Einführung in die Informatik

Haskell

Eine moderne, funktionale Programmiersprache

Wolfram Burgard

Motivation

- Während in **imperativen Programmiersprachen** das Konzept der **Variablen als Speicherbereich** für Werte im Vordergrund steht, ist in **funktionalen Programmiersprachen** der Begriff der **Funktion** von zentraler Bedeutung.
- Die zentrale Anweisung in **imperativen Programmiersprachen** ist die **Zuweisung**. Ein Problem wird durch die schrittweise Veränderung des Zustandsraums (d.h. der Werte der Variablen) gelöst. Dabei spielt die **Reihenfolge der Anweisungen** (Statements) eine **entscheidende Rolle**.
- In funktionalen Programmiersprachen wie Haskell werden **Berechnungen** hingegen vornehmlich durch die **Auswertung von Ausdrücken** durchgeführt.
- Ein **funktionales Programm** ist eine **Menge von (Funktions-) Definitionen**.
- Die **Ausführung** wird in **funktionalen Programmiersprachen** durch die **Auswertung von Ausdrücken** (in ihrer Umgebung) durchgeführt.

Zwei motivierende Beispiele

1. Betrachten wir die Gleichung

$$x^2 - 2x + 1 = 0$$

   In diesem Kontext bezeichnet x keineswegs eine Speicherzelle, deren Wert während der Auswertung verändert werden kann. Vielmehr sucht man als Lösung **einen** Wert, den man für x einsetzen kann, so dass die Gleichung erfüllt wird.

2. Betrachten wir das Programmstück

```
x = 3;
y = (++x)*(x--);
```

   Offensichtlich hängt der Wert von `y` von der Auswertungsreihenfolge ab.

3. In funktionalen Sprachen hat ein **Ausdruck** (abhängig von seiner Umgebung) **immer denselben Wert.** Dies bezeichnet man auch als **Referential Transparency**.

Literatur

- P. Hudak & J. Peterson. A Gentle Introduction to Haskell 98.
  http://ais.informatik.uni-freiburg.de/lehre/ws03/info1/material/haskell-tutorial.ps
- Rita Loogen. Deklarative Programmierung (Vorlesungsskript Univ. Marburg).
  http://ais.informatik.uni-freiburg.de/lehre/ws03/info1/material/SkriptPraktischeInformatikIIIWS0102.pdf
- http://www.haskell.org/
- Software:
  http://www.haskell.org/Hugs/

Einfache Haskell-Programme

1. Im einfachsten Fall sind Haskell-Programme **Mengen von Definitionsgleichungen**.

2. In jeder Definitionsgleichung wird eine Funktion definiert.

3. Beispielsweise kann eine Funktion zur Berechnung des euklidischen Abstands von zwei Punkten (x_1, y_1) und (x_2, y_2) folgendermaßen realisiert werden:

```
square x                = x * x
distance x1 y1 x2 y2    = sqrt((square (x2 - x1))
                                 + (square (y2 - y1)))
```

Hierbei ist `sqrt` eine in Haskell eingebaute Funktion.

Notation

- Im Gegensatz zur Notation in der (Schul-) Mathematik werden in **Haskell Funktionen** mit **Außenklammern** notiert
- Man schreibt `(f 4)` oder `(g 4 5)` anstelle von *f(4)* oder *g(4,5).*
- Wir unterscheiden **Präfix- und Infix-Funktionen**.
- Binäre arithmetische Funktionen wie `+`,`-` und `*` sind vordefinierte **Infix-Operatoren**, d.h. ihr Bezeichner steht zwischen den beiden Argumenten.
- **Benutzerdefinierte Funktionen** hingegen sind üblicherweise **Präfix**-Funktionen und werden mit Außenklammern geschrieben.

Wächter

- Häufig lassen sich Funktionen doch nicht durch eine einzelne Gleichung hinschreiben.
- Beispielsweise ist der Absolutbetrag folgendermaßen definiert:

$$\texttt{abs(x)} \begin{cases} \texttt{x} & \texttt{falls x >= 0} \\ \texttt{-x} & \texttt{sonst} \end{cases}$$

- Um diese Fälle zu unterscheiden, verwendet man in Haskell so genannte Wächter (engl. guards).

```
abs x | x >= 0     = x
      | otherwise  = (-x)
```

Typen

- In Haskell haben **alle Datenobjekte** einen **wohldefinierten Typ**.
- Ein **Typ** ist die **Zusammenfassung (Menge) von Objekten gleicher Art**.
- Ebenfalls aus Java bekannte Typen sind

`Int`	Menge aller ganzen Zahlen
`Bool`	Menge der Wahrheitswerte `True` und `False`
`Char`	Menge der Zeichen (ASCII-Zeichensatz)

- Neben diesen Basistypen gibt es auch Funktionstypen.
- Beispielsweise ist `Int -> Int` die **Menge aller einstelligen Funktionen von den ganzen Zahlen in die ganzen Zahlen**.
- Allerdings ist `Int -> Int -> Int` die Menge aller zweistelligen Funktionen über den natürlichen Zahlen.
- In Haskell verwenden wir die Notation `Int -> Int -> Int` anstelle von `Int x Int -> Int`.
- Haskell erlaubt es, **optional Typangaben** der Form `name :: type` **zu Funktionsdefinitionen** hinzuzufügen.

Beispiele für Typen

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

square :: Int -> Int
square x = x*x

distance :: Double -> Double -> Double -> Double -> Double
distance x1 y1 x2 y2 = sqrt((square (x2 - x1))
                            + (square (y2 - y1)))

epsilon :: Double
epsilon = 0.000001

allEqual :: Int -> Int -> Int -> Bool
allEqual n1 n2 n3 = (n1 == n2) && (n1 == n3)


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

Auswertung

- Bei der **Auswertung von Ausdrücken** werden die **Funktionsdefinitionen wie Ersetzungsregeln interpretiert** (ähnlich wie bei Grammatiken).
- In jedem (sogenannten) **Reduktionsschritt**, wird eine **Funktionsanwendung durch den Rumpf der entsprechenden Funktionsdefinition ersetzt**.
- Dabei werden die **Parameter** der Funktion in der Funktionsdefinition **durch die aktuellen Parameter** ersetzt.
- Ein **reduzierbarer Ausdruck** wird üblicherweise **Redex** (reducible expression) genannt.
- Hat ein **Ausdruck keine reduzierbaren Teilausdrücke** so ist er in **Normalform**.
- Die **Normalform eines Ausdrucks** ist das **Ergebnis seiner Auswertung**.

Beispiele

```
square 5
```

```
allEqual 2 3 5
```

```
maxInt 3 1
```

```
allEqual (maxInt 1 5) 5 (maxInt 4 2)
```

Rekursive Funktionen

1. In funktionalen Sprachen stellt die **Rekursion** die **wichtigste Kontrollstruktur** dar.
2. In **jeder Definitionsgleichung** wird eine **Funktion** definiert.
3. Konstanten sind nullstellige Funktionen.
4. Beispiel: Berechnung des ggTs

```
ggt a b | b == 0     = a
        | a == 0     = b
        | a >= b     = ggt (mod a b) b
        | otherwise  = ggt a (mod b a)
```

Beispiele

```
ggt 48 36
```

```
ggt a b | b == 0      = a
        | a == 0      = b
        | a >= b      = ggt (mod a b) b
        | otherwise   = ggt a (mod b a)
```

Lokale Definitionen

- Häufig treten in Funktionsdefinitionen **Teilausdrücke mehrfach** auf.
- Dies führt dann dazu, dass die entsprechenden **Teilausdrücke auch mehrfach ausgewertet** werden:

```
gaussSum n = (div (n * (n + 1)) 2)
squareGaussSum n = (gaussSum n) * (gaussSum n)
```

- Wenngleich diese Funktionen korrekt funktionieren, haben sie den Nachteil, dass der Teilausdruck `(gaussSum n)` im Rumpf zweifach ausgewertet wird.

- Die **Mehrfache Auswertung von Teilausdrücken umgehen**, indem man so genannte „**lokale Definitionen**" verwendet:

```
squareGaussSumFast n = n1 * n1
                       where n1 = gaussSum n
```

Ein Beispiel

Berechnung von $\prod_{i=m}^{n} i$ mit einer rekursiven Funktion durch wiederholte Halbierung des Intervalls:

```
prod m n | m == n   = m
         | m > n    = 1
         | m < n    = (prod m mid) * (prod (mid+1) n)
                       where
                      mid = (div (m + n) 2)
```

Operatoren

- **Binäre Funktionen** können in Haskell **infix oder präfix** definiert werden.
- Wir bezeichnen **binäre Infix-Funktionen** auch als **Operatoren**.
- Indem man den **Bezeichner einer Präfix-Funktion in Hochkommata einschließt**, kann man diesen auch **als Infix-Operator verwenden**.
- Umgekehrt kann man **Operatoren durch Verwendung von Klammern auch präfix** verwenden.
- Daher ist folgendes Programm zulässig:

```
f x y = x + 2 * y
x 'g' y = 1 + x 'f' y
h x = (g x x)
```

Listen

- **Listen** sind eine der **grundlegenden Datenstrukturen** in funktionalen Programmiersprachen.
- Haskell stellt **polymorphe Listen** zur Verfügung, d.h. Listen, in denen alle Elemente vom gleichen Typ sind.
- Typische Beispiele sind Listen von Integers oder Listen von Characters.
- Instanzen solcher Listen sind

```
[1,2,3]
['a','b','c']
```

- Allerdings ist `[2,'b']` nicht zulässig, da 2 und `'b'` nicht zu einem gemeinsamen Typ gehören.

Beispiele für Listen

```
[1, 2, 3, 2, 1]            :: [Int]
['a', 'b', 'c']            :: [Char]
[True, False, True]        :: [Bool]
[(+), (*), div, mod]       :: [Int -> Int -> Int]
```

Für den Listentyp `[Char]` wird auch das **Typsynonym** `String` verwendet:

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

Zeichenketten der Form `"String"` sind eine Kurzschreibweise für `['S', 't', 'r', 'i', 'n', 'g']`.

Datenkonstruktoren

- **Datenstrukturen** werden in funktionalen Sprachen wie Haskell mit **Datenkonstruktoren** gebildet.
- Datenkonstruktoren können als **spezielle Funktionssymbole** aufgefasst werden, die frei interpretiert werden.
- **Listen** werden mit den **zwei Konstruktorsymbolen** `[]` und `(:)` erzeugt.
- Der Konstruktor `[] :: [t]` wird auch **Nil** genannt. Er ist eine **nullstellige Konstruktorkonstante zur Darstellung der leeren Liste**.
- Der Konstruktor `[]` hat den Typ `[t]` für beliebigen Typen `t`. Wir bezeichnen hierbei `t` auch als **Typvariable**.
- Der Konstruktur `(:)` ist der so genannte **Cons-Operator**, der einen binärer Infix-Konstruktor realisiert.
- Der Cons-Operator **erzeugt aus einem Element `x` und einer Liste `l` eine neue Liste, die dadurch entsteht, dass man der Liste `l` das Element `x` voranstellt**.

Repräsentationen von Listen

- Die Standardrepräsentation für Listen ist `[1, 2, 3]`
- Dies ist allerdings nur eine Abkürzung für `1:(2:(3:[]))`.
- Die leere Liste wird durch die Konstante `[]`repräsentiert.
- `:` ist ein zweistelliger Infix-Operator, der eine Liste erzeugt, indem er das erste Argument der durch das zweite Argument gegebenen Liste voranstellt.
- `:` ist rechtsassoziativ. Wir können daher unsere Liste auch schreiben als `1:2:3:[]`.

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

[True,False,True] = True:False:True:[]

[div,mod,(+)] = div:mod:(+):[]
```

Funktionen für Listen

- Zur Definition von Funktionen für Listen (und andere strukturierte Objekte) benötigt man Testfunktionen.
- Darüber hinaus benötigen wir Funktionen, um auf die Elemente der Strukturen zugreifen zu können.
- Typische Tests sind, ob die Liste leer ist oder nicht:

```
null l    = (l == [])
nonNull l = (l /= [])
```

Selektorfunktionen für Listen

- Der Zugriff auf Elemente einer Liste wird mit den Standardfunktionen `head` und `tail` durchgeführt:

```
head (x:xs)  = x
tail (x:xs)  = xs
```

- Die Typen dieser Funktionen sind.

```
head :: [t]-> t
tail :: [t] -> [t]
```

Länge einer Liste

- Eine Standardfunktion für Listen ist `length`.
- Sie berechnet die Länge der Liste, d.h. die Anzahl der Elemente in der Liste.
- Wir verwenden hier eine rekursive Variante:
  - Die leere Liste repräsentiert durch `[]` hat die Länge 0.
  - Eine nicht-leere Liste hat eine um eins größere Länge als der Rest der Liste ab dem 2. Element (tail).

```
length l        | l == []   = 0
                | otherwise = 1 + length (tail l)
```

Eine analoge Java-Funktion

Wir implementieren eine entsprechende Hilfsmethode in der Klasse `SingleLinkedList`:

```
private int length(Node n){
    if (n == null)
        return 0;
    else
        return 1 + length(n.getNextNode());
}
```

Auf der Basis dieser Methode können wir jetzt eine Methode zur Bestimmung der Länge von Listenobjekten der Klasse `SingleLinkedList` realisieren:

```
public int length(){
    return length(this.head);
}
```

Kurznotationen für Listen

Für **Zahlenlisten** gibt es in Haskell eine praktische **Kurznotation**:

$$\texttt{[m..n]}\begin{cases}\texttt{[m,m+1,m+2,... ,n]}, \text{ falls } \texttt{m < n}\\ \texttt{[]}, \text{ sonst}\end{cases}$$

Beispiele: `[1..4] => [1,2,3,4], [4..1] => []`

Weiter gilt für `k < n`:

`[k,m..n] => [k,k+(m-k),k+2*(m-k), ..., n1]`
mit `n1 = k+j*(m-k) < n` und `j` maximal

Beispiele: `[1..4] => [1,2,3,4], [4,6..3] => []`

Pattern Matching

- Bei Datenstrukturen wie Listen tauchen in den **Wächtern** häufig **typische Bedingungen** auf, die beispielsweise zwischen der leeren und der nichtleeren Liste unterscheiden.
- Solche Bedingungen lassen sich auf der **linken Seiten der Funktionsgleichungen durch so genannte Muster (Pattern)** formulieren.
- Typische Muster sind z.B. `[]`oder `(x:xs)`.
- Bei der **Anwendung einer Funktion** (d.h. wenn eine Funktion aufgerufen wird) werden dann die **aktuellen Parameter mit diesen Mustern verglichen**.
- Diesen Prozess nennt man **Pattern Matching**.
- Dabei werden die **Gleichungen von oben nach unten durchsucht**.
- Die **erste passende Gleichung wird verwendet**.
- Dann wird der **Ausdruck auf der rechten Seite ausgewertet**, wobei das **Resultat als Ergebnis des Funktionsaufrufs** verwendet wird.
- Wird **keine passende Gleichung** gefunden, kommt es zu einer **Fehlermeldung**.

Länge einer Liste mit Pattern Matching

- Die leere Liste repräsentiert durch `[]` hat die Länge 0.
- Eine nicht-leere Liste mit erstem Element `x` und Rest `xs` hat eine um eins größere Länge als `xs`.

```
length []          = 0
length (x:xs)      = 1 + length xs
```

Verketten von Listen

Eine **wichtige Funktion** für Listen ist das **Verketten von zwei Listen** `l` und `ys`. Ergebnis soll eine Liste sein, die durch Anhängen von `ys` an `l` entsteht.

- Ist `l` die leere Liste, so ist `ys` das Ergebnis.
- Ist `l` nicht leer, d.h. hat es die Form `(x:xs)` so ist das Ergebnis die Liste mit erstem Element `x` und einem Rest, der sich aus der Verkettung von `xs` und `ys` ergibt.

```
append [] ys            = ys
append (x:xs) ys        = x:(append xs ys)
```

- Für das Verketten von Listen gibt es in Haskell den **eingebauten Operator `++`**.

Ein Anwendungsbeispiel

```
append [] ys      = ys
append (x:xs) ys  = x:(append xs ys)
```

```
append [1,2,3] [4,5,6]
```

Ein Anwendungsbeispiel

```
append [] ys       = ys
append (x:xs) ys   = x:(append xs ys)
```

- Für die Auswertung eines Aufrufs wird stets der auszuwertende Ausdruck durch die entsprechende rechte Seite der Gleichung ersetzt.
- Im Fall von `append [1,2,3] [4,5,6]` ergibt sich folgende Auswertung:

```
=> append [1,2,3] [4,5,6]
=> 1:append [2,3] [4,5,6]
=> 1:2:append [3] [4,5,6]
=> 1:2:3:append [] [4,5,6]
=> 1:2:3:[4,5,6]
```

- Der letzte Ausdruck entspricht `[1,2,3,4,5,6]`.

Verdoppeln der Werte aller Elemente einer Liste

- Ist `l` die leere Liste, so ist `[]` das Ergebnis.
- Ist `l` nicht leer, d.h. hat es die Form `(x:xs)` so ist das Ergebnis die Liste mit erstem Element `2*x` und einem Rest, der sich aus der Verdoppelung der Elemente in `xs` ergibt.

```
double []              = []
double (x:xs)          = (2*x):(double xs)
```

Das Invertieren einer Liste

- Ist `l` die leere Liste, so ist `[]` das Ergebnis.
- Ist `l` nicht leer, d.h. hat es die Form `(x:xs)`, so erhalten wir das Ergebnis, indem wir `xs` invertieren und daran `[x]` anhängen.

```
reverse []              = []
reverse (x:xs)          = reverse xs ++ [x]
```

Anwendung von `reverse`

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Aufwandsanalyse für `reverse` (1)

`reverse` $[x_1, \ldots, x_n]$ wird zunächst in n+1 Schritten reduziert zu

$(\ldots([\,] \mathbin{++} [x_n]) \mathbin{++} [x_{n-1}]) \mathbin{++} \ldots) \mathbin{++} [x_1]$

Dann werden folgende Reduktionen durchgeführt:

$$
\begin{array}{ll}
\Rightarrow (\ldots([x_n] \mathbin{++} [x_{n-1}]) \mathbin{++} \ldots) \mathbin{++} [x_1] & \text{1 Schritt} \\
\Rightarrow (\ldots([x_n,x_{n-1}] \mathbin{++} [x_{n-2}]) \mathbin{++} \ldots) \mathbin{++} [x_1] & \text{2 Schritte} \\
\Rightarrow (\ldots([x_n,x_{n-1},x_{n-2}] \mathbin{++} [x_{n-3}]) \mathbin{++} \ldots) \mathbin{++} [x_1] & \text{3 Schritte} \\
\ldots & \\
\Rightarrow [x_n,\ldots,x_2] \mathbin{++} [x_1] & \text{n-1 Schritte} \\
\Rightarrow [x_n,x_{n-1}\,,x_{n-2}] & \text{n Schritte}
\end{array}
$$

Aufwandsanalyse für `reverse` (2)

Insgesamt ergibt sich die folgende Anzahl notwendiger Reduktionen:

$$n+1+\sum_{i=1}^{n} i = \sum_{i=1}^{n+1} i = \frac{(n+1)\times(n+2)}{2} \in O(n^2)$$

Die **„naive“ Version von** `reverse` benötigt somit **quadratisch viele Schritte** in Abhängigkeit von der Länge der Liste.

Eine effizientere Variante von `reverse`

Eine **Linearzeit-Version** von `reverse` erhält man durch die Verwendung eines so genannten **Akkumulators**.

Dieser **Akkumulator** dient dazu, beim Durchlaufen der Liste, die **invertierte Liste der bereits besuchten Elemente schrittweise zu konstruieren**.

- Zu Beginn ist die bereits invertierte Teilliste leer.
- Ist die Liste leer, d.h. sind wir am Ende angelangt, ist das Ergebnis die bereits invertierte Teilliste.
- Ist `(x:xs)` die noch zu invertierende Liste und `ys` die invertierte Liste der bereits besuchten Elemente, so erhalten wir das Ergebnis, indem wir das Verfahren rekursiv auf `xs` und die Liste `(x:ys)` anwenden.

Invertieren mit Akkumulator

```
reverse xs  = aux xs []
              where
              aux [] ys     = ys
              aux (x:xs) ys = aux xs (x:ys)
```

Laufzeit dieser Version: $(n+2) \in O(n)$

Weitere Muster

- Betrachten wir die rekursive Definition der Addition für natürliche Zahlen auf der Basis der Inkrement-Operation:

$$\text{add}(x,y) = \begin{cases} y & \text{falls } x = 0 \\ \text{add}(x-1,y+1) & \text{falls } x > 0 \end{cases}$$

- Eine mögliche Realisierung in Haskell mit Hilfe von **Pattern** ist

```
add 0 b             = b
add (a+1) b         = add a (b+1)
```

Tupel

- **Tupel** sind **endliche Folgen von Elementen beliebigen Typs**:

  $\texttt{e}_1\texttt{, ..., e}_n \texttt{ :: (t}_1\texttt{, ..., t}_n\texttt{)}$ falls $\texttt{e}_\texttt{i}\texttt{::t}_\texttt{i}$ für alle $1 \leq \texttt{i} \leq n$

- Tupel können in Funktionsdefinitionen als **Pattern** auftreten:

```
fst :: (a,b) -> a
fst (x,y) = x
```

```
snd :: (a,b) -> b
snd (x,y) = y
```

- Beide Funktionen sind in Haskell vordefiniert.

List-Comprehensions

- List-Comprehensions stellen eine einfache Möglichkeit dar Listen zu definieren.
- Die Notation ist stark an die Notation von Mengen angelehnt.
- Beispielsweise kann die Menge der Quadratzahlen aller geraden Zahlen zwischen `0` und `20` sehr leicht in Haskell übesetzt werden.
- Das Haskell-Programm für die Menge

$$m = \{x^2 \mid x \in \{0...20\} \wedge x \bmod 2 == 0\}$$

ist

```
m = [xˆ2 | x <- [0..20], mod x 2 == 0]
```

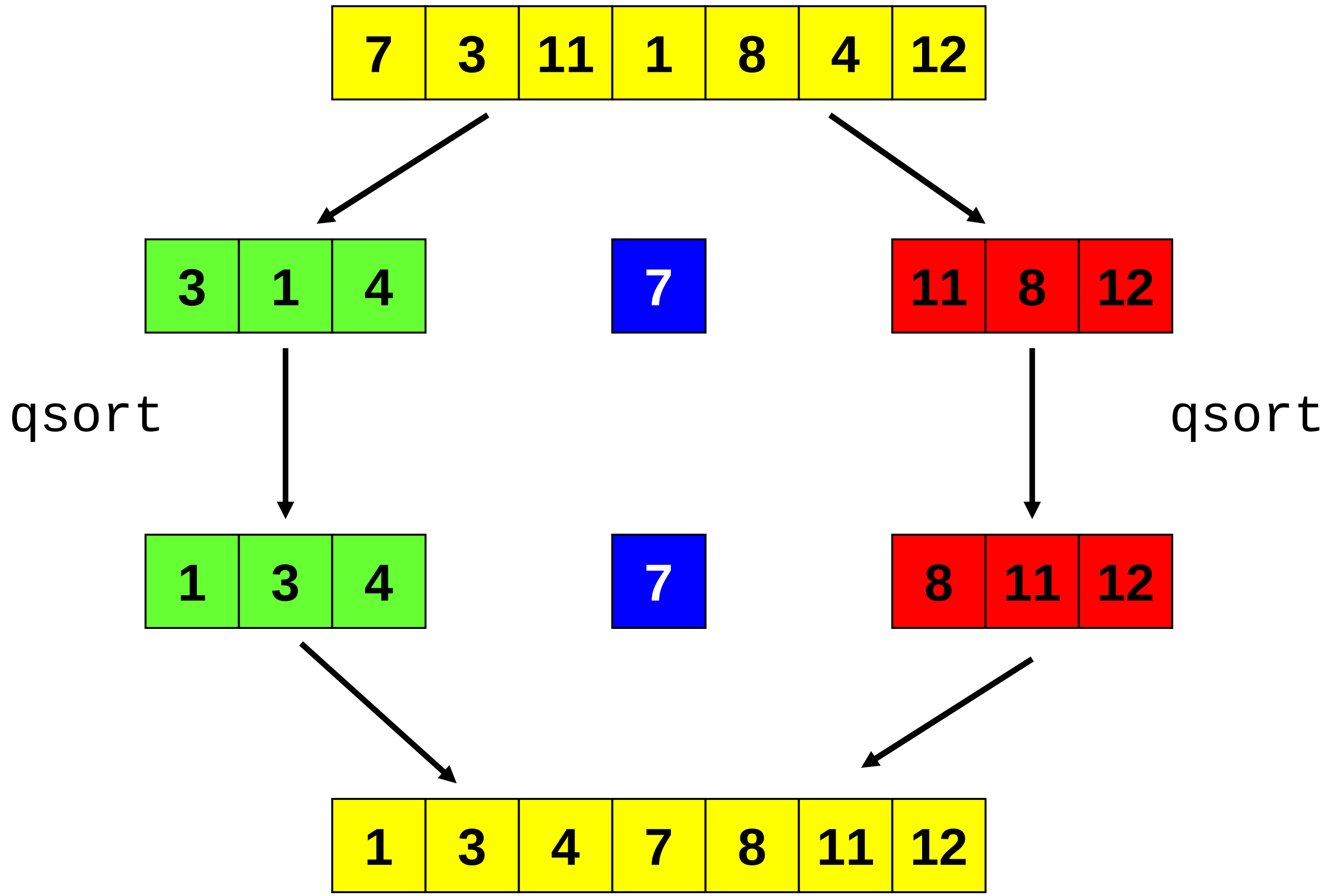
- Das Ergebnis der Auswertung von `m` ist dann:

```
[0,4,16,36,64,100,144,196,256,324,400]
```

Sortieren mit List-Comprehensions: Quicksort

- Eines der bekanntesten **Sortierverfahren** ist **Quicksort**.
- Es lässt sich folgendermaßen beschreiben:
  - Eine leere Liste ist fertig sortiert.
  - Um eine nicht leere Liste zu sortieren,
    1. nimmt man das erste Element x (**Pivot**-Element),
    2. **teilt den Rest der Liste** in zwei Listen s und b, die jeweils alle Elemente enthalten, die **kleiner bzw. nicht kleiner** sind als x (**Split-Operation**),
    3. **sortiert beide Teillisten** zu s' und b' (Rekursion) und **fügt die Ergebnislisten** in der Form s' ++ [x] ++ b' (oder s' ++ (x:b')) **zusammen**.

Funktionsweise von Quicksort



Quicksort mit List-Comprehensions

```
qsort []          = []
qsort (x:xs)      = qsort [b | b <- xs, b <= x]
                    ++ [x]
                    ++ qsort [b |b <- xs, b > x]
```

Hierbei ist der Operator ++ die Verkettung von Listen (analog zu + in Java).

Etwas effizienter ist die folgende Version:

```
qsort []          = []
qsort (x:xs)      = qsort [b | b <- xs, b <= x]
                    ++ (x:qsort [b |b <- xs, b > x])
```

Potenzmengenberechnung mit List-Comprehensions

- Die Potenzmenge 2^M einer Menge ist die Menge aller Teilmengen von M.
- Die Potenzmenge der leeren Menge enthält lediglich die leere Menge.
- Ist $M = M' \cup \{m\}$ eine nicht leere Menge, so besteht die Potenzmenge von M aus der Potenzmenge von M' vereinigt mit den Mengen, die aus der Potenzmenge von M' und deren Vereinigung mit $\{m\}$ erzeugt werden können:

$$2^M = 2^{M'} \cup \left\{ m' \cup \{m\} \mid m' \in 2^{M'} \right\}$$

- Das entsprechende Haskell-Programm ist:

```
powset []     = [[]]
powset (x:xs) = ys ++ [x:y | y <- ys]
                where
                ys = (powset xs)
```

Ein Anwendungsbeispiel

```
powset []     = [[]]
powset (x:xs) = ys ++ [x:y | y <- ys]
                where
                ys = (powset xs)
```

Algebraische Datenstrukturen

- Der Programmierer kann in Haskell eigene Datentypen definieren.
- Neue Typen werden mittels einer Datentypdefinition eingeführt.

$$\texttt{data T } a_1 \ \ldots\ a_k \ = \ C_1\ t_{11}\ \ldots\ t_{1m_1}$$
$$\mid\ \ldots$$
$$\mid\ C_n\ t_{n1}\ \ldots\ t_{nm_n}$$

- Die C_i heißen **Konstruktoren**. Mit ihrer Hilfe werden **Elemente des Datentyps** T **konstruiert**. Die a_i sind die **Typparameter** des Datentyps T.
- Eine solche Deklaration definiert n **Konstruktor(funktion)en**

$$C_j\ t_{j1}\ \ldots\ t_{jm_j} C \to T\ a_1\ \ldots\ a_k$$

wobei die Typvariablen in den Komponententypen aus $a_1\ \ldots\ a_k$ sind.

Beispiele

Vordefinierte Typen wie `Bool` und `Char` lassen sich beispielsweise folgendermaßen definieren:

```
data Bool        =  False | True
data Char        = '\NUL' | ... | '\255'
```

Dies sind so genannte **Aufzählungstypen**, wie beispielsweise in

```
data Color       = Red | Green | Blue
```

Tupel-Typen lassen sich folgendermaßen definieren:

```
data Point       = P int int
```

Typdefinitionen können auch rekursiv sein. Eine **explizite Deklaration der Listen** kann beispielsweise folgendermaßen durchgeführt werden:

```
List a           =  Nil | Cons a (List a)
```

Binärbäume

- Binärbäume lassen sich in Haskell folgendermaßen deklarieren:

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

- Ein Baum ist entweder ein Blatt oder er besteht aus einem Knoten mit einem linken und rechten Teilbaum gleichen Typs.

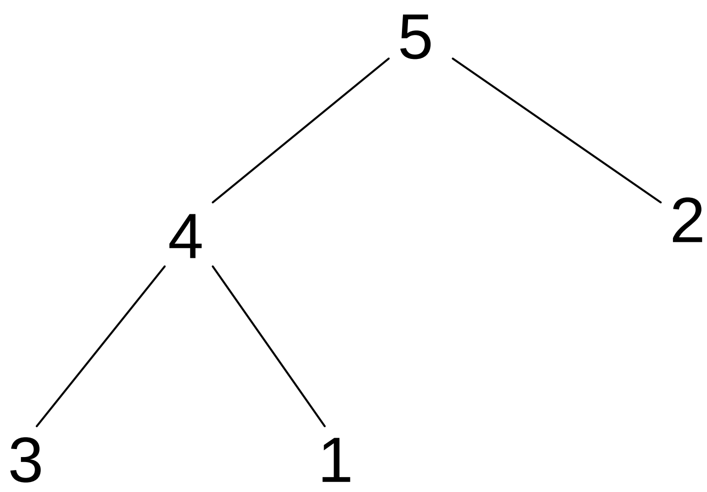
- Dabei enthalten sowohl Knoten als auch Blätter Informationen

- Typische Ausprägungen von Bäumen sind

```
(Leaf 3)
(Node 3 (Leaf 1) (Leaf 4))
(Node 'a' (Node 'b' (Leaf 'c') (Leaf 'd')) (Leaf 'e'))
```

Ein Beispiel

`(Node 5 (Node 4 (Leaf 3) (Leaf 1)) (Leaf 2)) :: Tree Int`



Funktionen für Binärbäume (1)

- Um alle Elemente eines Binärbaums aufzusummieren, können wir wie folgt vorgehen:

```
treeSum (Leaf x)              = x
treeSum (Node x left right) = x + treeSum left + treeSum right
```

- Eine Funktion, die wir schon kennen, ist das Berechnen der Inorder-Reihenfolge aller Knoteninhalte:

```
inOrder (Leaf x)              = [x]
inOrder (Node x left right) = inOrder left ++ (x:inOrder right)
```

- Beispielanwendungen:

```
treeSum  (Node 3 (Leaf 1) (Leaf 4)) => 8
inOrder  (Node 3 (Leaf 1) (Leaf 4)) => [1,3,4]
```

Funktionen für Binärbäume (2)

- Die Tiefe eines Binärbaums ist folgendermaßen definiert:

```
depth                         :: Tree a -> int
depth (Leaf x)                = 1
depth (Node x left right) = 1 + max (depth left) (depth right)

depth  (Node 3 (Leaf 1) (Leaf 4)) => 2
```

- Die folgende Funktion berechnet das kleinste Element in einem Baum

```
minTree                :: Tree Int -> Int
minTree (Leaf x)       = x
minTree (Node x l r) = min x (min (minTree l) (minTree r))

minTree  (Node 3 (Leaf 1) (Leaf 4)) => 1
```

- Die Funktionen `min` und `max` sind in Haskell vordefiniert.

Ein weiteres Beispiel: Arithmetische Ausdrücke

- Ein arithmetischer Ausdruck ist entweder eine Zahl oder kann durch die Addition oder Subtraktion von zwei Ausdrücken gebildet werden.

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

- Die Auswertung eines Ausdrucks kann in Haskell dann folgendermaßen auf die eingebauten Operationen zurückgeführt werden:

```
eval :: Expr -> Int
eval (Lit n) = n
eval (Add x y) = eval x + eval y
eval (Sub x y) = eval x - eval y
```

- Anwendung:

```
eval (Add (Lit 3) (Lit 4)) => 7
```

Funktionen höherer Ordnung (Motivation)

- Die bisher betrachteten Funktionen operieren alle auf Daten, d.h. Zahlen, Characters, Listen etc.
- Im Folgenden wollen wir Funktionen betrachten, deren Argumente selbst wieder Funktionen sein können.
- Solche Funktionen heißen **Funktionen höherer Ordnung** (engl. **higher-order functions**) oder **Funktionale**.
- Bei den oben betrachteten Listenoperationen haben wir in der Regel die Listen durchlaufen und immer dieselben Operationen durchgeführt.
- Offensichtlich ist dabei das Durchlaufen der Liste ein immer wieder vorkommendes Muster; lediglich die auszuführende Operation variiert.
- Funktionen höherer Ordnung abstrahieren nun von den auszuführenden Operationen, indem sie typische Operationsmuster zur Verfügung stellen.

Anwendung einer Funktion auf alle Elemente einer Liste

- Ein typisches Muster ist das Anwenden ein- und derselben Operation auf alle Elemente einer Liste.
- Ein Beispiel ist das Erhöhen aller Gehälter in einer Liste um 10%:

```
add10Percent x            = x * 1.1
add10PercentList []       = []
add10PercentList (x:xs) = (add10Percent x):add10PercentList xs
```

- Will man alle Elemente einer Liste quadrieren, geht man analog vor:

```
square x                 = x*x
squareList []            = []
squareList (x:xs)        = (square x):squareList xs
```

Die Funktion `map`: Abstraktion von der anzuwendenden Funktion

- Offensichtlich hatten wir in den oben betrachteten Funktionen folgendes Muster:

```
f      x          = ...
fList []          = []
fList (x:xs)      = (f x):fList xs
```

- Wenn wir nun die anzuwendende Funktion `f` als Argument übergeben können, erhalten wir die erforderliche Abstraktion.
- Dies leistet die Funktion `map`.

```
map f []          = []
map f (x:xs)      = (f x): map f xs
```

Anwendungen der Funktion `map`

- Mithilfe der Funktion `map` lassen sich nun die entsprechenden Listenoperationen einfach realisieren.
- Um alle Elementen um 10% zu erhöhen, verwenden wir folgendes Statement:
  `add10PercentList xs = map add10Percent xs`
- Analog können wir alle Elemente einer Liste quadrieren:
  `squareList xs = map square xs`

Eine Beispielauswertung

```
map square [1,2,3]
```

```
map f []       = []
map f (x:xs)   = (f x): map f xs
```

Eine Beispielauswertung

```
map square [1,2,3]
(square 1):map square [2,3]
1*1:map square [2,3]
1:map square [2,3]
1:(square 2):map square [3]
1:2*2:map square [3]
1:4:map square [3]
1:4:(square 3):map square []
1:4:3*3:map square []
1:4:9:map square []
1:4:9:[]
```

Die Funktion `foldr`

- Ein **weiteres (Rekursions-) Muster** stellen die Funktionen `sum` und `prod` dar, die die Summe bzw. das Produkt aller Elemente einer Liste berechnen.
- Typische, rekursive Definition dieser Funktionen sind:

```
sum []          = 0
sum (x:xs)      = x + sum xs
prod []         = 1
prod (x:xs)     = x * prod xs
```

- Wie bei der Funktion `map` **unterscheiden sich beide Funktionen lediglich durch den Operator** (`+` bzw. `*`).
- Ausgehend vom Startwert (`0` oder `1`) werden die Elemente von links nach rechts mit der entsprechenden Operation verrechnet.

Abstraktion durch `foldr`

- Wenn wir eine **allgemeine Form einer Funktion zur Akkumulation von Werten** über einer Liste formulieren wollen, müssen wir

  – die **Operation** und

  – den **Startwert**

  festlegen.

- Dann verfahren wir wie folgt:

  – Wenn die **Liste leer** ist, geben wir den **Startwert** zurück.

  – Ist die **Liste nicht leer**, erhalten wir das Ergebnis, indem wir die **Operation anwenden auf den Kopf der Liste und den Wert**, den wir **für den Rest der Liste** erhalten.

Realisierung der Methode `foldr`

- Die Implementierung der Methode `foldr` orientiert sich an dieser Definition:

```
foldr f e []            = e
foldr f e (x:xs)        = x ‘f‘ foldr f e xs
```

- Damit lassen sich die oben angegebenen Listenoperationen sehr einfach realisieren:

```
sumList xs              = foldr (+) 0 xs
prodList xs             = foldr (*) 1 xs
```

Eine Beispielanwendung

```
foldr (+) 0 [1,2,3]
```

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

Eine Beispielanwendung

```
sumList [1,2,3]
foldr (+) 0 [1,2,3]
1 + foldr (+) 0 [2,3]
1 + 2 + foldr (+) 0 [3]
3 + foldr (+) 0 [3]
3 + 3 + foldr (+) 0 []
6 + foldr (+) 0 []
6 + 0
6
```

Die Funktion `foldl`

- Zusätzlich zur Funktion `foldr` gibt es auch eine analoge Funktion `foldl`, bei der eine **Linksklammerung** durchgeführt wird:

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

- Ein Vergleich:

```
foldl + 0 [1,2,3]
=> foldl + (+ 0 1) [2,3]
=> foldl + (+ (+ 0 1) 2) [3]
=> foldl + (+ ( + (+ 0 1) 2) 3) []
=> (+ ( + (+ 0 1) 2) 3)
```

```
foldr + 0 [1,2,3]
=> + 1 foldr + 0 [2,3]
=> + 1 (+ 2 foldr + 0 [3])
=> + 1 (+ 2 (+ 3 foldr + 0 []))
=> + 1 (+ 2 (+ 3 0))
```

Vergleich von `foldr` und `foldl`

- Für **assoziative Funktionen** macht es natürlich **keinen Unterschied**, ob wir `foldr` oder `foldl` verwenden.
- Man kann aber Beispiele konstruieren, bei denen `foldr` **effizienter** ist als `foldl` und umgekehrt.
- Betrachten wir beispielsweise die folgenden (semantisch äquivalenten) Funktionen zum „Flachklopfen" von Listen von Listen.

```
concatr :: [[a]] -> [a]
concatr = foldr ++ []
```

```
concatl :: [[a]] -> [a]
concatl = foldl ++ []
```

- Während die Funktion `concatr` ***O(n)* Schritte** benötigt, um eine Liste der Länge n „flachzuklopfen", sind bei der Funktion `concatl` ***O*(n^2) Reduktionen** erforderlich.

Ein Beispiel zur Verdeutlichung

```
concatl [[1],[2],[3]]
```

```
foldl f e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```

```
concatr [[1],[2],[3]]
```

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

Ein Beispiel zur Verdeutlichung

```
foldl f e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```

```
concatl [[1],[2],[3]]
foldl ++ [] [[1],[2],[3]]
=> foldl ++ (++ [] [1]) [[2],[3]]
=> foldl ++ (++ (++ [] [1]) [2]) [[3]]
=> foldl ++ (++ (++ (++ [] [1]) [2]) [3]) []
=> (++ (++ (++ [] [1]) [2]) [3])
=> (++ (++ [1] [2]) [3])
=> (++ [1,2] [3])
=> [1,2,3]
```

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

```
concatr [[1],[2],[3]]
=> foldr ++ 0 [[1],[2],[3]]
=> ++ [1] foldr ++ [] [[2],[3]]
=> ++ [1] (++ [2] foldr ++ [] [3])
=> ++ [1] (++ [2] (++ [3] foldr ++ [] []))
=> ++ [1] (++ [2] (++ [3] []))
=> ++ [1] (++ [2] [3])
=> ++ [1] [2,3]
=> [1,2,3]
```

Filtern von Listen

- Mit Hilfe der zweistelligen Funktion `filter` kann man zu einer gegebenen Liste die Teilliste aller Elemente berechnen, die einen per Parameter übergebenem Test erfüllen:

```
filter p []     = []
filter p (x:xs) = if (p x) then (x:filter p xs)
                  else (filter p xs)
```

- Alternativ kann man diese Funktion auch mit einer Listenabstraktion implementieren:

```
filter p xs = [x| x<-xs, p x]
```

- Anwendung (`even x = (mod x 2) == 0`):

```
filter even [0..10] => [0,2,4,6,8,10]
```

Partielle Applikation

- In Haskell (wie auch in anderen funktionalen Programmiersprachen) ist es zulässig, dass **Funktionen auf weniger Argumente angewendet werden als für die Auswertung notwendig ist**.
- Dadurch **entstehen neue Funktionen**, die **Spezialfälle der ursprünglichen Funktionen** sind, da bestimmte Argumente bereits festgelegt sind.
- Gibt man für die resultierende Funktion dann die restlichen Argumente an, so erhält man eine Anwendung der ursprünglichen Funktion mit den gegebenen Parametern.
- Beispielsweise ist die Addition definiert als

```
add x y      = x + y
```

- Dann definiert die partielle Anwendung `(add 1)` die Nachfolgerfunktion.

Typen und partielle Applikation

- Die spezielle **Notation des Typs von Funktionen** erlaubt nun die **Definition von Funktionen mittels partieller Anwendung**.

- Die Notation t_1 `->` t_2 `-> … ` t_n `-> t` oder genauer

  t_1 `-> (`t_2 `-> … (`t_n`-> t) …)`

  verdeutlicht, dass jede **n-stellige Funktion als einstellige Funktion mit einem funktionalen Wertebereich** aufgefasst wird.

- Anders ausgedrückt: Durch die Anwendung einer n-stelligen Funktion auf ein Argument erhalten wir eine (n-1)-stellige Funktion.

- Allgemein: Durch die Anwendung einer n-stelligen Funktion auf k<n Argumente erhalten wir eine (n-k)-stellige Funktion.

Beispiele

- Die spezielle **Notation des Typs von Funktionen** erlaubt nun die **Definition von Funktionen mittels partieller Anwendung**.

- Die Funktion `map` hat den Typ `(a->b)->[a]->[b]` für alle Typen a und b.

- Da `square` den Typ `(int->int)`hat, erhalten wir für die partielle Anwendung von `map` auf `square` den Typ

```
map square :: [int] -> [int]
```

- Die letzten Parameter können, sofern Sie auf beiden Seiten gleich sind, weggelassen werden. Demnach sind folgende Definitionen zulässig:

```
sumList = foldr (+) 0
inc = add 1
```

Partielle Anwendung und Operatoren

- Auch für Operatoren ist die partielle Applikation zulässig.
- Dabei kann man die Applikation auf ein beliebiges der beiden Argumente des Operators vornehmen.
- Beispielsweise ist `(> 0)` eine Funktion, die `True` liefert, falls das Argument größer als Null ist. Andernfalls liefert sie `False`.

  Beispiele:

```
filter (> 5) [0..20]

qsort   []   = []
qsort (x:xs) = qsort (filter (<x) xs)
               ++ (x:qsort (filter (x<=) xs))
```

λ-Abstraktion

- Oft werden Funktionen lediglich als Argument einer Funktion höherer Ordnung verwendet.

- Mit Hilfe der λ-Abstraktion, kann man nun darauf verzichten, solche Funktionen im Rahmen einer Funktionsdefinition mit einem Bezeichner zu versehen.

- Für eine Funktion

  ```
  f x1 ... xn = e
  ```

  ist die entsprechende λ-Abstraktion

  ```
  λ x1 ... xn -> e
  ```

  was in Haskell folgendermaßen ausgedrückt wird:

  ```
  (\ x1 ... xn -> e)
  ```

Beispiele:

```
squareList = map (\ x -> x*x)
evens = filter (\ x -> ((mod x 2) == 0))
```

Weitere Funktionen höherer Ordnung

- **Funktionen höherer Ordnung** werden **nicht nur für Listenoperationen** verwendet.
- Ein typisches Beispiel aus der Mathematik für eine Funktion höherer Ordnung ist die **Komposition von Funktionen**:

$$f \circ g\ x = f(g(x))$$

- Diese Komposition kann auch in Haskell mit Hilfe des **Operators** `(.)` realisiert werden:

```
compose f g = f.g
```

- Der Typ von `compose` ist dementsprechend

```
compose:: (a->b) -> (b->c) -> (a->c)
```

- Eine weitere Standardfunktion ist dei zweifache Anwendung einer Funktion:

```
twice f x = f (f x)
```

Auswertung von Ausdrücken: Redexe

- Die **Ausführung eines funktionalen Programms** geschieht durch die **Auswertung eines Ausdrucks**.
- Dazu müssen wir stets eine **Abbildungsvorschrift** finden, die zu **einem Teilausdruck passt**.
- Ein solcher Teilausdruck heißt **Redex** (Reducible Expression).
- Um Ausdrücke auszuwerten, **ersetzen** wir stets **Redexe** durch rechte Seiten von Gleichungen.

Mehrere Redexe in einem Ausdruck

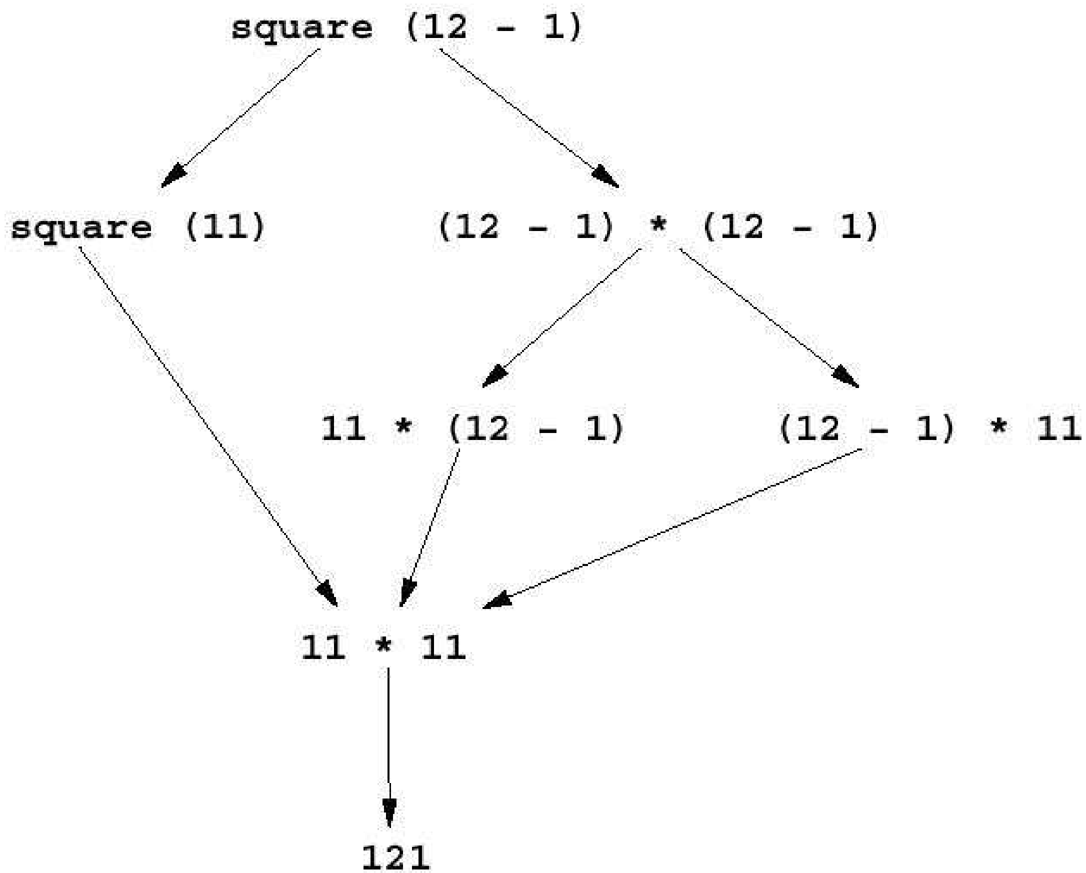
- Üblicherweise enthalten **Ausdrücke mehrere Redexe**.
- Beispielsweise können wir in dem Ausdruck

  ```
  square ((square 4) + (double 2))
  ```

  verschiedene Teilausdrücke ersetzen.
- Dies sind der innere und der äußere Aufruf von `square` sowie der Aufruf von `double`.

Auswertungsstrategien

- Oft gibt es **mehrere Möglichkeiten** einen **Ausdruck auszuwerten**.



- Eine **Auswertungsstrategie** ist ein **Algorithmus zur Auswahl des nächsten Redex**.

Lazy Evaluation: Motivation

- Betrachten wir die Funktion

```
double n = n + n
```

- Dann können wir den Ausdruck `double (double 4)` mindestens auf
- zwei verschiedene Arten auszuwerten:

`double (double 4)`	`double (double 4)`
`double (4 + 4)`	`(double 4) + (double 4)`
`double 8`	`(4 + 4) + (double 4)`
`8 + 8`	`8 + (double 4)`
`16`	`8 +(4 +4)`
	`8 +8`
	`16`

LI- und LO-Reduktion

- Oben haben wir den Ausdruck `double (double 4)` auf zwei verschiedene Arten vereinfacht.
- Auf der linken Seite wurde immer der am **weitesten links stehende Ausdruck**, der **keinen anderen Redex enthält** , ersetzt.
- Dies Strategie heißt **LI-Reduktion** (**Leftmost Innermost Reduction** oder **Eager Evaluation**).
- Auf der rechten Seite wurde immer der am **weitesten links stehende** Redex ersetzt, der **in keinem anderen Redex enthalten** ist.
- Dieses Verfahren heißt **LO-Reduktion** (**Leftmost Outermost Reduction**).

Call by Value und Call by Name

- LI-Reduktion entspricht dem Übergabemechanismus **Call by Value**, der auch in **Java** angewendet wird.
- Wird ein Funktionsaufruf **Call by Value** abgearbeitet, so werden zunächst die **aktuellen Parameter vereinfacht** .
- Die **Ergebnisse der Auswertung** werden dann **im Funktionsrumpf eingesetzt**, der anschließend abgearbeitet wird.
- Demgegenüber entspricht LO-Reduktion dem Mechanismus **Call by Name**.
- Dabei werden die aktuellen Parameter **unausgewertet im Rumpf eingesetzt**.
- Anschließend wird dann der dadurch entstehende Ausdruck ausgewertet.

Vor- und Nachteile dieser Verfahren (1)

- Offensichtlich ist LO-Reduktion der LI-Reduktion gegenüber im Nachteil, wenn ein Funktionsparameter im Rumpf der Definition mehrfach auftritt.
- Beispielsweise wird aus

  ```
  double (double 4)
  ```

  bei der LI-Reduktion

  ```
  double 8
  ```

  während wir ihn bei LO-Reduktion zu

  ```
  (double 4) + (double 4)
  ```

  auswerten.
- Bei der LO-Reduktion wird der aktuelle Parameter, der unausgewertet im Rumpf eingesetzt wird, **doppelt ausgewertet** .

Vor- und Nachteile dieser Verfahren (2)

- Tauchen hingegen Funktionsparameter im Rumpf nicht auf, ist die LO-Reduktion im Vorteil:

  ```
  first n m = n
  ```

- Die LI-Reduktion wertet bei dem Aufruf

  ```
  first 5 (4+4)
  ```

  beide Parameter aus, d.h. berechnet

  ```
  first 5 8
  ```

  und liefert dann das Ergebnis 5.

- Die LO-Reduktion ersetzt diesen Ausdruck hingegen direkt durch das Ergebnis.

Verbesserung der LO-Reduktion

- Wie oben gesehen, liegt der **Nachteil der LO-Reduktion** in der **Mehrfachauswertung von Ausdrücken**.
- Das **Problem der Mehrfachauswertung** lässt sich jedoch dadurch **lösen**, dass man nicht die kompletten Teilausdrücke übergibt, sondern immer nur **Referenzen auf Teilausdrücke**.
- Dies entspricht in Java der Übergabe von Referenzvariablen auf Objekte.
- Beispielsweise werden in Java immer nur Referenzen auf `Vector`-Objekte übergeben.
- Wird der Inhalt des Vektors geändert, sind davon alle Vorkommen dieser Referenzen betroffen.
- In Haskell geht man analog vor. Anstelle der Ausdrücke selbst werden immer nur Referenzen auf Ausdrücke verwendet.
- **Reduktionen von Redexen wirken sich somit in allen Vorkommen dieses Ausdrucks aus**.

Lazy Evaluation

- Da Referenzen auf einen Teilausdruck in mehreren Ausdrücken vorkommen können, wirken sich Reduktionen dieses Teilausdrucks in allen Ausdrücken mit dieser Referenz aus.
- Die LO-Reduktion mit Verwendung von Referenzen auf Ausdrücke bezeichnet man als **Lazy Evaluation**.
- Sie entspricht dem Auswertungsverfahren **Call by Need**.
- Diese Version der LO-Reduktion benötigt höchstens so viele Reduktionsschritte wie die LI-Reduktion.
- Das Motto dieser Strategie ist:

  Berechne den Wert eines Ausdrucks nur einmal und nur dann, wenn es unbedingt nötig ist.

Die Funktion `take`: Extraktion der ersten *n* Elemente

- Eine der sehr häufig benötigte Funktion im Zusammenhang ist die Funktion `take`.
- Diese Funktion dient dazu, die ersten *n* Elemente einer Liste *l* in Form einer Liste zurückzugeben.
- Das erste Argument bestimmt die Anzahl der Elemente.
- Das zweite Argument gibt die Anzahl der zu extrahierenden Elemente an.

1. Die ersten *0* Elemente einer Liste sind durch die leere Liste gegeben.
2. Ist *n* `>` *0* , die Liste aber bereits leer, so ist das Ergebnis die leere Liste.
3. Ist *n* `>` *0* und `l = (x:xs)` nicht leer so ist das Ergebnis die Liste `x:ys`, wobei `ys` die ersten *n* `- 1` Elemente der Liste `xs` sind.

Realisierung der Funktion `take` und ihre Anwendung

Wir müssen drei Fälle unterscheiden, nämlich dass `n = 0` oder `n > 0` und `l =[]`oder `l = x:xs`.

```
take 0 xs = []
take n [] = []
take n (x:xs) = x:take (n-1) xs
```

Anwendung der Funktion `take`:

```
take 3 [1,2,3,4]
```

Ein Anwendungsbeispiel

```
take 3 [1,2,3,4]
1:take 2 [2,3,4]
1:2:take 1 [3,4]
1:2:3:take 0 [4]
1:2:3:[]
[1,2,3]
```

Die Funktion `elementAt`

- Auch die Funktion `elementAt`, die wir bereits von `Vector`-kennen, lässt sich analog definieren:
  - Ist `n = 0` , ist das Ergebnis das erste Element der Liste.
  - Ist hingegen `n > 0` , ist das Ergebnis das `(n - 1)`-te Element der Restliste ab dem zweiten Element.

```
elementAt 0     (x:xs) = x
elementAt (n+1) (x:xs) = elementAt n xs
```

Unendliche Objekte

- Eine der faszinierendsten Möglichkeiten, die durch **Lazy Evaluation** erlaubt wird, ist die Verwendung und Verarbeitung von eigentlich **unendlich großen Objekten**.
- Die einfachste Gleichung für eine **unendliche Datenstruktur** ist eine unendlich lange Liste von Einsen:

```
ones = 1:ones
```

- Eine alternative Version ist die Verwendung einer List-Comprehension:

```
ones = [1,1..]
```

- Beide Funktionen `ones` berechnen eine unendliche Liste von Einsen.
- Wird der Ausdruck `ones` im Haskell-System eingegeben, wird eine nicht-terminierende Folge von Einsen ausgegeben, die nur durch Control-C angehalten werden kann.

List Comprehensions für unendliche Listen

- Wie oben bereits vorgeführt, können **List Comprehensions** dafür verwendet werden **unendliche Listen zu erzeugen**.

- Hierbei gelten folgende Regeln:

  - `[x..]` entspricht der Liste `[x,x+1,x+2,...]`.

  - `[x,y..]` entspricht einer Liste mit erstem Element `x`, wobei sich das `i`-te Element der Liste als `x + i * (y – x)` berechnet.

- Damit lassen sich beispielsweise sehr einfach die ungeraden und geraden Zahlen definieren:

```
evens = [0,2..]
odds = [1,3..]
```

Lazy Evaluation und unendliche Listen

- Durch die LO-Reduktion werden die Ausdrücke immer nur so weit wie nötig ausgewertet.
- Deswegen können wir uns mit `take` auch die ersten n einer unendlichen Liste berechnen lassen:

```
take 2 ones
```

- Die LO-Reduktion dieses Ausdrucks ergibt:

```
take 2 ones
take 2 1:ones
1:take 1 ones
1:take 1 1:ones
1:1:take 0 ones
1:1:[]
```

Weitere Beispiele: elementAt

- Berechnen des n-ten Elements.

```
elementAt 0     (x:xs) = x
elementAt (n+1) (x:xs) = elementAt n xs
```

```
elementAt 5 evens
```

Weitere Beispiele: sublist

- Berechnen einer Teilliste der Länge `m` beginnend bei Position `n` (analog zur `substring`-Methode in Java).

```
sublist n     m []     = []
sublist 0     m xs     = take m xs
sublist (n+1) m (x:xs) = sublist n m xs
```

```
subList 3 2 evens
```

Weitere Beispiele

```
sublist 3 2 evens        -->   [6,8]
sublist 2 2 [1,2,3]      -->   [3]
sublist 3 3 [1,2,3]      -->   []
```

Primzahlen

- In Java hatten wir bereits eine Methode zur Berechnung der Primzahlen kennen gelernt.
- Mit Hilfe von Lazy Evaluation lassen sich ähnliche Funktionen sehr elegant in Haskell hinschreiben.
- Betrachten wir zunächst den folgenden einfachen Test:

```
not_multiple x y = (mod y x) > 0
```

- Primzahlen erhalten wir nun entsprechend dem „Sieb des Eratosthenes“ dadurch, dass wir alle Vielfachen bereits gefundener Primzahlen herausfiltern:

```
sieve [] = []
sieve (x:xs) = x:sieve (filter (not_multiple x) xs)
```

- Dabei beginnen wir mit der Zahl 2:

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

Ein Anwendungsbeispiel

```
take 2 primes
```

```
sieve [] = []
sieve (x:xs) = x:sieve (filter (not_multiple x) xs)
take 0 xs = []
take n [] = []
take n (x:xs) = x:take (n-1) xs
filter p []     = []
filter p (x:xs) = if (p x) then (x:filter p xs)
                  else (filter p xs)
```

Ein Anwendungsbeispiel

```
sieve [] = []
sieve (x:xs) = x:sieve (filter (not_multiple x) xs)
```

```
   take 2 primes
⇒ take 2 (sieve [2..])
⇒ take 2 (2:sieve (filter (not_multiple 2) [3..]))
⇒ 2:take 1 sieve (filter (not_multiple 2) [3..])
⇒ 2:take 1 sieve (if (not_multiple 2 3) then (3:filter
   (not_multiple 2) [4..]) else (filter (not_multiple 2)
   [4..]))
⇒ 2:take 1 sieve (3: filter (not_multiple 2) [4..])
⇒ 2:take 1 (3: sieve filter (not_multiple 3) (filter
   (not_multiple 2) [4..]))
⇒ 2:3:take 0 sieve filter (not_multiple 3) (filter
   (not_multiple 2) [4..])
⇒ 2:3:[]
```

Programmieren mit unendlichen Objekten

- Bei der Programmierung von Funktionen muss nun bedacht werden, dass die Argumente ggf. unendlich sein können.
- Wird dieser Tatsache nicht Rechnung getragen, kann es zu unerwarteten Endlosschleifen kommen.
- Wir betrachten als Beispiel die Funktion `setOf`, die alle Duplikate aus einer Liste entfernt.
- Ein typisches Verfahren ist die Verwendung einer Tabelle in der man speichert, welche Elemente in der Liste vorkommen, wobei nur die zum ersten Mal auftretenden Elemente eingetragen werden.
- Ist die Liste leer, wird die Tabelle zurückgegeben.

Die Funktion `setOf`

- Eine Realisierung dieses Algorithmus ist

```
setOf1 xs = setOf1Aux xs []
setOf1Aux []     ys                  = ys
setOf1Aux (x:xs) ys | member x ys = setOf1Aux xs ys
setOf1Aux (x:xs) ys                  = setOf1Aux xs (x:ys)
```

- Die Funktion `setOf1` operiert korrekt auf endlichen Listen:

```
take 2 (setOf1 [2,3,2,3,4]) --> [4,3]
```

- Allerdings lässt sie sich nicht auf unendliche Listen anwenden, denn die Auswertung des Ausdrucks

```
take 1 (setOf1 ones)
```

terminiert nicht.

Eine Beispielanwendung von `setOf1`

```
setOf1 [2,3,2,3,4]
```

Eine alternative Funktion zur Berechnung der Menge

- Der Grund für die Endlosschleife liegt darin, dass das Ergebnis erst dann ausgegeben wird, wenn die Liste komplett abgearbeitet ist.
- Dies dauert bei unendlichen Listen jedoch unendlich lange.
- Eine Alternative besteht darin, neue Elemente sofort zurückzugeben und die Tabelle zu nutzen, um bereits ausgegebene Elemente zu speichern.

```
setOf2 xs = setOf2Aux xs []

setOf2Aux []     ys                = []
setOf2Aux (x:xs) ys | member x ys = setOf2Aux xs ys
setOf2Aux (x:xs) ys                = x:setOf2Aux xs (x:ys)
```

Anwendung der Methode `setOf2`

- Die Funktion `setOf2` kann auf endlichen und unendlichen Listen operieren:

```
setOf2 [2,3,2,3,4]            -->   [2,3,4]
take 1 (setOf2 ones)          -->   [1]
take 4 (setOf2 evens)         -->   [0,2,4,6]
```

- Allerdings wird auch durch Lazy Evaluation keine Terminierung garantiert. Beispielsweise terminiert die Reduktion der Ausdrücke

```
take 2 (setOf2 ones)
take 2 (setOf2 (append ones [2]))
```

auch mit Lazy Evaluation nicht.

Vor- und Nachteile der Standardstrategien

Strategie	Vorteile	Nachteile
leftmost innermost, eager evaluation	einfache Realisierung	Auswertung aller Teilausdrücke, Nichtterminieren bei unendlichen (auch nicht benötigten Teilausdrücken)
leftmost outermost lazy evaluation	bedarfsgesteuert, unendliche Datenstrukturen	aufwendige Realisierung

Typen in Haskell

- Im Gegensatz zu den meisten imperativen Programmiersprachen müssen wir **in Haskell keine Typen für Funktionen und Parameter (in Form von Prototypen) angeben**.
- **Im** Gegensatz zu imperativen und Objektorientierten Programmiersprachen **leitet Haskell den Typ einer Funktion automatisch her**.
- Betrachten wir beispielsweise die Funktion

```
identity x = x
```

- Hierfür ermittelt Haskell den folgenden Typ:

```
identity :: a -> a
```

- `identity` ist demnach eine Funktion, die ein **Element von einem beliebigen als Input** bekommt und (naturgemäß) ein **Ergebnis von demselben Typ** liefert.

Hindley-Milner Typsystem

- Den meisten funktionalen Programmiersprachen liegt das so genannte **Hindley-Milner Typsystem** zugrunde.
- Es **unterscheidet sich von Typsystemen anderer Sprachen** dadurch dass
  1. (parametrische) **Polymorphie von Funktionen und Datenstrukturen** zugelassen wird und dass
  2. die **Typinferenz entscheidbar** ist, so dass die **Typen automatisch hergeleitet werden können**.

Polymorphie

- Eine **Funktion** heißt **polymorph**, wenn sie für verschiedenartige Objekte, also **Objekte mit unterschiedlichen Typen**, **definiert ist**.
- Man unterscheidet zwischen **parametrischer** und **Ad-hoc-Polymorphie**.
- Von parametrischer Polymorphie spricht man, wenn Funktionen für eine ganze Klasse von Datenobjekten verwendet werden können.
- Beispielsweise können die Funktionen `length` und `append` für beliebige Listen (Listen von Zahlen, Listen von Listen, Zeichenketten ...) verwendet werden.
- Im Gegensatz dazu spricht man beispielsweise von **Ad-hoc-Polymorphie** im Kontext von Überladung (Overloading).
- Beispiele für **Ad-hoc-Polymorphie** sind der Additionsoperator in Java (Addition für ganze Zahlen, Fließkommazahlen oder die Verkettung von Strings).
- Hierbei steht **dasselbe Symbol für eine Reihe unterschiedliche Funktionen**, von denen je nach Typ der Argumente die passende ausgewählt wird.

Einfache polymorphe Funktionen

- Eine einfache polymorphe Funktion ist die bereits bekannte Identitätsfunktion:

  ```
  identity x = x
  ```

- Aber auch die bereits bekannten Projektionsfunktionen sind einfache Beispiele:

  ```
  first x y  = x
  second x y = y
  ```

- Diese **Funktionen sind unabhängig vom Typ der Argumente**.
- Allerdings **hängt der Ergebnistyp unmittelbar von dem (entsprechenden) Argumenttyp** ab.
- Beispielsweise ist der **Ergebnistyp von** `first` **immer identisch mit dem Typ des ersten Arguments**.
- Dies **gilt für alle Argumenttypen**.
- Also gilt für alle **Argumenttypen**, dass der **Ergebnistyp mit dem Typ des ersten Arguments übereinstimmt**.

Typvariablen

- Diese **Unabhängigkeit von dem tatsächlichen Typ** wird in Haskell **durch so genannte Typvariablen ausgedrückt**.
- Mathematisch gilt für die Funktionen `first`, `second` und `identity`:
  - $\forall \alpha\beta : \texttt{first} :: \alpha \rightarrow \beta \rightarrow \alpha$
  - $\forall \alpha\beta : \texttt{second} :: \alpha \rightarrow \beta \rightarrow \beta$
  - $\forall \alpha : \texttt{identity} :: \alpha \rightarrow \alpha$
- Haskell verwendet anstelle der griechischen Buchstaben Kleinbuchstaben.
- Darüber werden die **Allquantoren weggelassen, da alle Typvariablen allquantifiziert** sind. Demnach erhalten wir

```
first :: a -> b -> a

second :: a -> b -> b

identity :: a -> a
```

Polymorphe Datenstrukturen

- Ebenso wie Funktionen sind in Haskell auch **Datenstrukturen polymorph**.
- Beispielsweise haben die **Konstruktoren** für Listen einen **Typ**:
  - `[]` hat den Typ `[a]`
  - `(:)` hat den Typ `a->[a]->[a]`
- Dabei ist `[.]` ein **einstelliger Typkonstruktor**, d.h. dadurch dass wir `[Int]` hinschreiben charakterisieren wir den Typ „**Liste von** `Int`“.
- Listen mit Wahrheitswerten haben den Typ `[Bool]`.
- Listen von Listen von Zahlen haben den Typ `[[Int]]`.

Rangalphabet

- Ein **Rangalphabet** ist ein Paar (Σ,σ) wobei
  - $\Sigma = \{F_1, ..., F_m\}$ eine endliche Menge von **Operationssymbolen** und
  - $\sigma:\Sigma\rightarrow N$ (natürliche Zahlen incl. 0) eine Funktion ist, die jedem Operationssymbol F in Σ seine Stelligkeit $\sigma(F)$ zuordnet.
- Dabei ist `[.]` ein **einstelliger Typkonstruktor**, d.h. dadurch dass wir `[Int]` hinschreiben charakterisieren wir den Typ „**Liste von** `Int`“.
- Listen mit Wahrheitswerten haben den Typ `[Bool]`.
- Listen von Listen von Zahlen haben den Typ `[[Int]]`.

Schreibweisen und Beispiele

- $\Sigma^{(n)}=\{F\in \Sigma \mid \sigma(F) = n\}$ ist die Menge aller n-stelligen Operationssymbole in einem Rangalphabet (Σ, σ).
- $F^{(n)} \in \Sigma$ ist gleichbedeutend mit $F\in \Sigma^{(n)}$, wobei F ein n-stelliges Operationssymbol ist.
- Ein Rangalphabet für Zahlen mit Rechenoperationen ist
  - $F^{(0)}=\{0\}$
  - $F^{(1)}=\{succ,pred\}$ (einstellige Operationssymbole)
  - $F^{(2)}=\{+,-,*\}$
- Mögliche Ausdrücke für dieses Rangalphabet sind

```
+ succ 0 succ succ 0
```

```
+ * succ succ 0 succ succ 0 succ 0
```

Menge der polymorphen Typen

Definition: Sei $TVar = \{\alpha, \beta, \ldots\}$ eine abzählbare Menge von Typvariablen und $\Theta = \bigcup_{n \in N} \Theta^n$ ein Rangalphabet von Typkonstruktoren. Ein Typkonstruktor $\vartheta \in \Theta^n$ habe die Stelligkeit n. Die Menge Θ^0 der nullstelligen Typkonstruktoren enthalte die Menge der Basistypen $T_0 =$ `{Int, Bool, Char, ...}`.

Die Menge $Typ_\Theta(TVar)$ der **polymorphen Typen** über Θ und $TVar$ ist die kleinste Menge Typ, für die gilt:

1. $TVar \subseteq Typ$
2. mit $t_1, t_2 \in Typ$ ist auch $(t_1 \rightarrow t_2) \in Typ$
3. falls $\vartheta \in \Theta^n$ und $t_1, \ldots, t_n \in Typ$, so gilt auch $(\vartheta t_1 \ldots t_n) \in Typ$

Während 1) die **Typvariablen** festlegt und 2) die **Funktionstypgen** einführt, werden in 3) die **Strukturtypen** definiert.

Typinferenz

- Unter **Typinferenz** versteht man die **Bestimmung von Typen zu ungetypten Ausdrücken**.

- Im **Hindley-Milner-Typsystem** kann man **für jeden typisierbaren Ausdruck bei der Übersetzung des Programms** einen (bis auf Umbenennung von Typvariablen eindeutigen) **allgemeinsten Typ bestimmen**.

- Die **wesentlichen Vorteile** liegen darin, dass man

  1. **Laufzeitfehler vermeiden** und

  2. **Programmierfehler** (die sich durch Typfehler äußern) **frühzeitig erkennen kann**.

„well typed programs do not go wrong (due to type violations)“

Prinzipielle Vorgehensweise

1. Zuerst werden die **linken Seiten der Funktionsdefinitionen analysiert.** Dabei werden **Annahmen über die Typen der Funktionen, der Argumente und der Resultate erstellt**.

2. Dann folgt die **Analyse der rechten Seiten der Funktionsdefinitionen unter Verwendung der bereits gemachten Annahmen**. Gleichzeitig werden **Kompatibilitätsgleichungen der folgenden Art erstellt**:

    1. Der **Resultattyp** der Funktion **muss mit dem Typ des Ausdrucks auf der rechten Seite** der Funktionsdefinition **übereinstimmen**.

    2. In jeder **Funktionsanwendung** müssen die **Funktionen gemäß ihrem Typ verwendet werden**.

3. Schließlich werden die **Kompatibilitätsgleichungen gelöst**. Dabei wird der **Unifikationsalgorithmus von Robinson** angewendet.

Analyse der linken Seiten von Funktionsgleichungen

- Im **allgemeinen Fall** hat eine **Funktionsdefinition die folgende Struktur**:

  $$\texttt{f}\ \texttt{t}_1\ \ldots\ \texttt{t}_n = \texttt{e}$$

- Der **Typ von `f`** hat demnach die **allgemeine Form**

  $$\alpha_1 \to \alpha_2 \to \ldots \to \alpha_n \to \alpha_{n+1}$$

- Die **Parameterterme** $\texttt{t}_\texttt{i}$ können **Variablen oder Konstruktorapplikationen** sein.

- Da der **Typ eines Konstruktors stets in einer Typdeklaration vordefiniert** ist, kann der **Typ von `f` im Fall einer Konstruktoranwendung präzisiert** werden.

Beispiele

- Tritt `[]` oder `(x:xs)` als Parameterterm auf, so folgt, dass das entsprechende Argument vom Typ $[\alpha]$ sein muss.
- Der Parameterterm `[[]]` hingegen lässt auf $[[\alpha]]$ schließen.
- Natürlich sind in den verschiedenen definierenden Regeln für ein Funktionssymbol `f` in den einzelnen Parameterpositionen nur Terme desselben Typs zulässig.
- Betrachten wir erneut die Funktion `map`:

```
map f []          = []
map f (x:xs)      = (f x): map f xs
```

- Wir erhalten den Typ $\alpha_1 \rightarrow [\alpha_2] \rightarrow \alpha_3$.
- Gleichzeitig werden die folgenden Annahmen über die Typen der in den Gleichungen auftretenden Bezeichner festgehalten:

Bezeichner	`f`	`x`	`xs`	Rumpf
Typ	α_1	α_2	$[\alpha_2]$	α_3

Analyse der rechten Seiten von Funktionsgleichungen

- Bei der Analyse der Ausdrücke auf der rechten Seite werden die **Typannahmen aus der Analyse der linken Seiten mit verwendet**.
- **Für jeden Anwendungsausdruck** der Form `(e`$_1$ `e`$_2$`)` wird **eine Gleichung** der folgenden Form aufgestellt:

  $$typ(e_1) = typ(e_2) \to typ((e_1\ e_2))$$

- Aufgrund der Möglichkeit der partiellen Applikation können wir **mehrstellige Funktionen** auch als **Mehrfachanwendung von einstelligen Funktionen** (**Currying**) auffassen

  `(...(e`$_1$ `e`$_2$`) e`$_3$`)...) e`$_n$`).`

- Taucht ein **Ausdruck** *e* **bereits im Kopf** auf, so wird *typ(e)* **ersetzt durch den aus der Analyse der linken Seite hervorgegangenen Typ**.

Beispiel

- Die **Analyse der rechten Seite der ersten Gleichung** von

```
map f []          = []
map f (x:xs)      = (f x): map f xs
```

Bezeichner	`f`	`x`	`xs`	Rumpf
Typ	α_1	α_2	$[\alpha_2]$	α_3

ergibt, dass der **Ergebnistyp** von map, α_3 laut Annnahme bei der Typanalyse der linken Seiten, ein **Listentyp** sein muss.

- Also **schließen** wir

$$\alpha_3 = [\alpha_4]$$

Analyse der zweiten Gleichung

- Die **Analyse der rechten Seite der zweiten Gleichung**

```
map f (x:xs)      = (f x): map f xs
```

liefert gegeben

Bezeichner	f	x	xs	Rumpf
Typ	α_1	α_2	$[\alpha_2]$	α_3

folgende Gleichungen:

Ausdruck	Typgleichung
`(f x)`	$\alpha_1 = \alpha_2 \to \alpha_4$
`(map f)`	$\alpha_5 = \alpha_1 \to \alpha_6$
`((map f) xs)`	$\alpha_5 = \alpha_1 \to [\alpha_2] \to \alpha_7$
`(f x):((map f) xs))`	$\alpha_8 \to [\alpha_8] \to [\alpha_8] = \alpha_4 \to \alpha_7 \to \alpha_3$

Lösen der Typgleichungen

- Die **Lösung einer Menge von Typgleichungen**

$$\left\langle t_i = \tilde{t}_i \mid t_i, \tilde{t}_i \in Typ_\Theta(TVar), 1 \leq i \leq n \right\rangle$$

ist eine **Substitution** (Ersetzung) von **Typvariablen durch Typen oder andere Typvariablen**, also eine Abbildung

$$\sigma : TVar \rightarrow Typ_\Theta(TVar)$$

die alle Gleichungen erfüllt, d.h.

$$\hat{\sigma}(t_i) = \hat{\sigma}(\tilde{t}_i) \quad \forall i = 1 \ldots n$$

wobei

$$\hat{\sigma} : Typ_\Theta(TVar) \rightarrow Typ_\Theta(TVar)$$

die **Fortsetzung von σ auf die Typterme** $Typ_\Theta(TVar)$ ist.

Eigenschaften von $\hat{\sigma}$

$$\hat{\sigma}(\alpha) = \sigma(\alpha) \quad \alpha \in TVar$$

$$\hat{\sigma}(t_1 \to t_2) = \hat{\sigma}(t_1) \to \hat{\sigma}(t_2) \quad Funktionstypen$$

$$\hat{\sigma}((\vartheta t_1 \ldots t_n)) = (\vartheta \hat{\sigma}(t_1) \ldots \hat{\sigma}(t_n)) \quad Strukturtypen$$

- Statt $\sigma(t)$ schreibt man meist $t\sigma$.
- Die Hintereinanderausführung von Substitutionen wird durch $\sigma_1\sigma_2$ notiert.
- **Üblicherweise** existieren für eine Menge von Typgleichungen **mehrere Lösungen** (so genannte **Unifikatoren**).
- Dabei gibt es immer eine (bis auf Variablenumbenennung) **allgemeinste Lösung**.
- **Alle anderen entstehen aus dieser allgemeinsten durch Anwendung einer weiteren Substitution**.

Unifikatoren

Definition: Eine **Substitution σ** heißt **Unifikator** einer Menge von Typgleichungen

$$\left\langle t_i = \tilde{t}_i \mid t_i, \tilde{t}_i \in Typ_\Theta(TVar), 1 \leq i \leq n \right\rangle$$

falls

$$t_i\sigma = \tilde{t}_i\sigma \quad \forall i = 1 \ldots n$$

Ein **Unifikator σ** heißt **allgemeinster Unifikator**, falls für jeden anderen Unifikator σ' gilt, dass ein Substition ρ existiert, mit

$$\sigma' = \sigma\rho = \hat{\rho} \circ \sigma$$

Satz von Robinson

Satz: Für jede Menge von Typgleichungen

$$\left\langle t_i = \tilde{t}_i \mid t_i, \tilde{t}_i \in Typ_\Theta(TVar), 1 \leq i \leq n \right\rangle$$

existiert ein (bis auf Umbenennung von Variablen) eindeutiger allgemeinster Unifikator.

Der allgemeinste Unifikator lässt sich mit Hilfe des Unifkationsalgorithmus von Robinson berechnen.

Unifikationsalgorithmus von Robinson

$$
\begin{array}{l}
\mathit{unify} : \mathit{Typ}_\Theta(\mathit{TVar}) \times \mathit{Typ}_\Theta(\mathit{TVar}) \to [\mathit{TVar} \to \mathit{Typ}_\Theta(\mathit{TVar})] \cup \{\text{fail}\} \\
\mathit{unify}\ (\tau_1, \tau_2) \\
\quad = \textbf{if } \tau_1 \in \mathit{TVar} \wedge \tau_1 \text{ kommt nicht in } \tau_2 \text{ vor (sog. „occur check“) } \textbf{then } \{\tau_1 \mapsto \tau_2\} \\
\qquad \textbf{else if } \tau_2 \in \mathit{TVar} \wedge \tau_2 \text{ kommt nicht in } \tau_1 \text{ vor } \textbf{then } \{\tau_2 \mapsto \tau_1\} \\
\qquad \textbf{else if } (\tau_1 \in T_0 \vee \tau_2 \in T_0) \wedge \tau_1 = \tau_2 \textbf{ then } \{\} \\
\qquad \textbf{else if } \tau_1 = \tau_{11} \to \tau_{12} \wedge \tau_2 = \tau_{21} \to \tau_{22} \textbf{ then } \mathit{unifyList}([\tau_{11}, \tau_{12}], [\tau_{21}, \tau_{22}], \{\ \}) \\
\qquad \textbf{else if } \tau_1 = (\vartheta\ \tau_{11} \ldots \tau_{1n}) \wedge \tau_2 = (\vartheta\ \tau_{21} \ldots \tau_{2n}) \\
\qquad\qquad \textbf{then } \mathit{unifyList}([\tau_{11}, \ldots, \tau_{1n}], [\tau_{21}, \ldots, \tau_{2n}], \{\ \}) \\
\qquad \textbf{else } \text{fail} \\
\\
\mathit{unifyList} : [\mathit{Typ}_\Theta(\mathit{TVar})] \times [\mathit{Typ}_\Theta(\mathit{TVar})] \times ([\mathit{TVar} \to \mathit{Typ}_\Theta(\mathit{TVar})] \cup \{\text{fail}\}) \\
\hfill \to [\mathit{TVar} \to \mathit{Typ}_\Theta(\mathit{TVar})] \cup \{\text{fail}\} \\
\mathit{unifyList}\ ([\,], [\,], \sigma) = \sigma \\
\mathit{unifyList}\ ((\tau_1 : ts1), (\tau_2 : ts2), \sigma) \\
\quad = \textbf{let } \sigma_1 = \mathit{unify}(\tau_1\sigma, \tau_2\sigma) \textbf{ in} \\
\qquad \textbf{if } \sigma_1 = \text{fail } \textbf{then } \text{fail } \textbf{else } \mathit{unifyList}\ (ts1, ts2, \sigma\sigma_1)
\end{array}
$$

Anwendung auf die Gleichungen für `map`

Gleichungen:

$\alpha_1 = \alpha_2 \rightarrow \alpha_4$

$\alpha_5 = \alpha_1 \rightarrow \alpha_6$

$\alpha_5 = \alpha_1 \rightarrow [\alpha_2] \rightarrow \alpha_7$

$\alpha_8 \rightarrow [\alpha_8] \rightarrow [\alpha_8] = \alpha_4 \rightarrow \alpha_7 \rightarrow \alpha_3$

Ergebnis:

$\alpha_1 = \alpha_2 \rightarrow \alpha_4$

$\alpha_3 = [\alpha_8] = [\alpha_4]$

$\alpha_5 = \alpha_1 \rightarrow [\alpha_2] \rightarrow [\alpha_4]$

$\alpha_6 = [\alpha_2] \rightarrow [\alpha_4]$

$\alpha'_7 = [\alpha_4]$

$\alpha_8 = \alpha_4$

Resultierender Typ von `map`:

`map::` $(a_2 \rightarrow a_4) \rightarrow [a_2] \rightarrow [a_4]$

Zusammenfassung

- Haskell ist eine **funktionale Programmiersprache**.
- Eine wichtige Eigenschaft funktionaler Programmiersprachen ist, dass es **keine Seiteneffekte** gibt.
- Darüber hinaus gibt es **nicht das Konzept von Speicherzellen**, in denen Werte (Zustände) abgelegt werden.
- Haskell verwendet verzögerte Auswertung oder **Lazy Evaluation**.
- Dies wird durch **Leftmost Outermost Reduction** erreicht.
- Dabei werden **Ausdrücke nur so weit ausgewertet, wie es benötigt wird**.
- Wegen der Lazy Evaluation kann man auch mit **unendlichen Objekten** rechnen.
- Typen werden in Haskell **automatisch hergeleitet** .

THE END