

Objective Caml version 3.08.3

```

(* ----- *)
(* Eine höhere Funktion ist eine Funktion, die andere Funktionen als Argumente *)
(* hat, z.B. die Komposition, die zwei Funktionen f und g auf f o g abbildet. *)
(* ----- *)

# let composition (f: int -> int) (g: int -> int) = fun x -> f (g x);;
val composition : (int -> int) -> (int -> int) -> int -> int = <fun>

# let composition (f: int -> int) (g: int -> int) x = f (g x);;
val composition : (int -> int) -> (int -> int) -> int -> int = <fun>

# let succ = fun x -> x + 1;;
val succ : int -> int = <fun>

# let f = composition (fun x -> x * x) succ;;
val f : int -> int = <fun>

# f 4;;
- : int = 25

# composition (fun x -> x * x) succ 4;;
- : int = 25

# composition succ (fun x -> x * x) 4;;
- : int = 17

(* ----- *)
(* Es macht wenig Sinn, die Komposition nur für Funktionen vom Typ int -> int *)
(* zu definieren. Stattdessen genügt es zu fordern, dass der Argumenttyp von f *)
(* mit dem Resultatyp von g übereinstimmt. Das kann man entweder über die Pa- *)
(* rametertypen festlegen, oder man überlässt es dem Typinferenz-Algorithmus. *)
(* ----- *)

# let composition (f: 'b -> 'c) (g: 'a -> 'b) (x: 'a) = f (g x);;
val composition : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let composition f g x = f (g x);;
val composition : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let square x = x * x;;
val square : int -> int = <fun>

# let even = fun x -> x mod 2 = 0;;
val even : int -> bool = <fun>

# let g = composition even square;;
val g : int -> bool = <fun>

# let h = composition square even;;

Characters 27-31:
  let h = composition square even;;
                        ^^^^
This expression has type int -> bool but is here used with type int -> int

(* ----- *)
(* Wenn wir Mengen als Funktionen mit Resultatyp bool auffassen, dann lassen *)
(* sich die üblichen Mengenoperationen als höhere Funktionen implementieren. *)
(* ----- *)

# type int_set = int -> bool;;
type int_set = int -> bool

```

```

# let complement (p: int_set): int_set = fun x -> not (p x);;
val complement : int_set -> int_set = <fun>
# let odd = complement even;;
val odd : int_set = <fun>
# odd 5;;
- : bool = true
# odd 4;;
- : bool = false
# let union (p: int_set) (q: int_set): int_set = fun x -> p x || q x;;
val union : int_set -> int_set -> int_set = <fun>
# let intersection (p: int_set) (q: int_set): int_set = fun x -> p x && q x;;
val intersection : int_set -> int_set -> int_set = <fun>
# let negative = fun x -> x < 0;;
val negative : int -> bool = <fun>
# let nat = complement (intersection (union even odd) negative);;
val nat : int_set = <fun>
# nat 5;;
- : bool = true
# nat (-5);;
- : bool = false
# nat 0;;
- : bool = true

(* ----- *)
(* Auch hier macht es wenig Sinn, sich auf Mengen von integers zu beschränken. *)
(* Stattdessen kann man gleich Mengen mit beliebigem Elementtyp 'a betrachten. *)
(* Wenn man 'a set explizit als Parameter- bzw. Resultattyp angibt, dann wird *)
(* diese Schreibweise (die ja nur eine Abkürzung für 'a -> bool ist) auch vom *)
(* Compiler übernommen. *)
(* ----- *)

# let complement p = fun x -> not (p x);;
val complement : ('a -> bool) -> 'a -> bool = <fun>
# type 'a set = 'a -> bool;;
type 'a set = 'a -> bool
# let complement (p: 'a set): 'a set = fun x -> not (p x);;
val complement : 'a set -> 'a set = <fun>
# let union (p: 'a set) (q: 'a set): 'a set = fun x -> p x || q x;;
val union : 'a set -> 'a set -> 'a set = <fun>
# let intersection (p: 'a set) (q: 'a set): 'a set = fun x -> p x && q x;;
val intersection : 'a set -> 'a set -> 'a set = <fun>
# let odd = complement even;;
val odd : int set = <fun>
# let nat = complement (intersection (union even odd) negative);;

```

```
val nat : int set = <fun>

(* ----- *)
(* Natürlich können höhere Funktionen auch durch Rekursion definiert werden: *)
(* Die Funktion iterate berechnet aus den Argumenten n >= 0 und f: 'a -> 'a *)
(* die n-te Iteration von f, d.h. die Funktion f^n = f o ... o f (n-mal). *)
(* ----- *)

# let rec iterate n f = if n = 0 then fun x -> x else composition f (iterate (n-1) f);;

val iterate : int -> ('a -> 'a) -> 'a -> 'a = <fun>

# let rec iterate n f x = if n = 0 then x else f (iterate (n-1) f x);;

val iterate : int -> ('a -> 'a) -> 'a -> 'a = <fun>

# iterate 5 succ 0;;

- : int = 5

#
```