Užitečné definice, které budeme neustále předpokládat (v F# jsou)

```
let (>>) f g x = g (f x)
let (<<) g f x = g (f x)
let (|>) x f = f x
let (<|) f x = f x
```

Ntice

```
type a * b = a, b
type a * b * c = a, b, c

let f x y = x, y
let a, b = 1, 2

let fst (a,b) = a
let fst (a,_) = a
let snd (_,b) = b
```

Pattern matching

```
match co with
 [|] p1 [when cond1] -> e1
  |  p2 [when cond2] -> e2
  ...
  |  pn [when condn] -> en

let rec fib n = match n with
    0 -> 1
  | 1 -> 1
  | j -> fib (j - 2) + fib (j - 1)
```

Pro pohodlí je **function** jako **fun** `foo -> match foo with`

```
let rec fib = function
  | 0 | 1 -> 1
  | j -> fib (j - 2) + fib (j - 1)

let rec nonsense = function
  | 0, n | n, 0 -> n
  | ...

let is_lower = function
  | 'a' .. 'z' -> true
  | _ -> false
```

Neúplný rozbor

```
let is_odd n = match n mod 2 with
    0 -> true
  | 1 -> false
-> Warning P: this pattern-matching is not exhaustive.
   Here is an example of a value that is not matched:
   2
```

Nové datové typy

```
type name = typ
type name = C1 of typ | C2 of typ ...
type 'v1 ... 'vn name = ...
```

option aneb Maybe

```
type 'a option = None | Some of 'a
```

F# definuje

```
Option.is_none, Option.is_some : 'a option -> bool
Option.get : 'a option -> 'a
Option.map : ('a -> 'b) -> 'a option -> 'b option
Option.filter : ('a -> bool) -> 'a option -> 'a option
```

Seznamy

```
type 'a list = [] | 'a :: 'a list
[], 1 :: []
1 :: 2 :: 3 :: []
[1; 2; 3]

let rec len = function
    [] -> 0
  | x :: xs -> 1 + len xs
[1; 2] @ [3; 4]

List.length : 'a list -> int
List.hd : 'a list -> 'a
List.tl : 'a list -> 'a list
List.nth : 'a list -> int -> 'a
List.rev : 'a list -> 'a list
List.append : 'a list -> 'a list -> 'a list            neboli @
List.concat : 'a list list -> 'a list

List.map : ('a -> 'b) -> 'a list -> 'b list
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.for_all : ('a -> bool) -> 'a list -> bool
List.exists : ('a -> bool) -> 'a list -> bool
List.mem : 'a -> 'a list -> bool
List.mem_assoc : 'a -> ('a * 'b) list -> bool
List.assoc : 'a -> ('a * 'b) list -> 'b
List.remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
```

Předchozí funkce používají structural equality. Varianty `memq`, `mem_assq`, `assw` a `remove_assq` používají `(==)`

```
List.filter : ('a -> bool) -> 'a list -> 'a list
List.partition : ('a -> bool) -> 'a list -> 'a list * 'a list
List.split : ('a * 'b) list -> 'a list * 'b list        jmenuje se unzip v F#
List.combine : 'a list -> 'b list -> ('a * 'b) list    jmenuje se zip v F#

List.rev_append : 'a list -> 'a list -> 'a list
List.rev_map : ('a -> 'b) -> 'a list -> 'b list
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Tail rekurze

```
let len2 list =
  let rec len' acc = function
      [] -> acc
    | _ :: xs -> len' (acc + 1) xs
  in len' 0 list
```

Nebo lze použít to, že funkce jsou hodnoty

```
let len3 list =
  let (>>) f g x = g (f x) in
  let rec len' cont = function
      [] -> cont 0
    | _ :: xs -> len' ((+) 1 >> cont) xs
  in len' (fun x -> x) list
```

Value restriction

Proč `let len list = ... in len' [] list` místo eta–redukovaného `let len = ... in len' [] ?`

```
let id x = x;;
let id2 = (id id)           -> val : '_a -> '_a = <fun> *)
let test = id2 5, id2 "5"   -> This expression has type string but is here used with type int
```

V naprosté většině případů stačí použít eta–expanzi, tj.

```
let id2 x = (id id) x   nebo   let id2 x = x |> (id id)
```