Řešení domácích úkolů -- perm

```
let unrank len n =
  let els, p = Array.init len id, Array.zero_create len
  let rec step i n =
    if i = len then () else
    let ei = n % (len - i)
    p.[i] <- els.[ei]
    els.[ei] <- els.[len - i - 1]
    step (i+1) (n / (len - i))
  step 0 n
  p

let rank (p : int[]) =
  let len = p.Length
  let els, pos = Array.init len id, Array.init len id
  let rec step i n mul =
    if i = len then n else
    let ei = pos.[p.[i]]
    els.[ei] <- els.[len - i - 1]
    pos.[els.[ei]] <- ei
    step (i+1) (n + ei * mul) (mul * (len - i))
  step 0 0 1
```

Řešení domácích úkolů -- mul

```
let mul (a:ResizeArray<byte>) (b:ResizeArray<byte>) =
  let zeroify (d:ResizeArray<byte>) = if d.Count=0 then d.Add(0uy); d else d
  let norm (d:ResizeArray<byte>) =
    let mutable c = 0uy;
    for i = 0 to d.Count-1 do
      d.[i] <- d.[i] + c; c <- d.[i] >>> 4; d.[i] <- d.[i] &&& 15uy
    if c<>0uy then d.Add(0uy)
    while d.Count>0 && d.[d.Count-1]=0uy do d.RemoveAt(d.Count-1)
    zeroify d
  let split (a:ResizeArray<byte>) n =
    ResizeArray.sub a 0 n, ResizeArray.sub a n (a.Count-n) |> zeroify

  let (<<<) (a:ResizeArray<byte>) n =
    ResizeArray.append (ResizeArray.create n 0uy) a
  let (.*) (a:ResizeArray<byte>) (c:byte) =
    a |> ResizeArray.map (( * ) c) |> norm

  let (++) (a:ResizeArray<byte>) (b:ResizeArray<byte>) =
    let a, b = if a.Count > b.Count then a, b else b, a
    a |> ResizeArray.mapi (fun i ai -> if i < b.Count then ai+b.[i] else ai)
      |> norm
  let (--) (a:ResizeArray<byte>) (b:ResizeArray<byte>) =
    let a = a |> ResizeArray.copy
    while a.Count>0 && a.[a.Count-1]=0uy do a.RemoveAt(a.Count-1)
    for i = a.Count-2 downto 0 do a.[i+1] <- a.[i+1]-1uy; a.[i] <- a.[i]+16uy
    a |> ResizeArray.mapi (fun i ai -> if i < b.Count then ai-b.[i] else ai)
      |> norm

  let rec ( ** ) (a:ResizeArray<byte>) (b:ResizeArray<byte>) =
    let a, b = if a.Count > b.Count then a, b else b, a
    if b.Count = 1 then a .* b.[0] else
    let n = a.Count / 2
    let B, A = split a n
    let D, C = split b n
    let AC, BD = A ** C, B ** D
    let ADBC = (A++B) ** (C++D) -- AC -- BD
    (AC <<< n+n) ++ (ADBC <<< n) ++ BD

  zeroify a ** zeroify b
```

```
                                   Události
                                   --------
let t = new System.Timers.Timer(1000.);
t.Elapsed.Add (fun args -> printf "%A\n" args.SignalTime)
t.Start()
t.Stop()
t.Elapsed.AddHandler (fun obj args -> printf "%A %A\n" obj args.SignalTime)
let h = System.Timers.ElapsedEventHandler(fun obj arg -> ...)
t.Elapsed.AddHandler h
t.Elapsed.RemoveHandler h

type IDelegateEvent<'del (requires 'del :> Delegate)> = interface
   abstract member AddHandler : 'del -> unit
   abstract member RemoveHandler : 'del -> unit
end

type IEvent<'del,'a (requires delegate and 'del :> Delegate)> = interface
   inherit IDelegateEvent<'del>
   abstract member Add : event:('a -> unit) -> unit
end

type IEvent<'a> = IEvent<Handler<'a>,'a>

type Handler<'a> = delegate of obj * 'a -> unit
  Má member Invoke : sender:obj * 'a -> unit

Event.create : unit -> ('a -> unit) * IEvent<'a>
Event.listen : ('a -> unit) -> IEvent<'del,'a> -> unit
Event.filter : ('a -> bool) -> IEvent<'del,'a> -> IEvent<'a>
Event.map : ('a -> 'b) -> IEvent<'del,'a> -> IEvent<'b>
Event.scan : ('b -> 'a -> 'b) -> 'b -> IEvent<'del,'a> -> IEvent<'b>
Event.choose : ('a -> 'b option) -> IEvent<'del,'a> -> IEvent<'b>
Event.merge : IEvent<'del1,'a> -> IEvent<'del2,'a> -> IEvent<'a>
Event.partition : ('a -> bool) -> IEvent<'del,'a> -> IEvent<'a> * IEvent<'a>
Event.split : ('a -> Choice<'b,'c>) -> IEvent<'del,'a> -> IEvent<'b>*IEvent<'c>
Event.pairwise : IEvent<'del,'a> -> IEvent<'a * 'a>

form.MouseMove
   |> IEvent.filter (fun args -> args.X > 100)
   |> IEvent.listen (fun args -> printfn "Mouse at %A %A\n" args.X args.Y)

type DelegateEvent<'del (requires 'del :> Delegate)> =
   new : unit -> DelegateEvent<'del>
   member Trigger : args:obj [] -> unit
   member Publish : IDelegateEvent<'del>
type Event<'a> =
   new : unit -> Event<'a>
   member Trigger : arg:'a -> unit
   member Publish : IEvent<'a>
type Event<'del,'a (requires delegate and 'del :> Delegate)> =
   new : unit -> Event<'del,'a>
   member Trigger : sender:obj * a:'a -> unit
   member Publish : IEvent<'del,'a>

open System.Timers
type RandomTicker(approx) =
  let timer = new Timer()
  let rnd = System.Random(42)
  let triggerTickEvent, tickEvent = Event.create ()
  let chooseInterval() = approx * 3 / 2 - rnd.Next(approx)

  do timer.Interval <- chooseInterval()
  do timer.Tick.Add(fun args ->
    triggerTickEvent(timer.Interval)
    timer.Interval <- chooseInterval()

  member this.RandomTick = tickEvent
  member this.Start() = timer.Start()
  member this.Stop() = timer.Stop()
```

Computation expressions

```
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 |}).
```

Probability framework

```
type Probability = float
type Dist<'a> = list<'a * Probability>

let uniform : list<'a> -> Dist<'a> = function
  | [] -> []
  | xs -> let n = xs |> List.length |> float
          xs |> List.map (fun a -> a, 1. / n)
let die = uniform [1..6]

type Event<'a> = 'a -> bool
let ( |?| ) : Event<'a> -> Dist<'a> -> Probability = fun pred d ->
  d |> List.filter (fst >> pred) |> List.map snd |> List.sum
let die6 = ((=) 6) |?| die

let joinWith : ('a->'b->'c) -> Dist<'a> -> Dist<'b> -> Dist<'c> = fun f d d' ->
  [ for (x, p) in d do for (y, q) in d' do yield f x y, p * q]

let prod d d' = joinWith (fun a b -> a,b) d d'
let rec dice = function
  | 0 -> uniform [[]]
  | n -> joinWith (fun a b -> a::b) die (dice (n-1))
let dice66 = ((=) [6; 6]) |?| dice 2

type ProbBuilder() =
  member x.Zero() : Dist<'a> = []
  member x.Return a : Dist<'a> = [a, 1.]
  member x.Bind(d : Dist<'a>, f : 'a -> Dist<'b>) : Dist<'b> =
    [ for (x, p) in d do for (y, q) in f x do yield y, p*q]
let prob = new ProbBuilder()

let joinWith' f d d' = prob { let! x = d
                              let! y = d'
                              return f x y }
let selectOne cs = uniform [ for c in cs do yield c, List.filter ((<>) c) cs ]
let rec select n cs =
  match n with | 0 -> prob { return [], cs }
               | n -> prob { let! x, ds = selectOne cs
                             let! xs, es = select (n-1) ds
                             return x::xs, es }
type color = R | G | B
let selectRGB = ((=) [R; G; B]) |?| select 3 [R; R; G; G; B]
```