Řešení domácích úkolů

```
let rec ( *** ) e = function   0 -> 1 | 1 -> e
                            | n -> (e*e) *** (n lsr 1)  *  e *** (n land 1)
```

Nebo efektivněji jako

```
let rec ( *** ) e n =
  let rec pow acc e n =
    if n = 0 then acc
    else pow (if n land 1 = 1 then acc * e else acc) (e * e) (n lsr 1)
  in pow 1 e n


let bitsum n =
  let n = (n lsr  1 land 0x55555555) + (n land 0x55555555) in
  let n = (n lsr  2 land 0x33333333) + (n land 0x33333333) in
  let n = (n lsr  4 land 0x0F0F0F0F) + (n land 0x0F0F0F0F) in
  let n = (n lsr  8 land 0x00FF00FF) + (n land 0x00FF00FF) in
  let n = (n lsr 16 land 0x0000FFFF) + (n land 0x0000FFFF) in n

let bitsum2 n =
  let n = (n lsr  1 land 0x55555555) + (n land 0x55555555) in
  let n = (n lsr  2 land 0x33333333) + (n land 0x33333333) in
  let n = (n lsr  4 land 0x0F0F0F0F) + (n land 0x0F0F0F0F) in
  (n * 0x01010101) lsr 24
```

Opakování

```
module type SetSig = sig
  type 'a set
  val empty : 'a set
  val add : 'a -> 'a set -> 'a set
  val mem : 'a -> 'a set -> bool
end
```

```
module Set : SetSig = struct
  type 'a set = 'a list
  let empty = []
  let add x l = x :: l
  let mem x l = List.mem x l
end
```

Funktory aneb co si doma nezkoušejte

```
module type EqSig = sig
  type t
  val equal : t -> t -> bool
end

module type SetSig = sig
   type elt
   type t
   val empty : t
   val mem :  elt -> t -> bool
   val add :  elt -> t -> t
   val find : elt -> t -> elt
end
```

```
module MakeSet (Equal : EqSig) : SetSig = struct
   open Equal
   type elt = Equal.t
   type t = elt list
   let empty = []
   let mem x s List.exists (equal x) s
   let add = (::)
   let find x s = List.find (equal x) s
end

module StringCaseEqual = struct
  type t = string
  let equal a b = String.lowercase a = String.lowercase b
end
module SSet = MakeSet (StringCaseEqual);;
```

Nefunguje, je třeba, aby `elt` nebyl abstraktní typ:

```
module MakeSet (Equal : EqSig) : SetSig with elt = Equal.t = struct ... end

module type ValueSig = sig type value end
module type MapSig = sig
  type t
  type key
  type value
  val empty : t
  val add : t -> key -> value -> t
  val find : t -> key -> value
end
```

```
module MakeMap (Equal : EqualSig) (Value : ValueSig) : MapSig
    with type key = Equal.t
    with type value = Value.value
= struct
    type key = Equal.t
    type value = Value.value
    type item = Key of key | Par of key * value

    module EqualItem = struct
      type t = item
      let equal (Key a | Pair (a, _)) (Key b | Pair (b, _)) = Equal.equal a b
    end;;
    module Set = MakeSet (EqualItem);;
    type t = Set.t

    let empty = Set.empty
    let add map key value = Set.add (Pair (key, value)) map
    let find map key = match Set.find (Key key) map with
        Pair (_, value) -> value
      | Key _ -> failwith "find"
end
```

V standardní knihovně OCamlu je

```
module type {Set,Map}.OrderedType = sig
  type t
  val compare : t -> t -> int
end
module type Set.S = sig
  type elt type t val empty : t val add : elt -> t -> t ...
end
module type Map.S = sig
  type key type 'a t val empty : 'a t val add : key -> 'a -> 'a t -> 'a t ...
end
module {Set,Map}.Make : functor (Ord : OrderedType) -> {Set,Map}.S
  with type key = Ord.t

module type Hashtbl.HashedType = sig
  type t
  val compare : t -> t -> int
  val hash : t -> int
end
module type Hashtbl.S = sig
  type key type 'a t val create : int->'a t val add : 'a t->key->'a->unit ...
end
module Hashtbl.Make : functor (H : HashedType) -> Hashtbl.S
  with type key = Ord.t
```

Ale existuje i funkce

```
val Hashtbl.hash : 'a -> int
type ('a, 'b) Hashtbl.t
val create : int->('a, 'b) Hashtbl.t val add : ('a, 'b) Hashtbl.t->'a->'b->unit
```

Objekty

```
let circle = object
  val center = (50, 50)
  val radius = 10
  method draw = Graphics.fill_circle (fst center) (snd center) radius
end
circle#draw

circle má typ : <draw : unit>
let poly = object ... method draw = ... end
List.iter (fun item -> item#draw) [circle, poly]

let square = object ... method draw = ... method area = ... end
List.iter (fun item -> item#draw) [circle, poly, square]

type drawable = <draw : unit; ..>
type 'a drawable = <draw : unit; ..> as 'a
List.iter (fun (item:'a drawable) ->item#draw) [circle, poly, square]

let circle center radius = object
  val center = center
  method draw = Graphics.fill_circle (fst center) (snd center) radius
end

let new_collection () = object
  val mutable items = []
  method add item = items <- item :: items
end

let new_collection () = object (self : 'self) nebo jenom (self) nebo (a)
  val mutable items = []
  method add = items <- item :: items
  val add_many n x = if n > 0 then begin
    self#add x;
    self#add_many (n-1) x
  end
end

class circle center radius = object
  method draw = ()
end
let c = new circle (10,20) 30

class type drawable = object
  method draw : unit
end


class animal species = object method eat = print_endline (species^" eats") end
class pet species name = object
  inherit animal species
  method name : string = name
end
class pet_dog name = object
  inherit pet "dog" name
  method eat = print_endline (name^" barks and then eats")
end

let all_eat (animals : animal list) = List.iter (fun x -> x#eat) animals
all_eat [(new pet_dog "Alik" :> animal)]
all_eat [(new pet_dog "Alik" :> <eat : unit>)]

let to_animal a = (a :> animal) Má typ #animal -> animal

let all_eat (animals : #animal list) = List.iter (fun x -> x#eat) animals
```

```
class foo = object (self : 'self)
  method private x = ...
  method y = self#x
end

class virtual collection = object
  method virtual length : int
end
class virtual enum_coll = object (self : 'self)
  inherit collection
  method virtual iter : (element -> unit) -> unit
  method print = self#iter (fun e -> e#print)
end
```

Polymorfismus tříd

```
class ['key, 'value] map (compare : 'key -> 'key -> int) =
  let eq a (b, _) = compare a b = 0 in
  object (self : 'self)
    val mutable elements : ('key * 'value) list = []
    method add key value = elements <- (key, value) :: elements
    method find key = snd (List.find (eq key) elements)
  end

class ['value] int_map = [int, 'value] map compare_int

class ['a, 'b] pair (x0 : 'a) (y0 : 'b) = object (self : 'self)
  val mutable x = x0
  val mutable y = y0
  method set_fst x' = x <- x'
  method set_fst y' = y <- y'
  method value = x, y
end

(dog, dog) :> (animal, animal)

type (+'a, +'b) pair' = 'a * 'b
type (-'a, +'b) func = 'a -> 'b

class [+'a, +'b] pair (x0 : 'a) (y0 : 'b) = object (self : 'self)
  val mutable x = x0
  val mutable y = y0
  method value = x, y
end
```