Hauptmerkmale und Eigenschaften funktionaler Sprachen

* polymorphes Typsystem
* Funktionsdefinitionen und Pattern Matching
* Funktionen höherer Ordnung
* Nicht-strikte Semantik (lazy evaluation)

Polymorphes Typsystem (Hindley Milner)

- „Well-typed programs do not go wrong.“
- Typinferenz (type checker) bestimmt allgemeinsten Typ von Ausdrücken
- ... und entdeckt viele Programmierfehler
- keine Typüberprüfung zur Laufzeit notwendig

Beispiel: Haskell Typsystem

- **Basistypen**: Int, Float, Char, Bool ...
- **Tupel**
- **Listen** mit Konstruktoren
  - [ ] „leere Liste“
  - (:) „Cons“ Infixoperator zum Hinzufügen eines Kopfelementes an eine bestehende Liste

  `1:(2:(3:(5:(7:(11:[])))))` erzeugt die Liste `[1,2,3,5,7,11]`.

  (:) ist linkassoziativ, d.h. die Klammern können entfallen.
- benutzerdefinierte **algebraische Datenstrukturen**
  Bsp: `data Tree = Node Int Tree Tree | Leaf Int`
- **Funktionstypen** t_1 `->` t_2
- parametrische **Polymorphie**: Typvariablen repräsentieren beliebige Typen

Polymorphe Typen

Polymorphie erlaubt die Definition von Funktionen und Datenstrukturen über beliebigen Typen, die als Variablen repräsentiert werden.

Bsp: Listen mit beliebigem Eintragstyp: `forall a.[a]`

Typvariablen werden **implizit allquantifiziert:**
Man schreibt `[a]` statt `forall a.[a].`

Es gilt: `[] :: [a]` und `(:) :: a -> [a] -> [a].`

Algebraische Datenstrukturen werden i.a. polymorph deklariert:

```
data Tree a = Node a (Tree a) (Tree a)  |  Leaf a
```

Funktionsdefinitionen und Pattern Matching

- **verschiedene definierende Gleichungen für Funktionen je nach Art der Parameter**
- **Verbesserung der Lesbarkeit**

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

Muster für Listenargument
Konstruktorterm:
- Variable
- Konstruktor appliziert auf Terme

Pattern Matching Compiling

Ein Pattern Matching Compiler überführt die Definition von Funktionen mit mehreren Gleichungen in eine interne Repräsentation mit nur einer Gleichung und speziellem Pattern Matching-Konstrukt im Rumpfausdruck.

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

```
sum :: [Int] -> Int
sum l
 = case l of
     []     -> 0
     (x:xs) ->  x + sum xs
```

Vorteile:

- **Pattern Matching wird vereinfacht. Jeder Parameter wird höchstens einmal auf Vorkommen eines bestimmten Konstruktors getestet.**
- **Detailliertere Analysen möglich.**
- **-> einfache Funktionsdefinitionen der Form**

  $f\ x_1 \ldots x_r = e$ **statt** $\langle f\ t_{i1} \ldots t_{ir} \mid 1 \leq i \leq n \rangle$

Funktionen höherer Ordnung

- **generische Programmierung mit allgemeinem Berechnungsschema**
- **modulare Programmierung, Wiederverwendbarkeit wird verbessert**
- **Funktionen als „Objekte erster Klasse“ (first class citizens)**

wichtigste Funktionen höherer Ordnung:

```
map              :: (a->b) -> [a] -> [b]
map f []         =  []
map f (x:xs)     =  f x : map f xs

foldr            :: (a->b->b) -> b -> [a] -> b
foldr f e []     =  e
foldr f e (x:xs) =  f x (foldr f e xs)

filter           :: (a->Bool) -> [a] -> [a]
filter p xs      =  [ x | x <- xs, p x]
```

Nicht-strikte Semantik (Lazy evaluation)

- **bedarfsgesteuerte Auswertung**

  -> nur benötigte Teilausdrücke werden ausgewertet

  Bsp: .... if test then x else y ....

  Entweder x oder y werden ausgewertet, aber i.a. nicht beide, insbesondere gilt:

  [| if _ then _ else _ fi |] (true, val, ⊥) = val (nicht ⊥)

- **Für eine strikte Funktion gilt immer**

  f a_1 ... a_{i-1} ⊥ a_{i+1} ... a_n = ⊥ für beliebige Werte a_j

- **Nicht-strikte Semantik bedeutet:**

  Alle Funktionen bis auf vordefinierte Basisfunktionen sind nicht strikt!

Lazy Evaluation

- Ein Funktionsargument wird nur ausgewertet, wenn sein Ergebnis zur Bestimmung des Funktionsresultates benötigt wird (call-by-need!).
- Datenkonstruktoren sind Funktionen: „Cons should not evaluate its arguments.“
- Jeder Teilausdruck wird höchstens einmal ausgewertet (-> sharing!).
- Vorteile:
  - potentiell unendliche Datenstrukturen, z.B. Ströme
  - Trennung von Daten und Kontrolle
  - interaktive Programmierung (input = character stream)

Beispielprogramm: Das Sieb des Eratosthenes

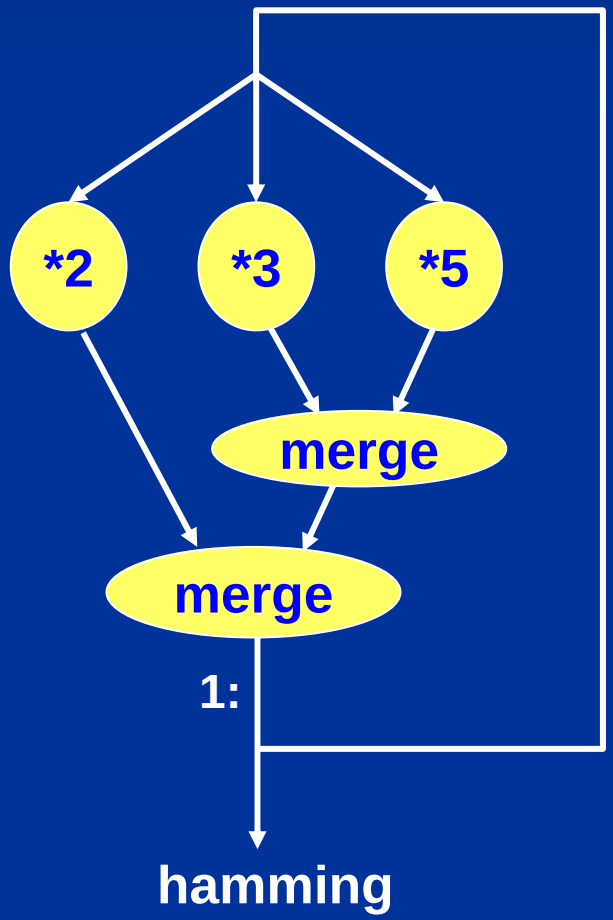
```
primes ::  [Int]
primes =  sieve [2..]

sieve  ::  [Int]  ->  [Int]
sieve  []       =  []
sieve  (x:xs)  =  x: (sieve (filter (nonMult x)  xs))

nonMult :: Int -> Int -> Bool
nonMult x y = y `mod` x > 0
```

Beispiel: Hamming-Zahlen

alle Zahlen der Form
$2^i 3^j 5^k$ (i,j,k >= 0)
in aufsteigender Reihenfolge
ohne Duplikate




```
hamming  :: [Int]
hamming  = 1: merge (map (*2) hamming)
                    (merge (map (*3) hamming)
                           (map (*5) hamming))
```

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