```
# 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)
```

Summentypen

Summentypen sind eine Verallgemeinerung von Aufzählungstypen, bei denen die Konstruktoren Argumente haben.

Beispiel: Hexadezimalzahlen

```
type hex = Digit of int | Letter of char;;
let char2dez c = if c >= 'A' && c <= 'F'
       then (Char.code c)-55
    else if c >= 'a' && c <= 'f'
       then (Char.code c)-87
    else -1;;
let hex2dez = function
              Digit n -> n
            | Letter c -> char2dez c;;
```

Char ist ein Modul, der Funktionalität für char sammelt :-)

Ein Konstruktor, der mit `type t = Con of <typ> | ...`
definiert wurde, hat die Funktionalität `Con : <typ> -> t` —
muss aber stets angewandt vorkommen ...

```
# Digit;;
The constructor Digit expects 1 argument(s),
but is here applied to 0 argument(s)
# let a = Letter 'a';;
val a : hex = Letter 'a'
# Letter 1;;
This expression has type int but is here used with type char
# hex2dez a;;
- : int = 10
```

Datentypen können auch rekursiv sein:

```
type folge = Ende | Dann of (int * folge)

# Dann (1, Dann (2, Ende));;
- : folge = Dann (1, Dann (2, Ende))
```

Beachte die Ähnlichkeit zu Listen ;-)

Rekursive Datentypen führen wieder zu rekursiven Funktionen:

```
# let rec n_tes = function
                    (_,Ende) -> -1
                  | (0,Dann (x,_)) -> x
                  | (n,Dann (_, rest)) -> n_tes (n-1,rest);;
val n_tes : int * folge -> int = <fun>

# n_tes (4, Dann (1, Dann (2, Ende)));;
- : int = -1
# n_tes (2, Dann (1, Dann(2, Dann (5, Dann (17, Ende)))));;
- : int = 5
```

Anderes Beispiel:

```
# let rec runter = function
                     0 -> Ende
                   | n -> Dann (n, runter (n-1));;
val runter : int -> folge = <fun>

# runter 4;;
- : folge = Dann (4, Dann (3, Dann (2, Dann (1, Ende))));;
# runter -1;;
Stack overflow during evaluation (looping recursion?).
```

Der Options-Datentyp

Ein eingebauter Datentyp in Ocaml ist `option` mit den zwei Konstruktoren `None` und `Some`.

```
# None;;
- : 'a option = None
# Some 10;
- : int option = Some 10
```

Er wird häufig benutzt, wenn eine Funktion nicht für alle Eingaben eine Lösung berechnet:

```
# let rec n_tes = function
                    (n,Ende) -> None
                  | (0, Dann (x,_)) -> Some x
                  | (n, Dann (_,rest)) -> n_tes (n-1,rest);;
val n_tes : int * folge -> int option = <fun>

# n_tes (4,Dann (1, Dann (2, Ende)));;
- : int option = None
# n_tes (2, Dann (1, Dann (2, Dann (5, Dann (17, Ende)))));;
- : int option = Some 5
```

3 Funktionen – näher betrachtet

- Endrekursion
- Funktionen höherer Ordnung
  - → Currying
  - → Partielle Anwendung
- Polymorphe Funktionen
- Polymorphe Datentypen
- Namenlose Funktionen

3.1 Endrekursion

Ein letzter Aufruf im Rumpf e einer Funktion ist ein Aufruf, dessen Wert den Wert von e liefert ...

```
let f x = x+5
let g y = let z = 7
          in if y>5 then f (-y)
             else z + f y
```

Der erste Aufruf in ein letzter, der zweite nicht :-)

$\Longrightarrow$ Von einem letzten Aufruf müssen wir nicht mehr zur aufrufenden Funktion zurück kehren.

$\Longrightarrow$ Der Platz der aufrufenden Funktion kann sofort wiederverwendet werden !!!

Eine Funktion f heißt endrekursiv, falls sämtliche rekursiven Aufrufe von f letzt sind.

Beispiele

```
let rec fac1 = function
      (1,acc) -> acc
    | (n,acc) -> fac1 (n-1,n*acc);;

let rec loop x = if x<2 then x
                 else if x mod 2 = 0 then loop (x/2)
                 else loop (3*x+1);;
```

Diskussion

- Endrekursive Funktionen lassen sich ähnlich effizient ausführen wie Schleifen in imperativen Sprachen :-)
- Die Zwischenergebnisse werden in akkumulierenden Parametern von einem rekursiven Aufruf zum nächsten weiter gereicht.
- In einer Abschlussregel wird daraus das Ergebnis berechnet.
- Endrekursive Funktionen sind insbesondere bei der Verarbeitung von Listen beliebt ...

Umdrehen einer Liste – Version 1:

```
let rec rev  list = match list
        with [] -> []
        | x::xs -> app (rev xs) [x]
```

Umdrehen einer Liste – Version 1:

```
let rec rev  list = match list
        with [] -> []
        | x::xs -> app (rev xs) [x]
```

`rev [0;...;n-1]` ruft Funktion `app` auf mit:

```
[]
[n-1]
[n-1; n-2]
...
[n-1; ...; 1]
```

als erstem Argument $\Longrightarrow$ quadratische Laufzeit :-(

Umdrehen einer Liste – Version 2:

```
let rev list = let rec r a l =
        match l
        with [] -> a
        | x::xs -> r (x::a) xs
    in r [] list
```

Umdrehen einer Liste – Version 2:

```
let rev list = let rec r a l =
        match l
        with [] -> a
        | x::xs -> r (x::a) xs
    in r [] list
```

Die lokale Funktion `r` ist end-rekursiv !

$\Longrightarrow$

lineare Laufzeit !!

3.2 Funktionen höherer Ordnung

Betrachte die beiden Funktionen

```
let f (a,b) = a+b+1;;
let g a b   = a+b+1;;
```

Auf den ersten Blick scheinen sich `f` und `g` nur in der Schreibweise zu unterscheiden. Aber sie haben einen anderen Typ:

```
# f;;
- : int * int -> int = <fun>
# g;;
- : int -> int -> int = <fun>
```

- Die Funktion `f` hat ein Argument, welches aus dem Paar `(a,b)` besteht. Der Rückgabewert ist `a+b+1`.
- `g` hat ein Argument `a` vom Typ `int`. Das Ergebnis einer Anwendung auf `a` ist wieder eine Funktion, welche, angewendet auf ein weiteres Argument `b`, das Ergebnis `a+b+1` liefert:

```
# f (3,5);;
- : int = 9
# let g1 = g 3;;
val g1 : int -> int = <fun>
# g1 5;;
- : int = 9
```

Haskell B. Curry, 1900–1982

Das Prinzip heißt nach seinem Erfinder Haskell B. Curry Currying.

→ g heißt Funktion höherer Ordnung, weil ihr Ergebnis wieder eine Funktion ist.

→ Die Anwendung von g auf ein Argument heißt auch partiell, weil das Ergebnis nicht vollständig ausgewertet ist, sondern eine weitere Eingabe erfordert.

Das Argument einer Funktion kann auch wieder selbst eine Funktion sein:

```
# let apply f a b = f (a,b);;
val apply ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
...
```

```
...
# let plus (x,y) = x+y;;
val plus : int * int -> int = <fun>
# apply plus;;
- : int -> int -> int = <fun>
# let plus2 = apply plus 2;;
val plus2 : int -> int = <fun>
# let plus3 = apply plus 3;;
val plus3 : int -> int = <fun>
# plus2 (plus3 4);;
- : int = 9
```

3.3 Funktionen als Daten

Funktionen sind Daten und können daher in Datenstrukturen vorkommen:

```
# ((+), plus3) ;
- : (int -> int -> int) * (int -> int) = (<fun>, <fun>);;
# let rec plus_list = function
                []    -> []
              | x::xs -> (+) x :: plus_list xs;;
val plus_list : int list -> (int -> int) list = <fun>
# let l = plus_list [1;2;3];;
val l : (int -> int) list = [<fun>; <fun>; <fun>]
```

// `(+) : int -> int -> int` ist die Funktion zum Operator `+`

```
...
  # let do_add n =
        let rec add_list = function
                      [] -> []
                  | f::fs -> f n :: add_list fs
    in add_list ;;
  val do_add : 'a -> ('a -> 'b) list -> 'b list = <fun>
  # do_add 5 l;;
  - : int list = [6;7;8]

  # let rec comp_list a = function
              [] -> a
        | f::fs -> f (comp_list fs);;
  val comp_list : ('a -> 'a) list -> 'a = <fun>
  # comp_list 0 l;;
  - : int = 6
```