

5.4 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**.

5.5 Polymorphe Funktionen

Das **Ocaml**-System inferiert folgende Typen für diese Funktionale:

```
map : ('a -> 'b) -> 'a list -> 'b list
fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
find_opt : ('a -> bool) -> 'a list -> 'a option
```

- 'a und 'b sind **Typvariablen**. Sie können durch jeden Typ ersetzt (**instanziiert**) werden (aber an jedem Vorkommen durch den gleichen Typ).

- Durch partielle Anwendung auf eine Funktion können die Typvariablen instanziiert werden:

```
# Char.chr;;  
val : int -> char = <fun>  
  
# map Char.chr;;  
- : int list -> char list = <fun>  
  
# fold_left (+);;  
val it : int -> int list -> int = <fun>
```

- Wenn man einem Funktional eine polymorphe Funktion als Argument gibt, ist das Ergebnis wieder polymorph:

```
# let cons_r xs x = x::xs;;
val cons_r : 'a list -> 'a -> 'a list = <fun>
# let rev l = fold_left cons_r [] l;;
val rev : 'a list -> 'a list = <fun>
# rev [1;2;3];;
- : int list = [3; 2; 1]
# rev [true;false;false];;
- : bool list = [false; false; true]
```

Ein paar der einfachsten polymorphen Funktionen:

```
let compose f g x = f (g x)
```

```
let twice f x = f (f x)
```

```
let iter f g x = if g x then x else iter f g (f x);;
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

```
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
```

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

```
# compose neg neg;;
```

```
- : bool -> bool = <fun>
```

```
# compose neg neg true;;
```

```
- : bool = true;;
```

```
# compose Char.chr plus2 65;;
```

```
- : char = 'C'
```

5.6 Polymorphe Datentypen

Man sich auch selbst polymorphe Datentypen definieren:

```
type 'a tree = Leaf of 'a  
            | Node of ('a tree * 'a tree)
```

- `tree` heißt **Typkonstruktor**, weil er aus einem anderen Typ (seinem Parameter `'a`) einen neuen Typ erzeugt.
- Auf der rechten Seite dürfen nur die Typvariablen vorkommen, die auf der linken Seite als Argument für den Typkonstruktor stehen.
- Die Anwendung der Konstruktoren auf Daten instanziiert die Typvariable(n):

```
# Leaf 1;;  
- : int tree = Leaf 1  
# Node (Leaf ('a',true), Leaf ('b',false));;  
- : (char * bool) tree = Node (Leaf ('a', true),  
                               Leaf ('b', false))
```

Funktionen auf polymorphen Datentypen sind typischerweise wieder polymorph ...

```

let rec size = function
    Leaf _      -> 1
  | Node(t,t') -> size t + size t'

let rec flatten = function
    Leaf x      -> [x]
  | Node(t,t') -> flatten t @ flatten t'

let flatten1 t = let rec doit = function
    (Leaf x, xs) -> x :: xs
  | (Node(t,t'), xs) -> let xs = doit (t',xs)
                        in doit (t,xs)
in doit (t,[])
end

...

```

```
...
val size : 'a tree -> int = <fun>
val flatten : 'a tree -> 'a list = <fun>
val flatten1 : 'a tree -> 'a list = <fun>

# let t = Node(Node(Leaf 1,Leaf 5),Leaf 3);;
val t : int tree = Node (Node (Leaf 1, Leaf 5), Leaf 3)

# size t;;
- : int = 3
# flatten t;;
val : int list = [1;5;3]
# flatten1 t;;
val : int list = [1;5;3]
```

5.7 Anwendung: Queues

Gesucht:

Datenstruktur 'a queue, die die folgenden Operationen unterstützt:

```
enqueue : 'a -> 'a queue -> 'a queue
dequeue : 'a queue -> 'a option * 'a queue
is_empty : 'a queue -> bool
queue_of_list : 'a list -> 'a queue
list_of_queue : 'a queue -> 'a list
```

1. Idee:

- Repräsentiere die Schlange als eine Liste:

```
type 'a queue = 'a list
```

Die Funktionen `is_empty`, `queue_of_list`, `list_of_queue` sind dann trivial :-)

1. Idee:

- Repräsentiere die Schlange als eine Liste:

```
type 'a queue = 'a list
```

Die Funktionen `is_empty`, `queue_of_list`, `list_of_queue` sind dann trivial :-)

- Entnehmen heißt Zugreifen auf das erste Element der Liste:

```
let dequeue = function  
    []      -> (None, [])  
  | x::xs  -> (Some x, xs)
```

1. Idee:

- Repräsentiere die Schlange als eine Liste:

```
type 'a queue = 'a list
```

Die Funktionen `is_empty`, `queue_of_list`, `list_of_queue` sind dann trivial :-)

- Entnehmen heißt Zugreifen auf das erste Element der Liste:

```
let dequeue = function  
    []      -> (None, [])  
  | x::xs  -> (Some x, xs)
```

- Einfügen bedeutet hinten anhängen:

```
let enqueue x xs = xs @ [x]
```

Diskussion:

- Der Operator @ konkateniert zwei Listen.
- Die Implementierung ist sehr einfach :-)
- Entnehmen ist sehr billig :-)
- Einfügen dagegen kostet so viele rekursive Aufrufe von @ wie die Schlange lang ist :-(
- Geht das nicht besser ??

2. Idee:

- Repräsentiere die Schlange als **zwei** Listen !!!

```
type 'a queue = Queue of 'a list * 'a list
let is_empty = function
    Queue ([],[]) -> true
    | _           -> false
let queue_of_list list = Queue (list,[])
let list_of_queue = function
    Queue (first,[])    -> first
    | Queue (first,last) ->
        first @ List.rev last
```

- Die zweite Liste repräsentiert das **Ende** der Liste und ist deshalb in **umgedrehter Anordnung** ...

2. Idee (Fortsetzung):

- Einfügen erfolgt deshalb in der zweiten Liste:

```
let enqueue x (Queue (first,last)) =  
    Queue (first, x::last)
```

2. Idee (Fortsetzung):

- Einfügen erfolgt deshalb in der zweiten Liste:

```
let enqueue x (Queue (first,last)) =  
    Queue (first, x::last)
```

- Entnahme bezieht sich dagegen auf die erste Liste :-)

Ist diese aber leer, wird auf die zweite zugegriffen ...

```
let dequeue = function  
    Queue ([],last) -> (match List.rev last  
        with [] -> (None, Queue ([],[]))  
         | x::xs -> (Some x, Queue (xs,[])))  
  | Queue (x::xs,last) -> (Some x, Queue (xs,last))
```

Diskussion:

- Jetzt ist Einfügen billig :-)
- Entnehmen dagegen kann so teuer sein, wie die Anzahl der Elemente in der zweiten Liste :-(
- Gerechnet aber auf jede Einfügung, fallen nur **konstante** Zusatzkosten an !!!

⇒ amortisierte Kostenanalyse

5.8 Namenlose Funktionen

Wie wir gesehen haben, sind Funktionen **Daten**. Daten, z.B. [1;2;3] können verwendet werden, ohne ihnen einen Namen zu geben. Das geht auch für Funktionen:

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

- **fun** leitet eine **Abstraktion** ein.
Der Name kommt aus dem **λ -Kalkül**.
- **->** hat die Funktion von **=** in Funktionsdefinitionen.
- **Rekursive** Funktionen können so nicht definiert werden, denn ohne Namen kann eine Funktion nicht in ihrem Rumpf vorkommen **:-)**