# Strukturelle Rekursion auf Listen als Higher-Order Funktion

```
sum  [ ]      =  0
sum  (x : xs) =  x  +  sum  xs
```

```
prod  [ ]      =  1
prod  (x : xs) =  x  *  prod  xs
```

- Die Listenfunktionen zum Summieren bzw. Multiplizieren von Listenelementen weisen dasselbe Rekursionsmuster auf, das sich mit Hilfe einer vordefinierten Funktion zum „Falten“ von zweistelligen Operatoren in Listen realisieren lässt:

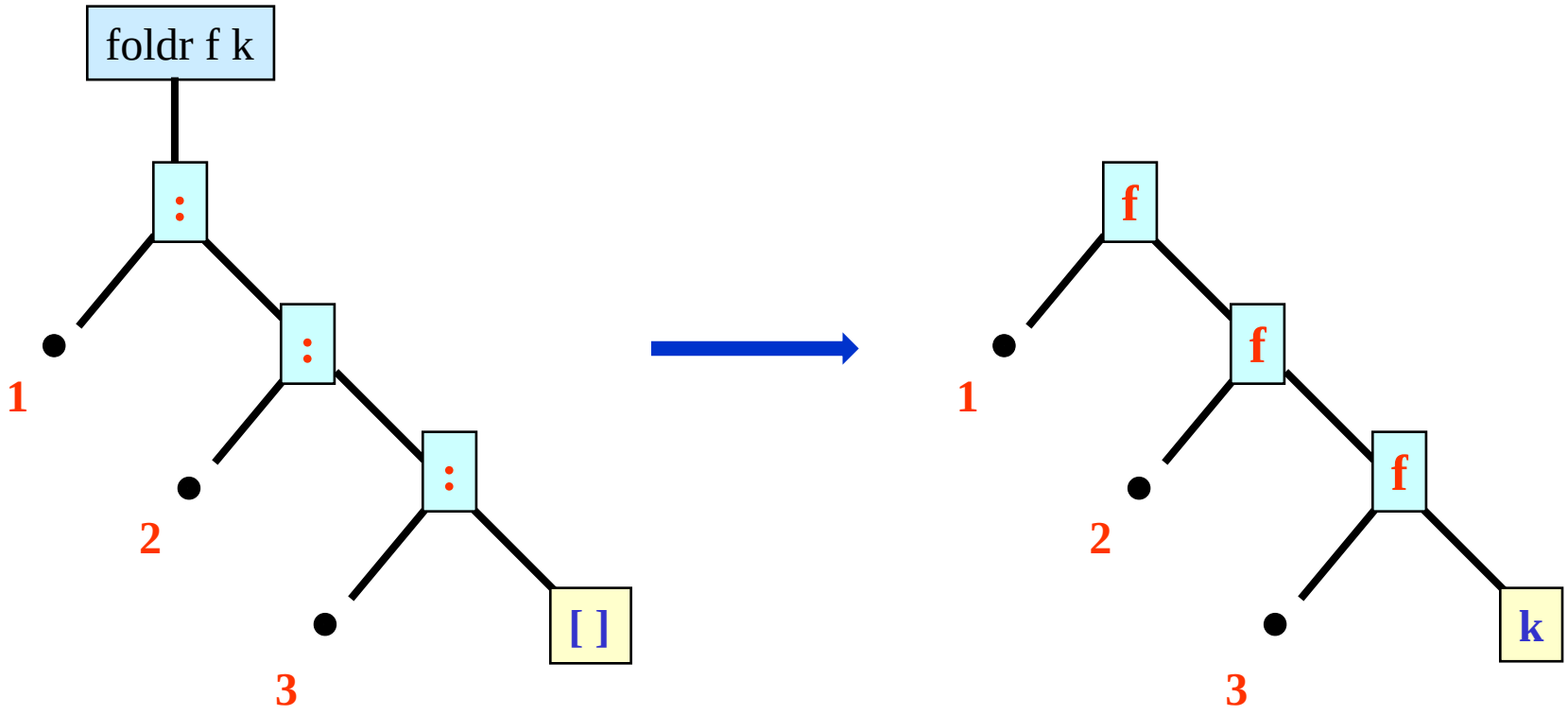
```
foldr  :: (a → b → b) → b → [a] → b
foldr  f  k  [ ]      =  k
foldr  f  k  (x : xs) =  f  x  (foldr  f  k  xs)
```

engl. „fold“: „falten“
(..r steht für „right“;
es gibt auch foldl)

- Zum Beispiel Definitionen von sum bzw. prod als Anwendung von foldr:

```
sum, prod :: [Int] → Int
sum  =  foldr  (+)  0
prod  =  foldr  (*)  1
```

# Visualisierung von foldr



# Weitere Beispiele für Verwendung von foldr

- Unter Verwendung von foldr sind vordefinierte logische Junktoren implementiert, die auf Listen von Booleschen Werten operieren:

```
and, or  :: [Bool] → Bool
and  =  foldr (&&) True
or   =  foldr (||) False
```

- „Quantoren“ über Listen sind als Verallgemeinerung dieser Junktoren mittels Komposition realisiert:

```
any, all  :: (a → Bool) → [a] → Bool
any p  =  or . map p
all p  =  and . map p
```

z.B.: `all (<100) [ x^2 | x ← [1 .. 19] ]`

# Allgemeine Strategie zur Verwendung von foldr

- Wann kann eine Funktion mittels foldr ausgedrückt werden?
- Wann immer es möglich ist, sie in folgende Form zu bringen:

```
g [ ]       = k
g (x : xs)  = f  x (g xs)
```

für irgendwelche k und f

- Dann:

```
g  =  foldr  f  k
```

- Dies liefert eine einfache Charakterisierung strukturell rekursiver Funktionen auf Listen!

# Eine linkslastige Variante von foldr

- Neben foldr gibt es:

```
foldl :: (b → a → b) → b → [a] → b
foldl f k [ ]       = k
foldl f k (x : xs)  = foldl f (f k x) xs
```