Struktury

```
type clovek = { name : string; height: float; female: bool; }

let milan = { name = "Milan"; height = 189.5; female = false }
let milan2 = { milan with height = 190.0 }
let jmeno, vyska = milan.name, milan.height
let { name = jmeno, height = vyska } = milan
let is_taller a b = match a,b with { height = a }, { height = b } -> a > b
```

Pozor na to, že jména položek žijí v jenom namespace, takže se předefinovávají:

```
type clovek2 = { name : string; }
```

`milan.name` –> This expression has type clovek but is here used with type clovek2

Side-effekty

Ve skutečnosti mohou mít funkce side–effekty, takže mohou libovolně vypisovat nebo i měnit hodnoty

```
let (;) unit a = a
let ignore x = ()
```

Vstup a výstup

```
print_char : char -> unit, print_string : char -> unit, print_int : int -> unit,
print_float : float->int, print_endline : string->unit, print_newline:unit->unit
```

Je–li místo `print` `prerr`, jde to na standardní chybový výstup

```
read_line : unit -> string, read_int : unit -> int, read_float : unit -> float

type in_channel, type out_channel
val stdin, stdout, stderr : in_channel * out_channel * out_channel
open_{in,out} : string -> {in,out}_channel
close_{in,out} : {in,out}_channel -> unit
input_char : in_channel -> char    output_char : out_channel -> char -> unit
input_line : in_channel -> string  output_string : out_channel -> string -> unit
```

Formátovaný výstup

```
Printf.printf : format -> parametry -> unit
Printf.sprintf : format -> parametry -> string
Printf.fprintf : out_channel -> format -> parametry -> unit
Printf.ksprintf : (string -> 'a) -> format -> parametry -> 'a
```

Další side-effekty -- odkazy

```
val ref : 'a -> 'a ref
val (:=) : 'a ref -> 'a -> unit
val (!) : 'a ref -> 'a

let factorial n =
  let f = ref 1 in
  for k = 2 to n do f := !f * k done;
  !f
```

Nebo `for k = n downto 2 do`

```
let factorial n =
  let f = ref 1 in
  let n = ref n in
  while !n > 0 do
    f := !f * !n;
    n := pred !n
  done;
  !f

type 'a elem = Nil | Elem of 'a * 'a elem ref
let new_elem what = Elem (what, ref Nil)

type 'a queue = 'a elem ref * 'a elem ref
let empty_queue () = (ref Nil, ref Nil)
```

```
let push (first, last) what =
  let e = new_elem what in match !first, !last with
      Nil, _ -> first := e; last := e
    | _, Elem (_, next) -> next := e; last := e
let pop (first, last) = match !first with
    Elem (what, next) -> first := !next; what
```

Pole

```
type 'a array = ...
[|1; 2; 3|] : int array
Array.length : 'a array -> int
[|1; 2; 3|].(0)
Array.of_list, Array.to_list
Array.init : int -> (int -> 'a) -> 'a array
Array.map, Array.fold_left, Array.fold_right, Array.sort, ...
```

Ale pole jsou měnitelná, takže `let z = [|1|] in z.(0) <- 42; z`

Struktury jsou také měnitelné

```
type clovek = { name : string; mutable height : float }
let milan = { name = "Milan"; height = 189. }
milan.height <- 190.;
```

Takže reference nejsou nic jiného než

```
type 'a ref = { mutable contents : 'a }
let ref value = { contents = value }
let (:=) ref value = ref.contents <- value
let (!) ref = ref.contents
```