Řešení domácích úkolů -- zs

```
let zslog (a : int[]) =
  let rec prod i j acc = if i>j then acc else prod (i+1) j (acc*a.[i])
  let rec zs’ i j acc =
    if i=j then Seq.singleton acc
    else let m = (i + j) / 2
         Seq.append (zs’ i m (acc * prod (m+1) j 1))
                    (zs’ (m+1) j (acc * prod  i  m 1))
  zs’ 0 (a.Length - 1) 1


let zssqrt (a : int[]) =
  let x_1k = a |> Seq.of_array |> Seq.scan ( * ) 1

  let x_kn =
    let rec blck i j x_jn = seq { if i<j then yield! blck i (j-1) (x_jn*a.[j-1])
                                   yield x_jn }
    let blen = a.Length |> float |> sqrt |> int
    let rec x_kn’ j x_jn = seq {
      if j > blen then yield! x_kn’ (j-blen) (Seq.hd (block (j-blen) j x_jn))
      yield! block (max 1 (j-blen+1)) j x_jn
    }
    x_kn’ a.Length 1
  Seq.map2 ( * ) x_1k x_kn
```

Řešení domácích úkolů -- short01

```
let short01 n =
  let pred = Array.create n None

  let rec prohledej zbytky =
     match [ for z in zbytky do
               for (c,y) in [0uy, z*10 % n; 1uy, (z*10+1) % n] do
                 if Option.is_none pred.[y] then
                   pred.[y] <- Some (c, z)
                   yield y
           ] with
     | [] | _ when Option.is_some pred.[0]-> ()
     | dalsi -> prohledej dalsi

  let rec vypis z =
    if z=1 then [1uy]
    else match Option.get pred.[z] with (c,y) -> c :: vypis y

  prohledej [1]
  vypis 0
```

Řešení domácích úkolů -- min1

```
...
  let rec prohledej zbytky =
     match [ for z in zbytky do
               let y = ref z
               while Option.is_none pred.[!y*10 % n] do
                 pred.[!y*10 % n] <- Some (0uy, !y)
                 y := !y*10 % n
                 yield !y
           ] with
     | [] | _ when Option.is_some pred.[0] -> ()
     | dalsi ->
     match [ for z in dalsi do
               if Option.is_none pred.[(z*10+1) % n] then
                 pred.[(z*10+1) % n] <- Some (1uy, z)
                 yield (z*10+1) % n
           ] with
     | [] | _ when Option.is_some pred.[0] -> ()
     | dalsi -> prohledej dalsi
...
```

Computation expressions -- Maybe

```
type MaybeBuilder() =
  member x.Return a = Some a
  member x.Bind(a, f) =
    match a with | None -> None
                 | Some a -> f a
let maybe = new MaybeBuilder()

let inc a = maybe { let! v = a
                    return v + 1 }
```

Computation expressions -- Parser

```
type ’a Parser = char list -> seq<’a * char list>

type ParserBuilder() =
  member x.Return a : ’a Parser = fun s -> Seq.singleton (a, s)
  member x.Bind(a, f) : ’a Parser =
    fun s -> a s |> Seq.map_concat (fun (b, s’) -> s’ |> f b)
let parser = new ParserBuilder()

let char : char Parser = function
  | [] -> Seq.empty
  | s::ss -> Seq.singleton (s, ss)


type ParserBuilder with
  member x.Zero() : ’a Parser = fun s -> Seq.empty
  member x.Delay a = fun s -> Seq.delay (fun () -> a () s)
  member x.Combine(a, b) : ’a Parser = fun s -> Seq.append (a s) (b s)

let sat pred = parser { let! c = char
                        if pred c then return c }
let space = sat System.Char.IsWhiteSpace
let digit = sat System.Char.IsDigit

let rec many p = parser { return! parser { let! r = p
                                           let! rs = many p
                                           return r::rs }
                          return [] }
let spaces = many space
let digits = many digit
let number = parser { let! ds = digits
                      return List.fold_left (fun n d -> n * 10 + int d - int ’0’
) 0 ds }

let addop = parser { let! op = char
                     if op = ’+’ then return (+)
                     if op = ’-’ then return (-) }

let aplusb = parser { let! a = number
                      let! _ = spaces
                      let! op = addop
                      let! _ = spaces
                      let! b = number
                      return op a b }

let parse parser str =
  for res in parser (List.of_seq str) do
    printf "%A %A\n" (fst res) (snd res)

parse aplusb "123 + 223"
```

Continuation passing style

```
let square x = x * x
let squareK x k = x * x |> k

type ContBuilder() =
  member this.Return(x) = fun k -> k x
  member this.Bind(a, f) = fun k -> a (fun l -> (f l) k)

  member this.Zero() = fun k -> k ()
  member this.Delay a = a ()
  member this.Combine(a, b) = fun k -> a (fun () -> b k)
let cont = new ContBuilder()
let runC k = k id

let squareC x = cont { return x*x }
let sqrtC n = cont { if n >= 0 then return n|>float|>sqrt|>int else return -1 }
let compC n = cont { let! k = sqrtC n
                     return! squareC (k+3) }

let callCC f = fun k -> f (fun l -> (fun _ -> k l)) k

let foo n = callCC <| fun k -> cont { let n’ = n*n + 3
                                      if n’ > 20 then return! k "over twenty\n"
                                      return string n’ + "\n" }
```

Výjimky pomocí callCC

```
let sqrtExcept n handler =
  callCC <| fun ok ->
    cont { let! err = callCC <| fun notOk ->
                                      cont { if n < 0. then return! notOk "!!!"
                                             return! ok (sqrt n)
                                           }
           return! handler err
         }

let tryCont k handler =
  callCC <| fun ok ->
    cont { let! err = callCC <| fun notOk -> cont { let! x = k notOk
                                                    return! ok x }
           return! handler err }

type SqrtException = LessThanZero

let sqrtExc n throw = cont { if n < 0. then return! throw LessThanZero
                             return sqrt n }

runC <| tryCont (sqrtExc -3.) (fun n -> print_any n; exit 1)
```