

# Programmierung 1

Vorlesung 8

*Livestream beginnt um 10:20 Uhr*

*Listen, Teil 3*  
*Sortieren*

Programmierung 1

# Faltung

$$\text{foldl} : (\alpha * \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha \text{ list} \rightarrow \beta$$

Verknüpfung

Startwert

Liste

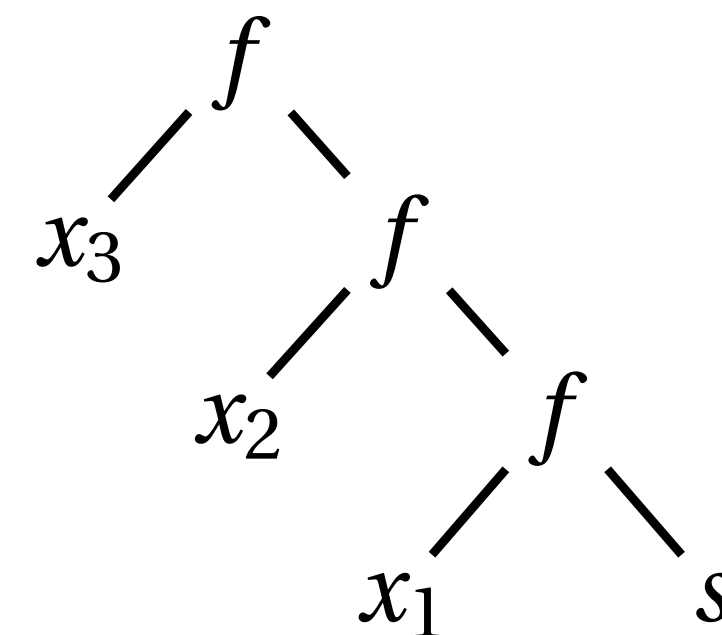


$$\alpha * \beta \rightarrow \beta$$

Argument

Akku

$$\text{foldl } f \ s \ [x_1, x_2, x_3] = f(x_3, f(x_2, f(x_1, s))) =$$



# Beispiele

## ► Reversierung:

```
fun rev nil      = nil  
  | rev (x::xr) = rev xr @ [x]
```

*val rev:  $\alpha$  list  $\rightarrow$   $\alpha$  list*

```
fun rev xs = foldl op:: nil xs
```

## ► Länge:

```
fun length nil      = 0  
  | length (x::xr) = 1 + length xr
```

*length:  $\alpha$  list  $\rightarrow$  int*

```
fun length xs = foldl (fn (x,n) => n+1) 0 xs
```

# Laufzeitunterschiede

- ▶ **Laufzeit** eines Prozeduraufrufs: zur Ausführung des Prozeduraufrufs benötigte Zeit.

```
fun rev nil      = nil  
  | rev (x::xr) = rev xr @ [x]
```

```
fun rev' xs = foldl op:: nil xs
```

- ▶ *rev'* hat eine sehr viel kürzere Laufzeit als *rev*.
- ▶ **Grund:** Kosten durch @
- ▶ mehr dazu in **Kapitel 11**

# Konkatenation?

## ► Append:

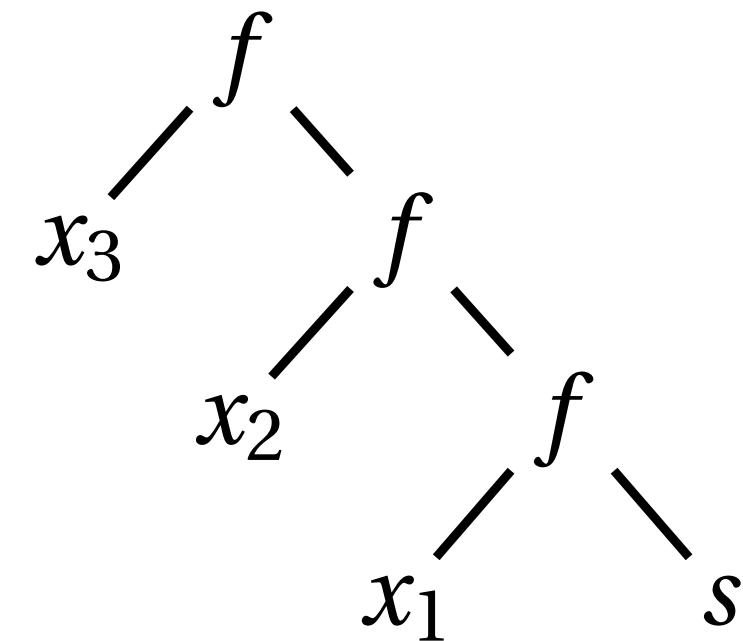
```
fun append (xs,ys) = if null xs then ys
                    else hd xs :: append(tl xs, ys)
```

*val append :  $\alpha$  list \*  $\alpha$  list  $\rightarrow$   $\alpha$  list*

```
foldl op:: [3,4] [1,2]
  = foldl op:: [3,4] (1::[2])
  = foldl op:: (op::(1,[3,4])) [2]
  = foldl op:: [1,3,4] (2::nil)
  = foldl op:: [2,1,3,4] nil
  = [2,1,3,4]
```

# Foldl und foldr

$$\text{foldl } f \ s \ [x_1, x_2, x_3] = f(x_3, f(x_2, f(x_1, s))) =$$

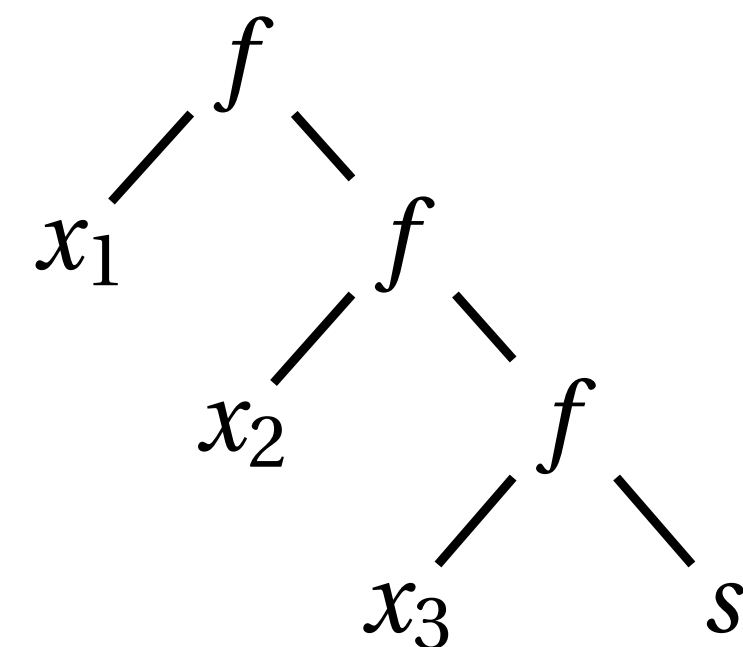


```
fun foldl f s nil      = s
```

```
  | foldl f s (x::xr) = foldl f (f(x,s)) xr
```

```
val foldl : (α * β → β) → β → α list → β
```

$$\text{foldr } f \ s \ [x_1, x_2, x_3] = f(x_1, f(x_2, f(x_3, s))) =$$



```
fun foldr f s nil      = s
```

```
  | foldr f s (x::xr) = f(x, foldr f s xr)
```

```
val foldr : (α * β → β) → β → α list → β
```

# Konkatenation

```
fun foldr f s nil      = s
  | foldr f s (x::xr) = f(x, foldr f s xr)
```

*val foldr : ( $\alpha * \beta \rightarrow \beta$ )  $\rightarrow \beta \rightarrow \alpha$  list  $\rightarrow \beta$*

```
foldr op:: [3,4] [1,2]
  = foldr op:: [3,4] (1::[2])
  = op::(1, foldr op:: [3,4] [2])
  = op::(1, foldr op:: [3,4] (2::nil))
  = op::(1, op::(2, foldr op:: [3,4] nil))
  = op::(1, op::(2, [3,4]))
  = op::(1, [2,3,4])
  = [1,2,3,4]
```



# Beispiele

## ► Konkatenation zweier Listen:

```
fun append (xs,ys) = if null xs then ys  
                    else hd xs :: append(tl xs, ys)
```

*val append :  $\alpha$  list \*  $\alpha$  list  $\rightarrow$   $\alpha$  list*

```
fun append (xs,ys) = foldr op:: ys xs
```

## ► Konkatenation zweier Elementlisten:

```
fun concat nil      = nil  
  | concat (x::xr) = x @ concat xr
```

*val concat :  $\alpha$  list list  $\rightarrow$   $\alpha$  list*

```
fun concat xs = foldr op@ nil xs
```

# Beispiele

## ► Map:

```
fun map f nil      = nil
  | map f (x::xr) = (f x) :: (map f xr)
```

$map : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$

```
fun map f = foldr (fn (x,yr) => (f x)::yr) nil
```

## ► Filter:

```
fun filter f nil      = nil
  | filter f (x::xr) = if f x then x :: filter f xr
                      else filter f xr
```

$val filter : (\alpha \rightarrow bool) \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ list}$

```
fun filter f =
  foldr (fn (x,ys) => if f x then x::ys else ys) nil
```

# Frage

**Sei  $xs$  eine beliebige Liste ganzer Zahlen.  
Welcher Ausdruck liefert den größeren Wert?**

▶ `foldl op+ 0 xs`



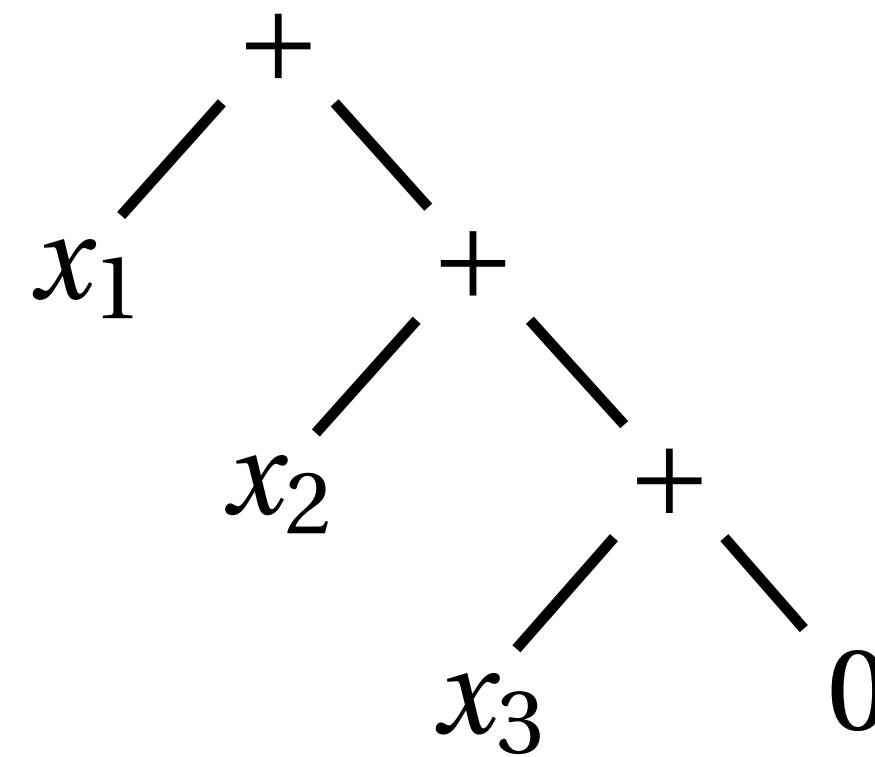
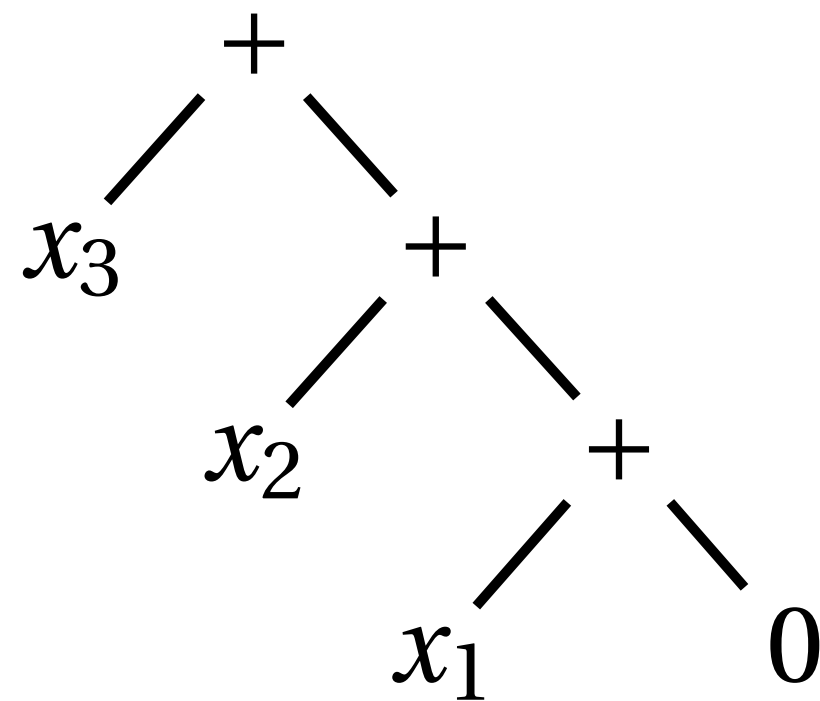
▶ `foldr op+ 0 xs`

▶ Die beiden Ausdrücke liefern den gleichen Wert.



# Foldl vs. foldr

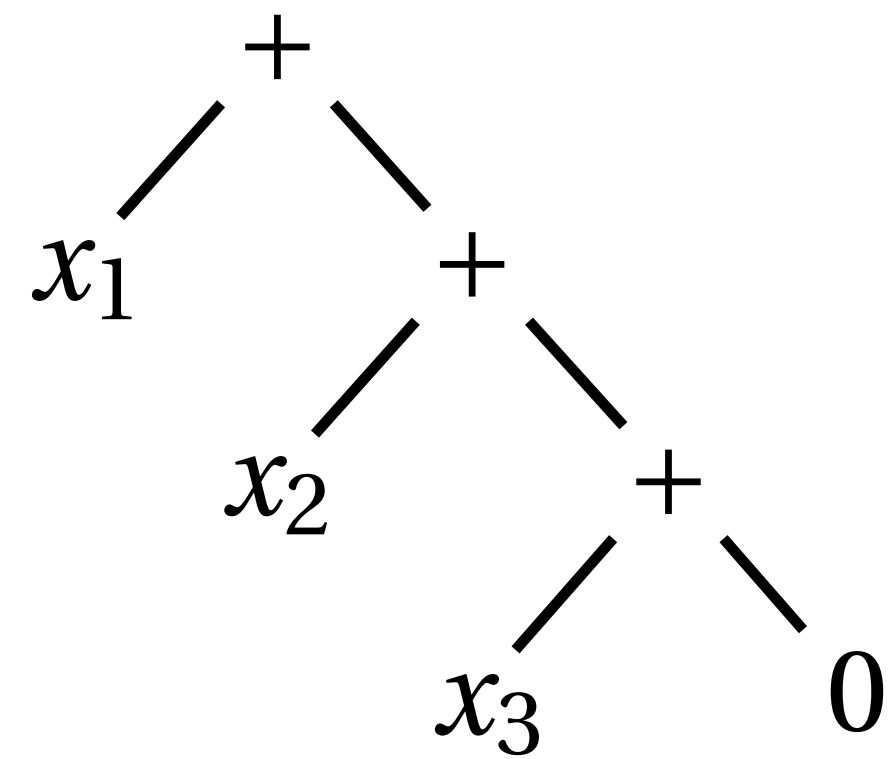
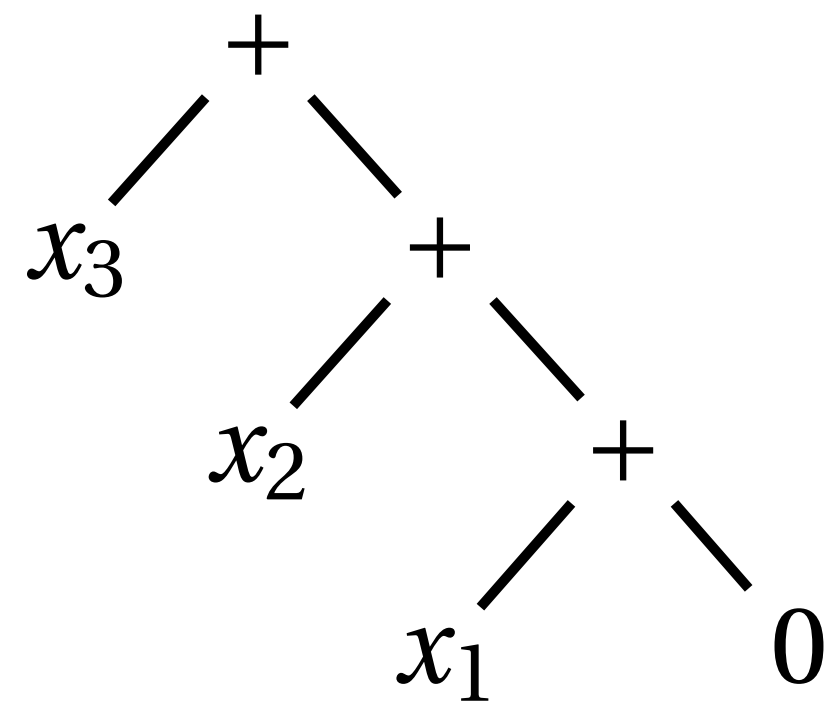
- ▶ **Summe der Elemente ganzzahliger Listen:**



`fun sum xs = foldl op+ 0 xs`

`fun sum xs = foldr op+ 0 xs`

- ▶ Wenn die **Verknüpfung kommutativ und assoziativ** ist, dann sind *foldl* und *foldr* **semantisch äquivalent**.
- ▶ Wenn man zwischen *foldl* und *foldr* wählen kann, sollte man sich für das endrekursive *foldl* entscheiden!



```
fun sum xs = foldl op+ 0 xs
```

```
fun sum xs = foldr op+ 0 xs
```

```
foldl op+ 0 [1,4,6]
= foldl op+ 1 [4,6]
= foldl op+ 5 [6]
= foldl op+ 11 nil
= 11
```

```
foldr op+ 0 [1,4,6]
= op+ (1, foldr op+ 0 [4,6])
= op+ (1, op+ (4, foldr op+ 0 [6]))
= op+ (1, op+ (4, op+ (6, foldr op+ 0 nil)))
= op+ (1, op+ (4, op+(6,0)))
= 11
```

# Kapitel 5

## Sortieren

# Sortierte Listen

- ▶ Eine Liste  $[x_1, x_2, \dots, x_n]$  über *int* heißt **aufsteigend sortiert** (oder kurz: **sortiert**) wenn  $x_1 \leq x_2 \leq \dots \leq x_n$ .
- ▶ Eine Liste  $[x_1, x_2, \dots, x_n]$  über *int* heißt **absteigend sortiert** wenn  $x_1 \geq x_2 \geq \dots \geq x_n$ .
- ▶ Eine Liste *xs* zu **sortieren** bedeutet, die **Anordnung** der Elemente in *xs* so zu verändern, dass die Liste sortiert ist.
- ▶ Die so entstandene Liste heißt die **Sortierung** von *xs*.
- ▶ **Beispiele:**

$$\text{sort } [] = []$$
$$\text{sort } [x] = [x]$$
$$\text{sort } [3, 4, 1, 4, 2, 3] = [1, 2, 3, 3, 4, 4]$$

# Sortieren durch Einfügen

- ▶ Idee: Nimm eine leere Liste  
und füge Element für Element an der richtigen Stelle ein
- ▶ Beispiel: `sort [3, 4, 1, 4, 2, 3]`

▶ [ ]

3

▶ [ 3 ]

4

▶ [ 3, 4 ]

1

▶ [ 1, 3, 4 ]

4

▶ [ 1, 3, 4, 4 ]

2

▶ [ 1, 2, 3, 4, 4 ]

3

▶ [ 1, 2, 3, 3, 4, 4 ]



# Sortieren durch Einfügen

- **Insert:** Füge in eine sortierte Liste einen Wert ein, so dass sich wieder eine sortierte Liste ergibt.

```
fun insert (x, nil)    = [x]
  | insert (x, y::yr) = if x<=y then x::y::yr else y::insert(x,yr)
val insert : int * int list → int list
```

```
insert(3, [0,1,2,3,4,5])
= 0::insert(3,[1,2,3,4,5])
= 0::1::insert(3,[2,3,4,5])
               = 0::1::2::insert(3,[3,4,5])
= 0::1::2::3::3::[4,5]
= [0,1,2,3,3,4,5]
```

# Sortieren durch Einfügen

$$\text{sort } [x_1, x_2, x_3] = \begin{array}{c} \text{insert} \\ \swarrow \quad \searrow \\ x_3 \quad \text{insert} \\ \swarrow \quad \searrow \\ x_2 \quad \text{insert} \\ \swarrow \quad \searrow \\ x_1 \quad [] \end{array} = \text{foldl insert [] } [x_1, x_2, x_3]$$

```
fun isort xs = foldl insert nil xs
```

```
val isort : int list → int list
```

## Beispiel:

```
isort [1,3,1] = foldl insert nil [1,3,1]
              = foldl insert (insert(1,nil)) [3,1]
              = foldl insert [1] [3,1]
              = foldl insert (insert(3,[1])) [1]
              = foldl insert [1,3] [1]
              = foldl insert (insert(1,[1,3])) nil
              = foldl insert [1,1,3] nil = [1,1,3]
```

# Frage

**Die Listen `xs` und `ys` seien wie folgt deklariert:**

```
val xs = List.tabulate(5000, fn x => x)
val ys = rev xs
```

**Welche Liste wird von `isort` schneller sortiert?**




- ▶ `xs`
- ▶ `ys`
- ▶ gleich schnell

# Frage

**Die Listen `xs` und `ys` seien wie folgt deklariert:**

```
val xs = List.tabulate(5000, fn x => x)
val ys = rev xs
```

**Welche Liste wird von `isort` schneller sortiert?**

- ▶ `xs` 
- ▶ `ys` 
- ▶ gleich schnell 

# Polymorphes Sortieren

```
fun pisort compare =  
  let  
    fun insert (x, nil)    = [x]  
      | insert (x, y::yr) = case compare(x,y) of  
                              GREATER => y::insert(x,yr)  
      | _                => x::y::yr  
  in  
    foldl insert nil  
  end  
val pisort : (α * α → order) → α list → α list
```

```
pisort Int.compare [5, 2, 2, 13, 4, 9, 9, 13, ~2]  
[~2, 2, 2, 4, 5, 9, 9, 13, 13] : int list
```

```
pisort Real.compare [5.0, 2.0, 2.0, 13.0, 4.0, 9.0]  
[2.0, 2.0, 4.0, 5.0, 9.0, 13.0] : real list
```

# Ordnungen

- **Typ Order:** drei mögliche Werte: *LESS, GREATER, EQUAL*

*Int.compare(2,6) = LESS*

*Int.compare(6,2) = GREATER*

*Int.compare(6,6) = EQUAL*

- **Inverse Ordnung:**

```
fun invert (compare : 'a * 'a -> order) (x,y) = compare (y,x)
```

*invert: ( $\alpha * \alpha \rightarrow order$ )  $\rightarrow \alpha * \alpha \rightarrow order$*

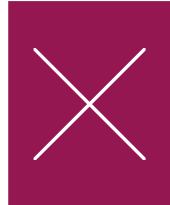



- **Absteigend Sortieren:**

```
pisort (invert Int.compare) [5, 2, 2, 13, 4, 9, 9, 13, ~2]
```

*[13, 13, 9, 9, 5, 4, 2, 2, ~2] : int list*

# Frage

Welche der folgenden Prozeduren liefern eine (aufsteigend) sortierte Liste?

- ▶ `pisort (invert Int.compare) o rev` 
- ▶ `rev o (pisort (invert Int.compare))` 
- ▶ `rev o (pisort Int.compare) o rev` 
- ▶ `rev o (pisort (invert Int.compare)) o rev` 

# Ordnungen auf Zeichen und Strings

- ▶ **Zeichen:** Vergleich basiert auf der Ordnung des Zeichenstandards

`#"B" < #"a"`

- ▶ **Strings: lexikalische Ordnung**

`"Aarlborg" < "Aaron"`

Die Zeichen zweier Wörter werden paarweise verglichen, bis zur **ersten Position**, in der **keine Gleichheit** vorliegt.

**Diese Position** entscheidet über die Ordnung der Wörter.

Wenn ein Wort in allen Positionen mit einem **längeren Wort** übereinstimmt, ist es kleiner als das längere Wort.

```
pisort String.compare ["ba", "bb", "b", "c", "d", "bac"]  
= ["b", "ba", "bac", "bb", "c", "d"]
```



# Lexikalische Ordnung auf Listen

Die Positionen zweier anzuordnender Listen werden solange paarweise verglichen, bis die **erste Position** erreicht ist, in der **keine Gleichheit** vorliegt.

**Diese Position** entscheidet dann über die Anordnung der Listen. Eine Liste, die auf allen Positionen mit einer **längeren Liste** übereinstimmt ist kleiner als die längere Liste.

```
fun lex (compare : 'a * 'a -> order) p = case p of
  (nil, _::_) => LESS
| (nil, nil)  => EQUAL
| (_::_, nil) => GREATER
| (x::xr, y::yr) => case compare(x,y) of
  EQUAL => lex compare (xr,yr)
  |      s      => s
```

$lex : (\alpha * \alpha \rightarrow order) \rightarrow \alpha list * \alpha list \rightarrow order$

# Sortieren durch Mischen (Mergesort)

## ► Idee:

Drei Schritte:

1. **Split:** Teile die Liste in zwei etwa gleich große Teile
2. **Sort:** Sortiere die Teile
3. **Merge:** Sortiere beim Zusammenfügen

## ► Beispiel: `sort [2, 8, 5, 3]`

- `split [2, 8, 5, 3] = ([5, 2], [3, 8])`
- `sort [5, 2] = [2, 5]`
- `sort [3, 8] = [3, 8]`
- `merge([2, 5], [3, 8]) = [2, 3, 5, 8]`

[www.prog1.saarland](http://www.prog1.saarland)