# Tutorium - Haskell in der Schule

Ralf Dorn - Dennis Buchmann - Felix Last - Carl Ambroselli

Otto-Nagel-Gymnasium
Berlin - Biesdorf

# Wer sind wir?



Otto-Nagel-Gymnasium in Berlin-Biesdorf

Hochbegabtenförderung und
MacBook-Schule

Leistungskurse seit 2005

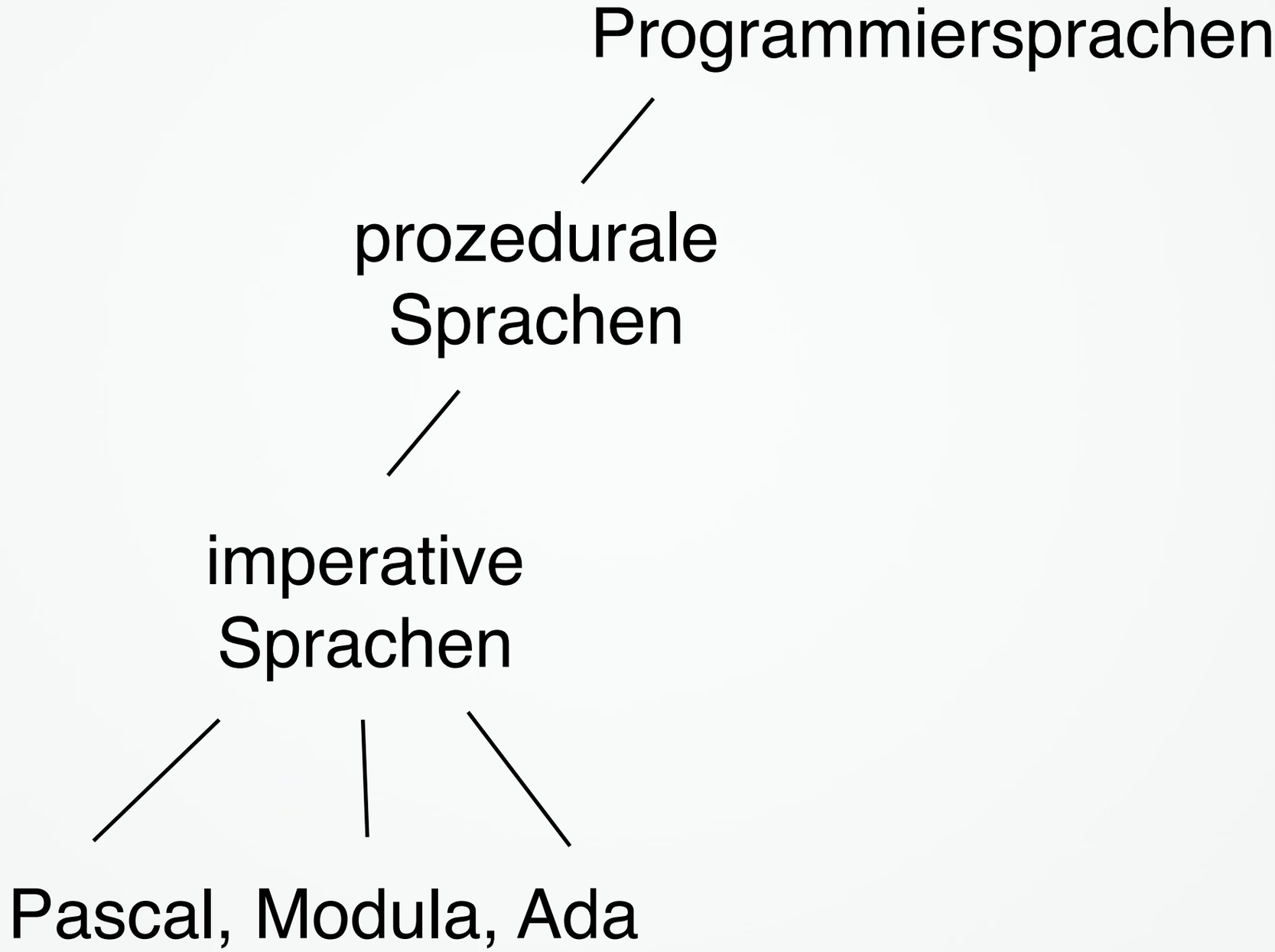
Ralf Dorn - Dennis Buchmann - Felix Last - Carl Ambroselli

# Was ist funktionale Programmierung?

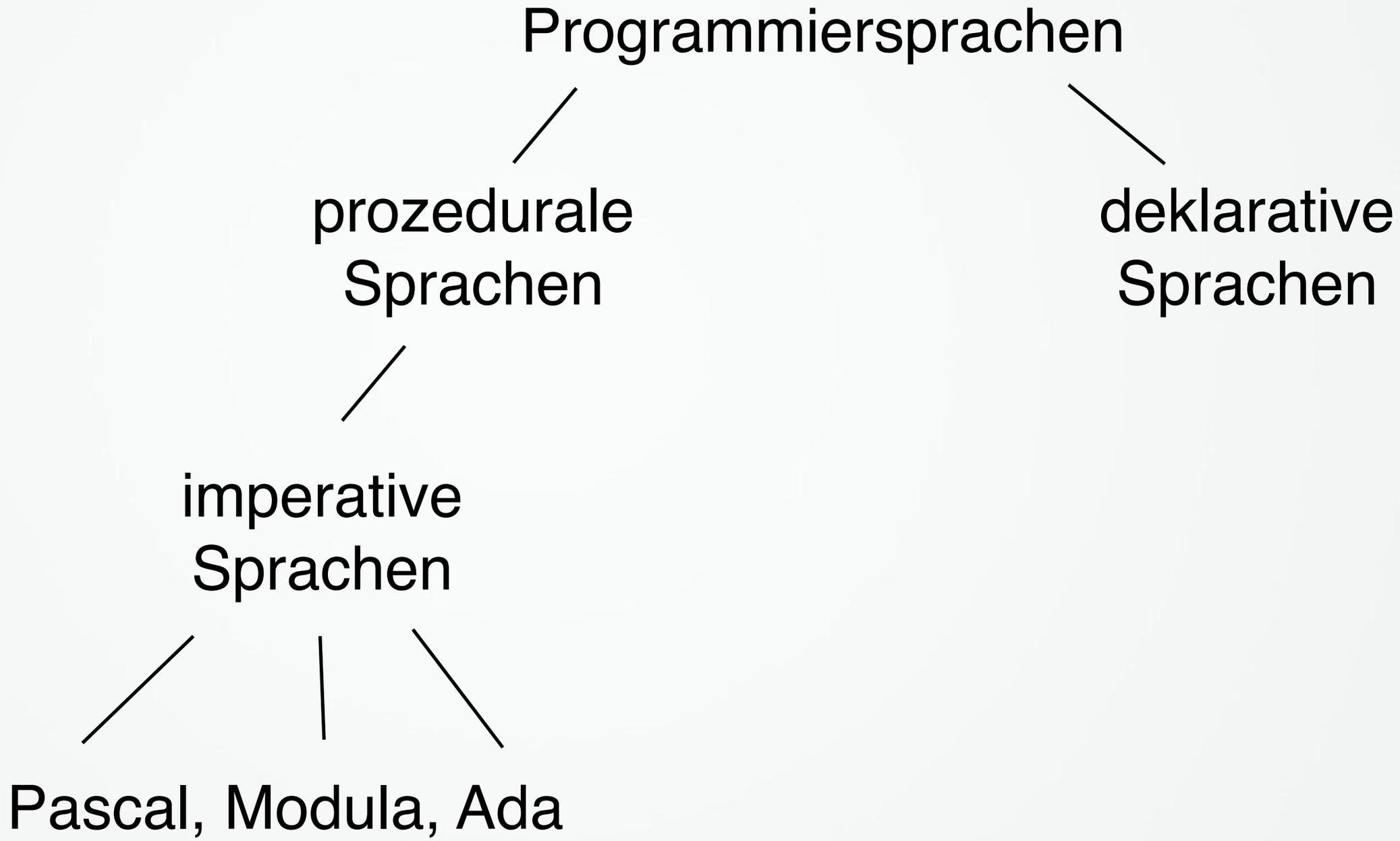
Ralf Dorn - Dennis Buchmann - Felix Last - Carl Ambroselli

Otto-Nagel-Gymnasium
Berlin - Biesdorf

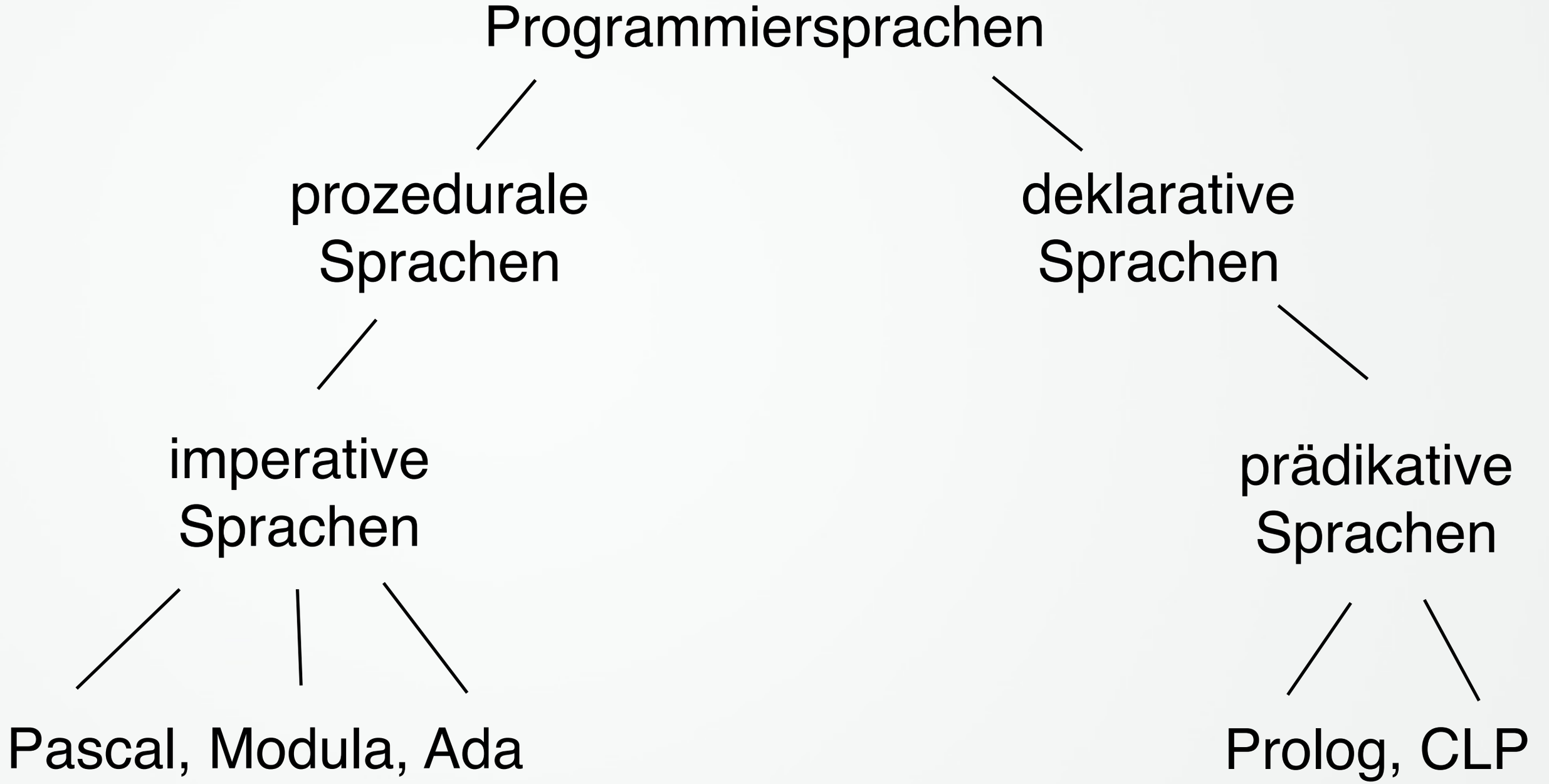
# Was ist funktionale Programmierung?



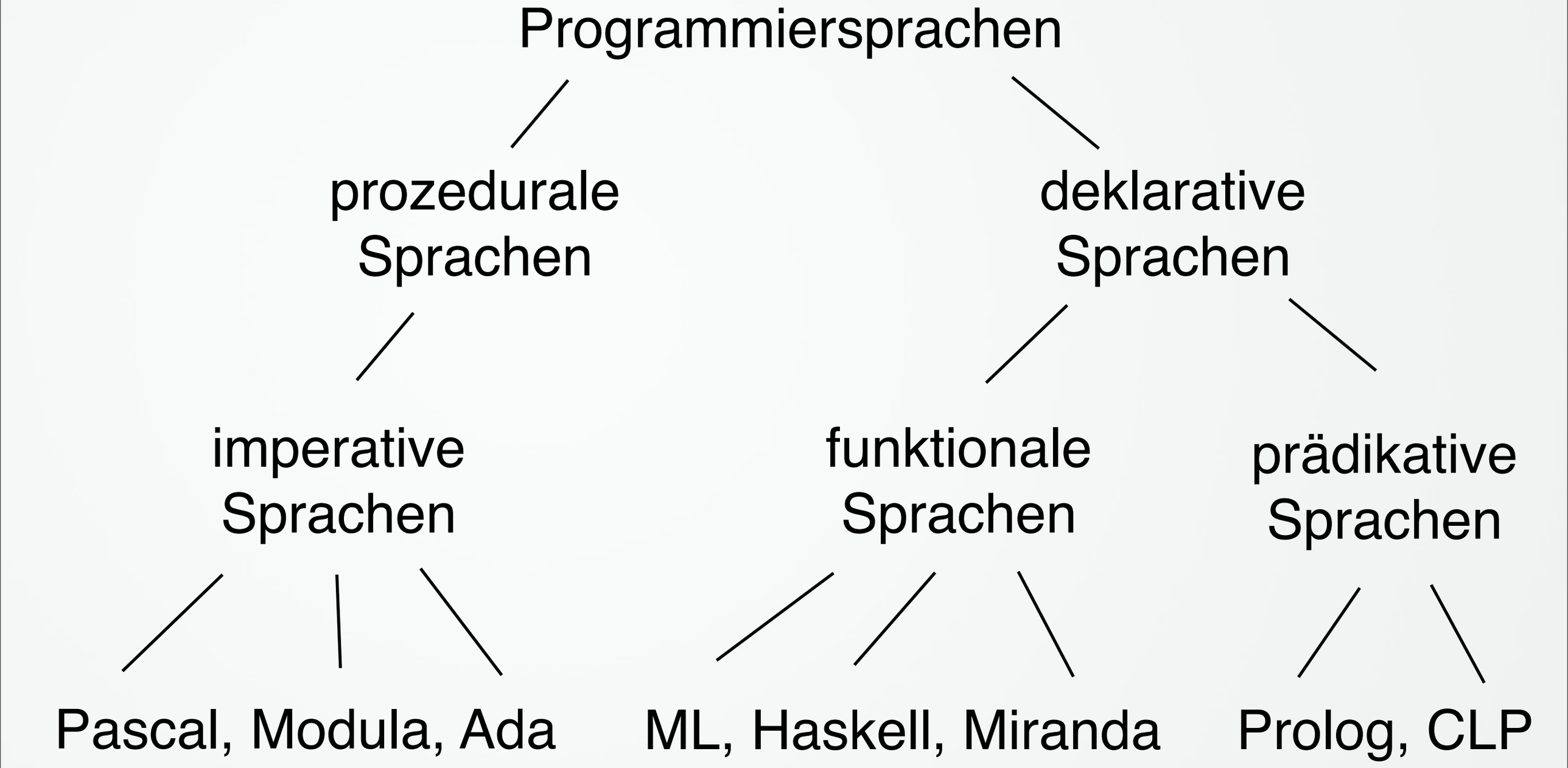
# Was ist funktionale Programmierung?



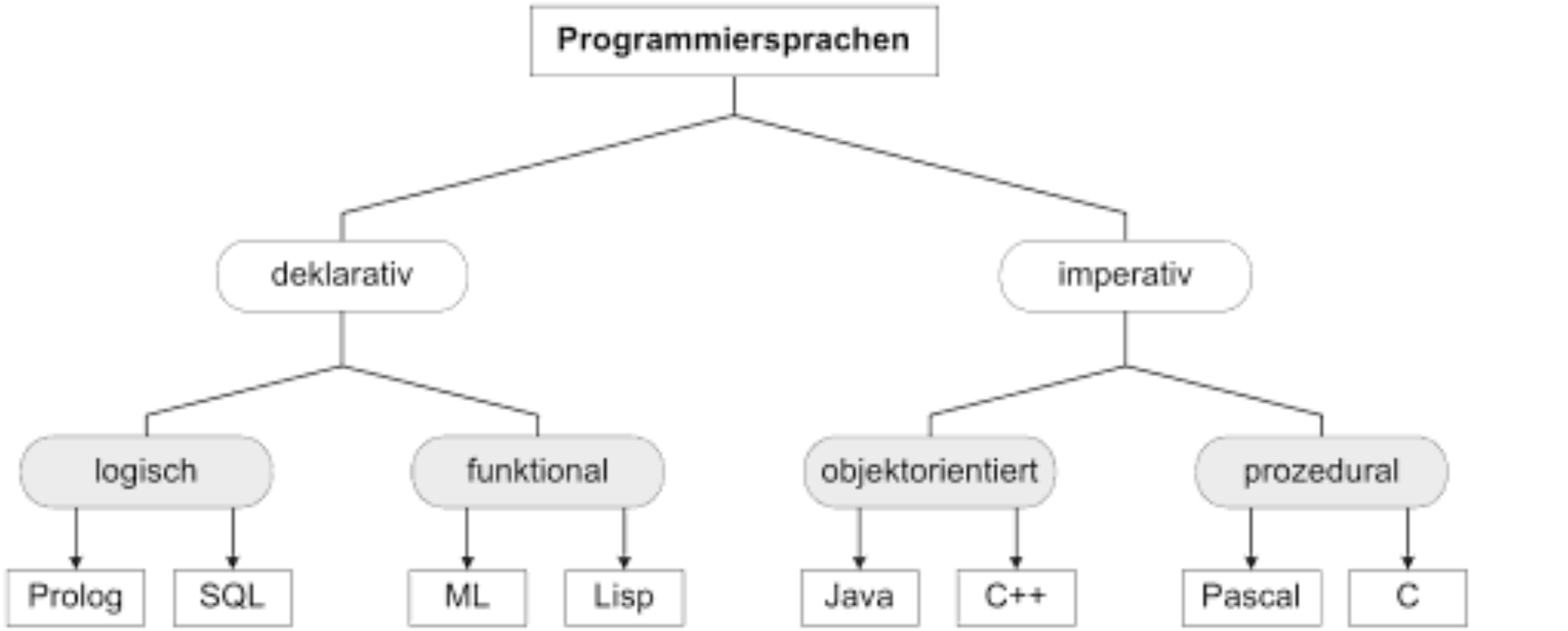
# Was ist funktionale Programmierung?



# Was ist funktionale Programmierung?



# Objektorientierte Programmierung?




Quelle: Heinrich Müller, Frank Weichert: *Vorkurs Informatik*.Der Einstieg ins Informatikstudium. Vieweg+Teubner, Wiesbaden 2011

# Einführung



http://www.haskell.org

Haskell is a computer programming language. In particular, it is a polymorphically statically typed, lazy, purely functional language, quite different from most other programming languages. The language is named for Haskell Brooks Curry, whose work in mathematical logic serves as a foundation for functional languages. Haskell is based on the lambda calculus, hence the lambda we use as a logo.

# In guter Tradition: Hallo Welt!

# In guter Tradition: Hallo Welt!

```
public class HalloWelt {
2:      // Ausgabe von "Hallo Welt!"
3:
4:      public static void main(String[] args) {
5:      System.out.println("Hallo Welt!");
6:      }
7: }    // Ende Klasse HalloWelt
```

# In guter Tradition: Hallo Welt!

```
public class HalloWelt {
2:      // Ausgabe von "Hallo Welt!"
3:
4:      public static void main(String[] args) {
5:      System.out.println("Hallo Welt!");
6:      }
7: }    // Ende Klasse HalloWelt
```

Haskell:

# In guter Tradition: Hallo Welt!

```
public class HalloWelt {
2:      // Ausgabe von "Hallo Welt!"
3:
4:      public static void main(String[] args) {
5:      System.out.println("Hallo Welt!");
6:      }
7: }    // Ende Klasse HalloWelt
```

## Haskell:

```
1: ausgabe = „Hallo Welt!“

oder

1: ausgabe = putStr „Hallo Welt!“
```

# Einfache Datentypen

## Char – Zeichen (ASCII)

## Num – Zahlen

Ganzzahlig: Int (begrenzt), Integer (unbegrenzt)
Fließkommazahlen: Float, Double

## String = [Char] – Zeichenkette

## Boolean – True/False (Wahrheitswert)

(&&) = „und“, (||) = „oder“, (==) = „gleich“, (/=) = „ungleich“

# Einfache Funktionen

Beispiel: Quadrieren

```
quad :: Num a => a -> a
```

Beispiel: Summe verdoppeln

```
sum2 :: Num a => a -> a -> a
```

# Einfache Funktionen

Beispiel: Quadrieren

```
quad :: Num a => a -> a

quad x = x*x
```

Beispiel: Summe verdoppeln

```
sum2 :: Num a => a -> a -> a
```

# Einfache Funktionen

Beispiel: Quadrieren

```
quad :: Num a => a -> a

quad x = x*x
```

Beispiel: Summe verdoppeln

```
sum2 :: Num a => a -> a -> a

sum2 x y = (x+y)*2
```

# Einfache Funktionen

Übung: Summe mal Produkt

```
sumPro :: Integer -> Integer -> Integer
```

Übung: Mittelwert von 3 Zahlen

```
mitt :: Float -> Float -> Float -> Float
```

# Einfache Funktionen

Übung: Summe mal Produkt

```
sumPro :: Integer -> Integer -> Integer
sumPro x y = (x+y)*(x*y)
```

Übung: Mittelwert von 3 Zahlen

```
mitt :: Float -> Float -> Float -> Float
```

# Einfache Funktionen

Übung: Summe mal Produkt

```
sumPro :: Integer -> Integer -> Integer

sumPro x y = (x+y)*(x*y)
```

Übung: Mittelwert von 3 Zahlen

```
mitt :: Float -> Float -> Float -> Float

mitt x y z = (x+y+z)/3
```

# Definieren von Variablen

Global:

```
a = 20
b = 3::float
c = True
d = ‘a‘
```

Lokal:

```
pyt a b = sqrt(quadA + quadB)
    where
    quadA = a*a
    quadB = b*b
```

# Definieren von Variablen



Übung: Summe zweier Zahlen hoch 3

```
sumPot :: Num a => a -> a -> a
```

# Definieren von Variablen

Übung: Summe zweier Zahlen hoch 3

```
sumPot :: Num a => a -> a -> a
sumPot x y = sum*sum*sum
  where sum = x+y
```

# Guards

Werden statt if-then-else-Blöcken genutzt

Beispiel: Betrag

```
betrag :: Integer-> Integer
betrag x
    |x >= 0 = x
    |otherwise = -x
```

Lösung mit if-then-else:

```
betrag2:: Integer -> Integer
betrag2 x = if x>=0 then x else -x
```

# Guards

## Übung: Prüfen auf Gleichheit

```
gleich :: Ord a => a -> a -> Bool
```

## Übung: Maximum von 3 Zahlen

```
max3 :: Ord a => a -> a -> a -> a
```

# Guards

Übung: Prüfen auf Gleichheit

```
gleich :: Ord a => a -> a -> Bool
gleich a b
    |a==b = True
    |otherwise = False
```

Übung: Maximum von 3 Zahlen

```
max3 :: Ord a => a -> a -> a -> a
```

# Guards

Übung: Prüfen auf Gleichheit

```
gleich :: Ord a => a -> a -> Bool
gleich a b
    |a==b = True
    |otherwise = False
```

Übung: Maximum von 3 Zahlen

```
max3 :: Ord a => a -> a -> a -> a
max3 x y z
    |x>=y && x>=z = x
    |y>=x && y>=z = y
    |z>=x && z>=y = z
```

# Rekursion

Beispiel: Potenzrechnung

```
pot :: Integer -> Integer -> Integer
pot x 0 = 1
pot x y = x*(pot x (y-1))
```

# Rekursion

## Übung: Fakultät

```
fak :: Integer -> Integer
```

## Übung: Summe aller Zahlen von x bis y

```
sumxy :: Integer -> Integer -> Integer
```

# Rekursion

## Übung: Fakultät

```
fak :: Integer -> Integer
fak 0 = 1
fak x = x*(fak (x-1))
```

## Übung: Summe aller Zahlen von x bis y

```
sumxy :: Integer -> Integer -> Integer
```

# Rekursion

## Übung: Fakultät

```
fak :: Integer -> Integer
fak 0 = 1
fak x = x*(fak (x-1))
```

## Übung: Summe aller Zahlen von x bis y

```
sumxy :: Integer -> Integer -> Integer
sumxy x y
    |x<y = y + (sumxy x (y-1))
    |x>y = x + (sumxy (x-1) y)
    |x==y = x
```

# Rekursion

Auswerten von „fak 4“:

```
fak 4 = 4*(fak (4-1)) = 4*(fak 3)
→  = 4*  3*(fak (3-1)) = 4*3*(fak 2)
→  = 4*3*  2*(fak (2-1)) = 4*3*2*(fak 1)
→  = 4*3*2*  1*(fak (1-1)) = 4*3*2*1*(fak 0)
→  = 4*3*2*1*  1 = 24
```

# Listen

Zusammenfassung beliebig vieler Objekte des gleichen Typs

Bsp. Liste von Zahlen, Liste von Buchstaben, Liste von Funktionen, Liste von Listen

# Listen

Kernfunktionalität von Haskell

Listendarstellung: eckige Klammern

Leere Liste (=Konstruktor): []

Liste von Zahlen: `[1,5,4] :: [Int]`

Liste von Listen:

`[['a','b','c'],['x']] :: [[Char]]`

# Listen

Listenkonstruktor: Doppelpunkt

(fügt Element vorne an die Liste an)

```
1:[2,3]   -> [1,2,3]
```

```
length:: [a] -> Int
length [] = 0
length (kopf:rest) = 1 + (length rest)
```

# Listen

Aufgaben

1. Funktion **`summe`**: berechnet Summe aller Elemente einer numerischen Liste

```
summe:: Num a => [a] -> a
```

2. Funktion **`kombiniere`**: fügt zwei Listen zusammen

```
--kombiniere [1,2] [3] -> [1,2,3]
kombiniere:: [a] -> [a] -> [a]
```

# Listen

Aufgaben

1. Funktion **`summe`**: berechnet Summe aller Elemente einer numerischen Liste

```
summe:: Num a => [a] -> a
```

# Listen

Aufgaben

1. Funktion **`summe`**: berechnet Summe aller Elemente einer numerischen Liste

```
summe:: Num a => [a] -> a
summe [] = 0
summe (x:xs) = x + summe xs
```

# Listen

Aufgaben

2. Funktion **`kombiniere`**: fügt zwei Listen zusammen

```
--kombiniere [1,2] [3] -> [1,2,3]
kombiniere:: [a] -> [a] -> [a]
```

# Listen

Aufgaben

2. Funktion **`kombiniere`**: fügt zwei Listen zusammen

```
--kombiniere [1,2] [3] -> [1,2,3]
kombiniere:: [a] -> [a] -> [a]
kombiniere [] y = y
kombiniere (x:xs) y = x:(kombiniere xs y)
```

# Listen

Aufgaben

2. Funktion **`kombiniere`**: fügt zwei Listen zusammen

```
--kombiniere [1,2] [3] -> [1,2,3]
kombiniere:: [a] -> [a] -> [a]
```

# Listen

Aufgaben

2. Funktion **`kombiniere`**: fügt zwei Listen zusammen

```
--kombiniere [1,2] [3] -> [1,2,3]
kombiniere:: [a] -> [a] -> [a]
kombiniere x y = x ++ y
```

# Listen

```
head :: [a] -> a

tail :: [a] -> [a]
```

# Listen

```
length :: [a] -> Int
```

# Listen

```
reverse :: [a] -> [a]
```

# Tupel

Zusammenfassung einer festen Anzahl von Elementen (mit unterschiedlichem Typ)

Bsp. (Matrikel-Nr., Name,Prüfung bestanden?)

(Zahl1, Zahl2)

# Tupel

Darstellung: runde Klammern

```
(916727,„Müller“,true)::

(Int,[Char],Boolean)
```

# Tupel

```
fst :: (a,b) -> a

snd :: (a,b) -> b
```

# Tupel

Aufgaben

1.Halbaddierer als Funktion umsetzen

```
--halbaddierer:: (x,y) -> (s,c)
halbaddierer:: (Bool,Bool) -> (Bool,Bool)
```

| x | y | s | c |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | I | I | 0 |
| I | 0 | I | 0 |
| I | I | 0 | I |

# Tupel

## Aufgaben

1.Halbaddierer als Funktion umsetzen

```
--halbaddierer:: (x,y) -> (s,c)
halbaddierer:: (Bool,Bool) -> (Bool,Bool)
halbaddierer (x,y) =
    (   ( (x||y) && (x!=y) ) , (x&&y)   )
```

# Höhere Funktionen

Überall dasselbe tun:

```
map :: (a -> b) -> [a] -> [b]
```

Beispiel:

```
map square [1,2,7,12,3,20]

=> [1,4,49,144,9,400]
```

# Suchproblematik

# Lineare Suche



```
linSearch :: Ord a => a -> [a] -> Bool
linSearch a [] = False
linSearch a (x:xs) | a == x = True
                   | otherwise = linSearch a xs
```

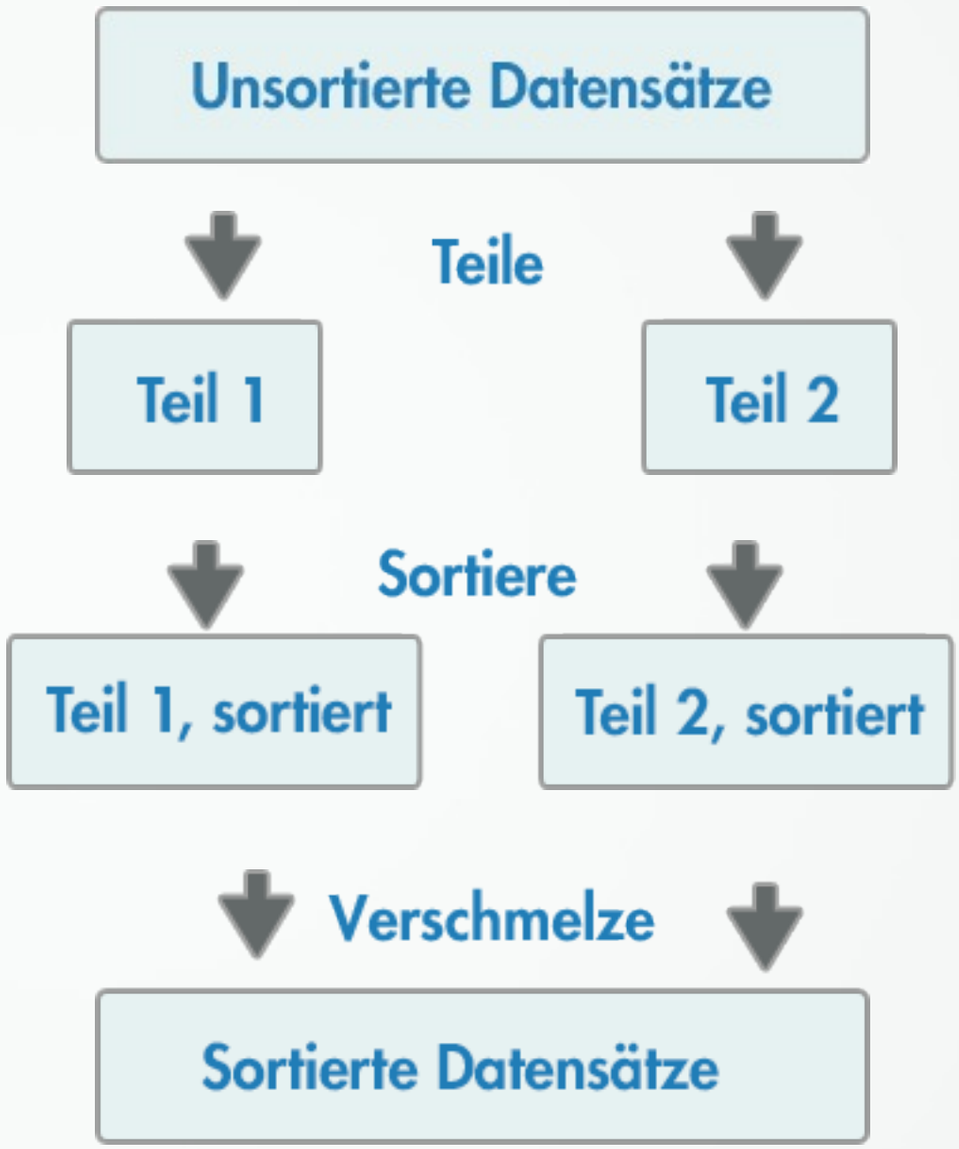
# Binäre Suche



```
binSearch :: Ord a => a -> [a] -> Bool
binSearch b [] = False
binSearch b x | (b==mid) = True
              | (b<mid) = binSearch b left
              | (b>mid) = binSearch b right
              where
              half = (length x) `div` 2
              left = take half x
              right = drop half x
              mid = head right
```

# Mergesort






Komplexität: O(n* log(n))

# Mergesort

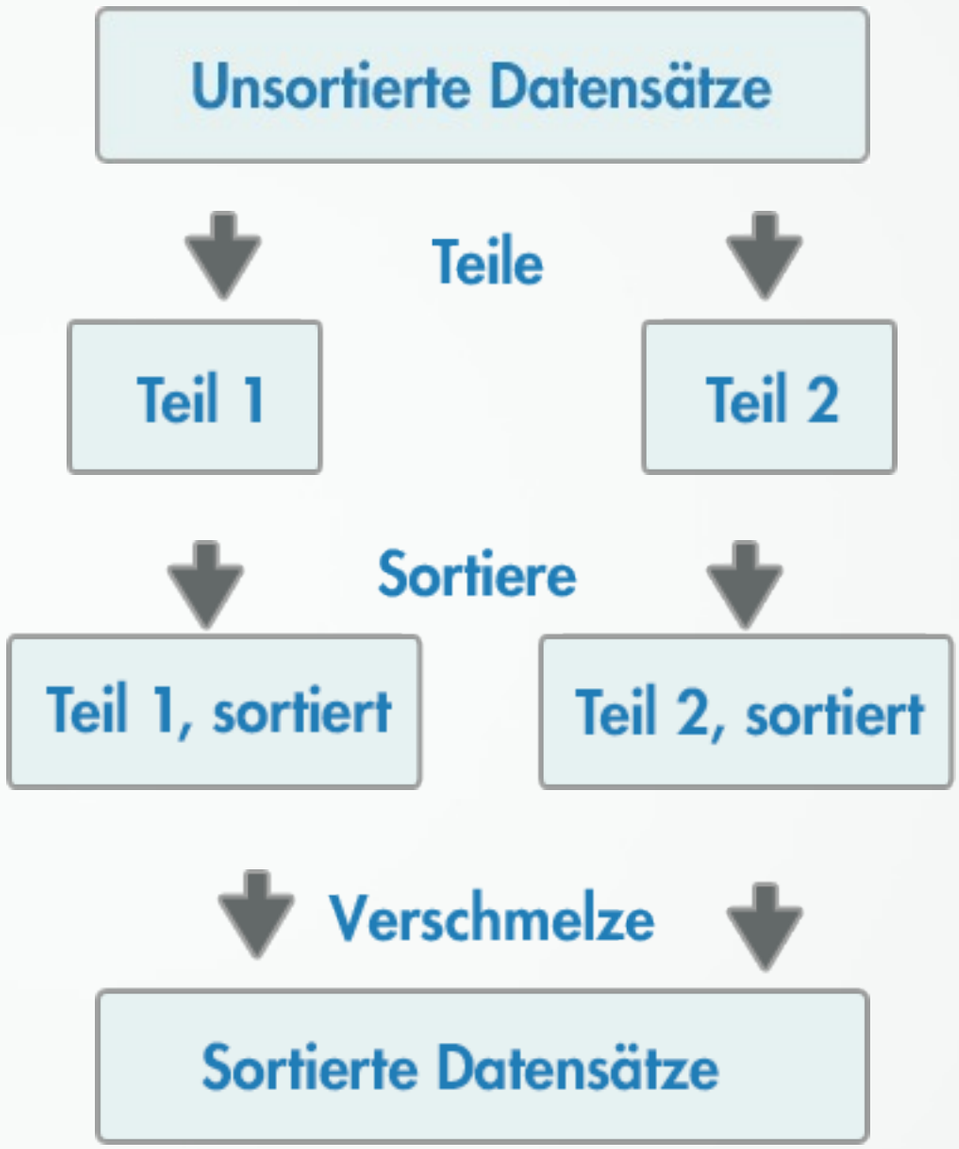


```
merge :: Ord a => [a] -> [a] -> [a]
```

```
mergesort :: Ord a => [a] -> [a]
```

# Mergesort






Komplexität: O(n* log(n))

# Mergesort



```
merge :: Ord a => [a] -> [a] -> [a]
merge [] a = a
merge a [] = a
merge (x:xs) (y:ys) | x <= y = x : merge xs (y:ys
                    | otherwise = y : merge (x:xs) ys
```

```
mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort x = merge (mergesort (take half x)) (mergesort
              (drop half x))

               where half = (length x) `div` 2
```

# Falten!

Ralf Dorn - Dennis Buchmann - Felix Last - Carl Ambroselli

Otto-Nagel-Gymnasium
Berlin - Biesdorf

# Fold

```
summe :: Num a => a -> [a] -> a
summe acc []     = acc
summe acc (x:xe) = summe (acc + x) xe
```

# Fold

```
summe :: Num a => a -> [a] -> a
summe acc []     = acc
summe acc (x:xe) = summe (acc + x) xe

> summe 0 [1,2,3,4]
= summe (0+1) [2,3,4]
= summe ((0+1)+2) [3,4]
= summe (((0+1)+2)+3) [4]
= summe ((((0+1)+2)+3)+4) []
= ((((0+1)+2)+3)+4) = 10
```

# Fold

```
foldl :: (a -> a -> a) -> a -> [a] -> a
foldl f acc []     = acc
foldl f acc (x:xe) = foldl (f acc x) xe
```

# Fold

```
foldl :: (a -> a -> a) -> a -> [a] -> a
foldl f acc []     = acc
foldl f acc (x:xe) = foldl (f acc x) xe
```

```
summe acc liste = foldl (+) acc liste
```

# Fold

```
foldl :: (a -> a -> a) -> a -> [a] -> a
foldl f acc []     = acc
foldl f acc (x:xe) = foldl (f acc x) xe
```

Füge alle Teilstrings zu einem String zusammen?

Ralf Dorn - Dennis Buchmann - Felix Last - Carl Ambroselli

Otto-Nagel-Gymnasium
Berlin - Biesdorf

# Füge alle Teilstrings zu einem String zusammen?

```
connect vorsatz liste = foldl (++) vorsatz liste
```

# Binäre Suchbäume



1. Ein Binärbaum ist eine rekursive Struktur.
2. Ein Binärbaum kann verschiedene, aber dann gleiche Daten verwalten.
3. Die Daten im Baum sind geordnet, d.h. linker Knoten < Wurzel < rechter Knoten (Invariante)

Ralf Dorn - Dennis Buchmann - Felix Last - Carl Ambroselli

Otto-Nagel-Gymnasium
Berlin - Biesdorf

# Binäre Suchbäume



1. Ein Binärbaum ist eine rekursive Struktur.
2. Ein Binärbaum kann verschiedene, aber dann gleiche Daten verwalten.
3. Die Daten im Baum sind geordnet, d.h. linker Knoten < Wurzel < rechter Knoten (Invariante)

Ralf Dorn - Dennis Buchmann - Felix Last - Carl Ambroselli

Otto-Nagel-Gymnasium
Berlin - Biesdorf

# Binäre Suchbäume



1. Ein Binärbaum ist eine rekursive Struktur.
2. Ein Binärbaum kann verschiedene, aber dann gleiche Daten verwalten.
3. Die Daten im Baum sind geordnet, d.h. linker Knoten < Wurzel < rechter Knoten (Invariante)

→
```
data Binbaum a = L | K (Binbaum a) a
                       (Binbaum a) deriving Show
```

# Beispielbäume

```
baum1 = K (K L 2 L) 5 L

baum2 = K (K L 2 L) 5 (K (K L 6 L) 8 (K
 (K L 12 L) 14 (K L 34 L)))

-- baum3 verletzt die Invariante
baum3 = K (K L 2 L) 5 (K (K L 6 L) 8 (K
 (K L 12 L) 14 (K L 4 L)))
```

# Beispielbäume

```
istleer::Binbaum a -> Bool
istleer L = True
istleer _ = False

lub::Binbaum a -> Binbaum a
-- linkerunterbaum
lub L =
lub (K l w _) = l

lwelem:: Binbaum a -> a
--linkeswurzelelement
lwelem L =
lwelem (K L w _) =
lwelem (K (K _ lw _) w r) = lw
```

# Aufgaben

Aufgabe: Gesucht ist eine Funktion **`knotenzahl::`** **`Binbaum a -> Int`**, welche die Anzahl der Knoten in einem binären Suchbaum bestimmt.

# Aufgaben

Aufgabe: Gesucht ist eine Funktion **`knotenzahl::`** **`Binbaum a -> Int`**, welche die Anzahl der Knoten in einem binären Suchbaum bestimmt.

Lösung:

```
knotenzahl :: Binbaum a -> Int
knotenzahl L = 0
knotenzahl (K l w r) = 1 + knotenzahl l
+ knotenzahl r
```

# Aufgaben



Aufgabe: Gesucht ist eine Funktion **`knotensumme:: Binbaum Int -> Int`**, welche die Summe der Knotenwerte in einem binären Suchbaum bestimmt.

# Aufgaben

Aufgabe: Gesucht ist eine Funktion **`knotensumme:: Binbaum Int -> Int`**, welche die Summe der Knotenwerte in einem binären Suchbaum bestimmt.

Lösung:

```
knotensumme :: Binbaum Int -> Int
knotensumme L = 0
Knotensumme (K l w r) = w +
        (Knotensumme l) + (knotensumme r)
```

# Aufgaben

Aufgabe: Gesucht ist eine Funktion **`istgueltig:: ???,`** welche die Gültigkeit der Invariante für binäre Suchbäume überprüft.

# Aufgaben

Lösung:

```
istgueltig:: Binbaum a -> Bool
istgueltig L = True
istgueltig (K L w L) = True
istgueltig (K L w (K l rw r)) = (w<rw)
&& istgueltig (K l rw r)
istgueltig (K (K l lw r) w R) = (lw<w)
&& istgueltig (K l lw r)
istgueltig (K (K ll lw lr) w (K rl rw
rr)) = istgueltig (K ll lw lr) && (lw<w)
&& (w<rw) && istgueltig (K rl rw rr)
```

# Aufgaben



Aufgabe: Gesucht ist eine Funktion **`doppel::`** **`Binbaum Int -> Binbaum Int`**, welche die alle Knotenwerte verdoppelt.

# Aufgaben

Aufgabe: Gesucht ist eine Funktion **`doppel::`** **`Binbaum Int -> Binbaum Int`**, welche die alle Knotenwerte verdoppelt.

Lösung:

```
doppel:: Binbaum a -> Binbaum a
doppel L = L
doppel (K l w r) = K (doppel l) (2*w)
(doppel r)

--oder mit map
????
```

# Aufgaben



Aufgabe: Gesucht ist eine Funktion **`doppel::`** **`Binbaum Int -> Binbaum Int`**, welche die alle Knotenwerte verdoppelt.

# Aufgaben

Aufgabe: Gesucht ist eine Funktion **`doppel::`** **`Binbaum Int -> Binbaum Int`**, welche die alle Knotenwerte verdoppelt.

Lösung mit map-Funktion:

```
doppel b = mapB (2*) b

mapB:: (a -> a) -> Binbaum a -> Binbaum a
mapB f L = L
mapB f (K l w r) = K (mapB f l) (f w)
 (mapB f r)
```

# Hausaufgabe ;)



Hausaufgabe: Gesucht ist eine Funktion **`anzahl::`** **`Blattbaum a -> Int`**, welche die Anzahl der Blätter in einem binären  Blattbaum bestimmt.

# Hausaufgabe ;)



Hausaufgabe: Gesucht ist eine Funktion **`anzahl::Blattbaum a -> Int`**, welche die Anzahl der Blätter in einem binären  Blattbaum bestimmt.