2 Funktionale Programmierung

In diesem Kapitel werden wir zunächst wichtige Aspekte rein funktionaler Programmiersprachen betrachten.

2.1 Funktionsdefinitionen

Grundidee: Programm	=	Menge von Funktionsdefinitionen (+ auszuwertender Ausdruck)
Funktionsdefinition:		Funktionsname
	+	formale Parameter
	+	Rumpf (Ausdruck)
Allgemein:		$f\ x_1 \ldots x_n \ = \ e$

Ausdrücke, die im Rumpf einer Funktionsdefinition vorkommen, sind wie folgt aufgebaut:

- konstante Ausdrücke: Zahlen `3`, `3.14159` (später mehr)
- Anwendung elementarer Funktionen: `3+4`, `1+5*7`
- allgemeine Funktionsanwendungen: $f \quad a_1 \ \ldots \ a_n$ (f ↑ *Funktion*; $a_1 \ldots a_n$ ↖ ↗ *aktuelle Parameter*)
- bedingte Ausdrücke: `if` b `then` e_1 `else` e_2

(diese Liste wird später noch erweitert)

Beispiel: Quadratfunktion:

```
square x = x*x
```

“`=`” steht für Gleichheit von linker und rechter Seite, somit können wir wie folgt „rechnen“:

```
square 9   =   9*9   =   81
```

Rechnen bedeutet in deklarativen Programmiersprachen, dass Gleiches durch Gleiches ersetzt wird, bis ein Wert herauskommt. Wie funktioniert dies aber genau, wenn man mit selbst definierten Funktionen rechnet?

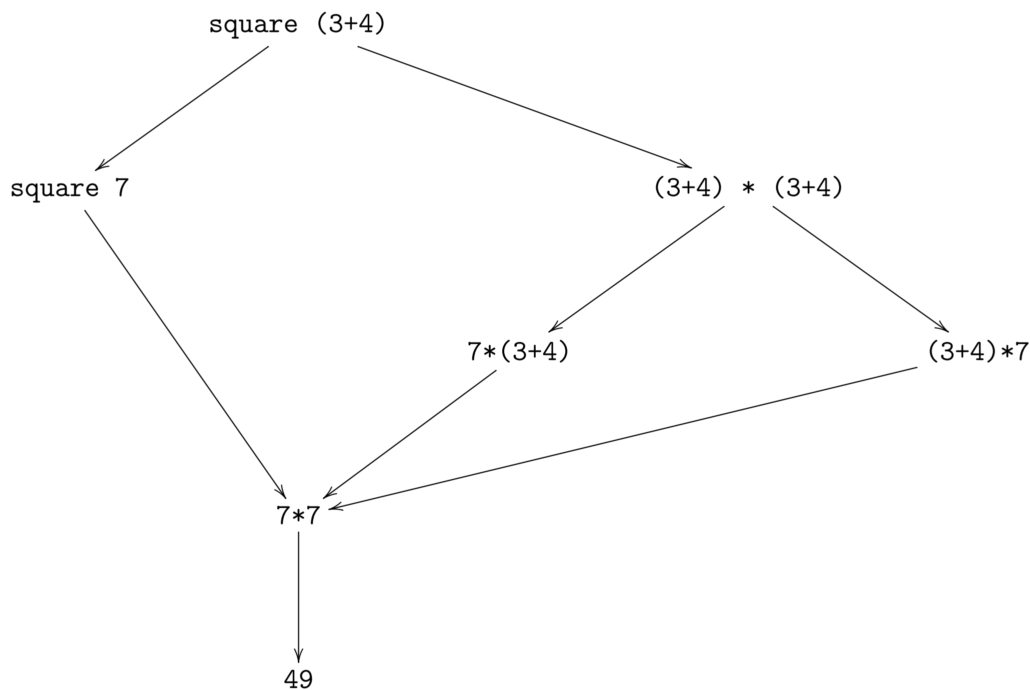
Auswerten von Ausdrücken:
Falls Ausdruck ein Aufruf einer elementaren Funktion (z.B. “`+`”) ist und alle Argumente

ausgewertet sind: Ersetze den Ausdruck durch den berechneten Wert
Sonst:

1. Suche im Ausdruck eine Funktionsanwendung (auch: **Redex**: **red**ucible **ex**pression)
2. Substituiere in entsprechender Funktionsdefinition formale Parameter durch aktuelle (auch im Rumpf!)
3. Ersetze Teilausdruck durch Rumpf

Wegen 2: *Variablen* in Funktionsdefinitionen sind *unbekannte Werte*, aber keine Speicherzellen!
(Unterschied: imperativ $\leftrightarrow$ deklarativ)

Beispiel: Auswertungsmöglichkeiten von `square (3+4)`
(bei Anwendung elementarer Funktionen: werte vorher Argumente aus)



Jede konkrete Programmiersprache folgt einer bestimmten **Auswertungsstrategie** (vgl. Punkt 1 in obiger Beschreibung). Daher können wir folgende grobe Klassifizierung funktionaler Programmiersprachen vornehmen:

Strikte Sprachen: Wähle immer linken inneren Teilausdruck, der Redex ist (Bsp.: linker Pfad)
(*leftmost innermost, call-by-value, application order reduction, eager reduction*)

+ einfache effiziente Implementierung

– berechnet evtl. keinen Wert, obwohl ein Ergebnis existiert (bei anderer Auswertung)

Nicht-strikte Sprachen Wähle immer linken äußeren Teilausdruck, der Redex ist (Bsp.: mittlerer Pfad)
(*leftmost outermost, call-by-name, normal order reduction*)
Besonderheit: Argumente sind evtl. unausgewertete Ausdrücke (nicht bei elementaren Funktionen)

+ berechnet Wert, falls einer existiert
+ vermeidet überflüssige Berechnungen
– evtl. Mehrfachauswertung (z.B. `(3+4)`)
  ⇒ Verbesserung durch *verzögerte/faule Auswertung* (*call-by-need*, *lazy evaluation*)

Beispiele funktionaler Programmiersprachen (nicht alle sind rein funktional!):

- LISP, Scheme, SML: strikte Sprachen
- Haskell: nicht-strikte Sprache (diese werden wir nachfolgend betrachten)

Fallunterscheidung: Für viele Funktionen notwendig, z.B.

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

Implementierung mit if-then-else (Beachte: `abs` vordefiniert!):

```
absif x = if x >= 0 then x else -x
```

Aufruf: `absif (-5)` $\rightsquigarrow$ `5`

Implementierung mit *bedingten Gleichungen* (guarded rules)

```
absg x | x >= 0     = x
       | otherwise  = -x
```

Bedeutung der Bedingungen:
Erste Gleichung mit erfüllbarer Bedingung wird angewendet (`otherwise` $\approx$ `True`)
Diese Schreibweise entspricht daher sehr stark der mathematischen Notation.

Wir erläutern die unterschiedlichen Möglichkeiten zur Definition einer Funktion mit Fallunterscheidungen an Hand der **Fakultätsfunktion**: Generell ist diese wie folgt definiert:

$$fac(n) \rightsquigarrow n * (n-1) * \cdots * 2 * 1$$

Rekursive Definition:

$$fac(n) = \begin{cases} 1 & \text{falls } n = 0 \\ n * fac(n-1) & \text{falls } n > 0 \end{cases}$$

Implementierung:

1. Mit `if-then-else`:

```
fac n = if n==0 then 1
                else n * fac (n-1)
```

   Nicht ganz korrekt, weil ein rekursiver Aufruf auch bei `n<0` möglich wäre und dies dann zu einer endlosen Berechnung führt.

2. Mit bedingten Gleichungen:

```
fac n | n==0  = 1
      | n>0   = n * fac (n-1)
```

   Diese Definition ist korrekt und präzise. Bei komplexeren Funktionen mit mehreren Argumenten kann die verwendung bedingter Gleichungen aber auch umständlich werden.

3. **Muster** in formalen Parametern (*pattern matching*):
   Bisher waren die formalen Parameter einer Funktionsdefinition nur Variablen. Nun lassen wir auch „Muster" zu. Intuitiv ist eine Gleichung nur anwendbar, wenn der aktuelle Parameter zum Muster „passt".

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
```

   Hierbei ist `(n+1)` ein Muster für alle positiven Zahlen. Allgemein steht das Muster `(n+`k`)` für ganze Zahlen $\geq k$, wobei k eine positive ganze Zahl sein muss.

   Da dieses Muster später selten gebraucht wird und eher nur für die Einführung in die musterorientierte Programmierung relevant ist, wird es standardmäßig durch den Glasgow Haskell Compiler (GHC) nicht mehr unterstützt. Um es zu benutzen, muss man den GHC mit der Option "`-XNPlusKPatterns`" aufrufen oder zu Beginn des Quellprogramms das Pragma

```
{-# LANGUAGE NPlusKPatterns #-}
```

   angeben.

   Beispielberechnung:

```
fac 3                  -- 3 passt auf Muster n+1 für n=2 ⇝
 = (2+1) * fac(2)
 = 3 * fac(2)          -- 2 passt auf Muster n+1 für n=1 ⇝
 = 3 * (1+1) * fac (1)
 = 3 * 2 * fac (1)
 = 3 * 2 * (1+0) * fac (0)
 = 3 * 2 * 1 * 1   ⇝ 6
```

Musterorientierter Stil:

- kürzere Definitionen (kein if-then-else)
- leichter lesbar
- leichter verifizierbar (vgl. `append`, Kapitel 1, S. 10)

Vergleiche hierzu die prägnante Definition

```
and True  x = x
and False x = False
```

mit der „if-then-else"-Definition

```
and x y = if x==True then y else False
```

oder auch

```
and x y | x==True  = y
        | x==False = False
```

Der musterorientierte Stil kann allerdings auch problematische Fälle haben, deren Bedeutung erklärt werden muss:

- Wie ist die Reihenfolge bei einer Musteranpassung?
  (links-nach-rechts, oben-nach-unten, best-fit?)
- Deklarativ sinnlose Definitionen möglich:

  ```
  f True  = 0
  f False = 1
  f True  = 2
  ```

  Die letzte Gleichung ist aus funktionaler Sicht sinnlos, da die erste Gleichung schon vorschreibt, dass der Wert der Funktion `f` angewendet auf `True` immer `0` sein soll.

Spezifikationssprache ↔ deklarative Programmiersprache

Die **deklarative Programmierung** basiert auf der Definition von Eigenschaften statt der Angabe des genauen Programmablaufs. Hieraus könnte man folgern, dass Spezifikationen identisch mit deklarativen Programmen sind.
Dies ist allerdings nicht der Fall, denn ein Programm ist effektiv ausführbar (z.B. Reduktion von Ausdrücken), während eine Spezifikation eine formale Problembeschreibung ist, die evtl. nicht ausführbar ist.

Da **Programmieren** als Überführung einer Spezifikation in eine ausführbare Form aufgefasst werden kann, kann die Programmierung allerdings durch deklarative Sprachen vereinfacht werden.

Beispiel: **Quadratwurzelberechnung nach Newton**

Wurzelfunktion: $\sqrt{x} = y$, sodass $y \geq 0$ und $y^2 = x$

Dies ist eine *Spezifikation*, die aber nicht effektiv ausführbar ist, denn wie sollen wir einen passenden Wert für y raten?

Ein konstruktives Approximationsverfahren ist das *Newtonsche Iterationsverfahren*:
Gegeben: Schätzung für y
bessere Schätzung: Mittelwert von y und $\frac{x}{y}$

Beispiel:

$x = 2$	$Schätzung:$	$Mittelwert:$
	1	$\frac{1+2}{2} = 1.5$
	1.5	$\frac{1.5+1.3333}{2} = 1.4167$
	1.4167	$\frac{1.4167+1.4118}{2} = 1.4142$

Funktionale Definition: Iteration mit Abbruchbedingung:

```
iter y x = if (ok y x) then y else iter (improve y x) x
              Abbruch                   Verbesserung
```

(Unterklammerung: `(ok y x)` = *Abbruch*, `(improve y x)` = *Verbesserung*)

```
improve y x = (y + x/y) / 2

ok y x = abs (y*y-x) < 0.001
```

Hierbei berechnet `abs` den Absolutbetrag einer Zahl. Damit erhalten wir als Gesamtlösung:

```
sqroot x = iter 1 x

> sqroot 9  ⇝ 3.00009
```

Nachteil dieser Lösung:

- `sqroot`, `iter`, `improve`, `ok` gleichberechtigte (globale) Funktionen
- Argument `x` muss „durchgereicht“ werden
- mögliche Konflikte bei Namen wie `ok`, `improve`

Eine aus Software-technischer Sicht verbesserte Definition erhalten wir durch Benutzung einer **where-Klausel**:

```
sqroot x = iter 1
  where iter y    = if ok y then y else iter (improve y)
        improve y = (y + x/y) / 2
        ok y      = abs (y*y-x) < 0.001
```

Hierbei sind die Deklarationen nach dem Schlüsselwort `where` *lokale Deklarationen*, die nur in der Gleichung für `sqroot` sichtbar sind, d.h. der **Gültigkeitsbereich** der `where`-Deklarationen ist nur die vorhergehende rechte Gleichungsseite (bzw. die Folge von bedingten Gleichungsseiten).
Weitere Form lokaler Deklarationen: **let-Ausdrücke**

`let` <Deklarationen> `in` <Ausdruck>

Die Deklarationen sind hierbei nur im Ausdruck sichtbar.

Beispiel: Definition der Funktion

$$f(x, y) = y(1 - y) + (1 + xy)(1 - y) + xy$$

in Haskell mittels `let`-Deklarationen:

```
f x y = let a = 1-y
            b = x*y
        in  y*a+(1+b)*a+b
```

Unklar ist allerdings bei der Verwendung lokaler Deklarationen, wann diese enden. Daher gibt es in Haskell folgende Festlegungen:

Formatfreie Schreibweise durch *Klammerung* lokaler Deklarationen:

```
let {...;...} in ...

f x = e where {...;...}
```

Formatabhängige Schreibweise durch die „*Abseitsregel*“ (*offside rule*, *layout rule*):

1. Erstes Symbol nach `let`, `where` (und `of` bei case-Ausdrücken) legt den *linken Rand* des Geltungsbereiches fest.
2. Beginnt neue Zeile rechts vom Rand ⇒ Fortsetzung der vorherigen Zeile
3. Beginnt neue Zeile genau am Rand ⇒ neue Deklaration
4. Beginnt neue Zeile links vom Rand ⇒ Geltungsbereich endet davor, Zeile gehört zum umfassenden Bereich

```
f x = e        ___________
  where g y = |           |
          ____|           |
         |                |
         |                |
         |________________|
        h y = |           |
```

```
          ____|            |
         |                 |
         |                 |
         |_________________|
          _________________
k x y = |                  |
  ______|                  |
 |                         |
 |                         |
 |_________________________|
```

- gewöhnungsbedürftig, aber:
- erlaubt kompakte, übersichtliche Schreibweise

2.2 Datentypen

Moderne funktionale Sprachen haben ein *strenges Typkonzept*, d.h.

- alle Werte sind klassifiziert in *Typbereiche* (boolesche Werte, Zeichen, ganze Zahlen, ...)
- jeder zulässige Ausdruck hat einen Typ: falls der Ausdruck zu einem Wert ausgerechnet wird, dann gehört dieser zur Wertemenge, die durch den Typen spezifiziert wird

Wir schreiben

e :: τ

um auszudrücken, dass der Ausdruck e den Typ τ hat. Nachfolgend betrachten wir kurz einige Standardtypen, die in Haskell vorhanden sind.

Basistypen

Wahrheitswerte: `Bool`
Konstanten: `True`, `False`
Funktionen: `&&` (und) `||` (oder) `not` (nicht)

Ganze Zahlen: `Int` (auch `Integer` für Zahlen beliebiger Größe)
Konstanten: `0, 1, -42, ...`
Funktionen: `+, -, *, /, div, mod, ...`

Gleitkommazahlen: `Float` (auch `Double` für größere Präzision)
Konstanten: `0.3 1.5 e-2 ...`

Ganze Zahlen und Gleitkommazahlen sind verschiedene Objekte und es erfolgt bei einem strengen Typsystem keine automatische Konversion zwischen `Int` und `Float`. Haskell hat aber ein mächtiges Typkonzept mit Überladung (Typklassen), das es erlaubt, dass man ganze Zahlkonstanten auch hinschreiben kann, wo Gleitkommazahlen erwartet werden. Allerdings ist ein Ausdruck wie “`div 4 2.0`” unzulässig.

Zeichen: `Char`
Konstanten: `'a' '0' '\n'`

Strukturierte Typen:

Listen (Folgen) von Elementen von Typ `t`: Der Listentyp wird wie folgt notiert: `[t]`

Die Elemente des Listentyps sind wie folgt definiert: Eine Liste ist

- entweder leer, d.h. die Konstante `[]`

- oder nicht-leer, d.h. sie besteht aus einem (Kopf-) Element `x` und einer Restliste `xs`, was wie folgt notiert wird: `x:xs`

Beispiel: Die Liste mit den Elementen `1,2,3` kann wie folgt aufgeschrieben werden:

```
1:(2:(3:[])) :: [Int]
```

Anmerkungen zur Notation von Listen:

1. “`:`” ist rechtsassoziativ: Somit können wir Klammern vermeiden:

   ```
   1 : 2:  3 : []
   ```

2. Wenn man alle Elemente kennt, kann man diese auch direkt aufzählen: `[1, 2, 3]`
   Somit:

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

3. Abkürzungen für Listen von Zahlen:

   ```
   [a..b]   == [a, a+1, a+2, ..., b]

   [a..]    == [a, a+1, a+2, ...        -- Unendlich!

   [a,b..c] == [a, a+s, a+2*s, ...,c]   -- mit Schrittweite s=b-a

   [a,b..]  == [a, a+s, a+2*s, ...      -- mit Schrittweite s=b-a
   ```

Operationen auf Listen (viele vordefiniert in der `Prelude`)

`length xs`: Berechne die Länge der Liste `xs`

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

“`++`”: Konkatenation zweier Listen (rechtsassoziativ)

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

Strings, Zeichenketten: Der Typ `String` ist identisch zu `[Char]`
Konstanten: `"Hallo" == ['H','a','l','l','o']`
Funktion: wie auf Listen

Tupel: $(t_1, t_2, \ldots, t_n)$: Struktur („Record“, kartesisches Produkt) mit Komponenten der Typen $t_1, t_2, \ldots, t_n$
Konstanten: `(1,'a',True) :: (Int,Char,Bool)`

Funktionale Typen: Der Typ “t_1->t_2” hat als Elemente Funktionen mit Argumenten vom Typ t_1 und Ergebnissen vom Typ t_2

Beispiel:

```
square :: Int → Int
```

Konstante: \ *x* -> *e*

\	*x* ->	*e*
↑	↑	↑
λ(*lambda*)	*Parameter*	*Rumpf*

äquivalent zu $f\ x = e$, falls f Name dieser Funktion

„λ-Abstraktion“: Definiert „namenlose“ Funktion

Beispiel:

```
\x → 3*x+4 :: Int → Int
```

Besonderheit funktionaler Sprachen:
Funktionale Typen haben gleichen Stellenwert wie andere Typen („Funktionen sind Bürger 1. Klasse“, „functions are first class citizens“), d.h. sie können vorkommen in

- Argumenten von Funktionen
- Ergebnissen von Funktionen
- Tupeln
- Listen

⋮

Falls Funktionen in Argumenten oder Ergebnissen anderer Funktionen vorkommen, nennt man diese auch **Funktionen höherer Ordnung**

```
polynomdiff:: (Float → Float) → (Float → Float)
```

Die **Anwendung (Applikation)** einer Funktion f auf Argument x wird in der Mathematik üblicherweise durch Klammerung geschieben, d.h. in der Form $f(x)$.
In Haskell lassen wir dagegen die Klammern weg: $f\ x$ (Anwendung durch Hintereinanderschreiben)

Funktionen mit mehreren Argumenten:
Funktionale Typen haben nur ein Argument. Wenn wir Funktionen mit mehreren Argumenten definieren wollen, gibt es zwei Möglichkeiten:

1. Mehrere Argumente als Tupel:

```
\ (x,y) → x+y+2 :: (Int,Int) → Int

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

2. „Curryfizierung“ (nach Schönfinkel und Curry)
   - Tupel sind kartesische Produkte: $(A, B) \hat{=} A \times B$
   - Beobachtung:
     Funktionsräume $(A \times B) \to C$ und $A \to (B \to C)$ sind isomorph
     Dies bedeutet, dass für jede Funktion $f : (A \times B) \to C$ auch eine äquivalente Funktion $f' :: A \to (B \to C)$ existiert.

   Beispiel:

```
add':: Int → (Int → Int)
add' x = \y → x+y

add (x,y) ⇝ x+y
(add' x) y ⇝ (\y →x+y) y ⇝ x+y
```

   Also: Statt Tupel von Argumenten Hintereinanderanwendung der Argumente
   Interessante Anwendung bei partieller Applikation:

   `(add 1)`: Funktion, die eins zu ihrem Argument addiert ≈ Inkrementfunktion
   ⇒ `(add 1) 3 = 4`

Die Benutzung "curryfizierter" Funktionen erlaubt die flexiblere Anwendungen von Funktionen mit mehreren Argumenten (vgl. generische Funktionen, Funktionen höherer Ordnung in Kapitel 2.4). Aus diesem Grund ist dies der übliche Weg in Haskell, Funktionen zu definieren.

$f\ x_1\ x_2 \ldots x_n \hat{=} (\ldots((f\ x_1)\ x_2) \ldots x_n)$ (Applikation linksassoziativ)

$t_1 \to t_2 \to \ldots \to t_n \hat{=} t_1 \to (t_2 \to (\ldots \to t_n) \ldots)$ (Funktionstyp rechtsassoziativ)

Daher:

```
add :: Int → Int → Int   =  Int → (Int → Int)
add x y = x+y
⋮
add 1 2 ⇝ ((  add 1  ) 2)
             ︸
           ::Int→Int
```

Achtung: "`add 1 2`" ist verschieden von `add (1,2)`, denn:

- Auf der linken Seite muss `add` den Typ `Int -> Int -> Int` haben.

- Auf der rechten Seite muss `add` den Typ `(Int, Int) -> Int` haben.

Aber: es gibt Umwandlungsfunktionen zwischen der Tupeldarstellung und der curryfizierten Darstellung (`curry`, `uncurry`).

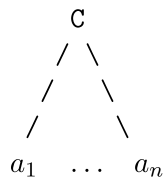
Benutzerdefinierte (algebraische) Datentypen:

Man kann in Haskell auch eigene Datentypen definieren. Die Definition eigener Datentypen basiert auf der Erkenntnis, dass jedes Objekt eines bestimmten Typs immer aufgebaut ist aus **Konstruktoren**, d.h. aus Funktionen mit dem entsprechenden Ergebnistyp, die nicht reduzierbar sind (frei interpretiert, keine auswertbaren Funktionsdefinition).

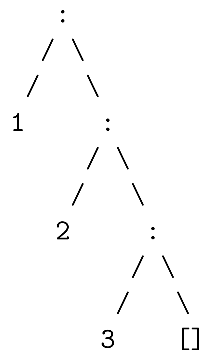
Beispiel:

Typ	Konstruktoren
`Bool`	`True False`
`Int`	`0 1 2...-3...`
Listen:	`[] :`

Konstruktion (`C` $a_1 \ldots a_n$) entspricht Baumstruktur:



Beispiel: Liste `1:2:3:[]` entspricht Baumstruktur:



Dagegen ist `++` eine Funktion, die Listen (Bäume) verarbeitet: `[1] ++ [2]` $\rightsquigarrow$ `[1,2]`

Konvention in Haskell:

- Konstruktoren beginnen mit Großbuchstaben
- Funktionen beginnen mit Kleinbuchstaben

Damit erfolgt die Definition neuer Datentypen durch Festlegung der Konstruktoren hierfür:

Datendefinition in Haskell:

$\texttt{data}\ t\ \texttt{=}\ C_1\ \texttt{t}_{11}\ \ldots\ \texttt{t}_{1n_1}\ \texttt{|}\ \ldots\ \texttt{|}\ C_k\ \texttt{t}_{k1}\ \ldots\ \texttt{t}_{kn_k}$

führt neuen Typ t und Konstruktoren $C_1, \ldots C_k$ mit

$$C_i :: t_{11} \to \ldots \to t_{in_i} \to t$$

ein.

Beispiel: Wir können eigene Zahlenlisten wie folgt definieren:

```
data List = Nil | Cons Int List
```

Hierdurch wird der Datentyp `List` und die Konstruktoren

```
Nil  :: List
Cons :: Int  → List  → List
```

eingeführt.

Spezialfälle:

Aufzählungstypen:

```
data Color = Red | Yellow | Blue
data Bool = True | False
```

Verbundtypen (Records):

```
data Complex = Complex Float Float

addc :: Complex  → Complex  → Complex
addc (Complex r1 i1) (Complex r2 i2) = Complex (r1+r2) (i1+i2)
```

Rekursive Typen (variante Records): Listen (s.o.)

Binärbäume mit ganzzahligen Blättern:

```
data Tree = Leaf Int | Node Tree Tree
```

Addiere alle Blätter:

```
addLeaves (Leaf x) = x
addLeaves (Node t1 t2) = (addLeaves t1) + (addLeaves t2)
```

Beachte: dies ist kein Baumdurchlauf (Strategie), sondern eine Spezifikation!