

Vorsicht bei redundanten und unvollständigen Matches!

```
# let n = 7;;
val n : int = 7
# match n with 0 -> "null";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 5, -13).
# match n
  with 0 -> "null"
       | 0 -> "eins"
       | _ -> "Soweit kann ich nicht zaehlen!";;
Warning: this match case is unused.
- : string = "Soweit kann ich nicht zaehlen!"
```

2.5 Listen

Listen werden mithilfe von `[]` und `::` konstruiert.

Kurzschreibweise: `[42; 0; 16]`

```
# let mt = [];;  
val mt : 'a list = []  
# let l1 = 1::mt;;  
val l1 : int list = [1]  
# let l = [1;2;3];;  
val l : int list = [1; 2; 3]  
# let l = 1::2::3::[];;  
val l : int list = [1; 2; 3]
```

Achtung:

Alle Elemente müssen den gleichen Typ haben:

```
# 1.0::1::[];;
```

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

Achtung:

Alle Elemente müssen den **gleichen** Typ haben:

```
# 1.0::1::[];;
```

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

tau list beschreibt Listen mit Elementen vom Typ **tau :-)**

Der Typ **'a** ist eine **Typ-Variable**:

[] bezeichnet eine leere Liste für **beliebige** Element-Typen **:-))**

Pattern Matching auf Listen:

```
# match l
  with []      -> -1
       | x::xs -> x;;
-: int = 1
```

2.6 Definitionen von Funktionen

```
# let double x = 2*x;;  
val double : int -> int = <fun>  
# (double 3, double (double 1));;  
- : int * int = (6,4)
```

- Nach dem Funktions-Namen kommen die Parameter.
- Der Funktionsname ist damit auch nur eine Variable, deren Wert eine Funktion ist :-)

- Alternativ können wir eine Variable einführen, deren Wert direkt eine Funktion beschreibt ...

```
# let double = fun x -> 2*x;;  
val double : int -> int = <fun>
```

- Diese Funktionsdefinition beginnt mit `fun`, gefolgt von den formalen Parametern.
- Nach `->` kommt die Berechnungsvorschrift.
- Die linken Variablen dürfen rechts benutzt werden :-)

Achtung:

Funktionen sehen die Werte der Variablen, die zu ihrem **Definitionszeitpunkt** sichtbar sind:

```
# let faktor = 2;;  
val faktor : int = 2  
# let double x = faktor*x;;  
val double : int -> int = <fun>  
# let faktor = 4;;  
val faktor : int = 4  
# double 3;;  
- : int = 6
```


Achtung:

Eine Funktion ist ein Wert:

```
# double;;  
- : int -> int = <fun>
```

Rekursive Funktionen:

Eine Funktion ist **rekursiv**, wenn sie sich selbst aufruft.

```
# let rec fac n = if n < 2 then 1 else n * fac (n-1);;
val fac : int -> int = <fun>
# let rec fib = fun x -> if x <= 1 then 1
                        else fib (x-1) + fib (x-2);;
val fib : int -> int = <fun>
```

Dazu stellt **Ocaml** das Schlüsselwort **rec** bereit :-)

Rufen mehrere Funktionen sich gegenseitig auf, heißen sie **verschränkt rekursiv**.

```
# let rec even n = if n=0 then "even" else odd (n-1)
      and odd  n = if n=0 then "odd"  else even (n-1);;
val even : int -> string = <fun>
val odd  : int -> string = <fun>
```

Wir kombinieren ihre Definitionen mit dem Schlüsselwort **and :-)**

Definition durch Fall-Unterscheidung:

```
# let rec len = fun l -> match l
                        with [] -> 0
                        | x::xs -> 1 + len xs;;

val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
```

Definition durch Fall-Unterscheidung:

```
# let rec len = fun l -> match l
                        with [] -> 0
                        | x::xs -> 1 + len xs;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
```

... kann kürzer geschrieben werden als:

```
# let rec len = function [] -> 0
                        | x::xs -> 1 + len xs;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
```

Fall-Unterscheidung bei mehreren Argumenten:

```
# let rec app l y = match l
                        with [] -> y
                           | x::xs -> x :: app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

Fall-Unterscheidung bei mehreren Argumenten:

```
# let rec app l y = match l
                        with [] -> y
                           | x::xs -> x :: app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

... kann auch geschrieben werden als:

```
# let rec app = function [] -> fun y -> y
                    | x::xs -> fun y -> x::app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

Lokale Definitionen

Definitionen können mit `let` lokal eingeführt werden:

```
# let      x = 5
  in let sq = x*x
  in      sq+sq;;
- : int = 50
# let facit n = let rec
  iter m yet = if m>n then yet
                else iter (m+1) (m*yet)
  in iter 2 1;;
val facit : int -> int = <fun>
```


2.7 Benutzerdefinierte Typen

Beispiel: Spielkarten

Wie kann man die Farbe und den Wert einer Karte spezifizieren?

1. Idee: Benutze Paare von Strings und Zahlen, z.B.

("Karo",10) ≡ Karo Zehn
("Kreuz",12) ≡ Kreuz Bube
("Pik",1) ≡ Pik As

Nachteile:

- Beim Test auf eine Farbe muss immer ein String-Vergleich stattfinden
—→ ineffizient!
- Darstellung des Buben als 12 ist nicht intuitiv
—→ unleserliches Programm!
- Welche Karte repräsentiert das Paar ("Kaor", 1)?
(Tippfehler werden vom Compiler nicht bemerkt)

Besser: Aufzählungstypen von **Ocaml**.

Beispiel: Spielkarten

2. Idee: Aufzählungstypen

```
# type farbe = Karo | Herz | Pik | Kreuz;;
type farbe = Karo | Herz | Pik | Kreuz
# type wert = Neun | Bube | Dame | Koenig | Zehn | As;;
type wert = Neun | Bube | Dame | Koenig | Zehn | As
# Kreuz;;
- : farbe = Kreuz
# let pik_bube = (Pik,Bube);;
val pik_bube : farbe * wert = (Pik, Bube)
```

Vorteile:

- Darstellung ist intuitiv.
- Tippfehler werden erkannt:

```
# (Kaor,As) ; ;
```


Unbound constructor Kaor
- Die interne Repräsentation ist **effizient** :-)

Bemerkungen:

- Durch **type** wird ein **neuer Typ** definiert.
- Die Alternativen heißen **Konstruktoren** und werden durch **|** getrennt.
- Jeder Konstruktor wird groß geschrieben und ist **eindeutig** einem Typ zugeordnet.

Aufzählungstypen (cont.)

Konstruktoren können verglichen werden:

```
# Kreuz < Karo;;  
- : bool = false;;  
# Kreuz > Karo;;  
- : bool = true;;
```

Pattern Matching auf Konstruktoren:

```
# let istTrumpf = function  
    (Karo,_)      -> true  
  | (_,Bube)     -> true  
  | (_,Dame)     -> true  
  | (Herz,Zehn)  -> true  
  | (_,_)        -> false;;
```

```
val istTrumpf : farbe * wert -> bool = <fun>
```

Damit ergibt sich z.B.:

```
# istTrumpf (Karo,As);;  
- : bool = true  
# istTrumpf (Pik,Koenig);;  
- : bool = false
```

Eine andere nützliche Funktion:

```
# let string_of_farbe = function
    Karo   -> "Karo"
  | Herz  -> "Herz"
  | Pik   -> "Pik"
  | Kreuz -> "Kreuz";;
val string_of_farbe : farbe -> string = <fun>
```

Beachte:

Die Funktion `string_of_farbe` wählt für eine Farbe in **konstanter Zeit** den zugehörigen String aus (der Compiler benutzt – hoffentlich – **Sprungtabellen** :-)

Jetzt kann **Ocaml** schon fast Karten spielen:

```
# let sticht = function
    ((Herz,Zehn),_)          -> true
  | (_,(Herz,Zehn))         -> false
  | ((f1,Dame),(f2,Dame))   -> f1 > f2
  | ((_,Dame),_)           -> true
  | (_,(_,Dame))           -> false
  | ((f1,Bube),(f2,Bube))   -> f1 > f2
  | ((_,Bube),_)           -> true
  | (_,(_,Bube))           -> false
  | ((Karo,w1),(Karo,w2))   -> w1 > w2
  | ((Karo,_),_)           -> true
  | (_,(Karo,_))           -> false
  | ((f1,w1),(f2,w2))       -> if f1=f2 then w1 > w2
                               else false;;
```



```

...
# let nimm (karte2,karte1) =
    if sticht (karte2,karte1) then karte2 else karte1;;

# let stich (karte1,karte2,karte3,karte4) =
    nimm (karte4, nimm (karte3, nimm (karte2,karte1)));;

# stich ((Herz,Koenig),(Karo,As), (Herz,Bube),(Herz,As));;
- : farbe * wert = (Herz, Bube)
# stich((Herz,Koenig),(Pik,As), (Herz,Koenig),(Herz,As));;
- : farbe * wert = (Herz, As)

```