Deklarace

```
let e = 2.72

let twopi =
  let pi = 4. * atan 1.
  2. * pi
```

Funkci můžeme vytvořit jako `fun i -> i + 1`

```
let inc = fun i -> i + 1
```

Pro zjednodušení je `let id v1 ... vn = e` zkratka za `let id = fun v1 ... vn -> e`

```
let koreny a b c =
  let d = b * b - 4. * a * c
  -b + sqrt d / 2., -b + sqrt d / 2.
```

Funkce jako argumenty

```
let derivace f x =
    let dx = 1e-5
    in (f (x + dx) - f (x - dx)) / (2. * dx)
-> val derivace : (float -> float) -> float -> float = <fun>

derivace (derivace (fun x -> x * x)) 10.
-> val it : float = 1.99996463834395377
```

Podmínky pomocí `if then else`

```
let is_even n = if n mod 2 = 0 then true else false
```

♣ Rekurze

```
let factorial n = if n = 1 then 1 else n * factorial (n-1)   Nefunguje!
let rec factorial n = if n = 1 then 1 else n * factorial (n-1)

let factorial n =
    let rec fac acc n = if n = 1 then acc else fac (acc*n) (n-1)
    in fac 1 n
```

♣ Předdefinované funkce

```
let id x = x
let ignore x = ()

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

♣ Operátory

```
5 + 5;;
(+) 5 5;;
let inc = (+) 1;;
let (+) = (-);;
```

prefixové operátory: `!něco` nebo `~něco`
infix operátory: `ostatní`
prefix i infix, při definici prefixové varianty začínají ~: `+ +. - -. & && % %%`

Datové deklarace

♣ typová synonyma

```
type dvojice = int * int
type 'a dvojice = 'a * 'a
type dvojice<'a> = 'a * 'a
type ('a, 'b) dvojice = 'a * 'b
type dvojice<'a, 'b> = 'a * 'b
```

♣ výčet hodnot

```
type den = Po = 1
         | Ut = 2
         | St = 3
int Po
enum<den> 1
```

♣ record, neboli struktura s pojmenovanými položkami

```
type bod = { x : float; y : float }
type bod<'a> = { x : 'a
                 y : 'a }
```

♣ algebraické datové typy – "to pravé"

```
type bod = Point of float * float
type tree<'a> = Nil
              | Node of tree<'a> * 'a * tree<'a>
```

F# – předdefinované typy

```
'a -> 'b

int * string     (1, "jedna")
fst : 'a * 'b -> 'a              snd : 'a * 'b -> 'b

'a option   nebo   option<'a>;    type 'a option = Option<'a>
type Option<'T> = None
                | Some of 'T  with member this.IsNone :  bool
                                   member this.IsSome :  bool
                                   member this.Value :  'T
Option.{get, isSome, isNone, count, map, iter, toArray, toList}

'a list nebo list<'a>;            type 'a list = List<'a>
type List<'T> = ( [] )
              | ( :: ) of 'T * 'T list with member this.Head :  'T
                                            member this.Tail :  'T list
[]                                          member this.IsEmpty :  bool
[1; 2; 3] = 1 :: 2 :: 3 :: []               member this.Item (int) :  'T
[1; 2] @ [3]                                member this.Length :  int
[1 .. 10]       [1..3..10]       [1. .. 10.]
[for i in 1..10 -> i*i]
let pairs = [for i in 1..10 -> i, i * i]
[for i in 1..10 do for j in 1..10 do yield i, j]
[for (i, j) in pairs do if isprime i then yield j]

List.{length, hd, tl, init, append, (min, max, sort, sum)[_by]}
List.{filter, map, map2, mapi, mapi2, iter, iter2, iteri, iteri2}
List.{fold, foldBack, fold2, foldBack2, scan, scanBack, reduce, reduceBack}
List.{zip, zip3, unzip, unzip3, concat}
List.{toArray, ofArray, toSeq, ofSeq}
```