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

- Rekursionsarten
- Funktionen höherer Ordnung
  - → Currying
  - → Partielle Anwendung
- Polymorphe Funktionen
- Polymorphe Datentypen
- Namenlose Funktionen
- Unendliche Datenstrukturen

3.1 Rekursionsarten

Je nach Art der rekursiven Aufrufe unterscheiden wir folgende Arten von Rekursion:

1. End-Rekursion: Jeder rekursive Aufruf liefert gleichzeitig den Rückgabewert.

```
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);;
```

2. Repetitive Rekursion: Es gibt in jedem Zweig höchstens einen rekursiven Aufruf.

```
let rec fac = function
          1 -> 1
        | n -> n * fac (n-1);;
```

3. Allgemeine Rekursion:

```
let rec fib = function
          0 -> 1
        | 1 -> 1
        | n -> fib (n-1) + fib (n-2);;

let rec f n = if n<2 then n
              else f (f (n/2));;
```

Das beste Speicherverhalten hat die End-Rekursion. Es wird zur allgemeinen Rekursion hin immer schlechter.

Der typische Fall von repetitiver Rekursion hat die folgende Form:

```
let rec f x = if t(x) then v else e(x,f(g(x))
```

Beispiel: Fakultät

```
let rec fac x = if x=1 then 1 else x * fac (x-1)
```

Hier ist `t(x)` $\equiv$ `x=1`, `v` $\equiv$ `1`, `g(x)` $\equiv$ `x-1` und `e(x,y)` $\equiv$ `x*y`.

Wie wir gesehen haben, gibt es auch eine end-rekursive Variante:

```
let rec fac1 (x,a) = if x=1 then a else fac1 (x-1,x*a)
```

Im allgemeinen Fall kann man `f` ersetzen durch:

```
let rec f1 (x,a) = if t(x) then a else f1 (g(x),e(x,a))
```

Dann ist `f1(x,v)` gleich `f x` — aber nur dann, wenn `e` eine assoziative, kommutative Funktion ist:

```
n*(...*(2*1)) = 1*(2*(...*n))
```

Der zusätzliche Parameter `a` heißt auch Akkumulator.

Vorsicht bei Listenfunktionen:

```
let rec f  x      = if x=0 then [] else x :: f (x-1)
let rec f1 (x,a) = if x=0 then a  else f1 (x-1,x::a)
```

Der Listenkonstruktor `::` is weder kommutativ noch assoziativ.
Deshalb berechnen `f x` und `f1 (x,[])` nicht den gleichen Wert:

```
# f 5;;
- : int list = [5; 4; 3; 2; 1]
# f1 (5,[]);;
- : int list = [1; 2; 3; 4; 5]
```

Selbst allgemeine Rekursion kann manchmal linearisiert werden:

```
let rec fib = function
          0 -> 1
        | 1 -> 1
        | n -> fib (n-1) + fib (n-2);;

let fib1 n =
   let rec iter x (m0,m1) =
            if x<=1 then m1 else iter (x-1) (m1,m0+m1)
   in iter n (1,1)
```

Das lässt sich aber nicht so einfach verallgemeinern :-)

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 sum = function
              [] -> 0
        | f::fs -> f (sum fs);;
  val sum : (int -> int) list -> int = <fun>
  # sum l;;
  - : int = 6
```

3.4 Einige Listen-Funktionen

```
let rec map f = function
            [] -> []
      | x::xs -> f x :: map f xs

let rec  fold_left f a = function
            [] -> a
      | x::xs -> fold_left f (f a x) xs

let rec fold_right f = function
            [] -> fun b -> b
      | x::xs -> fun b -> f x (fold_right f xs b)
```

```
let rec  find_opt f = function
           [] -> None
       | x::xs -> if f x then Some x
                  else find_opt f xs
```

Beachte:

- → Diese Funktionen abstrahieren von dem Verhalten der Funktion `f`. Sie spezifizieren das Rekursionsverhalten gemäß der Listenstruktur, unabhängig von den Elementen der Liste.
- → Daher heißen solche Funktionen Rekursions-Schemata oder (Listen-)Funktionale.
- → Listen-Funktionale sind unabhängig vom Typ der Listenelemente. (Diesen muss nur die Funktion `f` kennen :-)
- → Funktionen, die gleich strukturierte Eingaben verschiedenen Typs verarbeiten können, heißen polymorph.