1. Funktionen höherer Ordnung

Funktionen, die wiederum Funktionen als Argument oder Resultat haben.

square :: Int → Int Fkt. erster Ordnung

plus :: Int → (Int → Int) Fkt. höherer Ordn.

↑ Resultat ist eine Fkt

Bsp für Fkt, die Funktionen sowohl als Argument als auch als Ergebnis hat: Funktionskomposition (" ∘ ")

f ∘ g ist die Funktion, die aus der Komposition von f und g entsteht (führt erst g, dann f aus)

Typ b→c Typ a→b | f und g entstehen

Typ a→c

In Haskell: comp :: (b→c) → (a→b) → (a→c)

Ist in Haskell als Infix-Fkt. vordefiniert.

Wenn square Zahlen quadriert und half Zahlen halbiert.

(comp half square) 4 = $\frac{4^2}{2}$ = 8

(half . square) 4 = 8

Bsp curry und uncurry

Dann: curry (uncurry g) = g und uncurry (curry f) = f

Nutzen von Funktionen höherer Ordnung:

- Benutze eine Fkt. höherer Ordnung, um bestimmte Algorithmen-Struktur zu implementieren.
- Für Implementierung v. konkreten Algorithmen verwende dann diese Fkt. höherer Ordnung anstatt die Algorithmen-Struktur immer wieder neu zu implementieren.

1. Bsp: map

- suclist ersetzt alle Zahlen in einer Liste durch ihren Nachfolger.

```
suc :: Int -> Int
suc = plus 1
```

- sqrtlist arbeitet analog

⇒ Idee: formuliere eine Fkt., die die Alg-Struktur "Durchlaufe eine Liste und wende eine Fkt. auf jedes Element an" implementiert.

⇒ Abstrahiere von den Unterschieden zwischen suclist und sqrtlist
D.h: ersetze Unterschiede durch Variablen

- ersetze Int bzw. Float durch Typvariable (benötigt Sprache mit param. Polymorphismus)
- ersetze suc bzw. sqrt durch Variable (benötigt Sprache mit Fkt. höherer Ordnung)

Fkt-Variable g müsste ein Eingabeparameter von f sein.
↑ Typ a->b

map g entspricht der Funktion f

D.h. map g xs wendet die Fkt. g auf jedes Element der Liste xs an.

map ist vordef. in Haskell
⇒ Anstatt Rekursionsstruktur immer wieder neu zu implementieren, verwende map in konkreten Algorithmen.
↳ deutlich erhöhte Lesbarkeit

Implementierung v. suclist + sqrtlist:

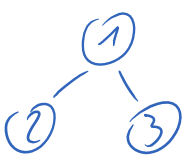
```
suclist :: [Int]->[Int]
suclist xs = map suc xs
```

```
sqrtlist :: [Float] -> [Float]
sqrtlist xs = map sqrt xs
```

Analog dazu kann map auch für eigene Datenstrukturen def. werden; z.B. für Binärbäume:

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

```
Node 1 (Node 2 Leaf Leaf)
       (Node 3 Leaf Leaf)
```




```
mapTree :: (a->b) -> (Tree a) -> (Tree b)
```

```
mapTree g Leaf = Leaf
mapTree g (Node x l r) = Node (g x) (mapTree g l)
                                   (mapTree g r)
```

2. Bsp für vordef. Fkt. höherer Ordnung: filter

- dropEven löscht alle geraden Zahlen aus einer Liste

  dropEven [1,2,3,4] = [1,3]

- dropUpper löscht alle Großbuchstaben aus einem String

  dropUpper "GmbH" = "mb"

Vordef. Fkt. sind in Libraries organisiert.
Fkt. odd ist in der Standard-Library "prelude".
Fkt. isLower ist im Modul Data.Char,
muss am Anfang des Files importiert werden.

```
import Data.Char                  ← importiert alle Funktionen des Moduls
oder  import Data.Char (isLower)  ← importiert nur isLower
```

Abstrahiere wieder von den Unterschieden:

- Ersetze Int bzw. Char durch Typvariable (benötigt param. Polymorph.)
- Ersetze odd bzw. isLower durch Variable (benötigt Fkt. höherer Ord.)

filter g xs löscht alle Elemente aus xs, bei denen g nicht wahr ist

2. Unendliche Datenobjekte

Aufgrund der outermost Auswertungsstrategie kann man in Haskell mit unendl. Datenobjekten programmieren.

from 2 = [2, 3, 4, ...] unendliche Liste

→ Auswertung von from 2 terminiert nicht.

take n [x_1, .., x_n, x_{n+1}, ...] = [x_1, ..., x_n]

(auch für unendliche Listen aufgrund der Auswertungsstrategie von Haskell).

⇒ Man kann mit unendl. Datenobjekten programmieren, wenn zur Berechnung des Ergebnisses nur ein endl. Teil des Datenobjekts gebraucht wird, dann terminiert das Prog. trotzdem.

Bsp: Berechne Liste aller Primzahlen

Generelles Vorgehen bei Algorithmen auf unendl. Datenobj.:

- Erzeuge zunächst (potentiell unendl.) Liste von Approximationen an die Lösung.

- Filtere daraus die wirkliche Lösung heraus.

z.B: Sieb des Eratosthenes

[2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ...]

Implementierung in Haskell:

drop_mult x xs löscht alle Vielfachen von x aus der Liste xs

y `mod` x ≠ 0 gdw. y ist kein Vielfaches von x

dropall: löscht alle Vielfachen des ersten Elements aus der Liste u. ruft sich rekursiv auf der entstehenden Liste ohne 1. Arg. auf

primes = [2, 3, 5, 7, ...] terminiert nicht

take 5 primes = [2, 3, 5, 7, 11]