

Objective Caml version 3.08.3

```

(* ----- *)
(* Deklarationen sind keine Zuweisungen, es wird kein Wert überschrieben, *)
(* sondern es wird höchstens ein Name "verdeckt". *)
(* ----- *)

# let x = 1
in (let x = 2 in x) + x;;

- : int = 3

(* ----- *)
(* Einem Namen, der für einen Wert steht, darf nichts zugewiesen werden: *)
(* ----- *)

# let x = 1;;

val x : int = 1

# x := 2;;

Characters 0-1:
  x := 2;;
  ^
This expression has type int but is here used with type 'a ref

(* ----- *)
(* Formal: Als erstes Argument des Zuweisungsoperators darf nur ein Element *)
(* vom Referenztyp verwendet werden. *)
(* ----- *)

# (:=);;

- : 'a ref -> 'a -> unit = <fun>

(* ----- *)
(* Elemente vom Referenztyp erhält man mit dem ref-Operator. *)
(* ----- *)

# ref;;

- : 'a -> 'a ref = <fun>

# let x = ref 0;;

val x : int ref = {contents = 0}

# x := 1;;

- : unit = ()

(* ----- *)
(* Auf den Inhalt eines solchen Elements greift man mit dem Inhaltsoperator ! *)
(* zu (explizite Dereferenzierung). *)
(* ----- *)

# ! x;;

- : int = 1

# (!);;

- : 'a ref -> 'a = <fun>

(* ----- *)
(* Der Name x allein steht für den Speicherplatz, nicht für seinen Inhalt. *)
(* ----- *)

# x;;

- : int ref = {contents = 1}

(* ----- *)
(* Mit Speicherplätzen kann man nicht "rechnen" (keine Pointer-Arithmetik) *)
(* ----- *)

# x + 1;;

```

Characters 0-1:

```
x + 1;;
^
```

This expression has type int ref but is here used with type int

```
# !x + 1;;
```

```
- : int = 2
```

```
# let y = ref 0;;
```

```
val y : int ref = {contents = 0}
```

```
# y := x;;
```

Characters 5-6:

```
y := x;;
^
```

This expression has type int ref but is here used with type int

```
# y := !x;;
```

```
- : unit = ()
```

```
# y;;
```

```
- : int ref = {contents = 1}
```

```
# x = y;;
```

```
- : bool = true
```

```
# x == y;;
```

```
- : bool = false
```

```
# y := !y + 1;;
```

```
- : unit = ()
```

```
# !y;;
```

```
- : int = 2
```

```
(* ----- *)
(* Durch die folgende Deklaration erhält man zwei Namen für ein und denselben *)
(* Speicherplatz (sogenanntes aliasing). *)
(* ----- *)
```

```
# let z = x;;
```

```
val z : int ref = {contents = 1}
```

```
# x := !x + 1;;
```

```
- : unit = ()
```

```
# !z;;
```

```
- : int = 2
```

```
(* ----- *)
(* Hintereinanderausführung mit ";" *)
(* ----- *)
```

```
# x := 5; x := 2 * !x;;
```

```
- : unit = ()
```

```
# !x;;
```

```
- : int = 10
```

```
# x := 5; x := 2 * !x; !x;;
```

```
- : int = 10
```

```

(* ----- *)
(* Weitere imperative Konstrukte: for-Schleife, while-Schleife *)
(* ----- *)

# x := 0;
for i = 1 to 100 do x := !x + i done;
!x;;

- : int = 5050

(* ----- *)
(* Man beachte: Die "Laufvariable" i ist vom Typ int, also sind Zuweisungen an *)
(* die Laufvariable nicht möglich. Sie ist lokal im Rumpf der for-Schleife, *)
(* außerhalb davon ist sie nicht bekannt. *)
(* ----- *)

# i;;

Characters 0-1:
  i;;
  ^
Unbound value i

(* ----- *)
(* Abwärts geht's auch. *)
(* ----- *)

# for i = 10 downto 0 do x := !x + i done; !x;;

- : int = 5105

(* ----- *)
(* while-Schleifen wie üblich. *)
(* ----- *)

# x := 1;
y := 0;
while !x < 1000 do x := !x + !x; y := !y + 1 done;
!y;;

- : int = 10

(* ----- *)
(* Referenzen sind first class citizens, dürfen also als Parameter oder *)
(* Resultate von Funktionen, als Elemente von Listen ... auftreten. *)
(* ----- *)

(* ----- *)
(* Insbesondere hat man damit "call-by-reference" als Spezialfall von call-by- *)
(* value. *)
(* ----- *)

# let rec imp_fact (n: int) (x: int ref): unit =
  if n = 0 then x := 1 else (imp_fact (n - 1) x; x := n * !x);;

val imp_fact : int -> int ref -> unit = <fun>

# imp_fact 6 x; !x;;

- : int = 720

# let rec imp_reverse (l1: 'a list ref) (l2: 'a list ref) =
while !l1 <> [] do l2 := List.hd (!l1) :: !l2; l1 := List.tl (!l1) done;;

val imp_reverse : 'a list ref -> 'a list ref -> unit = <fun>

(* ----- *)
(* Durch Listen von Referenzen kann man sich arrays simulieren (allerdings mit *)
(* ineffizienter Zugriffsfunktion). *)
(* ----- *)

# let rec new_ref_list (n: int) (x: 'a) =
  if n = 0 then [] else ref x :: new_ref_list (n - 1) x;;

val new_ref_list : int -> 'a -> 'a ref list = <fun>

# let new_array (n: int) (x: 'a) =
  let l = new_ref_list n x in function i -> List.nth l i;;

```

```

val new_array : int -> 'a -> int -> 'a ref = <fun>
# let a = new_array 10 0;;
val a : int -> int ref = <fun>
# a 5;;
- : int ref = {contents = 0}
# for i = 0 to 9 do a i := i * i done;;
- : unit = ()
# a 5;;
- : int ref = {contents = 25}
# a 6;;
- : int ref = {contents = 36}
# a 10;;
Exception: Failure "nth".

(* ----- *)
(* Auch "Objekte" (mit information hiding) kann man sich allein mit Referenzen *)
(* simulieren: Ein Objekt ist ein Tupel von Funktionen, die auf einer gemein- *)
(* samen Instanzvariablen arbeiten. *)
(* ----- *)

# let (inc, get) =
  let x = ref 0
  in ((fun () -> x := !x + 1),
      (fun () -> !x));;

val inc : unit -> unit = <fun>
val get : unit -> int = <fun>

# inc ();;
- : unit = ()

# get ();;
- : int = 1

(* ----- *)
(* oder besser: ein record von Funktionen: *)
(* ----- *)

# type counter =
  {inc: unit -> unit;
   get: unit -> int};;

type counter = { inc : unit -> unit; get : unit -> int; }

# let c =
  let x = ref 0
  in {inc = (fun () -> x := !x + 1);
      get = fun () -> !x};;

val c : counter = {inc = <fun>; get = <fun>}

# c.inc ();;
- : unit = ()

# c.get ();;
- : int = 1

(* ----- *)
(* Braucht man mehr als einen Zähler, so kann man sich einen "Zählergenerator" *)
(* definieren. *)
(* ----- *)

```

```
# let new_counter (): counter =
  let x = ref 0
  in {inc = (fun () -> x := !x + 1);
      get = fun () -> !x};;

val new_counter : unit -> counter = <fun>

# let c1 = new_counter ();;
val c1 : counter = {inc = <fun>; get = <fun>}

# let c2 = new_counter ();;
val c2 : counter = {inc = <fun>; get = <fun>}

# c1.inc (); c1.inc (); c2.inc ();;
- : unit = ()

# c1.get ();;
- : int = 2

# c2.get ();;
- : int = 1

#
```