

Objective Caml version 3.08.3

```

(* ----- *)
(* Binäre Bäume (deren Blätter mit integers markiert sind). *)
(* ----- *)

# type bintree =
  Leaf of int
  | Node of bintree * bintree;;

type bintree = Leaf of int | Node of bintree * bintree

# let t: bintree = Node (Node (Leaf 1, Leaf 2), Leaf 3);;

val t : bintree = Node (Node (Leaf 1, Leaf 2), Leaf 3)

(* ----- *)
(* Anstelle von integers kann man einen beliebigen Typ 'a zulassen. *)
(* ----- *)

# type 'a bintree =
  Leaf of 'a
  | Node of 'a bintree * 'a bintree;;

type 'a bintree = Leaf of 'a | Node of 'a bintree * 'a bintree

(* ----- *)
(* Dann ist bintree kein Typname mehr, sondern ein "Typkonstruktor", d.h. eine *)
(* Funktion, die Typen auf Typen abbildet. *)
(* ----- *)

# let t: bintree = Node (Node (Leaf 1, Leaf 2), Leaf 3);;

Characters 7-14:
  let t: bintree = Node (Node (Leaf 1, Leaf 2), Leaf 3);;
      ^^^^^^^^
The type constructor bintree expects 1 argument(s),
but is here applied to 0 argument(s)

# let t: int bintree = Node (Node (Leaf 1, Leaf 2), Leaf 3);;

val t : int bintree = Node (Node (Leaf 1, Leaf 2), Leaf 3)

(* ----- *)
(* Die (polymorphe) Funktion leaves sammelt die Blätter eines Baumes in einer *)
(* Liste auf. *)
(* ----- *)

# let rec leaves t =
  match t with
  | Leaf x -> [x]
  | Node (t1, t2) -> leaves t1 @ leaves t2;;

val leaves : 'a bintree -> 'a list = <fun>

# leaves t;;

- : int list = [1; 2; 3]

(* ----- *)
(* Will man neben den Blättern auch die inneren Knoten markieren (und dabei *)
(* Markierungen eines anderen Typs zulassen), so führt man einen zweiten Typ- *)
(* parameter 'b ein. *)
(* ----- *)

# type ('a,'b) bintree =
  Leaf of 'a
  | Node of 'b * ('a,'b) bintree * ('a,'b) bintree;;

type ('a, 'b) bintree =
  Leaf of 'a
  | Node of 'b * ('a, 'b) bintree * ('a, 'b) bintree

(* ----- *)
(* So kann man z.B. (einfache) arithmetische Ausdrücke als binäre Bäume auf- *)
(* fassen, deren Blätter mit integers und deren innere Knoten mit Funktionen *)
(* +, -, *, ... markiert sind. *)
(* ----- *)

```

```

# let t = Node ((-), Node ((+), Leaf 1, Leaf 2), (Node ((-), Leaf 3, Leaf 4)));;

val t : (int, int -> int -> int) bintree =
  Node (<fun>, Node (<fun>, Leaf 1, Leaf 2), Node (<fun>, Leaf 3, Leaf 4))

(* ----- *)
(* Die Funktion eval wertet einen solchen arithmetischen Ausdruckk aus. *)
(* ----- *)

# let rec eval t =
  match t with
  | Leaf x -> x
  | Node (f, t1, t2) -> f (eval t1) (eval t2);;

val eval : ('a, 'a -> 'a -> 'a) bintree -> 'a = <fun>

# eval t;;

- : int = 4

(* ----- *)
(* Will man Bäume mit beliebiger Verzweigung haben, so hat ein Knoten nicht *)
(* nur zwei Unterbäume, sondern eine LISTE von Unterbäumen. *)
(* ----- *)

# type 'a tree =
  Leaf of 'a
  | Node of 'a tree list;;

type 'a tree = Leaf of 'a | Node of 'a tree list

# let t = Node [Node [Leaf 1; Leaf 2; Leaf 3]; Leaf 4];;

val t : int tree = Node [Node [Leaf 1; Leaf 2; Leaf 3]; Leaf 4]

(* ----- *)
(* leaves sammelt wieder die Blätter auf. Die Funktion List.concat ist sozusam- *)
(* gen die Verallgemeinerung von List.append. Statt zwei Listen konkateniert *)
(* sie eine beliebige Liste von Listen. *)
(* ----- *)

# let rec leaves t =
  match t with
  | Leaf x -> [x]
  | Node l -> List.concat (List.map leaves l);;

val leaves : 'a tree -> 'a list = <fun>

# leaves t;;

- : int list = [1; 2; 3; 4]

(* ----- *)
(* So kann man jetzt arithmetische Ausdrücke darstellen, die nicht nur binäre *)
(* Funktionen enthalten, und wieder eine einfache Funktion eval zur Auswertung *)
(* solcher Ausdrücke definieren. *)
(* ----- *)

# type ('a, 'b) tree =
  Leaf of 'a
  | Node of 'b * ('a, 'b) tree list;;

type ('a, 'b) tree = Leaf of 'a | Node of 'b * ('a, 'b) tree list

# let add [x;y] = x + y;;

Characters 8-21:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  let add [x;y] = x + y;;
      ^^^^^^^^^^^^^^^
val add : int list -> int = <fun>

# let uminus [x] = -x;;

Characters 11-19:

```

```
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  let uminus [x] = -x;;
             ^^^^^^^^
val uminus : int list -> int = <fun>

# let t = Node (add, [Node (uminus, [Leaf 5]); Node (uminus, [Leaf 5])]);;

val t : (int, int list -> int) tree =
  Node (<fun>, [Node (<fun>, [Leaf 5]); Node (<fun>, [Leaf 5])])

# let rec eval t =
  match t with
  | Leaf x -> x
  | Node (f, l) -> f (List.map eval l);;

val eval : ('a, 'a list -> 'a) tree -> 'a = <fun>

# eval t;;

- : int = -10

#
```