>) (ys : ArraySlice<'a>) = match () with
     | _ when xs.Length > ys.Length -> kth k ys xs
     | _ when xs.Length=0 -> ys.[k]
     | _ when k = 0 -> min xs.[0] ys.[0]
     | _ when k = xs.Length+ys.Length-1 -> max xs.[xs.Length-1] ys.[ys.Length-1]
     | _ -> let mx = k * xs.Length / (xs.Length + ys.Length)
            let my = k - mx
            if xs.[mx] < ys.[my]
              then kth (k-mx) (xs.Slice(mx, xs.Length)) (ys.Slice(0, my))
              else kth (k-my) (xs.Slice(0, mx)) (ys.Slice(my, ys.Length))
  kth k (new ArraySlice<'a>(xs,0,xs.Length)) (new ArraySlice<'a>(ys,0,ys.Length)
```

Řešení domácích úkolů -- psrt

```
type Heap(n) =
  let h = Array.zero_create (n+1)
  let mutable len = 0
  let swap i j = let t = h.[i] in h.[i] <- h.[j]; h.[j] <- t

  member this.Count = len
  member this.Add(x) =
    let rec up i = if i>0 && h.[i/2]>h.[i] then swap i (i/2); up (i/2)
    len <- len + 1
    h.[len] <- x
    up len
  member this.Top =
    let rec down i =
      let j = if 2*i + 1 <= len && h.[2*i + 1] < h.[2*i] then 2*i + 1 else 2*i
      if j <= len && h.[j] < h.[i] then swap i j; down j
    let ret = h.[1]
    h.[1] <- h.[len]
    len <- len - 1
    down 1
    ret

let psrt = function
  | [] -> []
  | xs ->
      let rec rev_is_desc acc = function
        | [] -> Some acc
        | x::y::_ when y>x -> None
        | x::xs -> rev_is_desc (x::acc) xs
      let rec sort k (heap:Heap) acc = function
        | [] when heap.Count=0 -> acc
        | [] -> sort k heap (heap.Top :: acc) []
        | x::xs -> let acc' = if heap.Count < k then acc else heap.Top :: acc
                   heap.Add x
                   sort k heap acc' xs
      let rec test k xs = match xs |> sort k (new Heap(k)) [] |> rev_is_desc []
                          with | None -> test (k*k) xs
                               | Some xs -> xs
      test 4 xs
```

Objekty

Vše dědí od typu `obj`.
`box 5` je jako `(5 :> obj)`
`null : obj`
`unbox<int> (box 5)` je jako `(box 5 :?> int)`

Existují čtyři typy 'objektů':

```
type vector = { x : float; y : float } ...
type dog = class ... end
type pet = interface ... end
type pair = struct ... end

type vector =
  { x : float; mutable y : float }
  member v.Len = sqrt (v.x*v.x + v.y*v.y)
  static member SLen (v : vector) = v.Len
  member v.ChangeX x = { v with x = x }
  member internal v.ChangeY y = v.y <- y
  member private v.ChangeXY (x, y) = { x = x; y = y }
  member v.Y with get = 5. and private set = y <- 12.

type vector(x: float, y:float) as v =
  inherit obj()
  let mutable x = x
  let mutable y = y
  do printf "Constructor"
```

```
  static let z = 1
  static do printf "Static constructor"
  member this.Len = sqrt (x*x + y*y)
  new() as v = vector(0., 0.) then printf "helou"

type IInc =
  abstract Inc : int -> int
type IA = abstract A : int

type A() =
  ...
  interface IInc with
    member v.Inc x = x+1

(A :> IInc).Inc



type Arr(n) =
  let a : int[] = Array.zero_create n
  member this.Item
    with get i = a.[i]
    and  set i x = a.[i] <- x

type Test() =
  member this.Max(x) = x
  member this.Max(x,y) = max x y
  [<OverloadID("MaxInt")>]
  member this.Max(x : int, y : int, z : int) = max x y |> max z
  [<OverloadID("MaxFloat")>]
  member this.Max(x : float, y : float, z : float) = min x y |> min z

{ new obj() with member x.ToString() = "Hello, " + base.ToString() }
{ new IInc with member x.Inc a = a+1
  interface IA with member x.A = 5 }

[<AbstractClass>]
type TextOutputSink() =
  abstract WriteChar : char -> unit
  abstract WriteString : string -> unit
  default this.WriteString s = s |> String.iter x.WriteChar

{ new TextOutputSink() with override x.WriteChar c = System.Console.Write c}

{ new System.Collections.Generic.IEnumerator<int> with
    member this.Current = 1
  interface System.Collections.IEnumerator with
    member this.Current = box 1
    member this.MoveNext() = true
    member this.Reset() = failwith "reset"
  interface System.IDisposable with
    member this.Dispose() = () }
```

Generované rovnosti, porovnání a hešování

Všechny generované recordy, uniony, structy a výjimky mají defaultně

```
   override this.Equals(y:obj) = ...
   interface System.IComparable with
       member this.CompareTo(y:obj) : int = ...
   override this.GetHashCode() : int = ...
   interface Microsoft.FSharp.Core.IStructuralHash with
       member this.GetStructuralHashCode(nNumRemaining: int byref) : int = ...
[<ReferenceEquality>]
[<StructuralEquality; StructuralComparison(false)>]
```

Sequence expressions

```
seq { 1 .. 10 }
seq { 1 .. 2 .. 10 }
seq { for x in expr -> expr }
[  seq-expr  ] je Seq.to_list(seq {seq-expr})
[| seq-expr |] je Seq.to_array(seq {seq-expr})

seq { for x in [1..10] do if x%2 = 0 then yield x }

let rec ones = seq { yield 1; yield! ones }
let squares = seq { for i in ones do yield i*i }
let rec from n = seq { yield n; yield! from (n+1) }

let primes =
  let rec sieve xs =
    let p = Seq.hd xs
    seq { yield p
          yield! sieve (Seq.filter (fun x -> x%p <> 0) xs) }
  sieve (from 2)

expr { for pat in enum ... }
expr { let ... }
expr { let! ... }
expr { use ... }
expr { while ... }
expr { yield ... }
expr { yield! ... }
expr { return ... }
expr { return! ... }

builder-expr { cexpr } =
  let b = builder-expr in b.Run (b.Delay(fun () -> {| cexpr |}C))
```

Pokud `Run` nebo `Delay` neexistují, nezavolají se zde

Přepisovací pravidla

```
{| let binds in cexpr |}                 = let binds in {| cexpr |})
{| let! pat = expr in cexpr |}           = b.Bind(expr, (fun pat -> {| cexpr |}))
{| do expr in cexpr |}                   = expr; {| cexpr |}
{| do! expr in cexpr |}                  = b.Bind(expr, (fun () -> {| cexpr |}))
{| yield expr |}                         = b.Yield(expr)
{| yield! expr |}                        = expr
{| return expr |}                        = b.Return(expr)
{| return! expr |}                       = expr
{| use pat = expr in cexpr |}            = b.Using(expr, (fun pat -> {| cexpr |}))
{| use! v = expr in cexpr |}             = b.Bind(expr, (fun v ->
                                                b.Using(v,(fun v -> {| cexpr |})))
{| if expr then cexpr0 |}                = if expr then {| cexpr0 |} else b.Zero()
{| if expr then cexpr0 else cexpr1 |}    = if expr then {|cexpr0|} else {|cexpr1|}
{| match expr with p_i -> cexpr_i |}     = match expr with p_i -> {| cexpr_i |}
{| for pat in expr do cexpr |}           = b.For({|expr|}, (fun pat -> {|cexpr|}))
{| while expr do cexpr |}                = b.While((fun () -> expr), {|cexpr|}Del)
{| try cexpr with p_i -> cexpr_i |}      = b.TryWith({| cexpr |}Del, (fun v ->
                                           match v with | (p_i:exn) -> {|cexpr_i|}
                                                        | _ -> raise exn)
{| try cexpr finally expr |}     = b.TryFinally( {| cexpr |}Del, (fun () -> expr))
{| trans-cexpr0; cexpr1 |}       = b.Combine({| trans-cexpr0 |}, {| cexpr1 |}Del)
{| other-expr0 ; cexpr1 |}       = other-expr; {| cexpr1 |}
{| other-expr |}                 = other-expr; b.Zero()

where {| cexpr |}Del is b.Delay(fun () -> {| cexpr |}).
```