Softwaretechnik und Programmierparadigmen

01 Funktionale Programmierung

Prof. Dr. Sabine Glesner
Software and Embedded Systems Engineering
Technische Universität Berlin

Inhalt

Funktionale Programmierung

- Einführung
- Einführung in Haskell
- Rekursion
- Datentypen
- Funktionen höherer Ordnung
- Listenfunktionale
- Typklassen
- Unendliche Listen

Diese VL

Planung
- Entwicklungs-modelle
- Anforderungs management

Analyse und Entwurf
- Objekt-orientierter Entwurf (UML,OCL)
- Model Driven Develop ment

Implementierung
- Design Patterns
- Architekturstile
- Funktionale Programmierung (Haskell)
- Logische Programmierung (Prolog)

Qualitäts-sicherung
- Testen
- Korrektheit (Hoare-Kalkül)
- Code-Qualität

Unter-stützende Prozesse
- Konfigurations-Management
- Projekt-Management
- Deployment
- Betrieb, Wartung, Pflege
- Dokumentation

Softwaretechnik-Anteil | Programmierparadigmen-Anteil

Inhalt

Funktionale Programmierung

- Einführung
- Einführung in Haskell
- Rekursion
- Datentypen
- Funktionen höherer Ordnung
- Listenfunktionale
- Typklassen
- Unendliche Listen

Eigenschaften von Programmiersprachen

Syntax

- Legt die **wohlgeformten Ausdrücke** einer Programmiersprache fest
- Eine Menge von wohlgeformten Ausdrücken bildet ein **Programm**
- Die Menge aller möglichen Programme bildet die **Programmiersprache**

➢ Ist die **Rechtschreibung** und **Grammatik** einer Programmiersprache

Semantik

- Definiert **Bedeutung** von wohlgeformten Ausdrücken
- Gibt an wie Programme zu **interpretieren** sind

➢ Ist das **Wörterbuch** einer Programmiersprache

Achtung: Syntaktisch falsche Programme haben **keine** Bedeutung!

(Falsch geschriebene Wörter stehen nicht im Wörterbuch)

Programmierparadigmen

Programmierparadigmen Legen die **Grundregeln** einer höheren Programmiersprache fest

Behandlung und Repräsentation von…

- Statischen Elementen (Konstanten, Variablen, Methoden, Objekten)
- Dynamischen Elementen (Zuweisungen, Kontrollfluss, Datenfluss)
- „First Class Citizens“ (Grundelemente der Sprache)

…sowie Kriterien und **Empfehlungen** für

- Gute Lesbarkeit, Wartbarkeit, Effizienz
- Redundanzfreiheit und Modularität

Eine Sprache kann **mehrere** Paradigmen unterstützen!

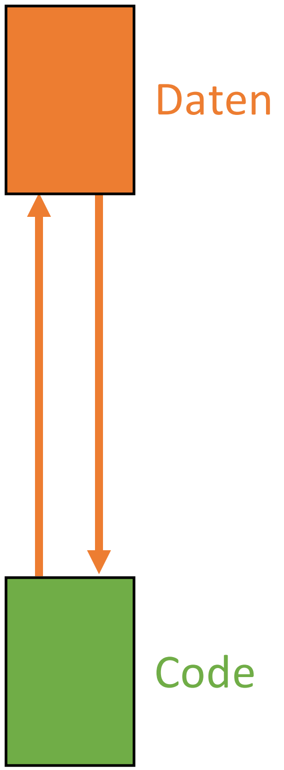
Maschinencode

Der **Zustand** ist vollständig definiert durch

- Den Programmzeiger
- Den Callstack
- Die Daten

Problem:

Mangelnde Abstraktion macht die Entwicklung umständlich, fehleranfällig und führt zu **redundantem Code**



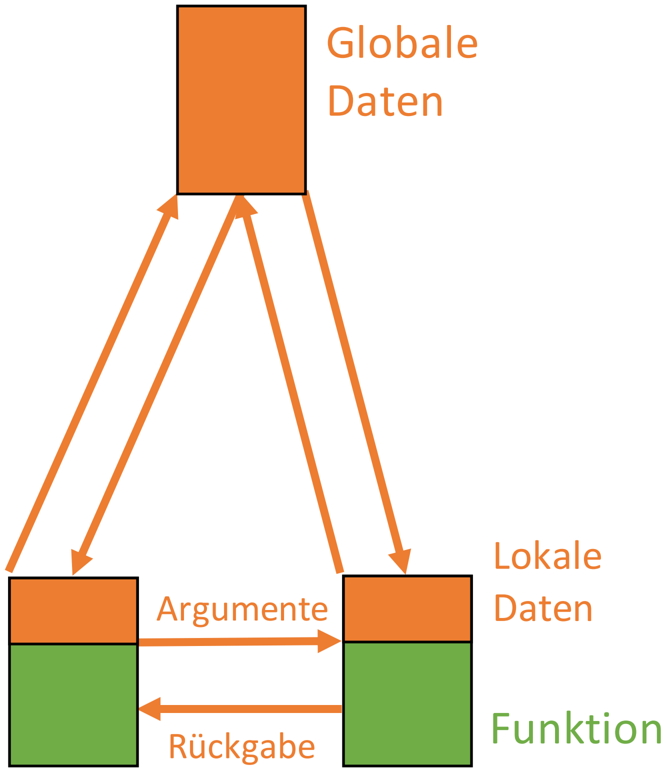
Imperativ prozedural

Der **Zustand** ist vollständig definiert durch

- Den Programmzeiger
- Den Callstack
- Die *globalen* Daten
- Die *lokalen* Daten

Problem:

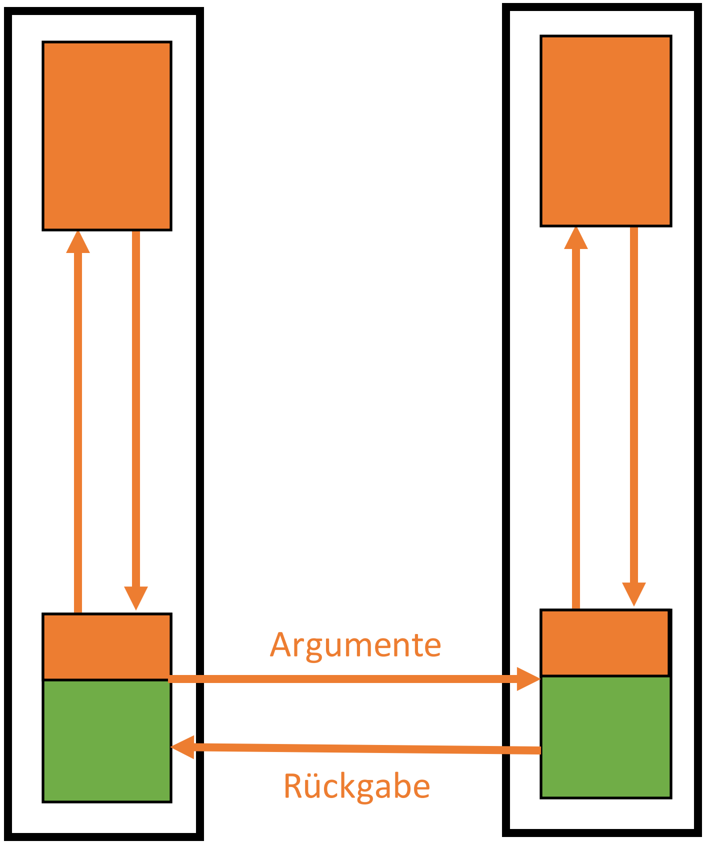
Der Informationsaustausch über **globale Daten** ist schwer nachvollziehbar und beeinflusst das Verhalten von Funktionen



Imperativ objekt-orientiert

Lösungsvorschlag 1:

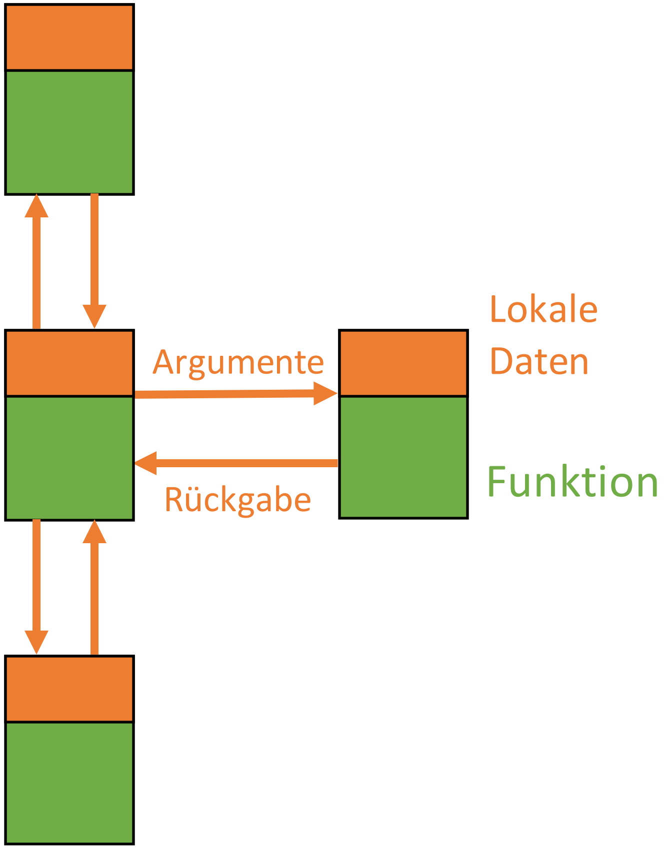
Kapselung der globalen Daten in **Objekte** mit den passenden Methoden



Funktional prozedural

Lösungsvorschlag 2:

Minimierung des Zustandsraumes durch Verzicht auf globale Daten



Imperativ vs. Funktional

Imperativ

- Ein imperatives Programm ist eine **Folge von Befehlen**, die angeben **wie** und in **welcher Reihenfolge** etwas getan wird
- **Zustandsbehaftet**, variabel
- Es gibt **Variablen**, die den Zustand zu einer bestimmten Zeit festhalten.
- Man kann **Schleifen** verwenden, um Berechnungen zu wiederholen

Funktional

- Ein deklaratives Programm ist eine **Ein-/Ausgaberelation**, die beschreibt **was** getan werden soll
- **Zustandslos**, zeitlos
- Es gibt keine Variablen außer den **Parametern**
- Man verwendet **Rekursion**, da Schleifen die Änderung des Zustands in der Zeit beschreiben würden

Funktionale Programmiersprachen

Funktionale Sprachen werden zunehmend in der Industrie eingesetzt

- Für die **nebenläufige** Verarbeitung **großer Datenmengen** wird häufig das **MapReduce-Schema** eingesetzt (z.B. Apache Hadoop)
- Twitter und Foursquare verwenden funktionale Sprachen in ihren **Backends**
- Moderne **Compiler** wie LLVM bieten Anbindung für funktionale Sprachen

Funktionale Programme sind nebeneffektfrei, zeitlos und elegant

Wichtige Vertreter funktionaler Programmiersprachen

- (Common) LISP
- ML
- GOFER
- OPAL
- HASKELL

Inhalt

Funktionale Programmierung

- Einführung
- **Einführung in Haskell**
- Rekursion
- Datentypen
- Funktionen höherer Ordnung
- Listenfunktionale
- Typklassen
- Unendliche Listen

Haskell

Haskell (1990) basiert auf den Lambda-Kalkül und wurde durch **Miranda** (1985), einer nicht-strikten funktionalen Sprache, inspiriert

Haskell ist…

- rein funktional (nebeneffektfrei)
- nicht-strikt (lazy evaluation)
- statisch getypt (mit Typinferenz)

Die wichtigste Implementierung ist der (frei verfügbare) **Glasgow Haskell Compiler** (GHC)

Konstante Funktionen

Getrennte **Deklaration** (Signatur) und **Definition** (Implementierung):

Deklaration	*Bezeichner* :: *Rückgabetyp*
Definition	*Bezeichner* = *Definition*

Beispiel: Konstante Funktion

```
goldenRatio :: Double
goldenRatio = 1.618
```

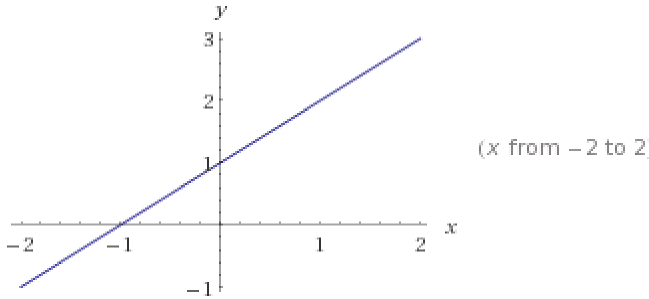
Funktionen mit Parametern

Getrennte **Deklaration** (Signatur) und **Definition** (Implementierung):

Deklaration	*Bezeichner* :: *Parametertyp -> Rückgabetyp*
Definition	*Bezeichner Parameter = Definition*

Beispiel: Nachfolgerfunktion

```
succ‘ :: Int -> Int
succ‘ x = x + 1
```



Weitere Funktionen

Beispiel: Verwendung von mehreren Parametern

```
add :: Int -> Int -> Int
add x y = x + y
```

Beispiel: Verschiedene Typen

```
div :: Int -> Int -> Double
div x y = x / y
```

Beispiel: Funktion mit Verzweigung

```
max :: Int -> Int -> Int
max x y = if x > y then x else y
```

Funktionsaufrufe

Funktionen werden auf Argumente **angewendet**

```
(funktion arg1 arg2)
```

Beispiel: mindestens 10

```
minTen :: Int -> Int
minTen v = max 10 v
```

Argumente von max mit Leerzeichen

Klammern um Argumente eindeutig zuzuordnen

Einige Operatoren verwenden **Infixnotation**

Beispiel: Präzise Konstante

```
goldenRatioPrecise :: Double
goldenRatioPrecise = ((sqrt 5) + 1) / 2
```

Typinferenz

Statisch typisiert

- d.h. der Typ jedes Ausdrucks ist zur **Compile-Zeit** bekannt.
- Macht den Code **sicherer**: Typfehler werden vor Ausführung ausgeschlossen

Typinferenz

- Haskell kann den Typ von Ausdrücken oft selbst erkennen, wenn sich das aus dem Ausdruck ergibt
- Spart Schreibarbeit, macht es übersichtlicher

Beispiel: `isSucc a b = b == succ‘ a`

- ➢ Durch `succ‘` steht `a :: Int` fest,
- ➢ durch `==` ergeben sich `b :: Int` und der Rückgabetyp `Bool`
- ➢ die Signatur `isSucc :: Int -> Int -> Bool` kann inferiert werden

Closures mit Let

Bestandteile einer Funktion können mit **let** definiert werden

```
let expr in ...
```

Beispiel: Heronische Flächenformel

```
area :: Double -> Double -> Double -> Double
area a b c = let s = (a + b + c) / 2
         in (sqrt(s * (s – a) * (s – b) * (s – c)))
```

$$A = \sqrt{s(s-a)(s-b)(s-c)}\ ; s = \frac{a+b+c}{2}$$

Inhalt

Funktionale Programmierung

- Einführung
- Einführung in Haskell
- Rekursion
- Datentypen
- Funktionen höherer Ordnung
- Listenfunktionale
- Typklassen
- Unendliche Listen

Schleifen?

Schleifen in imperativen Programmen **verändern den Zustand** in jeder Iteration

```
for(x = 0, s = 0; x < n; x++) s += n;
```

Funktionale Programme sind **zustandslos**

- Variablen können nur **einmal** belegt werden

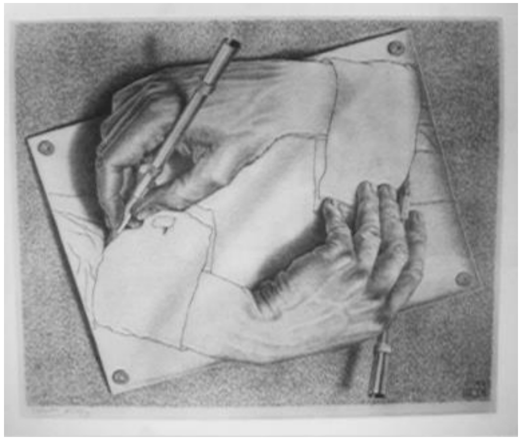
```
let x = x + 1 in x
```
geht nicht!

Statt **Schleifen** verwenden funktionale Sprachen **Rekursion.**

Rekursion!

Ein rekursives Programm **ruft sich selbst auf** (direkt oder indirekt)

Meistens enthält es eine **Abbruchbedingung**, die angibt, wann das Programm **aufhört**, sich selbst aufzurufen



„**Rekursion** kann man am besten mit **Rekursion** beschreiben."

Schleife vs. Rekursion

Imperativ (C)

```
float potenz(float x, int n) {
     float sum = 1;
     while (n--) sum *= x;
     return sum;
}
```

Funktional (Haskell)

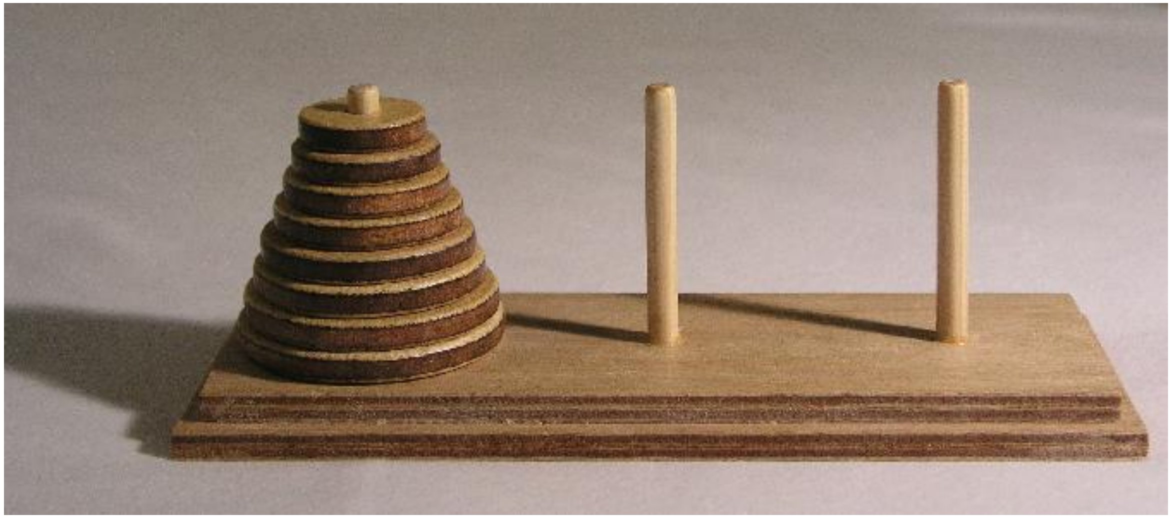
```
potenz :: Double -> Int -> Double
potenz x n = if n == 0 then 1     -- Rekursionsanker
                       else x * (potenz x (n - 1))
```

Rekursive Auswertung

```
potenz x n = if n == 0 then 1     -- Rekursionsanker
                       else x * (potenz x (n - 1))
```

```
(potenz 2 3)
(2 * (potenz 2 (3 – 1)))
(2 * (2 * (potenz 2 (2 – 1))))
(2 * (2 * (2 * (potenz 2 (1 – 1)))))
(2 * (2 * (2 * (1)))) -- Rekursionsanker
(2 * (2 * (2)))
(2 * (4))
(8)
```

Türme von Hanoi



Das Spiel besteht aus drei Stäben *A*, *B* und *C*, auf die mehrere gelochte Scheiben gelegt werden, alle verschieden groß. Zu Beginn liegen alle Scheiben auf Stab *A*, der Größe nach geordnet, mit der größten Scheibe unten und der kleinsten oben. Ziel des Spiels ist es, den kompletten Scheiben-Stapel von *A* nach *C* zu versetzen.

Bei jedem Zug darf die oberste Scheibe eines beliebigen Stabes auf einen der beiden anderen Stäbe gelegt werden, vorausgesetzt, dort liegt nicht schon eine kleinere Scheibe. Folglich sind zu jedem Zeitpunkt des Spieles die Scheiben auf jedem Feld der Größe nach geordnet.

Türme von Hanoi

Problem lässt sich durch rekursiven Algorithmus optimal lösen

N,A,B,C: Parameter

Bewege N Steine von A nach C (über B)

Rekursionsanker

Falls N = 1: Transportiere den Stein von A nach C

Rekursionsschritt

Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

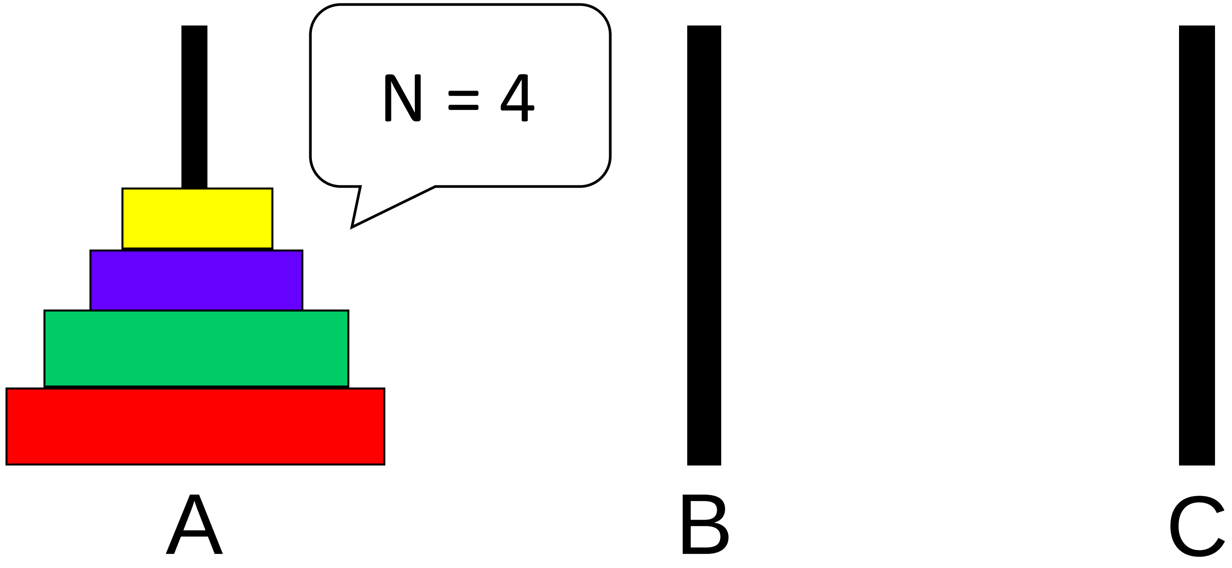
Rekursiver Aufruf

Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

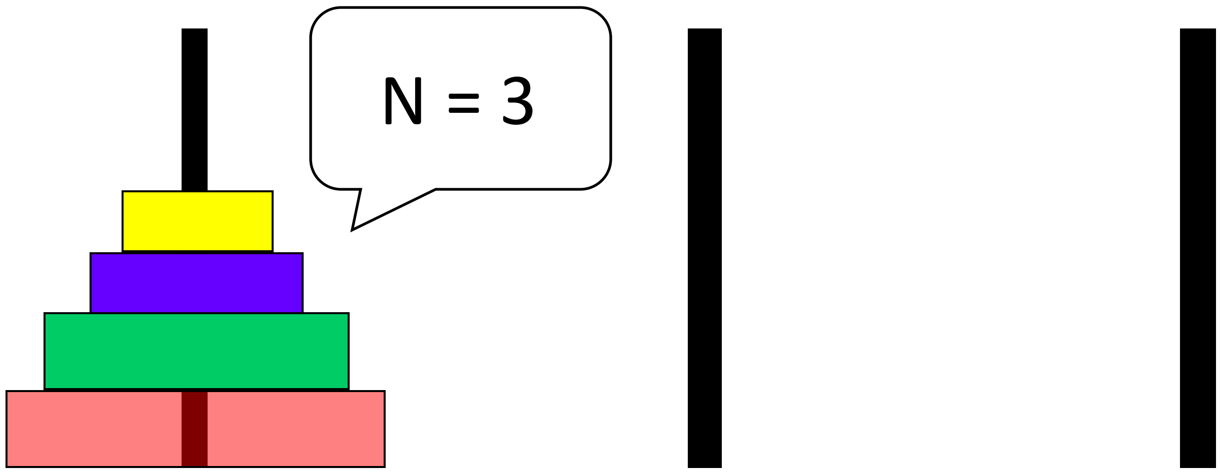


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

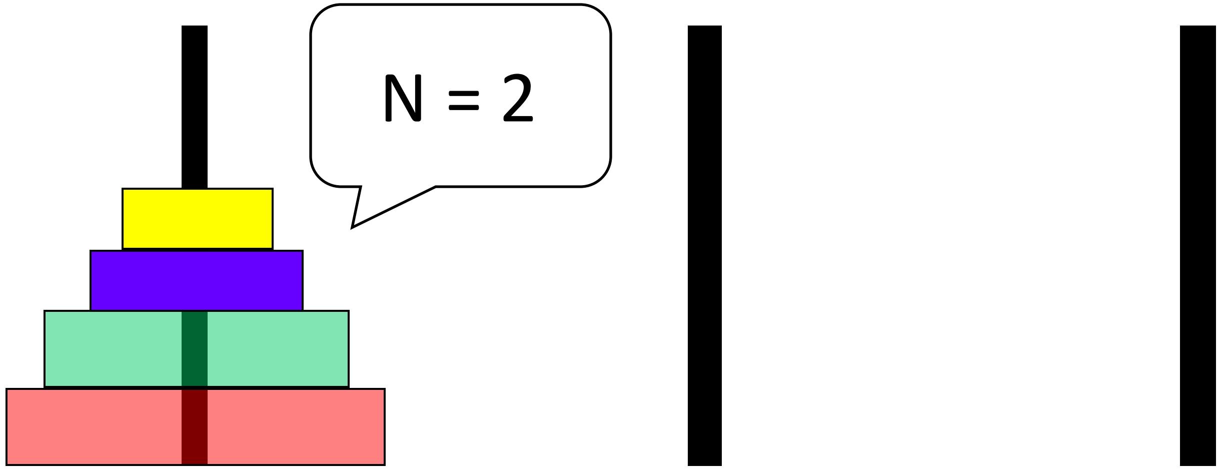


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

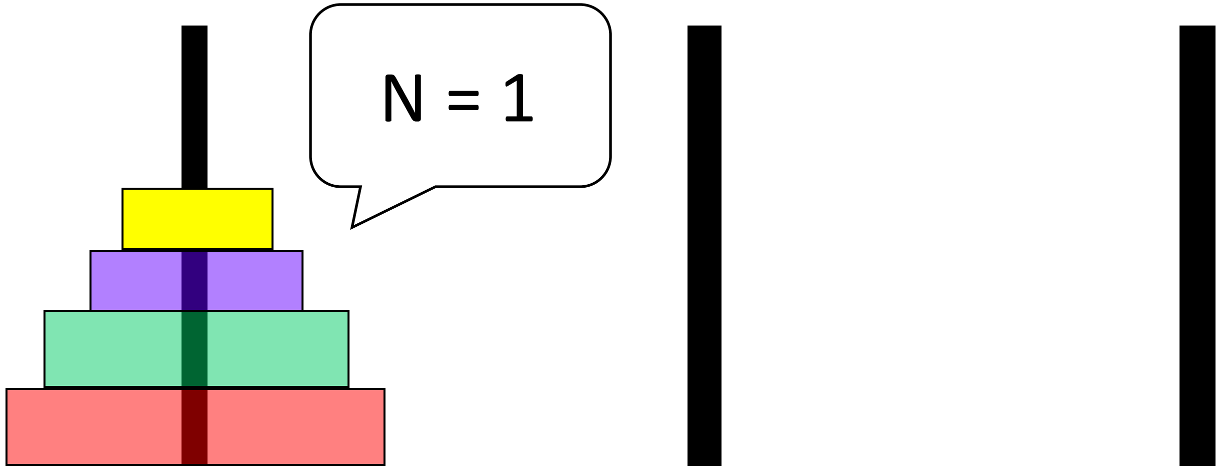


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

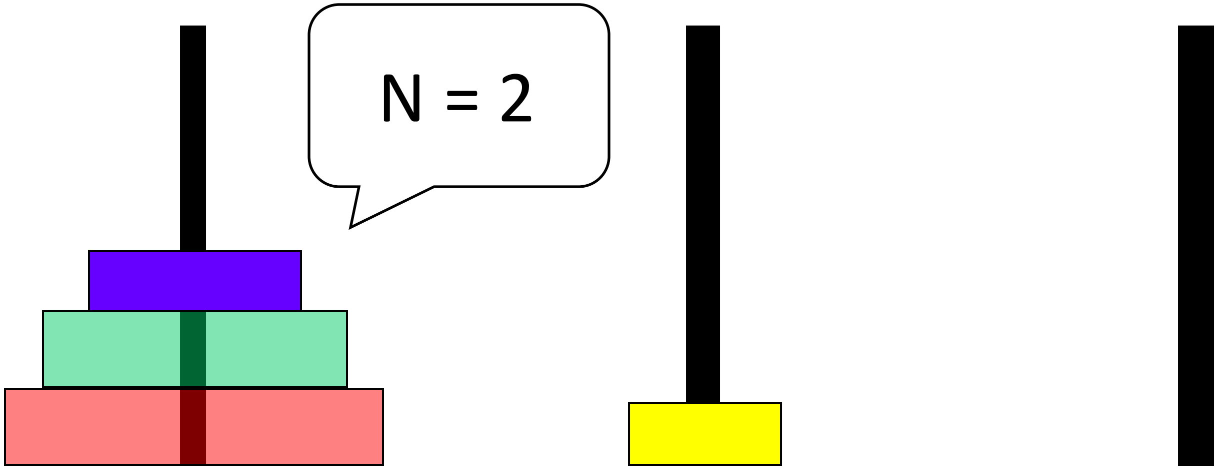


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

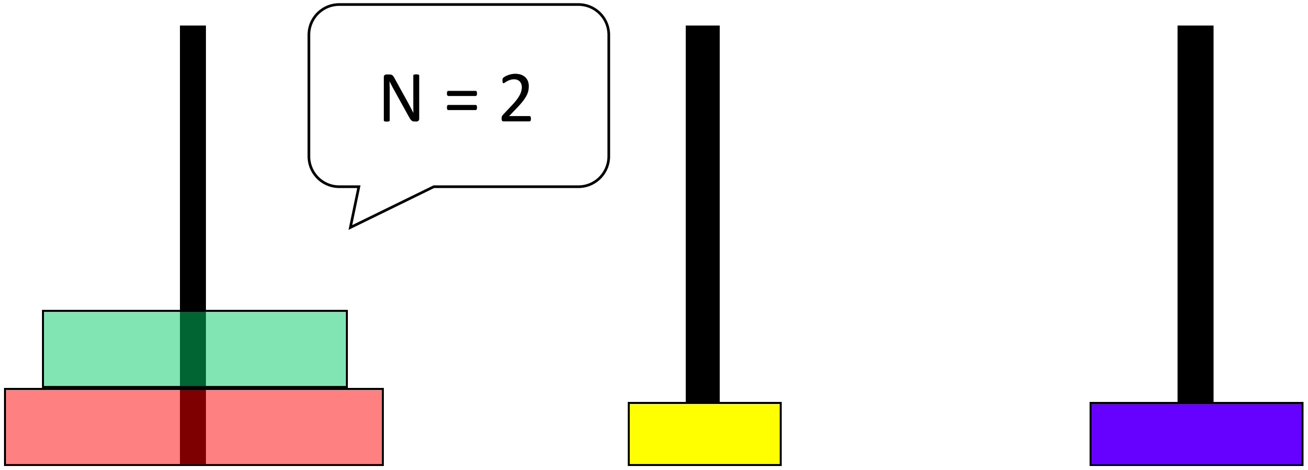


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

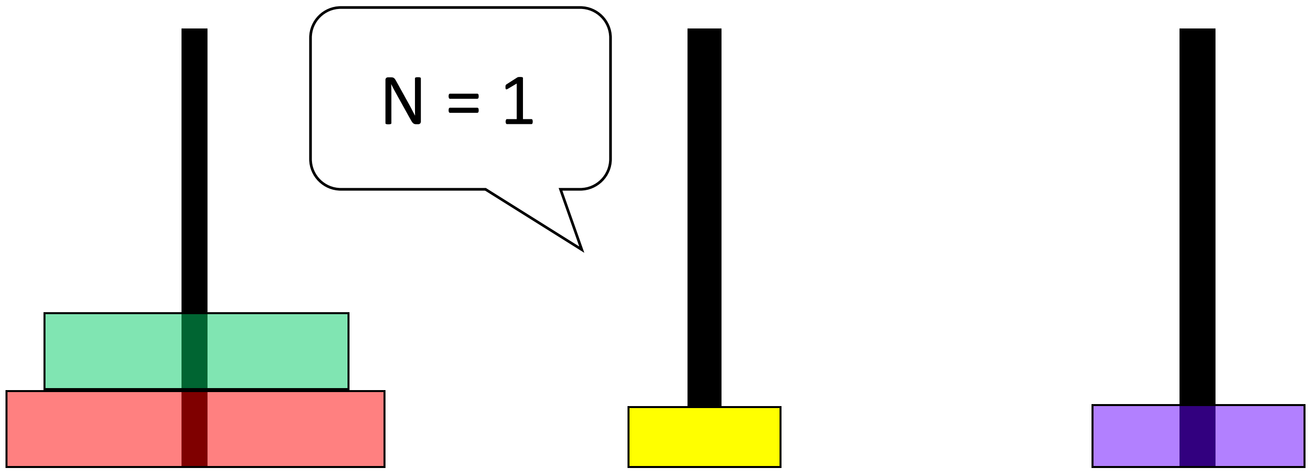


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

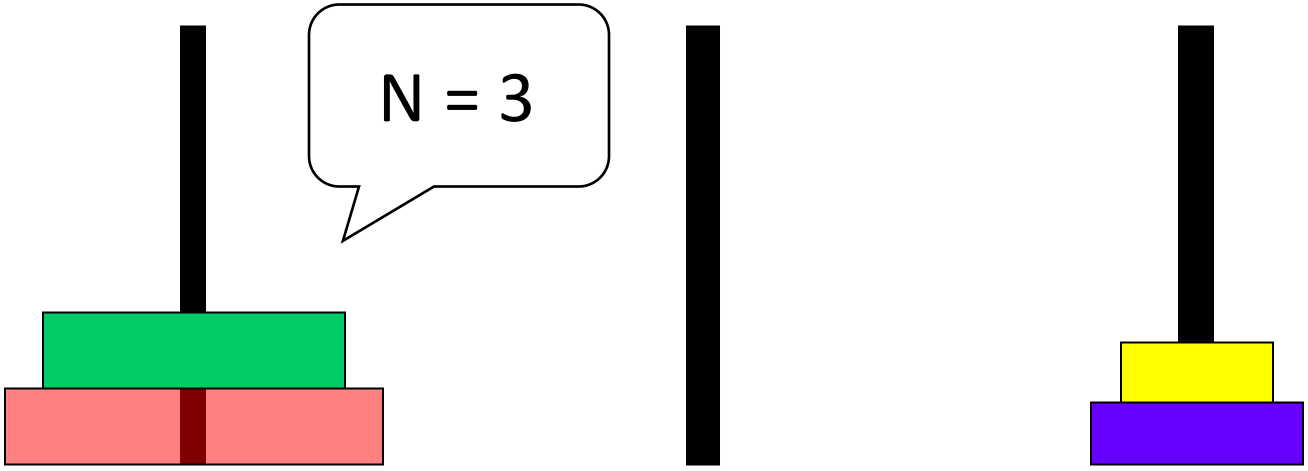


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

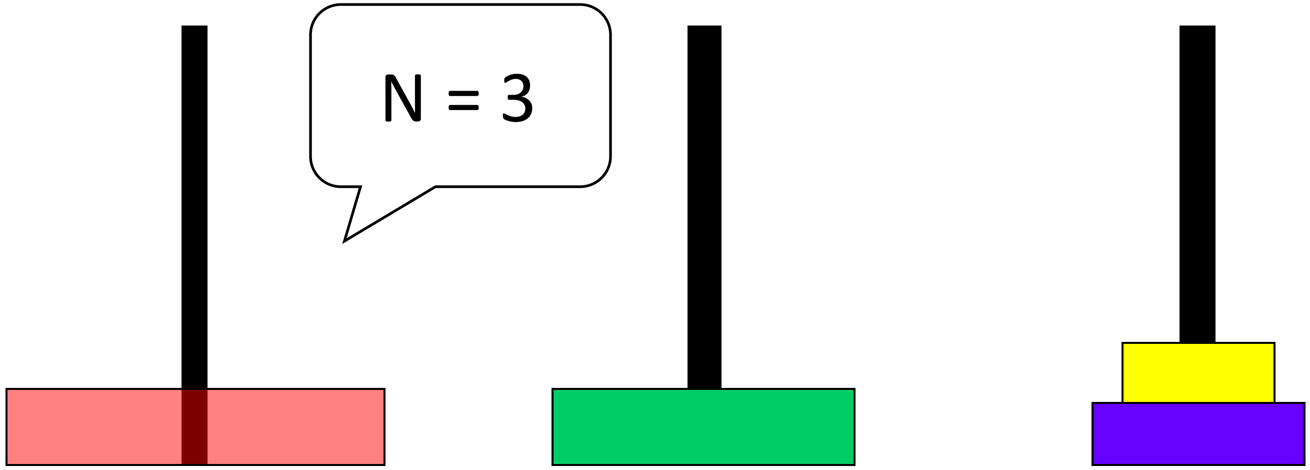


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

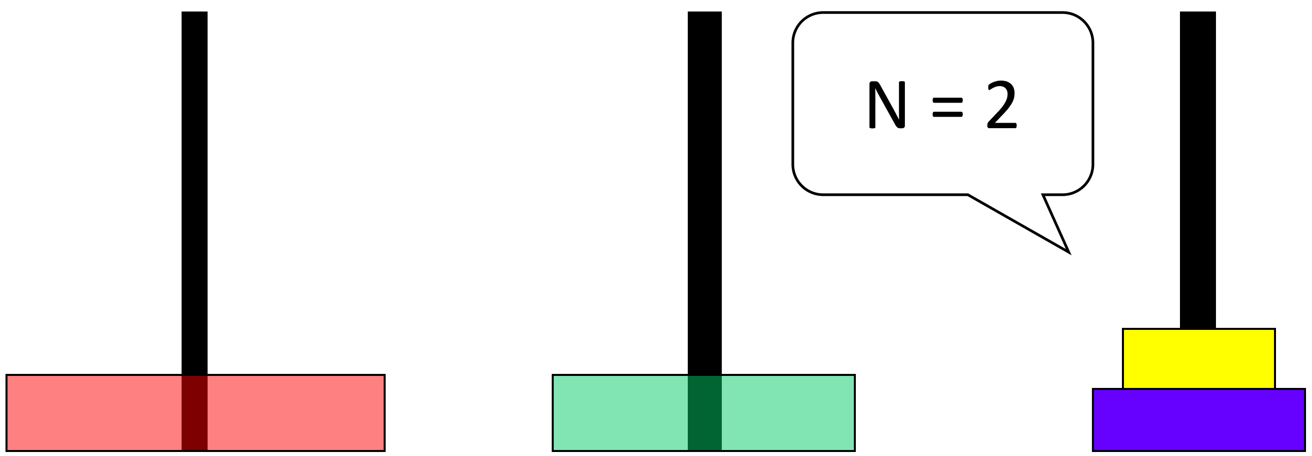


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

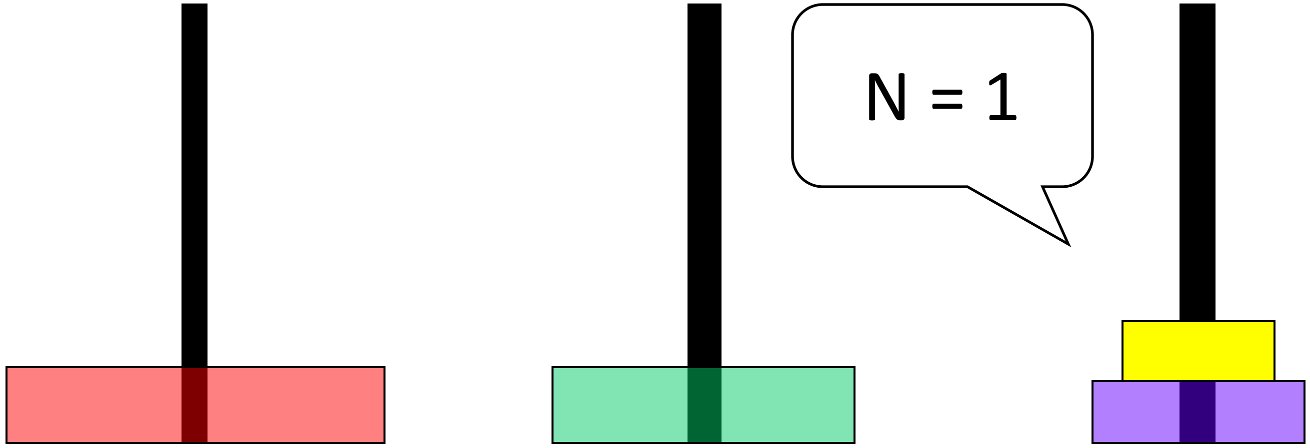


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

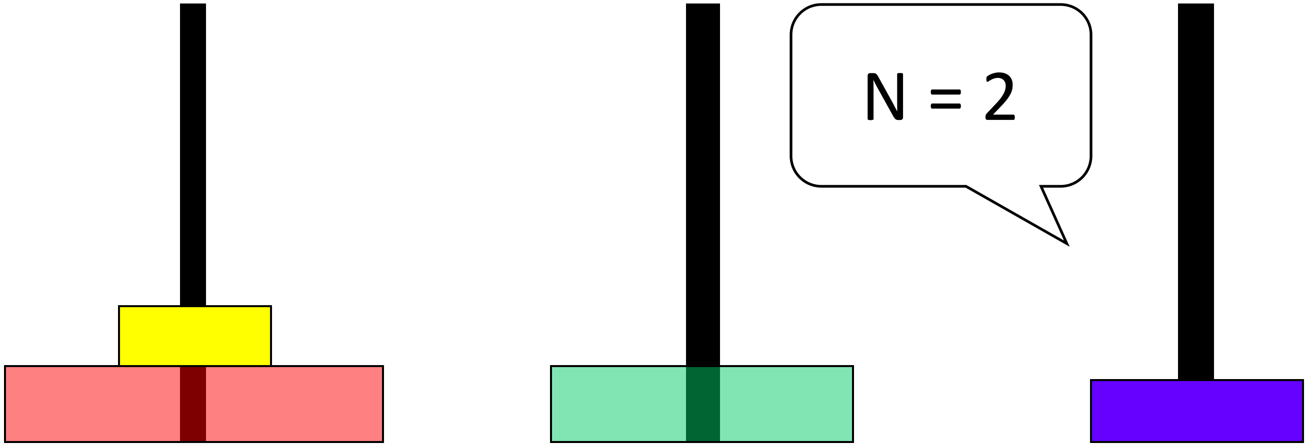


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

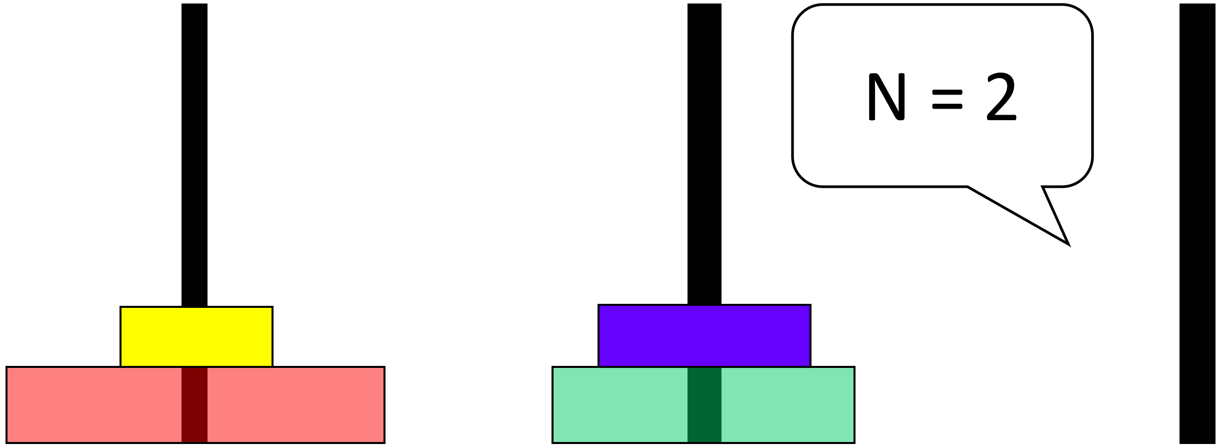


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

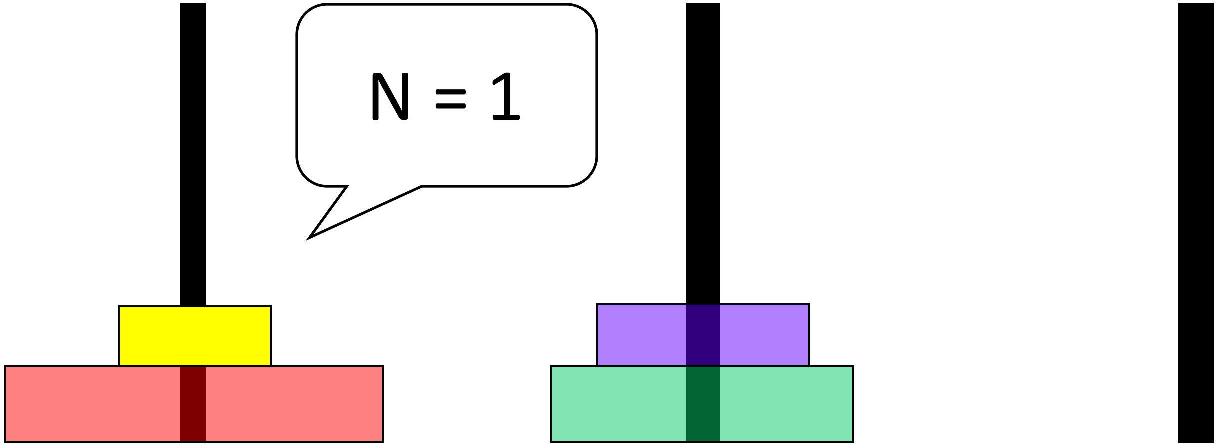


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)



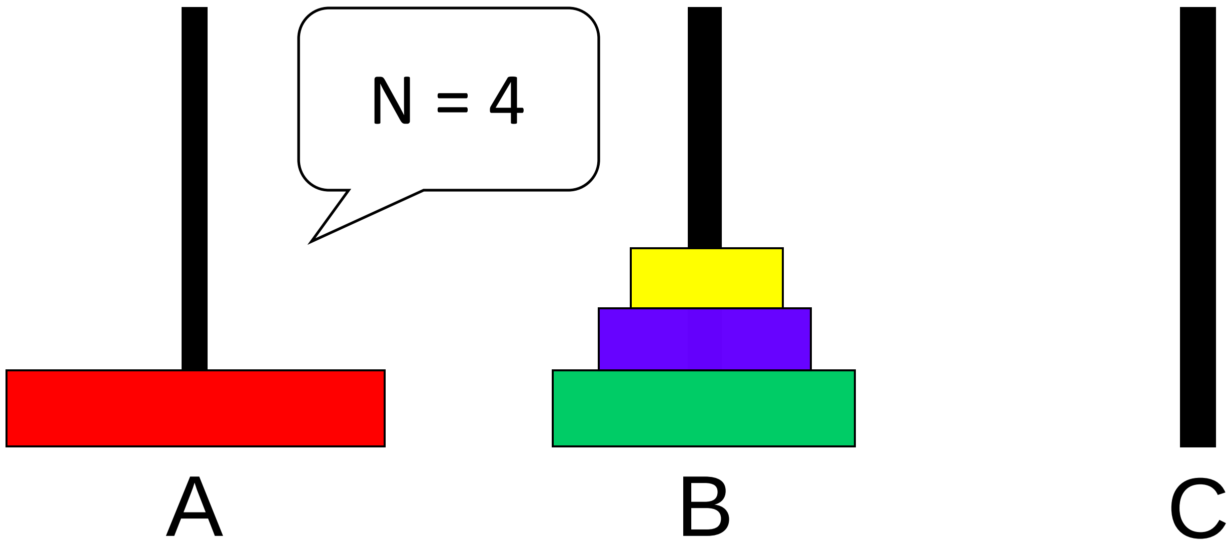
Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C

Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)

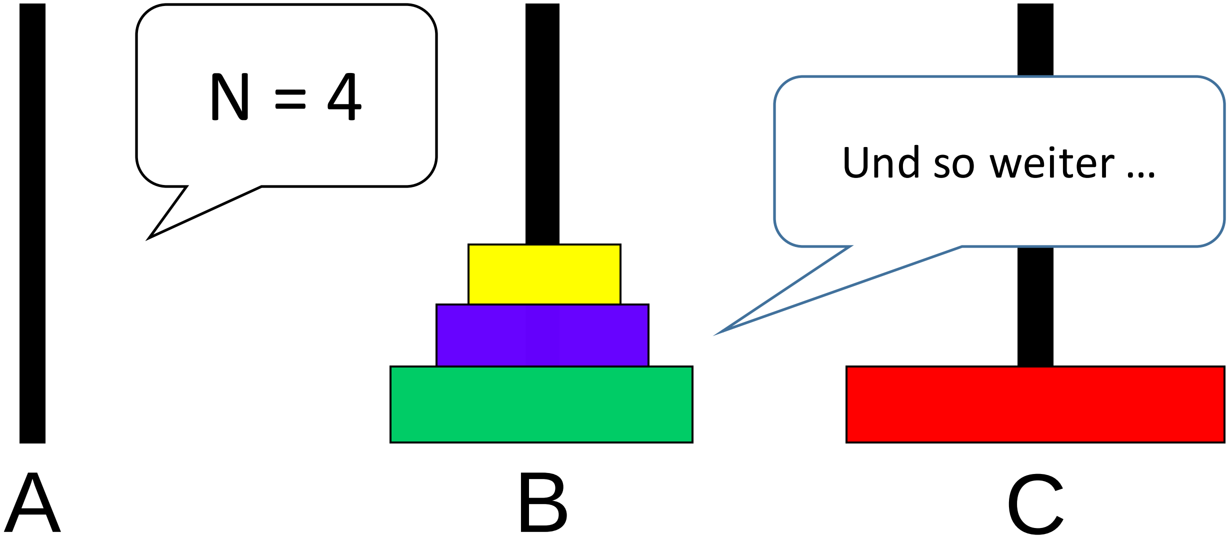


Türme von Hanoi

Bewege N Steine von A nach C (über B)

Falls N = 1: Transportiere den Stein von A nach C
Falls N > 1 gehe wie folgt vor:

- Bewege N-1 Steine von A nach B (über C);
- Transportiere den verbleibenden Stein von A nach C;
- Bewege N-1 Steine von B nach C (über A)



Pattern Matching

Durch **pattern matching** können verschiedene **Fälle** unterschieden werden

```
potenz :: Double -> Int -> Double
potenz x 0 = 1
potenz x n = x * (potenz x (n - 1))
```

Die Auswahl wird **zur Laufzeit** anhand der **Argumente** bestimmt

```
(potenz 2 0) == 1 -- Fall 1
(potenz 2 1) == 2 * (potenz 2 (1 – 1)) -- Fall 2
```

Achtung: Haskell verwendet **first-fit!**

```
potenz x n = x * (potenz x (n - 1))
potenz x 0 = 1 -- Dieser Fall wird überschattet!
```

Tail Recursion (Endrekursion)

In jedem Rekursionsschritt werden die Berechnungen zuerst gemacht, der rekursive Aufruf zuletzt

- ➢Zusätzlicher Parameter für Zwischenergebnisse nötig
- ➢Ergebnis ist im Rekursionsanker fertig

Ergebnis wird nach oben zurückgereicht

Übergebe berechnetes Zwischenergebnis an nächsten Schritt

```
potenztc x 0 s = s -- Rekursionsanker
potenztc x n s = (potenztc x (n - 1) (x * s))
potenz x n = potenztc x n 1 -- Startwert
```

Wrapper-Funktion initialisiert Ergebnisparameter

Tail Recursion (Endrekursion)

```
potenztc x 0 s = s -- Rekursionsanker
potenztc x n s = (potenztc x (n - 1) (x * s))
potenz x n = potenztc x n 1 -- Startwert

(potenz 2 3)
(potenztc 2 3 1)
(potenztc 2 2 (2 * 1))
(potenztc 2 1 (2 * (2 * 1)))
(potenztc 2 0 (2 * (2 * (2 * 1))))
(2 * (2 * (2 * (1)))) -- Rekursionsanker
(2 * (2 * (2)))
(2 * (4))
(8)
```

Der äußere Aufruf bleibt gleich…

…der Speicherverbrauch leider auch

Lazy vs. Eager Evaluation

Eager Evaluation oder „call-by-value“: Beim Aufruf einer Funktion werden zunächst die Argumente **vollständig ausgewertet**, bevor der Funktionsrumpf ausgewertet wird

Lazy Evaluation oder „call-by-need“: (Teil-) Ausdrücke werden erst dann ausgewertet, **wenn sie benötigt werden**

- (Common) LISP: Eager Evaluation
- ML: Eager Evaluation
- OPAL: Eager Evaluation
- GOFER: Lazy Evaluation
- **HASKELL: Lazy Evaluation**

In Haskell kann **Eager Evaluation erzwungen** werden

Tail-Call Optimierung

```
potenztc x 0 s = s -- Rekursionsanker
potenztc x n s = (potenztc x (n - 1) $! (x * s))
potenz x n = potenztc x n 1
```

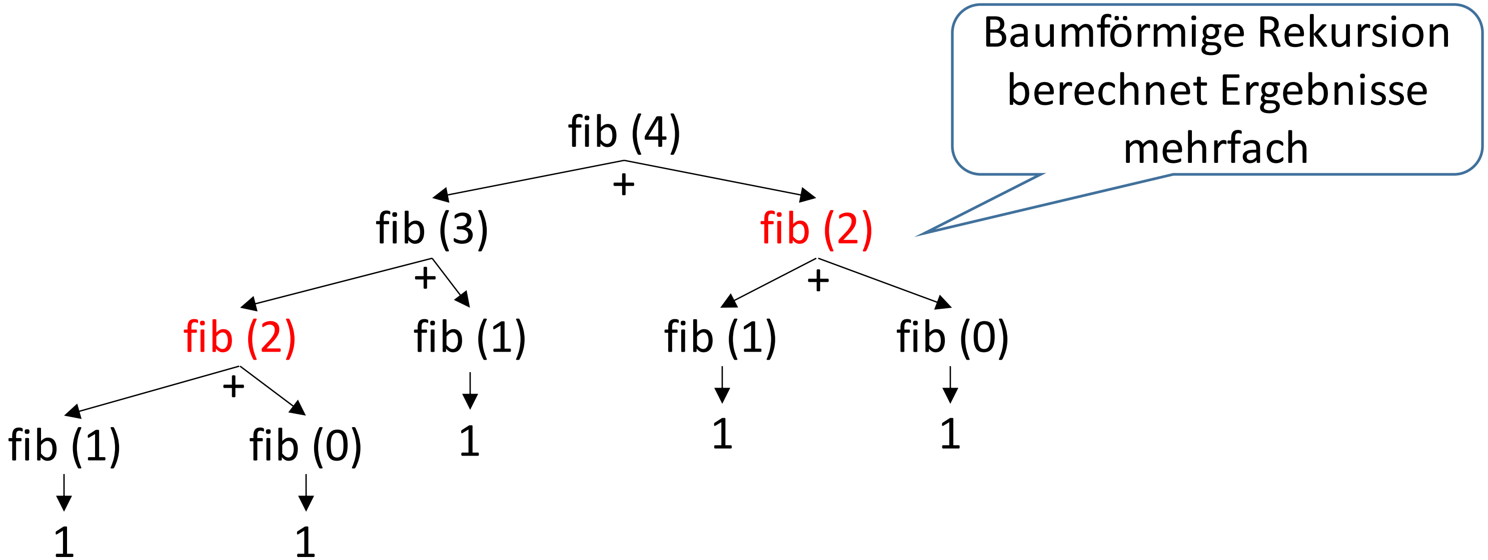
Erzwingt Eager Evaluation von (x * s)…

```
(potenz 2 3)
(potenztc 2 3 1)
(potenztc 2 2 2)
(potenztc 2 1 4)
(potenztc 2 0 8)
(8) -- Rekursionsanker
```

…und optimiert den Speicherbedarf (wird vom Compiler ohne Funktionsaufrufe umgesetzt)

Rekursive Fibonacci-Funktion

```
fib :: Int -> Int
fib 0 = 1 -- Rekursionsanker 1
fib 1 = 1 -- Rekursionsanker 2
fib n = fib (n - 1) + fib (n - 2)
```



Effiziente Fibonacci-Funktion

```
fibe :: Int -> (Int,Int)
fibe 0 = (1,1) -- Rekursionsanker 1
fibe 1 = (1,2) -- Rekursionsanker 2
fibe n = let (a,b) = fibe (n-1)
          in (b,a+b)
```

Rückgabe von Paaren
(fib n, fib (n+1))

```
(fibe 3)   ((1+2),(1+2)+2) → (3,5)
(fibe 2)   (2,1+2)
(fibe 1)   (1,2) -- Rekursionsanker 2
```

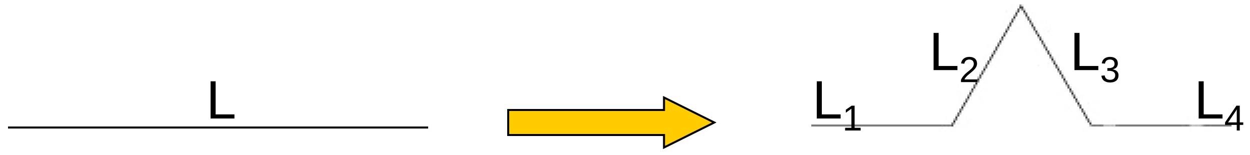
Weitere Rekursionen

Koch-Kurve (Monsterkurve)

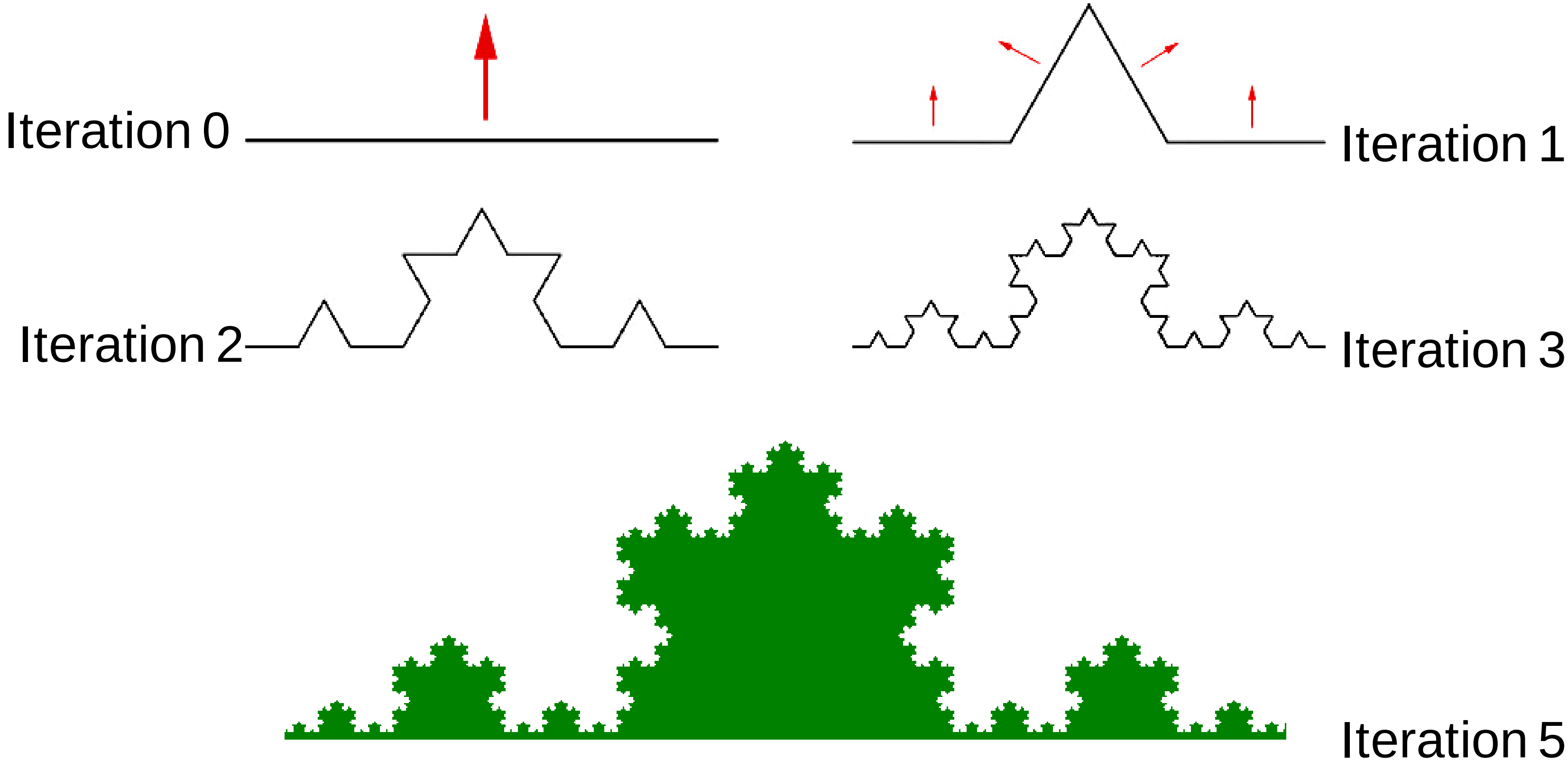
- überall stetig, nirgends differenzierbar

Konstruktionsprinzip:

1. Ersetze jede Linie L mit Länge x durch 4 Linien L_1, L_2, L_3, L_4 der Länge x/3
2. Ordne L_1, L_2, L_3, L_4 wie folgt an:



Koch-Kurve (Monsterkurve)



Inhalt

Funktionale Programmierung

Datentypen

Ermöglichen **Organisation** und **Strukturierung** von Daten

- Zusammenhängende Daten können **gruppiert** werden

Beispiele:

- Das Prüfungsamt kann Matrikelnummern, Namen und Leistungen verwalten. Diese werden gekapselt in der Datenstruktur Student
- Ein Punkt in der Ebene hat x- und y-Koordinate:

```
Point {x :: Double, y :: Double}
```

- Ein Datum kann durch Angabe des Tages, Monats und Jahres beschreiben werden:

```
Datum {tag :: Int, monat :: String, jahr :: Int}
```

Definition Produkttyp

Eine **Datenstruktur** der Art

Parameter

```
data Point = Point Double Double
```

Typkonstruktor (Daten-)Konstruktor

deklariert und definiert einen **Produkttyp**. Dabei wird der Typ Point und eine Konstruktorfunktion Point **automatisch** erzeugt:

```
Point                                   -- Typ
Point :: Double -> Double -> Point      -- Konstruktor
```

In Haskell ist oft Datenkonstruktor = Typname (wenn es nur einen gibt)

Definition Produkttyp

Verwendet man **Bezeichner** (Record-Syntax)

```haskell
data Point = Point {x :: Double, y :: Double}
```

Selektoren

werden **zusätzlich** Selektorfunktionen erzeugt:

```haskell
Point                                  -- Typ
Point :: Double -> Double -> Point     -- Konstruktor
x :: Point -> Double                   -- Selektor
y :: Point -> Double                   -- Selektor
```

Beispiel Produkttypen

```
data Rat =  Bruch {zaehler::Int, nenner::Int}
```

Der **Konstruktor** kann anders heißen als der **Typ**

```
data Datum =  Datum {tag::Int,monat::String,jahr::Int}
data Person = Person {name::String, geburt::Datum}
```

Datentypen können weitere komplexe **Datentypen** beinhalten

```
data Point = Point {x::Double, y::Double}
data Circle = Circle {center::Point, radius::Double}
```

```
data Tasche = Tasche Handy Flasche Block Stift
```

Definition Aufzählungstyp

Der **Aufzählungstyp** (Enumerator) ist ein **Spezialfall** des **Summentyps**

$$\texttt{\textbf{data} } \underbrace{\texttt{Color}}_{\text{Typ}} \texttt{ = } \underbrace{\texttt{Red | Green | Blue}}_{\text{Konstruktoren}}$$

deklariert und definiert einen **Aufzählungstyp**. Dabei wird der Typ Color und **alle** Konstruktorfunktionen **automatisch** erzeugt:

```
Color             -- Typ
Red :: Color      -- Konstruktor
Green :: Color    -- Konstruktor
Blue :: Color     -- Konstruktor
```

Definition Summentyp

Eine **Datenstruktur** der Art

```
data Shape = Circle {center::Point, radius::Double}
           | Square {anchor::Point, length::Double}
```

deklariert und definiert einen **Summentyp**. Dabei wird der Typ Shape, alle Konstruktorfunktionen **und** die Selektoren **automatisch** erzeugt:

```
Shape                                      -- Typ
Circle :: Point -> Double -> Shape         -- Konstruktor
Square :: Point -> Double -> Shape         -- Konstruktor
center :: Shape -> Point                   -- Selektor
radius :: Shape -> Double                  -- Selektor
anchor :: Shape -> Point                   -- Selektor
length :: Shape -> Double                  -- Selektor
```

Dekonstruktion

Bisher haben wir **pattern matching** nur über **Literalen** gesehen

```
mul _ 0 = 0
mul x 1 = x
ucase ‘a‘ = ‘A‘
```

Wildcard ‚_‘ kann für nicht benötigte Parameter eingesetzt werden

Pattern matching geht aber auch über Konstruktoren. Dabei werden die Datenstrukturen dekonstruiert

```
size :: Shape -> Double
size (Circle c r) = 3.14159 * r * r
size (Square a l) = l * l
```

Dekonstruktion lässt sich in Haskell an vielen Stellen einsetzen

```
size s = case s of
          (Circle c r) -> 3.14159 * r * r
          (Square a l) -> l * l
```

Selektoren

Selektorfunktionen werden in Haskell über **Dekonstruktion** definiert

```
 data Shape = Circle {center::Point, radius::Double}
            | Square {anchor::Point, length::Double}
```

```
center :: Shape -> Point
center (Circle c r) = c
radius :: Shape -> Double
radius (Circle c r) = r
```

Sie sind automatisch nur für den **passenden** Konstruktor definiert

`center (Square (Point 1 2) 3.0)` geht nicht!

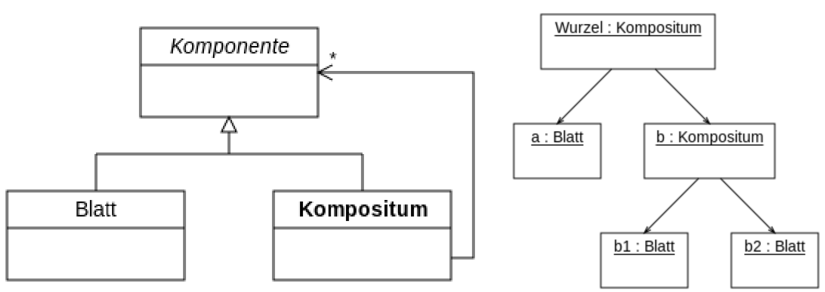
Rekursive Datenstrukturen

Datenstrukturen können **rekursiv** definiert werden

- Ermöglicht komplexe Strukturen **beliebiger Größe**

Beispiele:

- Das Prüfungsamt verwaltet alle Studenten in mehreren Listen, jedem **Eintrag** in der Liste folgt eine beliebig lange **Restliste**.
- GUIs setzen sich aus **Komponenten**-Bäumen zusammen, die entweder **weitere Komponenten** beinhalten oder **Blätter** darstellen.



Rekursive Datenstruktur: Liste

Listen können als **rekursive Summentypen** definiert werden

```
data IList = IList {element::Int, rest::IList}
                  | Empty
```

deklariert und definiert **Listen ganzer Zahlen**.

```
IList                               -- Typ
IList :: Int -> IList -> IList      -- Konstruktor
Empty :: IList                      -- Konstruktor
element :: IList -> Int             -- Selektor
rest :: IList -> IList              -- Selektor
```

Beispiel Liste ganzer Zahlen

```
data IList = IList {element::Int, rest::IList} | Empty
```

Listen werden **rekursiv** erzeugt

```
let l = (IList 1 (IList 2 (IList 3 (IList 4 Empty)))) in
```

Zugriff auf die Werte erhält man über die **Selektoren**

```
element (rest (rest (l))) = 3
```

Achtung: leere Listen beinhalten keine Elemente!

`element Empty` geht nicht!

`rest Empty` geht auch nicht!

Funktionen auf Listen

```
sum :: IList -> Int
sum (Empty) = 0
sum (IList v l) = v + sum l

sum (IList 1 (IList 2 (IList 3 (IList 4 Empty))))
1 + (sum (IList 2 (IList 3 (IList 4 Empty))))
1 + (2 + (sum (IList 3 (IList 4 Empty))))
1 + (2 + (3 + (sum (IList 4 Empty))))
1 + (2 + (3 + (4 + (sum (Empty)))))
1 + (2 + (3 + (4 + (0)))) -- Rekursionsanker
10
```

Definition Bäume

Bäume können ebenfalls als **rekursive Summentypen** definiert werden

```
data ITree = INode Int ITree ITree
           | Empty
```

Doppelt rekursiv

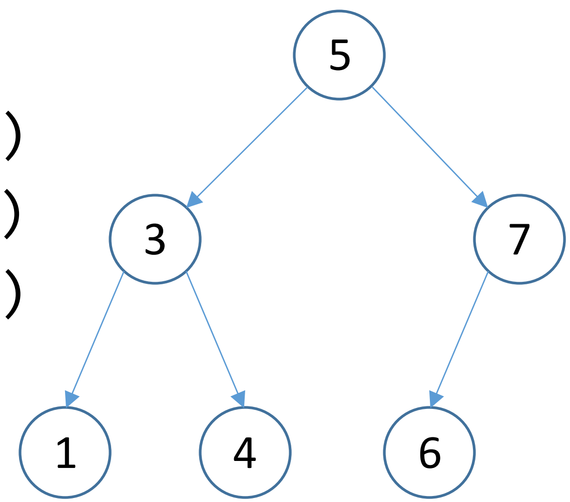
deklariert und definiert **Binärbäume ganzer Zahlen**

```
ITree                                  -- Typ
INode :: Int -> ITree -> ITree -> ITree-- Konstruktor
Empty :: ITree                         -- Konstruktor
```

Beispiel Bäume

```
data ITree = INode Int ITree ITree | Empty
```

```
INode 5
     (INode 3 (INode 1 Empty Empty)
              (INode 4 Empty Empty))
     (INode 7 (INode 6 Empty Empty)
              (Empty))
```




```
sumt :: ITree -> Int
sumt (Empty) = 0
sumt (INode v tl tr) = v + sumt tl + sumt tr
```

Doppelt rekursiv

Parametrisierte Datentypen

Neben `IList`, der **Liste ganzer Zahlen**, bräuchten wir noch…

`BList` für Listen von **Wahrheitswerten**,

`CList` für Listen von **Zeichen**,

`DList` für Listen von **Gleitkommazahlen** und

weitere Listen für **jeden neu definierten Typ**.

Dabei unterscheiden sich diese Listen nur in ihrer **Signatur**, die **Implementierung** ist stets dieselbe!

Wir brauchen eine **parametrisierbare Liste** für alle Typen!

Parametrisierte Liste

Listen können als **parametrisierter Datentyp** definiert werden

```
data List a = List {element::a, rest::(List a)}
                                 | Empty
```

Typparameter

deklariert und definiert **Listen aller Art**.

```
List a                                    -- Typ
List :: a -> List a -> List a             -- Konstruktor
Empty :: List a                           -- Konstruktor
element :: List a -> a                    -- Selektor
rest :: List a -> List a                  -- Selektor
```

Typparameter

Beispiel parametrisierte Liste

```
data List a = List {element::a, rest::(List a)} | Empty
```

Der **Typparameter** a gibt den Typ der Listenelemente vor

```
List 5 (List 7 Empty) :: List Int
List 3.2 (List 6.4 Empty) :: List Double
List True (List False Empty) :: List Bool
List ‘T‘ (List ‘U‘ Empty) :: List Char
```

Achtung: Typen können nicht gemischt werden!

`List ‘T‘ (List 5 Empty)` geht nicht!

Der Listenkonstruktor

Unsere **parametrisierte Liste** ist in Haskell bereits **vordefiniert**

Bisher: `data List a = List a (List a) | Empty`

Nativ: `data [a]      = a : [a] | []`

Konstruktor mit Infixnotation

Das heißt, was bei uns bisher z.B. so aussah:

```
List 5 (List 7 (List 9 (Empty)))
```

geht mit der Standard-Liste so:

```
5 : 7 : 9 : []
```

Infixnotation

Syntaktischer Zucker

Haskell erlaubt eine **vereinfachte Schreibweise** von Listen

```
5 : 7 : 9 : [] = [5,7,9]
```

Der **Listentyp** wird im Typparameter angegeben

```
[] :: [a] -- hier ist der Typ noch beliebig
[5,7,9] :: [Int]
[‘a‘,‘b‘,‘c‘] :: [Char]
```

Für **Zeichenketten** gibt es noch eine Vereinfachung

```
type String = [Char]
‘a‘:‘b‘:‘c‘:[] = [‘a‘,‘b‘,‘c‘] = “abc“
```

Parametrisierte Tupel

Paare definieren 2-Tupel für **zwei** beliebige Typen

```
data Pair a b = PairConstructor a b
```

deklariert und definiert **Tupel** mit **zwei** Typparametern.

```
PairConstructor 5 6 :: Pair Int Int
PairConstructor 5.0 6 :: Pair Double Int
PairConstructor ‘a‘ True :: Pair Char Bool
```

Tupel sind mit **runden** Klammern **vordefiniert**

```
(1, 4.0, ‘a‘, True) :: (Int, Double, Char, Bool)
```

Einfache Operationen für Paare und Listen

Paare (2-Tupel)

`fst :: (a, b) -> a` gibt das **erste Element** zurück

`snd :: (a, b) -> b` gibt das **zweite Element** zurück

Listen

`head :: [a] -> a` gibt das **erste Element** zurück

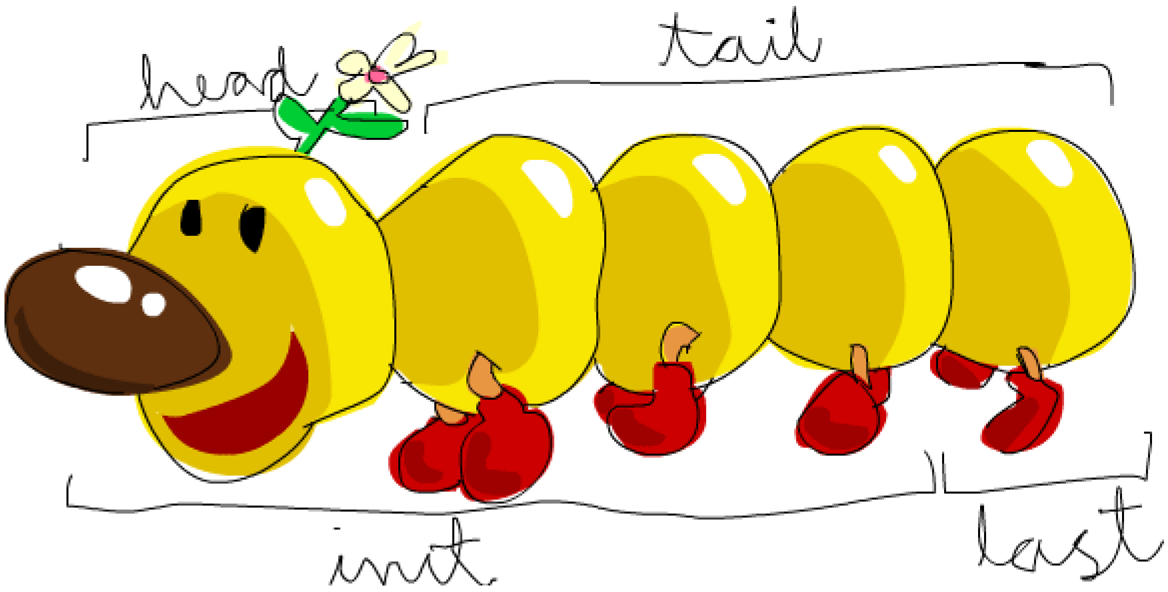
`tail :: [a] -> [a]` gibt die Liste **ohne das erste Element** zurück

`last :: [a] -> a` gibt das **letzte Element** zurück

`init :: [a] -> [a]` gibt die Liste **ohne das letzte Element** zurück

`null :: [a] -> Bool` überprüft ob die Liste **leer** ist

Head, Tail, Init und Last



siehe http://learnyouahaskell.com/starting-out

Rekursive Funktionen auf Listen

```
last :: [a] -> a
last xs = if (null (tail xs)) then (head xs)
         else last (tail xs)
```

```
last [1,2,3] -- head: 1, tail: [2,3]
last [2,3] -- head: 2, tail: [3]
last [3] -- head: 3, tail: [] Rekursionsanker!
3
```

Achtung: `last` ist für leere Listen nicht definiert!

```
last [] -- tail [] nicht definiert
```

Pattern matching mit dem Listenkonstruktor

Pattern matching kann über **Literale** und Konstruktoren erfolgen

- Listen haben die **Konstruktoren** : und [] (die leere Liste)

```
[1,2,3,4]  = 1:2:3:4:[]
```

Die Funktion **`last`** lässt sich also auch wie folgt ausdrücken

```
last (x:[]) = x
last (x:xs) = last xs
```

Mit **Wildcards**, wenn der Wert „egal“ ist

```
head (x:_) = x -- die Restliste wird verworfen
tail (_:xs) = xs -- das erste Element wird verworfen
```

Weitere Funktionen auf Listen

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + (length xs)

length [7,8,9]
length 7:8:9:[]
1 + (length 8:9:[])
1 + (1 + (length 9:[]))
1 + (1 + (1 + (length [])))
1 + (1 + (1 + (0)))
3
```

Weitere Funktionen auf Listen

```
append :: [a] -> [a] -> [a]
append [] ys = ys
append (x:xs) ys = x : (append xs ys)

append [1,2] [3,4]
1 : (append 2:[] [3,4])
1 : 2 : (append [] [3,4])
1 : 2 : [3,4]
1 : 2 : 3 : 4 : []
[1,2,3,4]
```

append ist in Haskell als (++)-Operator **vordefiniert**

Weitere Funktionen auf Listen

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]

reverse [1,2,3]
(reverse [2,3]) ++ [1]
((reverse [3]) ++ [2]) ++ [1]
(((reverse []) ++ [3]) ++ [2]) ++ [1]
[] ++ [3] ++ [2] ++ [1]
[3,2,1]
```

Definition Bäume

Bäume können ebenfalls **allgemein** definiert werden

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

deklariert und definiert **parametrisierte Binärbäume**

```
Tree a                                   -- Typ
Node :: a -> Tree a -> Tree a -> Tree a -- Konstruktor
Empty :: Tree a                          -- Konstruktor
```

Definition Maybe

Haskell kennt einen speziellen Typ zur Fehlerbehandlung, um die **Abwesenheit** eines Wertes zu überprüfen.

```
data Maybe a = Just a | Nothing
```

```
Maybe a                          -- Typ
Just :: a -> Maybe a             -- Konstruktor
Nothing :: Maybe a               -- Konstruktor
```

Die Definition kann man auch als eine ein-elementige Liste auffassen

Maybe Beispiel

In imperativen Sprachen gibt man im Fehlerfall z.B. *null* zurück

```
Integer getFirst(List<Integer> vals)
{
  if (vals.size() > 0)
    return vals.get(0);
  else return null;
}

boolean isFirst(List<Integer> vals,
Integer val)
{
  if (getFirst(vals) != null)
    return getFirst(vals) == val;
  else return false;
}
```

In Haskell verwenden wir an dieser Stelle **Maybe**

```
getFirst :: [Int] -> Maybe Int
getFirst (x:_) = Just x
getFirst _     = Nothing
```

```
isFirst :: [Int] -> Int -> Bool
isFirst vals val =
  case getFirst vals of
    Just x  -> x == val
    Nothing -> False
```

Typsynonyme

Das Schlüsselwort **type** gibt einem Typausdruck einen neuen Namen

```
type String = [Char]
```

- „String“ liest sich besser als „Liste von Zeichen“
- Das war schon alles über Type

Inhalt

Funktionale Programmierung

- Einführung
- Einführung in Haskell
- Rekursion
- Datentypen
- Funktionen höherer Ordnung
- Listenfunktionale
- Typklassen
- Unendliche Listen

Funktionen höherer Ordnung

In **funktionalen** Sprachen sind **Funktionen** first-class citizens

- **Funktionen** können Argumente von **Funktionen** sein

```
applyTo :: (a -> a) -> a -> a
applyTo f x = (f x)
```

Damit können Funktionalitäten **generalisiert** werden

```
applyTo double 5   = (double 5)  = 2 * 5 = 10
applyTo square 5   = (square 5)  = 5 * 5 = 25
applyTo cube 5     = (cube 5)    = 5 * 5 * 5 = 125
```

Beispiele

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)

addTwo    = applyTwice succ      = 1 + (1 + x)
times4    = applyTwice double    = 2 * (2 * x)
quarter   = applyTwice half      = (x / 2) / 2
powFour   = applyTwice square    = (x * x) * (x * x)
dropTwo   = applyTwice tail      = tail (tail xs)
```

Anonyme Funktionen

Funktionen **ohne explizite Signatur** heißen anonyme Funktionen

- Sie basieren auf der **Lambda-Abstraktion** und sind **unbenannt**

```
\ x -> x * x
```

\ steht in Haskell für λ

beschreibt die **Quadratfunktion** als anonyme Funktion. Sie dienen meist als **Argumente** für **Funktionen höherer Ordnung**

```
applyTwice (\x -> x * x) 5 = (5 * 5) * (5 * 5) = 625
```

Die **Typsignatur** wird aus der **Definition** gefolgert **(Typinferenz)**

Currying

In **funktionalen** Sprachen sind **Funktionen** first-class citizens

- **Funktionen** können das Ergebnis von **Funktionen** sein

```
add ::  Int -> (Int -> Int)
add x y = x + y
```

Bei der **Anwendung** einer Funktion entstehen **weitere Funktionen**

```
add 5 6 = (add 5) 6 = (5 + y) 6 = (5 + 6) = 11
```

add 5 wird automatisch gebildet

Der Pfeil „->" ist **rechtsassoziativ**, die Anwendung **linksassoziativ**, deswegen werden die Klammern **weggelassen**

Isomorphie

Grundlage des currying, ist die “**Isomorphie”** der beiden Mengen

```
{ f | f :: a -> b -> c }
{ g | g :: (a, b) -> c }
```

In vielen Programmiersprachen ist `g` die übliche(einzige) Form

In Haskell werden **alle Funktionen** als **curried** betrachtet

Die Funktionen `curry` und `uncurry` **wandeln** eine Funktion **um**

```
f = curry g          g = uncurry f
```

Curry und Uncurry

Auch die Umwandlungsfunktionen für die Curry- und Tupelformen sind Funktionen höherer Ordnung

Curry bekommt eine Funktion in Tupelform

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f x y = f (x, y)
```

Diese Klammern sind nicht notwendig!

Uncurry bekommt eine Funktion in Curry-Form

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (x, y) = f x y
```

Partial Application

Currying erlaubt partial application: das Anwenden einer Funktion auf eine **nicht-ausreichende** Menge von **Argumenten**

Dabei entsteht eine neue Funktion mit **weniger Parametern**

```
add :: Int -> Int -> Int
add 5 :: Int -> Int
```

In der Definition werden die **Parameter** durch die **Argumente** ersetzt

```
add x y = x + y
(add 5) y = 5 + y
```

Beispiele

Durch partial application können **Spezialfälle** abgeleitet werden

```
succ :: Int -> Int
succ = add 1

mul :: Int -> Int -> Int
mul x y = x * y
double = mul 2

equal :: Int -> Int -> Bool
equal x y = x == y
isZero = equal 0
```

Prefix- und Infixnotation

Funktionen in **Prefixnotation** werden von Links nach rechts angewendet, Funktionen in **Infixnotation** je nach Sektor

```
exp2 = (2^) -- Linker Sektor belegt: 2^x
pow2 = (^2) -- Rechter Sektor belegt: x^2
```

Alle **Operatoren** sind grundsätzlich in Infixnotation, durch **Klammern** kann man die Prefixnotation nutzen

```
addSeven x = 7 + x = (+) 7 x = (7+) x
```

infix

prefix

Alle anderen **Funktionen** sind grundsätzlich in Prefixnotation, durch **Hochkommas** kann man die Infixnotation nutzen

```
subTen x = sub x 10 = x `sub` 10 = (`sub` 10) x
```

prefix

infix

Weitere Beispiele

Bei **kommutativen Operatoren** spielt der Sektor keine Rolle

```
double    = (2*)          gleich (*2)
isZero    = (==0)         gleich (0==)
wrong     = (&&False)     gleich (False&&)
succ      = (1+)          gleich (+1)
```

Bei **nicht kommutativen** dagegen schon

```
half      = (/2)          ungleich (2/)
over5     = (>5)          ungleich (5>)       = under5
prefix    = ("TU"++)      ungleich (++"TU")   = suffix
pred      = (`sub` 1)     ungleich (1 `sub`)
```

(-) ist mehrdeutig

Komposition

Kompatible Funktionen können mit Komposition „verkettet“ werden

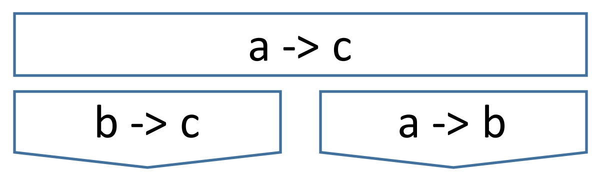
Seien A, B, C beliebige Mengen und $f : A \to B$ sowie $g : B \to C$ Funktionen, so heißt die Funktion

$$g \circ f : A \to C, x \mapsto (g \circ f)(x) := g(f(x))$$

die Komposition von f und g.

In Haskell existiert der `(.)` Operator, eine **Funktion höherer Ordnung**

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```




```
even = \x -> (0 == (x `mod` 2))   = (0==) . (`mod` 2)
odd  = \x -> (not (even x))       = not . even
```

Komposition

Kompatible Funktionen können mit Komposition „verkettet“ werden

Seien A,B,C beliebige Mengen und $f : A \to B$ sowie $g : B \to C$ Funktionen, so heißt die Funktion

$$g \circ f : A \to C, x \mapsto (g \circ f)(x) := g(f(x))$$

die Komposition von f und g.

In Haskell existiert der `(.)` Operator, eine **Funktion höherer Ordnung**

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Int -> Bool

Int -> Bool | Int -> Int

```
even = \x -> (0 == (x `mod` 2))   = (0==) . (`mod` 2)
odd  = \x -> (not (even x))       = not . even
```

Inhalt

Funktionale Programmierung

- Einführung
- Einführung in Haskell
- Rekursion
- Datentypen
- Funktionen höherer Ordnung
- Listenfunktionale
- Typklassen
- Unendliche Listen

Motivation

Die (rekursive) **Verarbeitung** von **rekursiven Datentypen** ist ein zentraler Bestandteil der funktionalen Programmierung

```
doubleAll :: [Int] -> [Int]
doubleAll [] = []
doubleAll (x:xs) = (double x) : doubleAll xs

doubleAll [1,2,3] = [(2*1),(2*2),(2*3)] = [2,4,6]
```

Problem

Die **Rekursion** sieht dabei häufig **gleich** aus

```
doubleAll :: [Int] -> [Int]
doubleAll [] = []
doubleAll (x:xs) = (double x) : doubleAll xs
squareAll :: [Int] -> [Int]
squareAll [] = []
squareAll (x:xs) = (square x) : squareAll xs
cubeAll   :: [Int] -> [Int]
cubeAll   [] = []
cubeAll   (x:xs) = (cube x) : cubeAll xs
```

Wir brauchen **Funktionen höherer Ordnung**!

Listenfunktionale

Für das Arbeiten mit **rekursiven Datentypen** existieren einige fundamentale **Funktionen höherer Ordnung**

`map`	Für das **Manipulieren** von Listenelementen
`filter`	Für das **Aussortieren** von Listenelementen
`zip`	Für das **Zusammenführen** von Listenelementen
`reduce`	Für die **Zusammenfassung** von Listenelementen

In Haskell heißt die Funktion für das **Zusammenfassen** `fold` und die Funktion für das **Zusammenführen** `zipWith`

Map

`map` **wendet** eine Funktion auf alle Elemente einer Liste **an**

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

Ähnlich wie OCL's **collect**!

```
map succ [1,2,3]
(succ 1) : map succ [2,3]
(succ 1) : (succ 2) : map succ [3]
(succ 1) : (succ 2) : (succ 3) : map succ []
(succ 1) : (succ 2) : (succ 3) : []
2 : 3 : 4 : [] = [2,3,4]
```

Map

Mit `map` lassen sich unsere **Spezialfälle** generalisieren

```
map double [1,2,3] = [2,4,6]
map square [1,2,3] = [1,4,9]
map cube [1,2,3] = [1,8,27]
```

Dabei kann sich auch der **Typparameter** der Liste ändern

```
map :: (a -> b) -> [a] -> [b]
show :: Show a => a -> String
```

$$\texttt{map show (}\underbrace{\texttt{[1,2,3,4,5]}}_{\texttt{[Int]}}\texttt{) = }\underbrace{\texttt{["1","2","3","4","5"]}}_{\texttt{[String]}}$$

Filter

`filter` **sortiert** Elemente nach einer vorgegebenen Definition **aus**

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs) = if (f x)  then x : filter f xs
                            else filter f xs
```

Das **Auswahlkriterium** wird durch das **Funktional** bestimmt

```
filter even [1,2,3,4,5,6] = [2,4,6]
filter odd [1,2,3,4,5,6] = [1,3,5]
```

Ähnlich wie OCL's **select**!

Guards

Bedingte Anweisungen lassen sich in Haskell auch übersichtlich in guarded expressions darstellen (ähnlich wie switch in JAVA)

- Dabei gilt wie beim pattern matching **first fit**

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs)
    | (f x)        = x : filter f xs
    | otherwise    = filter f xs
```

`otherwise` ist einfach als **True** definiert und dient als Default

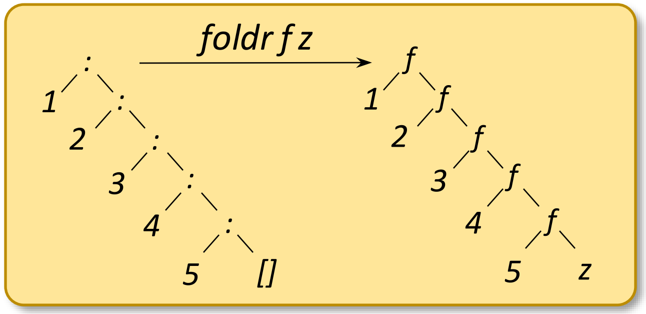
Rechtes Fold

`foldr` **fasst** die Elemente einer Struktur zu einem Wert **zusammen**

- Für **Listen** ist `foldr` bereits vordefiniert

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

`z` dient als **Startwert** und ersetzt die leere Liste `[]`
`f` **fässt** je **zwei** Werte **zusammen** und ersetzt den Konstruktor `(:)`

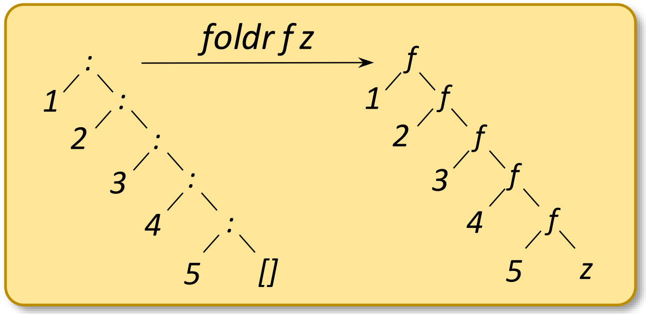


Ähnlich wie OCL's **iterate**!

Summen durch Fold

Die **Summe auf Listen**

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



mit **`foldr`** definiert sieht z.B. so aus

```
sum = foldr (+) 0
```

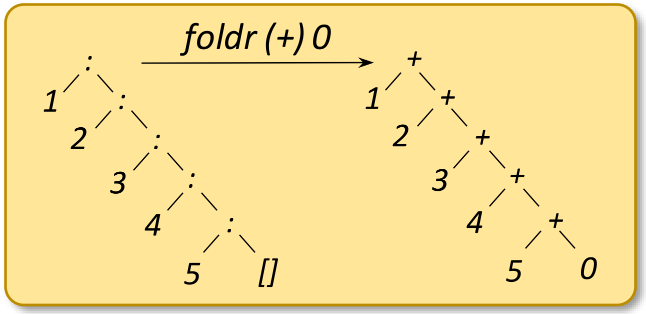
Hier wird durch partial application ein **Spezialfall** von `foldr` **abgeleitet**

```
(foldr (f) z) [] = z
(foldr (f) z) (x:xs) = (f) x ((foldr (f) z) xs)
```

Summen durch Fold

Die **Summe auf Listen**

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



mit **`foldr`** definiert sieht z.B. so aus

```
sum = foldr (+) 0
```

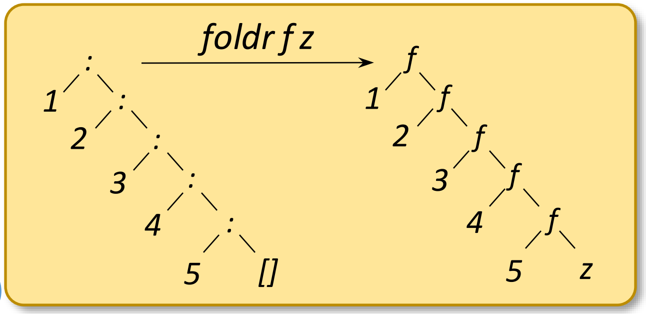
Hier wird durch partial application ein **Spezialfall** von `foldr` **abgeleitet**

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

Produkt durch Fold

Das **Produkt auf Listen**

```
product :: [Int] -> Int
product [] = 1
product (x:xs) = x * (product xs)
```



mit **foldr** definiert sieht z.B. so aus

```
product = foldr (*) 1
```

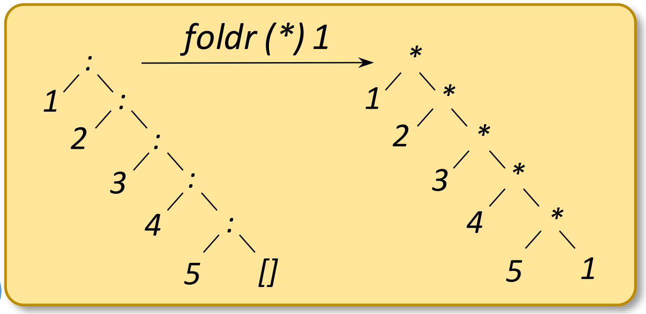
Wieder wird durch partial application ein **Spezialfall abgeleitet**

```
(foldr (f) z) [] = z
(foldr (f) z) (x:xs) = (f) x ((foldr (f) z) xs)
```

Produkt durch Fold

Das **Produkt auf Listen**

```
product :: [Int] -> Int
product [] = 1
product (x:xs) = x * (product xs)
```



mit **foldr** definiert sieht z.B. so aus

```
product = foldr (*) 1
```

Wieder wird durch partial application ein **Spezialfall abgeleitet**

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

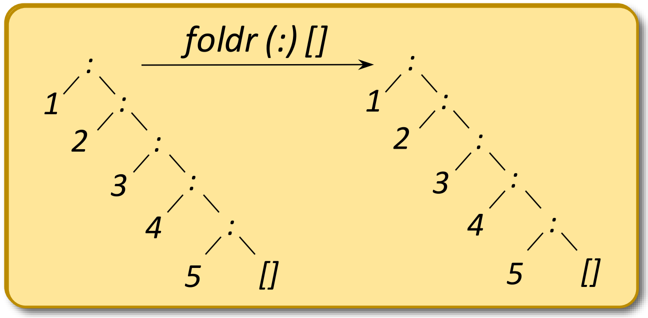
Strukturerhaltendes Fold

Übergibt man einem `foldr` auf Listen die **Listenkonstruktoren** bleibt die Listenstruktur **erhalten**

```
keep :: [a] -> [a]
keep = foldr (:) []
```

```
(foldr (:) []) [] = []
(foldr (:) []) (x:xs) = x : ((foldr (:) []) xs)
```

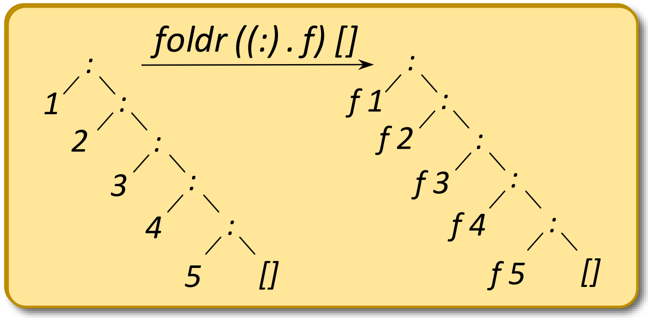
```
keep [1,2,3,4,5] = [1,2,3,4,5]
```



Map durch Fold

Mit der Übergabe der **Listenkonstruktoren** und einem **Funktional** lässt sich `map` durch `foldr` **ausdrücken**

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

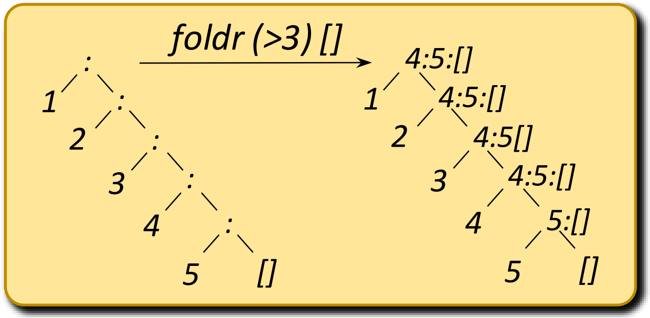


Filter durch Fold

Wird der Listenkonstruktor **bedingt** angewendet kann `filter` durch `foldr` **definiert** werden

```
filter :: (a -> Bool) -> [a] -> [a]
filter f = foldr (\x y -> if f x then x:y else y) []
```

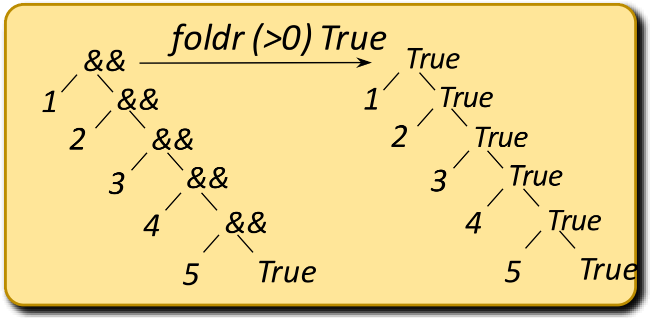
```
filter (>3) [1,2,3,4,5] = [4,5]
```



ForAll durch Fold

Ergebnis der Teilausdrücke kann auch ein anderer Datentyp sein, z.B. Bool

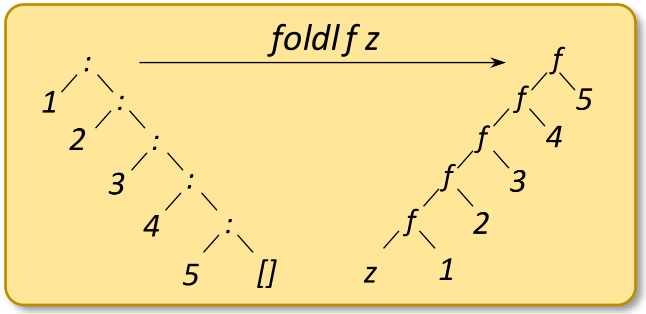
```
forAll :: (a -> Bool) -> [a] -> Bool
forAll p l = foldr (\x acc -> (p x) && acc) True l

forAll (>0) [1,2,3,4,5] = True
```



Linkes Fold

Neben dem **rechten Fold** `foldr` gibt es noch das **linke Fold** `foldl`

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

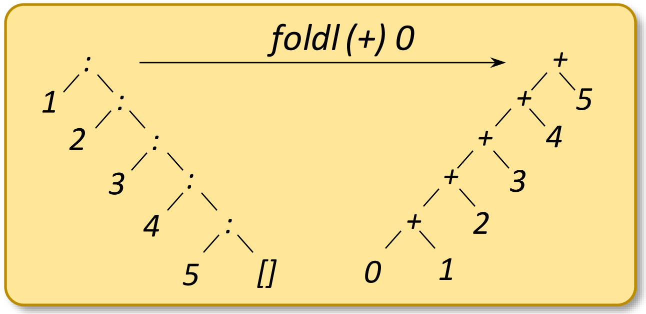


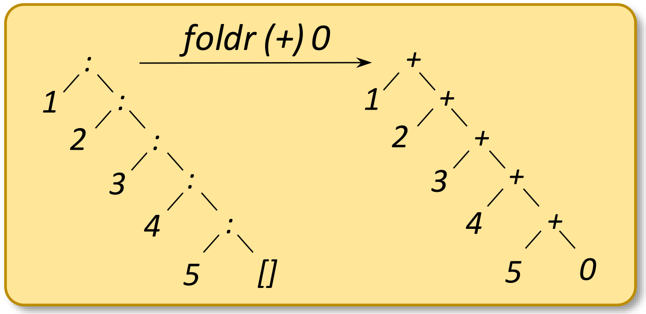
Foldr und Foldl

Linkes uns rechtes Fold unterscheiden sich in der **Reihenfolge** der **Auswertung** und **Anwendung** des Funktionals

```
sumr = foldr (+) 0
suml = foldl (+) 0
sumr [1,2,3,4,5] = 1+(2+(3+(4+(5+0)))) = 15
suml [1,2,3,4,5] = ((((0+1)+2)+3)+4)+5 = 15
```

Die **linke Summe** ermöglicht Optimierung durch **tail recursion**!





ZipWith

`zipWith` **führt** zwei Listen mit Hilfe eines Funktionals **zusammen**

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = (f x y) : zipWith f xs ys
```

Damit lassen sich zum Beispiel zwei **Tabellenspalten kombinieren**

```
cell = [19,21,20,29,23] -- Handyrechnung/Monat
food = [220,254,189,312,234] -- Lebensmittel/Monat
total= zipWith (+) cell food = [239,275,209,341,257]
```

Zip und Unzip

Haskell kennt außerdem eine Funktion `zip` die **zu Tupeln kombiniert**

```
zip :: [a] -> [b] -> [(a, b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Das lässt sich auch mit partial application von `zipWith` **ableiten**

```
zip = zipWith (,)
```

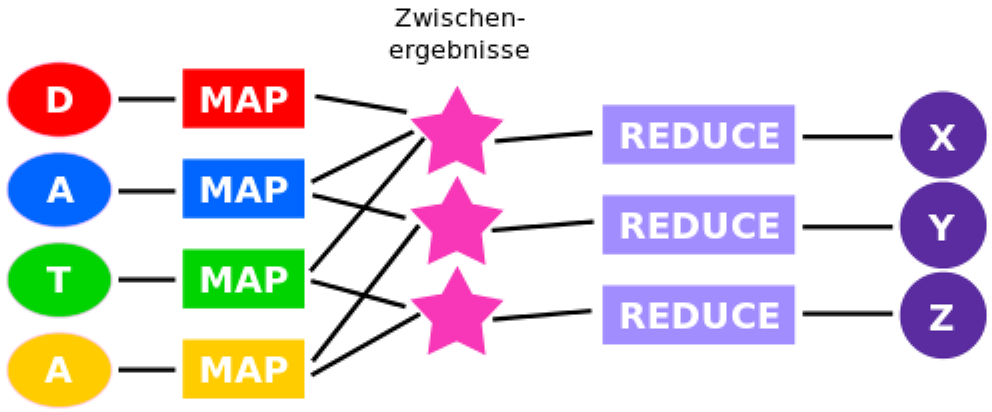
Umgekehrt gibt es die Funktion `unzip` um die Listen zu **trennen**

```
unzip :: [(a,b)] -> ([a],[b])
unzip [] = ([],[])
unzip ((x,y):zs)= let (xs,ys) = unzip zs in (x:xs,y:ys)
```

Listenfunktionale im Einsatz

Wegen der **Zustandsfreiheit** eignet sich die **funktionale Programmierung** besonders für **nebenläufige** Berechnungen

- Das Listenfunktional `map` lässt sich sequenziell oder parallel anwenden
- Für **verteilte** Berechnungen über **großen Datenmengen** auf Rechnerclustern hat **Google** das MapReduce Programmiermodell etabliert



- Die Map-Phase wird parallel ausgeführt, die Zwischenergebnisse in einer Reduce-Phase gesammelt und zum Ergebnis kombiniert

siehe: https://de.wikipedia.org/wiki/MapReduce

Listenfunktionale in anderen Sprachen

Einige andere **Programmiersprachen** unterstützen **Listenfunktionale**

Python

`lambda x, y: x + y` ist eine anonyme Funktion

```
map(lambda x: x + 1, [1,2,3,4]) = [2,3,4,5]
filter(lambda x: x % 2, [1,2,3,4,5,6]) = [1,3,5]
reduce(lambda x, y: x + y, [47,11,42,13]) = 113
```

Java 8

```
Arrays.asList(1,2,3,4).stream().map(x -> x + 1) = [2,3,4,5]
Arrays.asList(1,2,3,4,5,6).stream().filter(x -> x % 2 != 0) = [1,3,5]
Arrays.asList(47,11,42,13).stream().reduce(0, (x, y) -> x + y) = 113
```

Inhalt

Funktionale Programmierung

- Einführung
- Einführung in Haskell
- Rekursion
- Datentypen
- Funktionen höherer Ordnung
- Listenfunktionale
- Typklassen
- Unendliche Listen

Typen und Typklassen

Was ist mit **typspezifischen Funktionen**?

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

Macht nur für **bestimmte Typen** Sinn:

```
isum :: [Int] -> Int
dsum :: [Double] -> Double
rsum :: [Real] -> Real …
```

Wir brauchen **Typklassen** um **Gemeinsamkeiten** auszudrücken!

Definition Typklassen

Typklassen **deklarieren** welche Funktionen für einen Typen definiert sein müssen, damit er eine Instanz der Typklasse ist.

- Sie ähneln vom Prinzip her dem Konzept der **interfaces** in JAVA (besonders ab JAVA 8)

Die **Typklasse** Eq (Äquivalenz) deklariert **zwei Funktionen**

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Über eine **Standardimplementierung** wird Redundanz vermieden

```
    a == b = not (a /= b)
    a /= b = not (a == b)
```

Instanziierung von Typklassen

Um zu einer Typklasse zu gehören, müssen Instanzen der Typklasse alle deklarierten Funktionen **definieren**.

```
data Rat = Bruch {zaehler::Int, nenner::Int}
```

Der **Typ** Rat kann zu der der **Typklasse** Eq a hinzugefügt werden

```
instance Eq Rat where
(Bruch xz xn) == (Bruch yz yn) = (xz*yn) == (yz*xn)
```

Int ist bereits Instanz von Eq

Ungleichheit (/=) ergibt sich aus der Standardimplementierung

Verwendung von Typklassen

Ist ein Typ eine **Instanz einer Typklasse**, können die deklarierten Funktionen **verwendet** werden

```
(Bruch 3 6) == (Bruch 1 2) = True
(Bruch 4 7) == (Bruch 1 2) = False
(Bruch 4 7) /= (Bruch 1 2) = True
```

Typklassen können **Typparameter einschränken**

```
same :: Eq a => [a] -> [a] -> Bool
same [] [] = True
same [] ys = False
same xs [] = False
same (x:xs) (y:ys) = x == y && (same xs ys)
```

Beispiel Nummern

Die **Addition** ist für alle Instanzen der Typklasse **Num** definiert

```
class Num a where
   (+), (-), (*) :: a -> a -> a
   negate, signum, abs :: a -> a
   fromInteger :: Integer -> a
```

Damit lässt sich unsere Summenfunktion sinnvoll **einschränken**

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

Beispiele

Listen

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

`sum [1,2,3] = 6`, aber `sum [‘a‘,‘b‘,‘c‘]` geht nicht!

Int ist Instanz von Num

Char nicht

Bäume

```
sumt :: Num a => Tree a -> a
sumt (Empty) = 0
sumt (Node a tl tr) = a + sumt tl + sumt tr
```

Weitere Einschränkungen

Typklassen können von anderen Typklassen **erben**

```
class Eq a => Ord a where
      (<),(<=),(>=),(>)  :: a -> a -> Bool
      max, min           :: a -> a -> a
```

erbt (==) und (/=) von der Typklasse Eq a

Instanziierungen können ebenfalls **eingeschränkt** werden

```
instance Eq a => Eq [a] where
      [] == [] = True
      (x:xs) == (y:ys) = x == y && xs == ys
      xs == ys = False -- sonst
```

Typklasse Show

Instanzen der Typklasse **Show** lassen sich in **Strings** konvertieren

- Ähnelt der **toString-Methode** in JAVA

```
class Show a where
    show :: a -> [Char]
```

```
data Rat = Bruch {zaehler::Int, nenner::Int}
instance Show Rat where
    show (Bruch z n) = show z ++ "/" ++ show n
```

Int ist auch Instanz von Show

```
show (Bruch 7 8) = "7/8"
```

Typklasse Enum

Die Typklasse Enum umfasst **sequenziell geordnete Typen**

- Elemente von Typen, die Enum instanziieren, können **aufgezählt** werden

Elemente eines Enum Typs lassen sich auf **natürliche Zahlen** abbilden

```
class Enum a where
    fromEnum :: a -> Int
    toEnum :: Int -> a
```

Für sie ist auch die **Vorgänger-** und **Nachfolgerfunktion** definiert

```
    succ v = toEnum (succ (fromEnum v))
    pred v = toEnum (pred (fromEnum v))
```

…und zurück

Abbildung auf Int…

Typklasse Foldable

Das **Zusammenfassen** von Elementen kann für viele **rekursive Datenstrukturen** definiert werden

- Diese **Gemeinsamkeit** wird durch die Typklasse `Foldable` ausgedrückt

`Foldable` Typen unterstützen neben `foldr` **weitere Funktionen**

```
null      :: Foldable t =>              t a -> Bool
length    :: Foldable t =>              t a -> Int
sum       :: (Foldable t, Num a) =>     t a -> a
maximum   :: (Foldable t, Ord a) =>     t a -> a
elem      :: (Foldable t, Eq a) => a -> t a -> Bool
```

Für uns ist die **Liste** die wichtigste **Instanz** von `Foldable`

Inhalt

Funktionale Programmierung

- Einführung
- Einführung in Haskell
- Rekursion
- Datentypen
- Funktionen höherer Ordnung
- Listenfunktionale
- Typklassen
- Unendliche Listen

Ranges

Weil alle Elemente eines Enum Typs die **Nachfolgerfunktion** definieren, können diese Elemente in **Ranges** verwendet werden

```
range :: Enum a => a -> a -> [a]
range x y
     | (fromEnum x) > (fromEnum y) = []
     | otherwise = x : range (succ x) y

range 3 9 = [3,4,5,6,7,8,9]
range ‘a’ ‘e’ = “abcde”
```

List Comprehensions

In Haskell sind **ranges** als list comprehension **vordefiniert**

```
[3..9] = [3,4,5,6,7,8,9]
```

Die **Schrittweite** wird aus der **Differenz** der ersten Werte **abgeleitet**

```
odds = [1,3..9] = [1,3,5,7,9]
sevenDown = [35,28..0] = [35,28,21,14,7,0]
```

Weitere list comprehensions erlauben das **kombinieren** und **filtern**

```
roll7 = [(a,b)| a <- [1..6], b <- [1..6],a + b == 7]
= [(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)]
```

beschreibt **alle Möglichkeiten** mit zwei Würfeln **eine 7 zu würfeln**

Unendliche Listen

Haskell lässt auch **offene Ranges** zu

```
[0..] =
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,2
0,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37
,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,
55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,7
2,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89
,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,1
05,106,107,108,109,110,111,112,113,114,115,116,117,1
18,119,120,121,122,123,124,125,126,127,128,129,130,
,131,132,133,134,135,136,137,138,139,140,141,142,143
,144,145,146,147,148,149,150,151,152,153,154,155,...
```

Es ergibt sich eine **unendliche Liste**

Lazy Evaluation

Unendliche Listen sind möglich, weil Haskell nicht-strikt **evaluiert**

- Die Liste wird immer **nur so weit** tatsächlich erstellt, wie sie **gebraucht** wird

```
zeros :: [Int]
zeros = 0 : zeros -- kein Rekursionsanker!
zeros = 0:0:0:0:0:0:… = [0,0,0,0,0,0,… = [0,0..]
```

Das erlaubt es auch, auf unendlichen Listen zu **arbeiten**

```
ones :: [Int]
ones = map (1+) zeros -- zeros ist unendlich!
ones = (1+0):(1+0):(1+0):(1+0):… =[1,1,1,… =[1,1..]
```

Die natürlichen Zahlen

Auch die Liste der natürlichen Zahlen lässt sich so **konstruieren**

```
nats :: [Int]
nats = 0 : map (1+) nats
```

```
nats
0 : map (1+) nats
0 : map (1+) (0 : map (1+) nats …
0 : (1+0 : (1+1+0 : (1+1+1+0 : …
[0,1,2,3,…
```

Aus unendlichen Listen kann man mit `take` und `drop` **auswählen**

```
take 5 nats = [0,1,2,3,4] -- die ersten Elemente
drop 5 nats = [5,6,7,8,9… -- die Restliste
```

Beispiel

Wir wollen eine **Liste aller Primzahlen** aufstellen

Nach **Euklid** ist die Menge aller Primzahlen **unendlich**

- Der Nachfolger des Produkts aller Primzahlen ist entweder selbst eine Primzahl oder besitzt eine Primfaktor, der von allen anderen verschieden ist

Wir **berechnen** die Liste der Primzahlen mit dem Sieb des Eratosthenes

1. **Schreibe** alle natürlichen Zahlen ab 2 in einer Liste (p:as) **auf**
2. **Behalte** das erste Element aus der Liste als Primzahl p
3. **Entferne** alle Vielfachen dieser Zahl aus der Restliste as
4. **Wiederhole** Schritt 2 mit der gefilterten Restliste as

Beispielablauf

(p:as) = [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...

(p:as) = 2:[3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...

(2:as) = 2:[3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,...

(2:(p:as)) = 2:3:[5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35, ...

(2:(3:as)) = 2:3:[5,7,11,13,17,19,23,25,29,31,35,37,41,43,47,49,...

(2:(3:(p:as))) = 2:3:5:[7,11,13,17,19,23,25,29,31,35,37,41,43,47,...

(2:(3:(5:as))) = 2:3:5:[7,11,13,17,19,23,29,31,37,41,43,47,49,53,...

(2:(3:(5:(p:as)))) = 2:3:5:7:[11,13,17,19,23,29,31,37,41,43,47,49,...

(2:(3:(5:(7:as)))) = 2:3:5:7:[11,13,17,19,23,29,31,37,41,43,47,53,...

2:3:5:7:11:13:17:19:23:29:31:37:41:43:47:53:59:61:67:71:73:79:...

Alle Primzahlen

Wir **berechnen** die Liste der Primzahlen mit dem Sieb des Eratosthenes

1. **Schreibe** alle natürlichen Zahlen ab 2 in einer Liste (p:as) **auf**

   primes = sieve [2..]

2. **Behalte** das erste Element aus der Liste als Primzahl p

   sieve (p:as) = p : …

3. **Entferne** alle Vielfachen dieser Zahl aus der Restliste as

   … (filter (\x -> (x `mod` p) /= 0) as)

4. **Wiederhole** Schritt 2 mit der gefilterten Restliste

   sieve (p:as) = p : sieve (filter (\x -> (x `mod` p) /= 0) as)

primes = sieve [2..] =
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,
103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,…

Primzahlen

2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,1
07,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199,2
11,223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,313,3
17,331,337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,433,4
39,443,449,457,461,463,467,479,487,491,499,503,509,521,523,541,547,557,563,5
69,571,577,587,593,599,601,607,613,617,619,631,641,643,647,653,659,661,673,6
77,683,691,701,709,719,727,733,739,743,751,757,761,769,773,787,797,809,811,8
21,823,827,829,839,853,857,859,863,877,881,883,887,907,911,919,929,937,941,9
47,953,967,971,977,983,991,997,1009,1013,1019,1021,1031,1033,1039,1049,105
1,1061,1063,1069,1087,1091,1093,1097,1103,1109,1117,1123,1129,1151,1153,11
63,1171,1181,1187,1193,1201,1213,1217,1223,1229,1231,1237,1249,1259,1277,1
279,1283,1289,1291,1297,1301,1303,1307,1319,1321,1327,1361,1367,1373,1381,
1399,1409,1423,1427,1429,1433,1439,1447,1451,1453,1459,1471,1481,1483,148
7,1489,1493,1499,1511,1523,1531,1543,1549,1553,1559,1567,1571,1579,1583,15
97,1601,1607,1609,1613,1619,1621,1627,1637,1657,1663,1667,1669,1693,1697,1
699,1709,1721,1723,1733,1741,1747,1753,1759,1777,1783,1787,1789,1801,1811,
1823,1831,1847,1861,1867,1871,1873,1877,1879,1889,1901,1907,1913,1931,193
3,1949,1951,1973,1979,1987,1993,1997,1999,2003,2011,2017,2027,2029,2039,20
53,2063,2069,2081,2083,2087,2089,2099,2111,2113,2129,2131,2137,2141,2143,2
153,2161,2179,2203,2207,2213,2221,2237,2239,2243,2251,2267,2269,2273,2281,
2287,2293,2297,2309,2311,2333,2339,2341,2347,2351,2357,2371,2377,2381,238
3,2389,2393,2399,2411,2417,2423,2437,2441,2447,2459,2467,2473,2477,2503,...

Lernziele 1

- ❑ Was ist der Unterschied zwischen Syntax und Semantik?
- ❑ Was versteht man unter einem Programmierparadigma?
- ❑ Was ist der Unterschied zwischen imperativ und funktional?
- ❑ Welche Programmierparadigmen unterstützt Haskell?
- ❑ Welche Rolle spielen Funktionen in Haskell?
- ❑ Was verwendet man statt Schleifen in funktionalen Sprachen?
- ❑ Was ist Pattern Matching?
- ❑ Welche Grundlegenden Datentypen gibt es in Haskell?
- ❑ Was versteht man unter Konstruktion/Dekonstruktion?
- ❑ Wie funktioniert die Selektion von Daten in Haskell?
- ❑ Was ist ein rekursiver Datentyp?
- ❑ Wofür benötigt man parametrisierte Datentypen?
- ❑ Welche wichtigen Funktionen benötigt man zum Arbeiten auf Listen?
- ❑ Welche Vorteile bietet Lazy-Evaluation?

Lernziele 2

- ❑ Was ist der Unterschied zwischen einem Typ und einer Typklasse?
- ❑ Wie werden Typklassen instanziiert?
- ❑ Wie kann man mit Typklassen Signaturen einschränken?
- ❑ Was ist Currying? Welche Form verwendet Haskell, welche C?
- ❑ Wie funktioniert Partial Application? Wann verwendet man es?
- ❑ Was versteht man unter Funktionen höherer Ordnung?
- ❑ Was sind Listenfunktionale und wofür benötigt man sie?
- ❑ Welches Listenfunktional kann die anderen beiden ersetzen?
- ❑ Wodurch kann man auf begrenztem Speicher unendliche Listen anlegen?
- ❑ Wie arbeitet man mit unendlichen Listen?