

## Wiederholung

- Basistypen recht gewöhnlich
- `let` bindet Variablen, keine Zuweisung
- Funktionen immer einstellig (`int -> bool -> string`)
- Ausdrücke, keine Statements
- Rekursion statt Schleifen
- Polymorphe Funktionen akzeptieren Werte beliebigen Typs
- Listen sind die zentrale Datenstruktur

## Ablauf von insert

```
# let rec insert elt lst =
  match lst with
  [] -> [elt]
  | head :: tail ->
    if elt <= head
    then elt :: lst
    else head :: insert elt tail;;

# insert 3 [1;2;4;7]
-> 1 :: insert 3 [2;4;7]
-> 1 :: 2 :: insert 3 [4;7]
-> 1 :: 2 :: 3 :: [4;7]
- [1; 2; 3; 4; 7] : int list
```

## Mehr Listenfunktionen

Elementtest für Listen

```
# let rec mem e l =
  match l with
  [] -> false
  | hd::tl -> if e = hd then true else mem e tl;;
```

```
val mem : 'a -> 'a list -> bool = <fun>
```

```
# mem 1 [1;2;3];;
- : bool = true
```

## Funktionen als Objekte erster Klasse

Funktionen lassen sich als Parameter übergeben

```
# let deriv f dx x = (f(x +. dx) -. f(x)) /. dx;;
val deriv : (float -> float) ->
  float -> float -> float = <fun>
```

```
# deriv sin 1e-6 pi;;
- : float = -1.00000000013961143
```

oder in Datenstrukturen speichern:

```
# [sin;cos;tan];;
- : (float -> float) list = [<fun>; <fun>; <fun>]
```

## Higher-Order Funktionen

Funktion auf alle Elemente einer Liste anwenden

```
# let rec map f l =
  match l with
  [] -> []
  | hd :: tl -> f hd :: map f tl;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map square [0;1;2;3;4];;
- : int list = [0; 1; 4; 9; 16]
```

## Listen filtern

Prädikat bestimmt, welche Elemente drin bleiben dürfen

```
# let rec filter p l =
  match l with
  [] -> []
  | hd::tl ->
    if p hd
    then hd::(filter p tl)
    else filter p tl;;
val filter :
  ('a -> bool) -> 'a list -> 'a list = <fun>

# filter even [1;2;3;4];;
- : int list = [2; 4]
```

## Weitere Higher-Order-Funktionen

```
let rec exists pred l =
  match l with
  [] -> false
  | hd::tl -> if pred hd
              then true
              else exists pred tl

val exists : ('a -> bool) -> 'a list -> bool = <fun>

let rec for_all pred l =
  match l with
  [] -> true
  | hd::tl -> if pred hd
              then for_all pred tl
              else false

val for_all : ('a -> bool) -> 'a list -> bool = <fun>
```

## Iteration über Listen

fold\_left faltet Liste

```
let rec fold_left f accu l =
  match l with
  [] -> accu
  | hd::tl -> fold_left f (f accu hd) tl

val fold_left :
  ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# fold_left (fun ac el -> ac + el) 0 [1;3;7];;
- : int = 11
```

## Anonyme Funktionen

Ausdruck, der zu Funktion auswertet: `fun var ... -> expr`

```
# fun x -> x + 1;;
- : int -> int = <fun>

# map (fun x -> x + 1) [0;1;2;3;4];;
- : int list = [1; 2; 3; 4; 5]

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

# let cos2 = compose (fun x -> x *. x) cos;;
val cos2 : float -> float = <fun>
```

## Records

Datentyp mit benannten Feldern

Syntax: `type name = { field ; ... }`

Wobei `field ::= name : type`

```
# type ratio = {num: int; denum: int};;
type ratio = { num: int; denum: int }
```

## Alternative: Mit Pattern-Matching

```
function case | case | ...

# function [] -> 0
      | hd::[] -> 1
      | hd::mid::tl -> 2

- : 'a list -> int = <fun>
```

## Records erzeugen

Syntax: `{ name = expr ; ... }`

```
# let add_ratio r1 r2 =
      {num = r1.num * r2.denum + r2.num * r1.denum;
       denum = r1.denum * r2.denum};;
val add_ratio : ratio -> ratio -> ratio

# let elevenfifteen = add_ratio {num=1; denum=3}
                                {num=2; denum=5};;
val elevenfifteen : ratio = {num = 11; denum = 15}
```

## Recordfelder lesen

Auf Felder zugreifen: `expr . name`

```
# elevenfifteen.num;;  
- : int = 11
```

## Records mit veränderbaren Feldern

Alternative für `field ::= ... | mutable name : type`

```
# type colleague = {name: string;  
                    mutable married: bool};;  
type colleague = { name : string;  
                  mutable married : bool; }
```

```
# let mandeep = {name="Mandeep"; married = false};;  
val mandeep : colleague = {name = "Mandeep";  
                           married = false}
```

Verändern: `expr <- expr`

```
# mandeep.married <- true;;  
- : unit = ()  
# mandeep;;  
- : colleague = {name = "Mandeep"; married = true}
```

## = und ==

Beide Operatoren vergleichen zwei beliebige Werte gleichen Typs

(Ausnahme: Funktionen)

Unterschied:

- `=` testet auf strukturelle Gleichheit
- `==` testet auf physische Gleichheit

Umgangssprachlich:

- `a = b`  $\Rightarrow$  `a` ist gleich `b`
- `a == b`  $\Rightarrow$  `a` ist dasselbe wie `b`

## Unterschied = und ==

Läßt sich beobachten, wenn Dinge verändert werden können:

Wirkt sich die Veränderung an `a` auf `b` aus?

```
# let new_mandeep = {name="Mandeep"; married = true}  
# mandeep = newMandeep;;  
- : bool = true  
# mandeep == newMandeep;;  
- : bool = false
```

## Algebraische Datentypen

Typen, deren Werte von unterschiedlichen Konstruktoren erzeugt werden

Konstruktornamen müssen mit Großbuchstabe beginnen

Syntax: `type name = constr | ...`

Wobei `constr ::= Name of type ...`

```
# type number = Int of int | Float of float | Error;;
type number = | Int of int | Float of float | Error
# Int 42;;
- : number = Int 42
```

## Beispiel: Vorzeichen

```
# type sign = Positive | Negative;;
type sign = Positive | Negative

# let sign_int n = if n >= 0 then Positive
                  else Negative;;

val sign_int : int -> sign = <fun>

# sign_int 42;;
- : sign = Positive
```

## Funktionen auf algebraischen Datentypen

Zugriff stets mit `match`

```
# let add_num n1 n2 =
  match (n1, n2) with
  | (Int i1, Int i2) ->
    if sign_int i1 = sign_int i2 &&
       sign_int(i1 + i2) <> sign_int i1
    then Float(float i1 +. float i2)
    else Int(i1 + i2)
  | (Int i1, Float f2) -> Float(float i1 +. f2)
  | (Float f1, Int i2) -> Float(f1 +. float i2)
  | (Float f1, Float f2) -> Float(f1 +. f2)
  | (_, _) -> Error;;

val add_num : number -> number -> number = <fun>

# add_num (Int 1) (Float 2.2);;
- : number = Float 3.2
```

## Polymorphe algebraische Datentypen

Datentyp über Typen parametrisiert

Syntax: `type name var .. = constr | ...`

`constr` verwendet Variablen

```
# type 'a option = Some of 'a
                  | None;;

# let head lst =
  match lst with
  | [] -> None
  | hd::_ -> Some hd;;

head : 'a list -> 'a option
```

## mapfilter

option-Funktion auf jedes Element anwenden, nur **some** Elemente behalten

```
# let rec mapfilter p l =
  match l with
  [] -> []
  | hd::tl ->
    match p hd with
    None -> mapfilter p tl
    | Some e -> e::mapfilter p tl;;

val mapfilter :
  ('a -> 'b option) -> 'a list -> 'b list = <fun>

# mapfilter head [[];[1;2];[3];[];[4;5]];;
- : int list = [1; 3; 4]
```

## Lokale Bindungen

Syntax: `let var = expr in expr`

```
# let f x =
  let x = x + 1 in
  let y = x + 1 in
  let z = x + y in
  z;;

# f 1;;
- : int = 5
```

Analog: `let rec ... and ... in expr`

## Rekursive algebraische Datentypen

Definition kann den definierten Typ verwenden

```
# type 'a btree = Empty
  | Node of 'a * 'a btree * 'a btree;;
type 'a btree = | Empty
  | Node of 'a * 'a btree * 'a btree
```

Funktionen darüber sind dann i.d.R. rekursiv

```
# let rec depth btree =
  match btree with
  Empty -> 0
  | Node(y, left, right) ->
    1 + max (depth left) (depth right)
val depth : 'a btree -> int = <fun>
# depth (Node (42, Node (23, Empty, Empty), Empty));;
- : int = 2
```

## Ausnahme-Behandlung

Exception definieren: `exception Name`

```
# exception Empty_list;;
exception Empty_list
```

Exception werfen: `raise Name`

```
# let head l =
  match l with
  [] -> raise Empty_list
  | hd :: _ -> hd;;
val head : 'a list -> 'a = <fun>

# head [];;
Uncaught exception: Empty_list
```

## Exception abfangen

Syntax: `try expr with Name -> expr`

```
# let name_of_binary_digit digit =
  try
    List.assoc digit [(0, "zero"); (1, "one")]
  with Not_found ->
    "not a binary digit";;
val name_of_binary_digit : int -> string = <fun>

# name_of_binary_digit 0;;
- : string = "zero"
# name_of_binary_digit 2;;
- : string = "not a binary digit"
```

## Zugriff auf andere Module

Expliziter Zugriff: `Modulname . Name`

Implizit: `open Modulname` macht die Namen des Moduls verfügbar

Das Modul `Pervasives` ist immer geöffnet

## Module

Jede Datei ist ein eigenes Modul

Name des Moduls = Dateiname ohne `.ml` und erster Buchstabe groß

Namen sind nur im Modul sichtbar

Module können einzeln kompiliert werden

System ist eigentlich viel mächtiger

## Schnittstellen

Zu jedem Modul gehört eine Schnittstelle

Schnittstelle bestimmt, welche Namen von außen sichtbar sind

Default-Schnittstelle: Alle Namen sichtbar

Explizite Schnittstelle: `.mli`-Datei mit Exporten

Export:

- `val name : type`
- Typdefinition
- Exceptiondefinition

## Compilieren von Modulen

Datei `sort.ml`

```
let rec insert = ...  
let rec sort = ...
```

Zu Modul `sort` kompilieren:

```
ocamlc -c sort.ml
```

⇒ `sort.cmo`

## Compilieren von Schnittstellen

Datei `foo.mli`

```
val sort : 'a list -> 'a list
```

Kompilieren:

```
ocamlc -c sort.mli
```

⇒ `sort.cmi`

## Verwenden

Datei `main.ml`

```
... Sort.sort [2;1] ...
```

Datei `foo.ml`

```
open Sort  
sort [1;2]
```

## Laden

```
ocaml foo.cmo
```

oder interaktiv

```
# #load "foo.cmo";;
```

Macht Modul `Foo` verfügbar für `open` und expliziten Zugriff