```
                                 F# – základní typy
                                 ------------------
bool    System.Boolean   true, false
  && , || : bool -> bool -> bool               not : bool -> bool
  =, !=, <, >, <=, >= : ’a -> ’a -> bool when ’a : equality
  min, max : ’a -> ’a -> ’a when ’a : equality

unit           Core.Unit       ()         void         System.Void
sbyte          System.SByte    0y         byte         System.Byte     0uy
int16          System.Int16    0s         uint16       System.UInt16   0us
int,int32      System.Int32    0          uint32       System.UInt32   0u
int64          System.Int64    0L         uint64       System.UInt64   0UL
nativeint      System.IntPtr   0n         unativeint   System.UIntPtr  0un
float32,single System.Single   0.0f       float,double System.Double   0.0
decimal        System.Decimal  0M
bigint         System.Numerics.BigInteger 0I
  +, -, *, **, /, %, ~- : overloaded
  &&&, |||, ^^^, ~~~, <<<, >>> : overloaded
  abs, acos, asin, atan, atan2, ceil, cos, cosh, exp, floor, log, log10,
  pown, round, sign, sin, sinh, sqrt, tan, tanh, truncate : overloaded
  nan, infinity : double        nanf, infinityf : float
  byte, sbyte, int16, uint16, int, int32, ..., decimal : conversions

char    System.Char    ’a’, ’\t’, ...; konverze pomoci char
string  System.String  "ahoj", "C:\\c", @"C:\c", "abc"B : byte[]
  konverze pomoci string
  + , ^ : string -> string -> string
  System.Text.StringBuilder, metody
    Append, Insert, Remove, Replace, EnsureCapacity, ToString, Chars
  printf, printfn, sprintf

                               F# – strukturované 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>
Option.{get, isSome, isNone, count, map, iter, toArray, toList}

’a list nebo list<’a>;            type ’a list = List<’a>
[], x :: xs, [1; 2; 3], xs @ ys
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}

’a []
[||]; [|1; 2; 3|]
Array.{length, init, create, zeroCreate, append, (min, max, sort, sum)[_by]}
Array.{filter, map, map2, mapi, mapi2, iter, iter2, iteri, iteri2}
Array.{fold, foldBack, fold2, foldBack2, scan, scanBack, reduce, reduceBack}
Array.{zip, zip3, unzip, unzip3, concat}
Array.{toList, ofList, toSeq, ofSeq}
Array.empty<’a> : ’a []

’a [,], ’a [,,], ’a [,,,]
Array[234].{length1, length2, length3, length4, create, zeroCreate, init}
Array[23].{iter, iteri, map, mapi}

’a ref
ref : ’a -> ’a ref
(!) : ’a ref -> ’a
(:=) : ’a ref -> ’a -> unit
incr, decr : ’a ref -> unit
```

```
'a lazy, lazy<'a>;   type 'a lazy = Lazy<'a>
(lazy exp : 'a lazy).Force ()
let force (a : Lazy<'a>) = a.Force ()

'a seq nebo seq<'a>;   type 'a seq = IEnumerable<'a>
IEnumerable<'a> ma funkci GetEnumerator vracejici IEnumerator<'a>
IEnumerator<'a> ma vlastnost Current a funkci MoveNext
Seq.{length, append, concat, filter, fold, hd, skip, skipWhile, take, takeWhile}
Seq.{map, map2, mapi, iter, iter2, iteri, fold, reduce, scan, zip, zip3}
Seq.{(max, min, sort, sum)[_by], cache}
```

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 id x = x
  let d = b * b - 4. * a * c
  -b + sqrt d / 2., -b + sqrt d / 2.                      let ignore x = ()
```

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
```

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
```

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

```
type dvojice = int * int                                   type den = Po = 1
type 'a dvojice = 'a * 'a                                           | Ut = 2
type ('a, 'b) dvojice = 'a * 'b                                     | St = 3
                                                           int Po
type bod = { x : float; y : float }                        enum<den> 1

type tree<'a> = Nil
              | Node of (tree<'a> * 'a * tree<'a>)
```