Funktionale Programmierung

Mitschrift von www.kuertz.name

Hinweis: Dies ist **kein offizielles Script**, sondern nur eine private Mitschrift. Die Mitschriften sind teweilse **unvollständig, falsch oder inaktuell**, da sie aus dem Zeitraum 2001–2005 stammen. Falls jemand einen Fehler entdeckt, so freue ich mich dennoch über einen kurzen Hinweis per E-Mail – vielen Dank!

Mihhail Aizatulin (avatar@hot.ee)
und Klaas Ole Kürtz (klaasole@kuertz.net)

Inhaltsverzeichnis

1 Einführung **1**

2 Einführung in Haskell **3**
- 2.1 Funktions- und Typdefinitionen . . . 3
  - 2.1.1 Funktionsdefinitionen . . . 3
  - 2.1.2 Basisdatentypen . . . 6
  - 2.1.3 Typannotationen . . . 7
  - 2.1.4 Algebraische Datentypen . . . 7
- 2.2 Polymorphismus . . . 9
- 2.3 Pattern-Matching . . . 12
  - 2.3.1 Case-Ausdrücke . . . 13
  - 2.3.2 Guards . . . 14
- 2.4 Funktionen höherer Ordnung . . . 14
  - 2.4.1 Anonyme Funktionen (Lambda-Abstraktion) . . . 15
  - 2.4.2 Generische Programmierung . . . 17
  - 2.4.3 Kontrollstrukturen (Programmschemata) . . . 19
  - 2.4.4 Funktionen als Datenstrukturen . . . 20
  - 2.4.5 Wichtige Funktionen höherer Ordnung . . . 21
- 2.5 Typklassen und Überladung . . . 22
  - 2.5.1 Die Klasse `Read` . . . 24
- 2.6 Lazy Evaluation . . . 25
- 2.7 List Comprehensions . . . 28

3 Monaden **32**
- 3.1 die IO-Monade . . . 32
- 3.2 allgemeine Monaden . . . 36
- 3.3 Implementierung der IO-Monade . . . 41
- 3.4 Erweiterung der IO-Monade um Zustände . . . 42
- 3.5 Zustandsmonaden . . . 43

4 Theoretische Grundlagen: Der Lambda-Kalkül **45**
- 4.1 Syntax des Lambda-Kalküls . . . 45
- 4.2 Substitution . . . 45
- 4.3 Reduktionsregeln . . . 46
- 4.4 Datenobjekte im reinen Lambda-Kalkül . . . 49
- 4.5 Mächtigkeit des Kalküls . . . 50
- 4.6 Angereicherter Lambda-Kalkül . . . 51

5 Parserkombinatoren **52**
5.1 Kombinatorbibliothek für Parser 52
5.2 Monadische Parserkombinatoren 56
5.3 Anmerkung zum Parsen 58

6 Debugging **59**
6.1 Debuggen mit Observations (Hood) 60
6.2 Implementierung von Observations für Daten 61
6.3 Implementierung von Observations für Funktionen 64
6.4 Andere Ansätze 66

7 Rekursion **68**
7.1 Attributierte Grammatiken 68
7.2 Continuation Passing Style 76

8 Algorithmen und Datenstrukturen **79**
8.1 Listen 79
8.2 Stacks (LIFO) 81
8.3 Queues (FIFO) 82
8.4 Arrays 82
8.5 Höhenbalancierte Suchbäume 85
8.6 Tries 86
8.6.1 Nested Datatypes 90

9 Nebenläufige Programmierung **93**
9.1 ConcurrentHaskell 93
9.2 Mutable Variables 94
9.3 Scheduling 95
9.4 Semaphoren 96
9.5 Message Passing 97
9.6 Memory Transactions 99

10 Erlang **104**
10.1 Nebenläufige Programmierung 106
10.2 Verteilte Programmierung 108
10.3 Robuste Programmierung 111

1 Einführung

Vorteile von funktionalen Sprachen sind:

- hohes Abstraktionsniveau, keine Manipulation von Speicherzellen
- keine Seiteneffekte, dies führt zu höherer Verständlichkeit und besseren Möglichkeiten zur Code-Optimierung
- Programmierung über Eigenschaften, nicht über den zeitlichen Ablauf
- implizite Speicherverwaltung (u.a. Garbage Collection)
- einfachere Korrektheitsbeweise
- kompakterer Source-Code (kürzere Entwicklungszeiten, lesbarere Programme, bessere Wartbarkeit)
- modularer Programmaufbau, Polymorphismus, Funktionen höherer Ordnung, damit auch eine hohe Wiederverwertbarkeit des Codes

Strukturen in Funktionalen Programmen sind

- Variablen entsprechen unbekannten Werten (nicht Speicherzellen!)
- Programme entsprechen Mengen von Funktions- und Typdefinitionen
- Speicher ist nicht explizit verwendbar, sondern wird automatisch alloziert und freigegeben (Garbage Collection)
- Programmablauf entspricht einer Reduktion von Ausdrücken (nicht einer Sequenz von Anweisungen)

Historie der funktionalen Sprachen:

- **Grundlage** ist die mathematische Theorie des λ-Kalküls (Church 1941)
- LISP (McCarthy 1960): Datenstruktur ist vor allem Listen, Verwendung heute in der KI und in Emacs
- Scheme: LISP-Dialekt mit statischer Bindung
- SASL (Turner 1979): Lesbares LISP für die Lehrer
- ML (Milner 1978): polymorphes Typsystem
- KRC (Turner 1980): Guards, Pattern-Matching, Laziness

- Miranda (Turner 1985): Erweiterung von KRC um ein polymorphes Typsystem, Modulkonzept, aber: geschützte Sprache, damit geringe Verbreitung
- Haskell: Entwicklung als public-domain-Variante von Miranda, Festsetzung als Standard Haskell 98, seitdem jedoch viele Weiterentwicklungen (Typsystem, Nebenläufigkeit, verteilte Programmierung, explizite Speichermanipulation, ForeignFunction-Interace
- StandardML (1997): Standardisierung, seitdem aber auch viele Weiterentwicklungen
- Erlang: entwickelt von der Firma Ericsson zur Implementierung von Telekommunikationsanwendungen; Aufbau auf PROLOG, spezielles Konzept zur nebenläufigen bzw. verteilten Programmierung

In der Vorlesung wird zunächst Haskell 98 behandelt.

2 Einführung in Haskell

2.1 Funktions- und Typdefinitionen

- In der Mathematik stehen **Variablen** für unbekannte Werte, beispielsweise ist $x^2 - 4x + 4 = 0$ eine Aussage, aus der sich $x = 2$ ableiten lässt. Aber in imperativen Programmiersprechen stehen Anweisungen wie `x = x + 1`, was im Widerspruch zur mathematischen Notation steht! **Idee** der funktionalen Programmierung: Variablen werden wie in der Mathematik verwendet!

- In der Mathematik verwendet man **Funktionen** für Berechnungen, in der imperativen Welt werden diese jedoch (mit einer anderen Bedeutung: Seiteneffekte!) zur Modularisierung eingesetzt, **Lösung** aus der funktionalen Welt: Programmierung ohne Seiteneffekte!

2.1.1 Funktionsdefinitionen

Die Definition einer Funktion geschieht folgendermaßen:

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

Dabei ist `f` der Funktionsname, die x_i sind formale Parameter und `e` ist der Funktionsrumpf. Mögliche **Ausdrücke** dabei sind:

- Zahlen (`3`, `3.1415`...)
- Basisoperationen (`3+4`, `6*7`)
- formale Parameter (Variablen)
- Funktionsanwendung (`(f e1 ... en)`) mit einer Funktion `f` und Ausdrücken `ei`
- bedingte Ausdrücke (`if b then e1 else e2`)

Beispiele für Funktionen:

- Quadratfunktion:

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

- Minimum:

  ```
  min x y = if x <= y then x else y
  ```

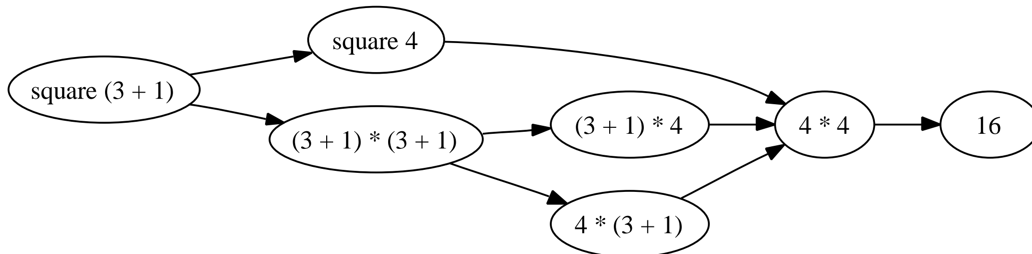
- **Fakultätsfunktion**: Mathematisch ist die Funktion definiert als:

$$n! = \begin{cases} 1 & \text{falls} \quad n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

  In Haskell:

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

Betrachte zur **Auswertung** die Funktionsdefinition als orientierte Gleichung (Reduktionssemantik): Binde die formalen Parameter an die aktuellen Parameter und ersetze den Funktionsaufruf durch die rechte Seite. Beispiel für die Auswertung von `square (3 + 1)`:



Weiteres Beispiel: **Fibonacci-Funktion**

```haskell
fib1 n = if n == 0
            then 0
            else if n == 1
                    then 1
                    else fib1(n - 1) + fib1(n - 2)
```

Dies entspricht der mathematischen Definition, ist aber sehr ineffizient ($\mathcal{O}(2^n)$).
Verbesserung: Berechnung der Fibonacci-Folge „von unten", bis der n-te Wert berechnet ist – diese Programmiertechnik heißt *Akkumulatortechnik*: Zur Berechnung von `fib n` werden die Ergebnisse von `fib(n - 1)` und `fib( n - 2)` benötigt, behalte diese als Akkumulator-Parameter.

```haskell
fib2' fibn fibnp1 n =
      if n == 0 then fibn
                else fib2' fibnp1
                           (fibn + fibnp1)
                           (n - 1)

fib2 n = fib2' 0 1 n
```

Aus softwaretechnischer Sicht ist dies unschön: `fib2'` kann auch (falsch) verwendet werden, besser ist eine **lokale Definition**:

```
fib2 n = fib2' 0 1 n
     where fib2' fibn fibnp1 n = ...
```

Hierbei ist `where` immer gültig für die letzte Gleichung! Weitere Möglichkeit:

```
fib2 n = let fib2' fibn fibnp1 n = ...
         in fib2' 0 1 n
```

Das Konstrukt `let ... in` ist auch in beliebigen Ausdrücken möglich, z.B:

```
(let x = 3
     y = 1
 in  x + y) + 2
```

Zur Vermeidung von Klammern und Trennzeichen wird die **Off-Side-Rule** benutzt: Das nächste Symbol, das kein Whitespace ist hinter `where` und `let`, definiert einen Block:

```
f x = e
  where g y = {Rumpf von g}
          {Rumpf von g geht weiter}
          {Rumpf von g geht weiter}
        h z = {Rumpf von h}
          {Rumpf von h geht weiter}
          {Rumpf von h geht weiter}
k x y = ...
```

Sobald ein Zeichen auf der gleichen Höhe wie `g` erscheint, wird es als Anfang einer neuen Definition in `where` betrachtet, hier also `h`. Das erste Zeichen links von `g` (in diesem Fall `k`) wird dem übergeordneten Block, hier der Top-Level-Deklaration zugeordnet. Aber auch geschachtelte `let`- und `where`-Deklarationen sind möglich.

In verschiedenen Blöcken im obigen Beispiel sind folgende Variablen und Funktionen **sichtbar**:

- Im Rumpf von `g`: Variablen `x, y`, Funktionen `f, g, h, k`
- Im Rumpf von `h`: Variablen `x, z`, Funktionen `f, g, h, k`
- Im Rumpf von `k`: Variablen `x, y`, Funktionen `f, k`

Vorteile von lokalen Deklarationen sind:

- Vermeidung von Namenskonflikten und falscher Verwendung von Hilfsfunktionen
- bessere Lesbarkeit wegen des kleineren Interfaces
- Vermeidung von Mehrfachberechnung, betrachte beispielsweise die Funktion $f(x, y) = y(1 - y) + (1 + xy)(1 - y) + xy$, diese kann umgesetzt werden als

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

- Einsparung von Parametern der Hilfsfunktion, beispielsweise beim Test auf Primzahleigenschaft

```
isPrime n = n /= 1 &&
            checkDiv (div n 2)
      where checkDiv m =
            m == 1 || mod n m /= 0
                      && checkDiv (m - 1)
      -- && bindet stärker als ||
```

  Der Operator `&&` ist ein **nicht striktes „und“**, z.B. gilt `False &&`⊥ `= False` (wobei ⊥ eine nicht-terminierende Berechnung bezeichnet). Genauso ist `||` ein nicht striktes „oder“. Die Funktion `div` ist die ganzzahlige Division. Kommentare fangen mit `--` an.

2.1.2 Basisdatentypen

Bereits verwendet:

- Ganze Zahlen:
  - `Int`, Werte von $-2^{16} + 1$ bis $2^{16} - 1$
  - `Integer`, beliebig klein/groß (nur durch Speicher beschränkt)

  Operationen: `+, -, *, div, mod`; Vergleiche: `<=, >=, <, >, /=, ==` .

- Boolesche Werte: `Bool`; Werte: `True, False`; Operationen: `&&, ||, ==, /=, not`

- Fließkommazahlen: `Float`, geschrieben `0.3, -1.5e-2`; Operationen wie auf `Int/Integer`, mit `/` statt `div` und ohne `mod`
- Zeichen (ASCII): `Char`, Werte: `'a', '\n', '\NUL', '\214'`. Operationen: `chr, ord`

2.1.3 Typannotationen

Alle Werte bzw. Ausdrücke in Haskell haben einen **Typ**, welcher auch mittels `::` **annotiert** werden kann:

```
3::Int
3::Integer
(3 == 4) || True :: Bool
square :: Int -> Int
square x = (x :: Int) * (x :: Int)::Int
```

Aber: was ist der Typ von `min`? In ML (Java) wäre der Typ `min :: (Int, Int)-> Int`, die Schreibweise in Haskell ist jedoch *curryfiziert*: `min :: Int -> Int -> Int`.

2.1.4 Algebraische Datentypen

Eigene Datentypen können als neue Datentypen definiert werden. Werte werden mittels *Konstruktoren* (frei interpretierte Funktionen) aufgebaut. Die Definition sieht allgemein wie folgt aus:

$$\textbf{data}\ \tau = c_1\ \tau_{11} \ldots \tau_{1n_1} \mid \ldots \mid c_k\ \tau_{k1} \ldots \tau_{kn_k}$$

wobei

- τ der neu definierte Typ ist,
- $c_1, \ldots, c_k$ definierte Konstruktoren sind
- τ_{i1} bis τ_{in_i} die Argumenttypen des Kontruktors c_i sind, also $c_i :: \tau_{i1} \to \ldots \to \tau_{in_i} \to \tau$.

Beachte: Sowohl Typen, als auch Konstruktoren müssen mit Großbuchstaben beginnen. **Beispiele:**

- Aufzahlungstyp (nur 0-stellige Konstruktoren)

  ```
  data Color = Red | Blue | Yellow
  Red::Color
  ```

- Verbundtyp (nur ein Konstruktor)

```
data Complex = Complex Float Float
Complex 3.0 4.7 :: Complex
```

Namen für Typen und Konstruktoren sind in getrennten Namensräumen, deshalb gleiche Namen erlaubt.

Wie ist die Selektion von Komponenten möglich? In Haskell: *Pattern-Matching* statt expliziter Selektionsfunktionen. **Beispiel:**

```
addC :: Complex -> Complex -> Complex
addC (Complex r1 i1)
     (Complex r2 i2) =
          (Complex (r1 + r2) (i1 + i2))
```

- Listen kann man wie folgt definieren:

```
data List = Nil | Cons Int List

append :: List -> List -> List
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

`Nil` stellt die leere Liste dar, `Cons` erzeugt eine neue Liste durch Anhängen eines neues Elements. Die Funktion `append` wird mit Hilfe mehrerer Gleichungen definiert, wobei die erste passende gewählt wird.

In Haskell sind Listen vordefiniert:

```
data [Int] = [] | Int : [Int]
```

Der Operator `:` ist rechts-assoziativ, also `1:(2:(3:[]))= 1:2:3:[]`. Abkürzende Schreibweise: Aufzählung der Elemente, z.B. `[1, 2, 3]`. Außerdem: Operator `++` statt `append`:

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

Bemerkung: Operatoren sind zweistellige Funktionen, die infix geschrieben werden und mit einem Sonderzeichen beginnen. Durch Klammerung werden sie zu normalen Funktionen, z.B. `[1] ++ [2]` ist äquivalent zu `(++) [1] [2]`. Umgekehrt können zweistellige Funktionen in `‘...‘` infix verwendet werden: `div 4 2` ist das Gleiche wie `4 ‘div‘ 2`.

Beispiel: Berechnung der Länge einer Liste:

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

Bemerkung: Für selbstdefinierte Datentypen besteht nicht automatisch die Möglichkeit, diese vergleichen bzw. ausgeben zu können. Hierzu:

```
data MyType = ...
       deriving (Eq, Show)
```

`Eq` stellt `==` und `/=` für den Typen zur Verfügung, `Show` bietet die Möglichkeit zur Ausgabe.

2.2 Polymorphismus

Betrachtet man die Definitionen von `length` und `(++)`, so fällt auf, dass beide zwar auf Listen arbeiten, aber der Typ der Listenelemente nicht relevant ist. Möglich wäre sowohl

```
length :: [Int] -> Int
```

als auch

```
length :: [[Int]] -> Int
```

Allgemein könnte man sagen:

```
length :: ∀ Typen τ [τ] -> Int
```

was in Haskell durch Typvariablen ausgedrückt wird:

```
length :: [a] -> Int
```

was für alle Typen `a` bedeutet. Der Name einer Typvariablen muss mit einem kleinen Buchstaben anfangen. Der Typ von `(++)` ist somit

```
(++) :: [a] -> [a] -> [a]
```

Es können nur Listen über gleichen Elementtypen zusammengehängt werden. Für **Definitionen** von polymorphen Datenstrukturen verwendet man *Typkonstruktoren* zum Aufbauen von Typen:

$$\textbf{data}\ \mathcal{K}\ a_1 \ldots a_n = c_1\ \tau_{11} \ldots \tau_{1n_1} \mid \ldots \mid c_k\ \tau_{k1} \ldots \tau_{kn_k}$$

Dies ist ein Datentypkonstruktor wie zuvor, dabei sind

- $\mathcal{K}$ ein Typkonstruktor

- $a_1, \ldots, a_n$ Typvariablen
- τ_{ik} Typen (auch polymorphe)

Funktionen und Konstruktoren werden auf Werte (Ausdrücke) angewendet. Analog werden Typkonstruktoren auf Typen bzw. Typvariablen angewendet.
Beispiel: Partielle Werte

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

Dann lassen sich folgendes Typen benutzen:

```
Maybe Int
Maybe (Maybe Int)
```

Zugehörige Werte sind `Nothing`, `Just 42` bzw. `Just Nothing` oder `Just (Just 42)`. Bei der Applikation von Typkonstruktoren auf Typvariablen erhalten wir *polymorphe Typen*:

```
isNothing :: Maybe a -> Bool
isNothing Nothing = True
isNothing (Just _) = False
```

Das Pattern `_` steht für einen einen beliebigen Wert. **Beispiele:**

- **Binärbäume**

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

  height :: Tree a -> Int
  height Empty = 0
  height (Node tl _ tr) = 1 + (max (height tl)
                                   (height tr))
  ```

- In Haskell vordefiniert: **Listen**.

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

  Dies ist ein Pseudocode, eine eigene Definition ist so nicht erlaubt. Die erste und die dritte Verwendung von `[]` sind Typkonstruktoren, die zweite dagegen ein Listenkonstruktor. Damit sind `[Int]`, `[Char]`, `[[Int]]` Applikationen des Typkonstruktors `[]` auf Typen und `[a]` der polymorphe Typ, der *alle* Listentypen repräsentiert.

  Einige Funktionen auf Listen:

```haskell
head :: [a] -> a
head (x _) = x

tail :: [a] -> [a]
tail (_:xs) = xs

last :: [a] -> a
last [x] = x
last (_:xs) = last xs

concat :: [[a]] -> [a]
concat [] = []
concat (l:ls) = l ++ concat ls

(!!) :: [a] -> Int -> a
(x:xs) !! n = if n == 0
                then x
                else xs !! (n - 1)
```

- **Strings** sind in Haskell als Typsynonym für Listen von Zeichen definiert:

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

  Schreibweise: `"Hallo"= 'H':'a':'l':'l':'o':[]`. Vorteil gegenüber speziellen vordefinierten Typ: alle Listenfunktionen für `String` auch verwendbar, z.B. `length ("Hallo"++ "Leute")`.

- **Vereinigung** zweier Typen:

  ```haskell
  data Either a b = Left a | Right b
  ```

  Damit können z.B. Werte „unterschiedlicher“ Typen in einer Liste zusammengefasst werden:

  ```haskell
  [Left 42, Right "Hallo"] :: [Either Int String]
  ```

- **Tupel** (mit Mixfixnotation):

  ```haskell
  data (,) a b = (,) a b -- oder (a, b)
  data (,,) a b c = (,,) a b c
  -- usw
  ```

Das linke Vorkommen von `(,)` ist ein Typkonstruktor; `(,)` rechts ist ein Wertkonstruktor. Einige Funktionen:

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

snd :: (a, b) -> b
snd (_, y) = y

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

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

Bemerkung: Es gilt nicht immer `unzip (zip` $l_1\, l_2$`)= (`l_1, l_2`)`.

2.3 Pattern-Matching

Pattern-Matching ist ein komfortabler Programmierstil, alternativ zur Verwendung von Selektoren. Dabei entspricht die linke Seite einer Funktionsdefinition einem „Prototyp“ des Funktionsaufrufs. Die Funktionen werden durch mehrere Gleichungen definiert:

$$
\begin{array}{l}
f\ \mathrm{pat}_{11}\ \ldots\ \mathrm{pat}_{1n} = e_1 \\
\quad \ldots \\
f\ \mathrm{pat}_{k1}\ \ldots\ \mathrm{pat}_{kn} = e_k
\end{array}
$$

Semantik: Wähle die erste Regel mit passender linken Seite und wende diese an. Dabei sollte man *überlappende Regeln vermeiden!*

Aufbau der Pattern:

- `x` (Variable): passt immer, `x` wird an aktuellen Ausdruck gebunden
- `_` (Wildcard): passt immer, keine Bindung
- `c` $\texttt{pat}_1 \ldots \texttt{pat}_k$ mit `c` k-stelliger Konstruktor und $\texttt{pat}_1, \ldots, \texttt{pat}_k$ Pattern: passt, falls aktueller Wert $(c\ e_1 \ldots e_k)$ ist und $\texttt{pat}_1$ auf alle e_i passt. Zahlen und Zeichen (Char) können als 0-stellige Konstruktoren

verwendet werden; `(:)` kann auch infix angegeben werden, Tupel auch mixfix.

- `x@pat` („as pattern"): Passt, falls `pat` passt, zusätzlich Bindung von `x` an aktuellen Ausdruck. Benutzer zum einen zum Benennen gröberer Strukturen, zum anderen zur Verfeinerung von Variablen.
- `n + k`: Passt auf alle Zahlen größer gleich k, wobei n an aktuellen Wert minus k gebunden wird. **Beispiel:**

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

Nicht erlaubt ist mehrfaches Auftreten einer Variable in einem Pattern, mehrere Wildcards können jedoch auftreten:

```
eq :: a -> a -> Bool
eq x x = True  -- verboten
eq _ _ = False -- erlaubt
```

Pattern können auch an anderen Stellen verwendet werden, und zwar als Konstantendefinitionen, z.B. in der Definition von `unzip`:

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

Hier wird im `let` Pattern-Matching benutzt. Genauso ist Pattern-Matching im `where` möglich oder auch auf Toplevel:

```
(dimX, dimY) = evalFieldSize testField
           -- wird nur einmal berechnet
```

2.3.1 Case-Ausdrücke

Manchmal ist Pattern-Matching mit Verzweigung auch in Ausdrücken praktisch – die Definition einer Hilfsfunktionen zum Verzweigen wäre umständlich. Dann werden *case-Ausdrücke* verwendet:

```
case e of
      pat_1 -> e_1
       ...
      pat_n -> e_n
```

Beachte dabei die Einrückungen gemäß Offside-Rule! Obiges definiert einen Ausdruck mit dem Typ von e_1 (der gleich dem Typ von $e_2, \ldots, e_n$ sein muß). Außerdem muss der Typ von e, `pat`$_1$, `pat`$_n$ ebenfalls gleich sein.

Beispiel: Extraktion der Zeilen eines Strings:

```
lines :: String -> [String] -- vordefiniert
lines "" = []
lines ('\n':cs) = "" : lines cs
lines (c:cs) = case lines cs of
                 (str:strs) -> (c:str):strs
                 []         -> [[c]]
```

2.3.2 Guards

Programmieren mit mehreren Regeln ist oft schöner als mit Bedingungen im Ausdruck. *Guards* erlauben es, beliebige Bedingungen auf die linke Seite zu schieben. **Beispiel:**

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

Semantik: Guards werden der Reihe nach ausgewertet. Die rechte Seite des ersten erfüllten Guards wird gewählt, dabei ist `otherwise = True` vordefiniert. Falls kein Guard erfüllt ist, wird die nächste Regel gewählt.

Beispiel: die ersten n Elemente einer Liste:

```
take :: Int -> [a] -> [a]
take n _ | n <= 0   = [] -- totale Funktion
take _ []           = []
take (n + 1) (x:xs) = x : take n xs
```

Guards sind bei jedem Pattern Matching erlaubt, also auch bei `case`-, `let`- und `where`-Ausdrücken.

2.4 Funktionen höherer Ordnung

Idee: Funktionen sind in funktionalen Programmiersprachen „Bürger erster Klasse“, d.h. können wie alle anderen Werte überall verwendet werden. Anwendungen sind unter anderem generische Programmierung und Programmschemata (Kontrollstrukturen).

Vorteile: Höhere Wiederverwendbarkeit und große Modularität.

Beispiel: Die Ableitung ist eine Funktion, die eine Funktion nimmt und eine Funktion berechnet:

```
ableitung :: (Float -> Float) -> (Float -> Float)
```

Numerische Berechnung:

$$f'(x) = \lim_{dx \to 0} \frac{f(x + dx) - f(x)}{dx}$$

Implementierung: Wähle kleines dx

```
dx = 0.0001
ableitung f = f'
      where f' x = (f(x + dx) - f x) / dx

(ableitung sin) 0.0 ⇝ 1.0
(ableitung square) 1.0 ⇝ 2.0003
```

2.4.1 Anonyme Funktionen (Lambda-Abstraktion)

„Kleine" Funktionen müssen nicht global definiert werden. Stattdessen kann man schreiben

```
ableitung (\x -> x * x)
```

Allgemein:

```
\pat₁ ... patₙ -> e
```

wobei `pat`$_1$, ..., `pat`$_n$ Pattern und e ein Ausdruck sind. Nun können wir schreiben:

```
ableitung f = \x -> (f(x + dx) - f x) / dx
```

Die Funktion `ableitung` ist nicht die erste Funktion, die ein funktionales Ergebnis hat. Betrachte z.B. die folgende Funktion:

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

Eine andere mögliche Definition:

```
add = \x y -> x + y
```

Also kann `add` auch als Konstante mit funktionalem Wert gesehen werden! Noch eine weitere Definition:

```
add x = \y -> x + y
```

Nun ist `add` eine einstellige Funktion mit funktionalem Ergebnis.

Mit der dritten Definition von `add` sollte auch `ableitung (add 2)`$\triangleq$`\x -> 1.0` möglich sein. Dies ist bei allen Implementierungen von Haskell möglich.

Bei der ersten Implementierung bezeichnet man `add 2` als *partielle Applikation* (man übergibt bei der Applikation weniger Argumente, als die Stelligkeit der Funktion es fordert). Ferner sollte gelten: `add x y` $\triangleq$`(add x)y`. In Haskell sind beide Ausdrücke sogar semantisch gleich: Die Funktionsapplikation ist *linksassoziativ*.

Entsprechend verhalten sich auch die Typen:

$$a \rightarrow b \rightarrow c == a \rightarrow (b \rightarrow c)$$

Der Typkonstruktor $\rightarrow$ ist also *rechtsassoziativ*. **Beachte**:

$$a \rightarrow b \rightarrow c \neq (a \rightarrow b) \rightarrow c$$

Die partielle Applikation ist möglich wegen der *curryfizierten*[1] Schreibweise. Wegen der Möglichkeit der partiellen Applikation ist die curryfizierte Definition von Funktionen der über Tupeln **vorzuziehen**!
Beispiele für partielle Applikation:

- `take 42 :: [a] -> [a]` gibt die ersten 42 Elemente jeder Liste zurück
- `(+) 1 :: Int -> Int` ist die Inkrement- oder Nachfolgerfunktion
- `(2-) :: Int -> Int` ist die Funktion `\x -> 2 - x`
- `(-2) :: Int -> Int` ist die Funktion `\x -> x - 2`
- `(/b) a` = `(a/) b` = `a/b`

Beachte: Bei Operatoren gibt es zusätzliche Schreibweisen, und es ist auch eine partielle Applikation auf ein zweites Argument möglich.

[1]Currying geht zurück auf die Mathematiker Curry und Schönfinkel, die gleichzeitig, aber unabhängig voneinander in den 1940ern folgende Isosmorphie feststellten: $[A \times B \rightarrow C] \simeq [A \rightarrow [B \rightarrow C]]$

2.4.2 Generische Programmierung

Beachte folgende Haskell-Funktionen:

```
inclist :: [Int] -> [Int]
inclist [] = []
inclist (x:xs) = (x+1) : inclist xs

code :: Char -> Char
code c | c == 'Z'  = 'A'
       | c == 'z'  = 'a'
       | otherwise = chr(ord c + 1)

codestr :: String -> String
codestr "" = ""
codestr (c:cs) = (code c) : codestr cs
```

Beide Funktionen haben „ähnliche“ Struktur, beide applizieren `code` bzw. inc auf alle Listenelemente. **Idee** ist nun, `code` und `inc` als Parameter zu übergeben. **Verallgemeinerung**:

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

Nun lassen sich obige Funktionen ganz einfach definieren:

```
inclist = map (1+)
codestr = map code
```

Beachte ein weiteres Beispiel:

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

checksum :: String -> Int
checksum [] = 1
checksum (c:cs) = ord c + checksum cs
```

Auch hier ist eine ähnliche Struktur erkennbar, die man verallgemeinern kann:

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

Nun lassen sich obige Funktionen wieder einfacher definieren:

```
sum = foldr + 0
checksum = foldr (\c sum -> ord c + sum) 1
```

Weitere Anwendung wären zum Beispiel Konjunktion und Disjunktion auf Listen von Booleschen Werten:

```
and = foldr (&&) True
or  = foldr (||) False
```

Entsprechend: `concat = foldr (++)[]`.

Allgemeines Vorgehen: Suche ein allgemeines Schema und realisiere es durch eine Funktion höhere Ordnung.

Weiteres Schema:

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

Die Umwandlung einer Liste in eine Menge (d.h. Entfernen doppelter Einträge) kann jetzt erfolgen durch:

```
nub :: [Int] -> [Int]
nub [] = []
nub (x:xs) = x : nub(filter (/= x) xs)
```

Sortieren von Listen mittels *QuickSort*:

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

Hier wäre es besser, statt der doppelten Anwendung eines Filters auf jedes Argument die Funktion `split` zu verwenden – siehe Übung.

Auch `filter` ist mittels `foldr` definierbar!

```
filter = foldr (\x ys -> if p x then x:ys
                                else   ys) []
```

Beachte: `foldr` ist ein *sehr* allgemeines **Skelett** (Programmiermuster), es entspricht einem Katamorphismus der Kategorientheorie. Dabei entspricht `foldr f e (x_1:x_2:...:[])` der Auswertung

$$(f x_1 (f x_2 \ldots (f\, x_n\, e)))$$

Dies hat aber manchmal auch **Nachteile**: Das äußerste `f` kann oft erst berechnet werden, wenn die ganze Struktur durchlaufen wurde, zum Beispiel wird bei `foldr (+)0 [1,2,3]` zunächst $1+(2+(3+0))$ auf den Stack gelegt und kann dann erst abgearbeitet werden.

„Bessere" Lösung ist hier definierbar mit der Akkumulatortechnik:

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

Hier wird nun (effizienter!) $((0+1)+2)+3$ berechnet statt $1+(2+(3+0))$. Diese Lösung zeigt wieder ein Programmierschema, was wieder verallgemeinert wird:

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

Dabei entspricht `foldl f e (`x_1`:`x_2`:...:[])` der Auswertung $f\dots(f(fex_1)x_2)\dots x_n$.

2.4.3 Kontrollstrukturen (Programmschemata)

In einer imperativen Sprache würde folgendes benutzt werden:

```
x = 1;
while (x < 100)
  x = x * 2;
```

Ähnliches funktioniert in Haskell mit Funktionen höherer Ordnung:

```
while :: (a -> Bool) -> (a -> a) -> a -> a
while p f x | p x = while p f (f x)
            | otherwise = x
```

Beispiel: Der Aufruf `while (<100)(2*)1` wird zu `128` ausgewertet.

Damit könnte man imperative Programmierung als Programmierung mit einem begrenzten Satz an Funktionen höherer Ordnung, den Kontrollstrukturen, auffassen.

2.4.4 Funktionen als Datenstrukturen

Definition: Eine *abstrakte Datenstruktur* ist ein Objekt mit:

- Konstruktoren (z.B. `[]`, `(:)` bei Liste)
- Selektoren (z.B. `head`, `tail`)
- Testfunktionen (z.B. `null`)
- Verknüpfungen (z.B. `(++)`)

Wichtig hierbei ist die **Funktionalität** der Schnittstelle und nicht die Implementierung. Somit entspricht eine Datenstruktur einem Satz von Funktionen. Die einfachste Implementierung ist also mittels Funktionen.

Beispiel: Arrays (Felder mit beliebigen Elementen).

- Konstruktoren:

  ```
  emptyArray :: Feld a

  -- fügt Wert in Feld an Indexposition ein
  putIndex :: Feld a -> Int -> a -> Feld a
  ```

- Selektoren

  ```
  -- liefert Wert an Indexposition
  getIndex :: Feld a -> Int -> a
  ```

Implementierung:

```
type Feld a = Int -> a

emptyArray i = error ("Zugriff auf nicht" ++
      "initialisierte Arraykomponente" ++ show i)

getIndex a i = a i

putIndex a i v = a'
      where a' j | i == j = v
                 | otherwise = a j
```

Vorteil ist die konzeptuelle Klarheit (Implementierung entspricht der Spezifikation), dies ist ideal für Rapid Prototyping. **Nachteil:** Die Zugriffszeit

ist abhängig von Anzahl der `putIndex`-Operationen. Durch einen abstrakten Datentyp ist später aber eine Optimierung möglich.

2.4.5 Wichtige Funktionen höherer Ordnung

- Hintereinanderausführung von Funktionen mittels `(.)`:

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

  Beispiel: Die folgende Funktion liefert die Anzahl der Elemente einer Liste von Listen

  ```
  length.concat :: [[a]] -> Int
  ```

- Vertauschung der Reihenfolge von Argumenten durch `flip`:

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

  Beispiel: `flip (:)[1] 2` ⇝ `[2, 1]`

- `curry` und `uncurry`:

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

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

  Beispiel:

  ```
  map (uncurry (+)) (zip [1, 2, 3] [4, 5, 6])
      ⇝ [5, 7, 9]
  ```

- Konstante Funktion `const`:

  ```
  const :: a -> b -> a
  const x _ = x
  ```

2.5 Typklassen und Überladung

Betrachte folgende Funktionsdefinition:

```
elem x [] = False
elem x (y:ys) = x == y || elem x ys
```

Was soll der Typ von `elem` sein? Möglichkeiten wären z.B.:

```
elem :: Int -> [Int] -> Bool
elem :: Bool -> [Bool] -> Bool
elem :: Char -> String -> Bool
```

Leider sind diese Typen nicht allgemein genug. Eine Möglichkeit wäre noch

```
elem :: a -> [a] -> Bool
```

Dies ist zu allgemein, da `a` zum Beispiel Funktionen repräsentieren kann, für die die Gleichheit nicht entschieden werden kann. Deswegen machen wir **Einschränkung auf Typen**, für die die Gleichheit definiert ist:

```
elem :: Eq a => a -> [a] -> Bool
```

Dabei ist `Eq a` ein *Klassenkonstraint*, der `a` einschränkt. [2] Die Klasse `Eq` ist wie folgt definiert:

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

In einer *Klasse* werden mehrere Typen zusammengefasst, für die die Funktionen der Klasse definiert sind (hier `(==)` und `(/=)`). Typen werden zu *Instanzen* einer Klasse wie folgt:

```
data Tree = Empty | Node Tree Int Tree

instance Eq Tree where
  Empty == Empty = True
  (Node tl1 n1 tr1) == (Node tl2 n2 tr2) =
        tl1 == tl2 && n1 == n2 && tr1 == tr2
  t1 /= t2 = not (t1 == t2)
```

Die Gleichheit auf polymorphen Datentypen kann nur definiert werden, falls Gleichheit für Subtypen definiert ist:

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

[2] Bei mehreren Konstraints muss man klammern: `(Eq a, Ord a)`.

```
instance Eq a => Eq (Tree a) where
        ...
```

Beachte: Auf diese Weise werden unendlich viele Typen jeweils Instanz der Klasse `Eq`.

Es gibt die Möglichkeit, vordefinierte Funktionen einer Klasse anzugeben: Die Definition von `(/=)` wird in fast jeder Instanzdefinition gleich aussehen. Deshalb verschieben wir sie in die Klassendefinition. Die eigentliche Definition von `Eq` ist wie folgt:

```
class Eq a where
        (==), (/=) :: a -> a -> Bool
        x1 == x2 = not (x1 /= x2)
        x1 /= x2 = not (x1 == x2)
```

Dabei kann jeweils eine oder auch beide der vordefinierten Funktionen überschrieben werden.

Es gibt die Möglichkeit, Klassen zu erweitern. **Beispiel:** Die Klasse, die die totale Ordnung definiert, ist eine Erweiterung der Klasse `Eq`:

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

data Ording = (LT | EQ | GT)
```

Eine minimale Instanziierung dieser Klasse implementiert entweder `compare` oder `<=`, es ist aber auch möglich, weitere Funktionen zu überschreiben.

Weitere vordefinierten Klassen sind:

- `Num` zum Rechnen mit Zahlen (definiert `(+)`, `(-)`, `(*)`, `abs` etc.)
- `Show` zum Umwandeln in Strings:

  ```
  show :: Show a => a -> String
  ```

- `Read` zum Konstruieren aus Strings

Vordefinierte Klassen (außer `Num`) werden automatisch instanziiert mittels `deriving` $\mathcal{K}_1, \ldots, \mathcal{K}_n$ hinter der Datentypdefinition, Beispiel:

```
data Tree = Node Tree Int Tree | Empty
   deriving Show
```

2.5.1 Die Klasse `Read`

Die Funktion `read` hat den Typ `Read a =>String -> a`, d.h. der Aufruf `read "(3, a)"` wird ausgewertet zu `(3, 'a')`. Der Aufruf setzt sich zusammen aus drei unterschiedlichen `read`-Funktionen:

```
read :: String -> Int
read :: String -> Char
read :: (Read a, Read b) => String -> (a, b)
```

Idee: die Eingabe wird zeichenweise abgearbeitet, das Ergebnis enthält den Reststring. Die Reihenfolge der Aufrufe ist z.B.:

```
readTuple "(3, 'a')"
      readInt "3, 'a')" ⇝ (3, ", 'a')")
      readChar "'a')"   ⇝ ('a', ")")
⇝ ((3, 'a'), "")
```

Auch Fehler müssen berücksichtigt werden. Eine Möglichkeit wäre:

```
read :: Read a => String -> Maybe (a, String)
```

In Haskell wird es anders gemacht:

```
type ReadS a = String -> [(a, String)]
class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a] -- vordefiniert
```

Das erste Parameter von `readsPrec` ist die Ausdruckstiefe (Anzahl der zu erwartenden Klammern). In der `Prelude` sind definiert:

```
reads :: Read a => ReadS a
reads = readsPrec 0
read :: Read a => String -> a
read xs = case reads xs of
            [(x, "")] -> x
            _         -> error "no parse"
```

Beispiel:

```
reads "(3, 'a')" :: [((Int, Char), String)]
  ⇝ [((3, 'a'), "")]
reads "3, 'a')" :: [(Int, String)]
  ⇝ [(3, ", 'a')")]
```

2.6 Lazy Evaluation

Betrachte folgendes Haskell-Programm:

```
f x = 1
h = h
```

mit der Anfrage:

```
f h
```

Nicht jeder Berechnungspfad dieser Anfrage terminiert. Es gibt aber zwei ausgezeichnete Reduktionen:

- *Leftmost-Innermost (LI, strikte Funktionen)*: Alle Argumente eines Funktionsaufrufs müssen ausgewertet sein, bevor die Funktion angewendet werden kann.
- *Leftmost-Outermost (LO, nicht-strikte Funktionen)*: Die Funktionen werden jeweils vor Argumenten ausgewertet.

Vorteil von Leftmost-Outermost ist die *Berechnungsvollständigkeit*: Alles, was mit irgendeiner Reduktion berechenbar ist, wird auch mit LO berechnet. Dies ist praxisrelevant u.a. für:

- *Vermeidung überflüssiger Berechnungen.* **Beispiel:**

  ```
  head ([1,2] ++ [3,4])
  ```

  Die Auswertung mit beiden Strategien ist in der Abb. 1 dargestellt.

- *Rechnen mit unendlichen Datenstrukturen.* **Beispiel:** Die Funktion

  ```
  from :: Num a => a -> [a]
  from n = n : from (n + 1)
  ```

  definiert eine unendliche Liste aller Zahlen ab n. Betrachte nun

  ```
  take :: Int -> [a] -> [a]
  take n _   | n <= 0 = []
  take n []           = []
  take n (x:xs)       = x:take (n - 1) xs
  ```

  Jetzt können wir schreiben

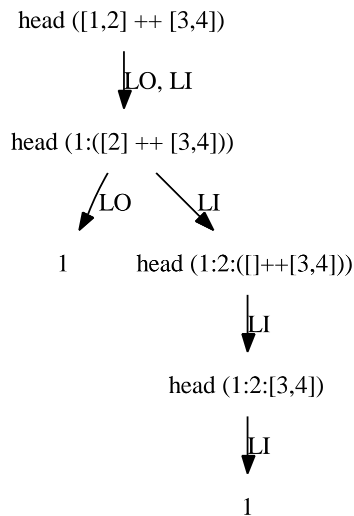


Abbildung 1: Auswertung von `head ([1,2] ++ [3,4])`

```
take 3 (from 1)
⇝ take 3 (1:from(1+1))
⇝ 1:take (3-1) (from(1+1))
 ...
⇝ 1:2:3:take 0 (from 4)
⇝ 1:2:3:[]
```

Vorteil: Trennung von Kontrolle (`take 3`) und Daten (`from 1`).

- Ein weiteres **Beispiel:** Primzahlberechnung mittels Sieb des Eratosthenes. Idee:

  1. Erstelle Liste aller Zahlen größer gleich 2.
  2. Streiche Vielfache der ersten (Prim-)Zahl aus der Liste.
  3. Das 1. Listenelement ist Primzahl, weiter bei (2) mit Restliste.

  In Haskell:

```
sieve :: [Int] -> [Int]
sieve (p:xs) = p:sieve
      (filter (\x -> x `mod` p > 0) xs)
```

```
primes = sieve (from 2)
take 10 primes
⇝ [2,3,5,7,11,13,17,19,23,29]
```

- Die Programmierung mit unendlichen Datenstrukturen kann auch als *Alternative zur Akkumulatortechnik* verwendet werden. **Beispiel:** Liste aller Fibonacci-Zahlen.

  ```
  fibgen :: Int -> Int -> [Int]
  fibgen n1 n2 = n1:fibgen n2 (n1 + n2)

  fibs :: [Int]
  fibs = fibgen 0 1
  fib :: Int -> Int
  fib n = fibs !! n

  fib 10 ⇝ 34
  fib 9 ⇝ 25
  ```

  Bei der zweiten Auswertung ist die neue Generierung der Liste nicht nötig, da Konstanten nur einmal berechnet werden. Diese Technik zum Merken von aufwändigen Berechnungen (*Memorization*) führt allerdings gegebenenfalls zur Speicherverschwendung.

Nachteil der LO-Strategie: Berechnungen können dupliziert werden, z.B. in `double x = x + x`. Aus Effizienzgründen verwendet daher keine Programmiersprache die reine LO-Strategie.

Eine **Optimierung** ist die sogenannte *Lazy-Auswertung*: Statt Termen werden Graphen reduziert, die Variablen des Programms entsprechen Zeigern auf Ausdrücke. Die Auswertung eines Ausdrucks gilt für alle Variablen, die an diesen gebunden sind (*sharing*). Der Mechanismus der *Normalisierung* führt Variablen für jeden Teilausdruck ein.

Die Formalisierung ist die sogenannte *Launchbury-Semantik*. Die Lazy-Auswertung ist optimal bezüglich der Länge der Ableitung, allerdings benötigt sie oft viel Speicher. Deswegen benutzen Sprachen wie ML, Erlang, Scheme oder Lisp die LI-Reduktion.

Layziness ist auch in **strikten** Sprachen möglich! **Idee:** Verwendung von Funktionen zur Verzögerung *(Lazy-Datenstrukturen)*. Wir verwenden dabei den *Unit-Typ*, der nur `()` als einzigen Wert enthält und als Dummy-Argument verwendet wird. **Beispiel:**

```haskell
type List a = () -> ListD a
data ListD a = Nil | Cons a (List a)

nil :: List a
nil () = Nil

cons :: a -> List a -> List a
cons x xs () = Cons a xs

headL :: List a -> a
headL xs = let (Cons x _) = xs () in x

tailL :: List a -> List a
tailL xs = let (Cons _ ys) = xs () in ys

isNil :: List a -> Bool
isNil xs = case xs () of
             Nil -> True
             _   -> False

app :: List a -> List a -> List a
app xs ys = if isNil xs
              then ys
              else cons (head xs)
                        (app (tailL xs) ys)

from n () = cons n (from (n + 1)) ()
```

Dies kann man dann beispielsweise wie folgt anwenden:

```haskell
   head (from 1)
⇝ let (Cons x _) = from 1 () in x
⇝ let (Cons x _) = cons 1 (from (1 + 1)) () in x
⇝ let (Cons x _) = cons 1 (from 2) () in x
⇝ let (Cons x _) = Cons 1 (from 2) in x
⇝ 1
```

2.7 List Comprehensions

Wir haben bereits syntaktischen Zucker für Aufzählungen kennengelernt:

```
[1..4] ⇝ enumFromTo 1 4 ⇝ [1,2,3,4]
[4,2..] ⇝ [4,2,0,-2,-4,...
```

Noch komfortabler sind *List Comprehensions:* Mengen schreibt man in der Mathematik gerne wie folgt:

$$\{(i, j) \mid i \in \{1, \ldots, 3\}, j \in \{2, 3, 4\}, i \neq j\}$$

In Haskell ist eine ähnliche Schreibweise für Listen möglich. Eine Liste, die obiger Menge entspricht, erhält man mit:

```
[ (i, j) | i ← [1..3], j ← [2,3,4], i /= j ]
⇝ [(1,2), (1,3), (1,4), (2,3), (2,4), (3,2), (3,4)]
```

Die allgemeine Form der List Comprehensions ist:

```
[ e | lce₁, lce₂, ..., lceₙ ]
```

Dabei ist `e` ein beliebiger Ausdruck über die in $\texttt{lce}_i$ definierten Variablen, und die Ausdrücke $\texttt{lce}_i$ sind von der folgenden Form:

- Generatoren: `pat ← e`, wobei `e` ein Ausdruck vom Listentyp ist
- Tests: `e` vom Typ `Bool`
- lokale Definitionen: `let pat = e` (ohne das `in e'`)

In einem Ausdruck $\texttt{lce}_i$ können alle Variablen benutzt werden, die in $\texttt{lce}_j$ mit $j < i$ definiert wurden.

Beispiele:

- Anfangsstücke natürlicher Zahlen

  ```
  [ [0..n] | n ← [-1..] ]
  ⇝ [[], [0], [0,1], [0,1,2], ...
  ```

- Konkatenation von Listen:

  ```
  concat :: [[a]] -> [a]
  concat xss = [ x | xs ← xxs, x ← xs ]
  ```

  Beachte, dass die Generatoren geschachtelten Schleifen entsprechen! Das führt ggf. zu **Problemen** bei unendlichen Listen:

```
[ (i,j) | i ← [1..3], j ← [1..] ]
⇝ [(1,1), (1,2), (1,3), ...
```

Hier sollte man die Generatoren vertauschen:

```
[ (i,j) | j ← [1..], i ← [1..3] ]
⇝ [(1,1), (2,1), (3,1), (1,2), ...
```

- Transponieren einer Matrix (gegeben als Liste von Listen):

```
transpose :: [[a]] -> [[a]]
transpose [] = []
transpose ([]:xss) = transpose xss
transpose ((x:xs):xss) =
                (x  : [ h | (h:t) ← xss ])
  : transpose (xs : [ t | (h:t) ← xss ])
```

- Acht-Damen-Problem: Acht Damen sollen auf einem Schachfeld so platziert werden, dass keine eine andere schlagen kann. Kodiert werden die Lösungen hier durch einen Positionsvektor, Beispiel:

[	0,	1,	2,	3,	4,	5,	6,	7	]
	⋆	–	–	–	–	–	–	–	
	–	⋆	–	–	–	–	–	–	
	–	–	⋆	–	–	–	–	–	
	–	–	–	⋆	–	–	–	–	
	–	–	–	–	⋆	–	–	–	
	–	–	–	–	–	⋆	–	–	
	–	–	–	–	–	–	⋆	–	
	–	–	–	–	–	–	–	⋆	

```
queens' :: Int -> [[Int]]
queens' 0 = [[]]
queens' (n+1) = [ q:b | b ← queens' n,
                        q ← [0..7],
                        not (elem q b) ]
          -- korrekt bis auf Diagonalen

diagsOK :: [[Int]] -> [[Int]]
diagsOK = filter (\b -> and
   [ not (j - i == abs (b !! i - b !! j))
```

```
            | i ← [0..7], j ← [i+1..7] ])
            -- Diagonalen korrigieren

queens' :: [[Int]]
queens = diagsOK (queens 8)

head queens ⇝ [3,1,6,2,5,7,4,0]
```

3 Monaden

Monaden sind eine Technik zur (lazy) Sequentializierung in funktionalen Sprachen. Zunächst beschränken wir uns auf Ein-/Ausgabe. Wann soll in Lazy-Sprachen Ein- und Ausgabe stattfinden? **Beispiel:**

```
main = let str = getline in
           putStr str
```

Was ist der Wert von `main`? Es ist der Wert von `putStr str`, eine Möglichkeit wäre `()`. Betrachte aber folgendes Programm:

```
man = getline ++ getline
```

Hier passiert nur eine Eingabe, da `getline` eine Konstante ist. Besser wäre:

```
main = getline () ++ getline ()
```

was aber nicht äquivalent zum folgenden ist:

```
main = let x = getline () in
           x ++ x
```

Ein weiteres Problem:

```
main = let str = getline () ++ getline () in
           head str
```

Hier passiert eine Eingabe, falls die erste Eingabe mindestens ein Zeichen enthält und zwei Ausgaben sonst. Noch problematischer wird folgendes häufig verwendete Szenario:

```
main = let database = readDBFromUser
           request  = readRequestFromUser
       in  lookupDB request database
```

Wegen Laziness wird `database` erst eingelesen, nachdem `request` eingelesen wurde und auch nur soweit, wie nötig um `lookup` durchzuführen; später wird die Datenbank dann ggf. weiter eingelesen für weitere Requests. Dies ist natürlich unpraktisch.

3.1 die IO-Monade

Wichtig für Ein- und Ausgabe sind weniger die Werte, als vielmehr die *Aktionen* und ihre *Reihenfolge*. Die Aktionen müssen sequentialisiert werden. In Haskell gibt es dafür den abstrakten Datentyp `IO ()`, z.B.

```haskell
putChar :: Char -> IO ()
```

IO-Werte sind *First-Order-Werte*, werden aber nur ausgeführt, wenn sie in die IO-Aktion von `main` bzw. interaktive Anfrage eingebaut sind.

Das **Zusammensetzen** von IO-Aktionen geschieht mittels des *Sequenz-Operators:*

```haskell
(>>) :: IO () -> IO () -> IO ()
```

Beispiele:

- Die Funktion

  ```haskell
  main :: IO ()
  main = putChar 'a' >> putChar 'a'
  ```

  verhält sich genauso wie

  ```haskell
  main = let p = putChar 'a' in p >> p
  ```

  oder wie

  ```haskell
  main = let l = repeat (putChar 'a')
         in  l!!1 >> l!!2
  ```

  Die Aktionen werden nur in der `main`-IO-Aktion ausgeführt.

- Ausgabe von Zeilen:

  ```haskell
  putStr :: String -> IO ()
  putStr "" = return ()
  putStr (c:cs) = putChar c >> putStr cs
  ```

  Dabei ist `return ()` eine IO-Aktion, die nichts tut. Andere Definitionsmöglichkeit:

  ```haskell
  putStr = foldr (\c->(putChar c >>)) (return ())
  ```

Für die **Eingabe** hat der Typkonstruktor `IO` zusätzlich einen Typparameter für die Übergabe von Werten an nachfolgende Aktionen, z.B:

```haskell
getChar :: IO Char
return :: a -> IO a
```

Sequenzierung mit Weitergabe erfolgt mittels des *Bind-Operators*:

```haskell
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Beispiele:

```haskell
(getChar >>= putChar) :: IO ()

getline :: IO String
getline = getchar >>= \ c ->
            if c == '\n'
            then return ""
            else getline >>= \ cs -> return (c:cs)

getline >>= putStr
```

Die `do`-**Notation** ist eine vereinfachte Schreibweise, die statt der Operatoren `(>>=)`, `(>>)` nur *Sequenzen* verwendet. Eingeleitet wird sie mit dem Schlüsselwort `do`, die Länge des Blockes bestimmt die Off-Side-Rule. Mögliche Ausdrücke sind:

```haskell
pat ← e
let pat = e -- ohne in
e
```

Beispiel:

```haskell
main = do
  c ← getChar
  putChar c
```

Dies entspricht

```haskell
main = getChar >>= \c -> putChar c
```

Die Variablen sind ab ihrer Einführung gültig. Die Pattern links von ← müssen fehlerfrei matchen, sonst wird ein Laufzeitfehler erzeugt – deswegen verwendet man meist Variablen.

Weitere **Beispiele:**

- `getline` in `do`-Notation

  ```haskell
  getline = do
    c ← getChar
    if c == '\n' then return ""
    else do cs ← getline
            return (c:cs)
  ```

- Ausgabe aller Zwischenergebnisse von `fac`:

```
fac :: Int -> IO Int

fac 0 = do putStr "1"
           return 1
fac (n + 1) = do
    facN ← fac n
    let facNp1 = (n + 1) * facN
    putStr (' ' : show facNp1)
    return facNp1

do x ← fac 7; print x
```

  Beachte: Ausgeführt werden nur Ausdrücke vom Typ `IO ()`, deswegen wäre bei der Auswertung von `fac 7` nur `<<IO Action>>` ausgegeben.

  Die eben verwendete Funktion `print` ist definiert als

```
print :: Show a => a -> IO ()
print = putStrLn . show
```

Allgemeiner **Programmierstil** sollte es sein, möglichst wenig IO-Operationen zu benutzen:

1. Einlesen in IO-Monade.

2. Berechnung auf Eingabe außerhalb der IO-Monade.

3. Ergebnis der Berechnung ausgeben.

Noch ein **Beispiel**: Betrachte ein Programm, was so lange Strings einliest, bis er den String `"."` erhält, danach gibt es die Strings in umgekehrter Reihenfolge wieder aus.

```
main ins = do
  str ← getline
  if str == "."
     then mapIO_ putStrLn ins
     else main (str:ins)

mapIO_ :: (a -> IO()) -> [a] -> IO()
mapIO_ f     []  = return ()
mapIO_ f (x:xs) = do f x; mapIO_ f xs
```

```haskell
-- oder alternativ:
mapIO_ f = foldr (\x as -> f x >> as) (return ())
```

Alternativ:

```haskell
readLines :: IO [String]
readLines = do
  str ← getLine
  if str == "."
     then return []
     else do ins ← readLines
             return (str:ins)
main = do
  ins ← readLines
  mapIO_ putStrLn (reverse ins)
```

Ganz ohne Datenstruktur kommt folgende Lösung aus:

```haskell
main = do str ← getLine
          if str == "."
             then return ()
             else do main; putStrLn str
```

Ohne do-Notation ergibt sich:

```haskell
main = getLine >>= \str ->
  if str == "."
     then return ()
     else main >> putStrLn str
```

3.2 allgemeine Monaden

Untersucht man die IO-Monade und ihre Verknüpfungen genauer, erkennt man die Gültigkeit folgender Gesetze:

```haskell
return () >> m  =  m
m >> return ()  =  m
m >> (n >> o)   =  (m >> n) >> o
```

Hierbei fällt auf, dass `(>>)` assoziativ ist und `return ()` ein neutrales Element ist, d.h. beide ergeben zusammen ein Monoid:

Definition: Ein **Monoid** ein Tripel $(M, \cdot, e)$ mit einer Menge M, einer Verknüpfung $\cdot : M \times M \to M$ und einem Element $e \in M$, so dass gilt:

- für alle $m \in M$ ist $e \cdot m = m \cdot e = m$, und
- für alle $m, n, o \in M$ ist $(m \cdot n) \cdot o = m \cdot (n \cdot o)$

Entsprechend gilt für `return` und `>>=` folgendes:

```
return v >>= \x -> m = m[x/v]
m >>= \x -> return x = m
m >>= \x -> (n >>= \y -> o) =
     (m >>= \x -> n) >>= \y -> o   -- falls x ∉ free(o)
```

Diese Struktur werden wir *Monade* nennen. Der Monaden-Begriff stammt aus der Kategorientheorie, wo er allgemeiner von einem Funktor und zwei natürlichen Transformationen spricht und wo die Monadengesetze mathematisch formuliert sind[3]. Das Wort Monade wurde von Leibniz' entlehnt[4].
Eine Monade ist in Haskell also ein Typkonstruktor m mit den Operationen `return`, `(>>=)` (und `fail`), welche die obigen Eigenschaften erfüllen. Dies definiert man in Haskell als Klasse:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  fail   :: String -> m a
  fail s = error s
  (>>)   :: m a -> m b -> m b
  p >> q = p >>= \_ -> q
```

Die `do`-Notation ist in Haskell syntaktischer Zucker für alle Monaden, `IO` ist Instanz der Klasse `Monad`. Weitere Instanz: `Maybe`.

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

Beispiel: Auswertung arithmetischer Ausdrücke[5]:

---

[3] `http://en.wikipedia.org/wiki/Monad_(category_theory)`

[4] Das Wort *Monade* kommt aus dem Griechischen und bedeutet *Eines*. Zu Leibniz' Idee der Monade: „*Eine Monade – der zentrale Begriff der Leibnizschen Welterklärung – ist eine einfache, nicht ausgedehnte und daher unteilbare Substanz, die äußeren mechanischen Einwirkungen unzugänglich ist. Das gesamte Universum bildet sich in den von den Monaden spontan gebildeten Wahrnehmungen (Perzeptionen) ab. Sie sind eine Art spirituelle Atome, ewig, unzerlegbar, einzigartig.*“ – nach `http://de.wikipedia.org/wiki/Leibniz`

[5] Operatoren als Konstruktoren sind möglich, müssen aber mit einem Doppelpunkt beginnen!

```
data Expr = Expr :+: Expr
          | Expr :/: Expr
          | Num Float
```

Problem ist hier die Vermeidung von Laufzeitfehlern (hier beim Teilen durch Null), Ziel ist eine Funktion `eval` mit folgenden Eigenschaften:

```
eval (Num 3 :+: Num 4) ⇝ Just 7.0
eval (Num 3 :/: (Num (−1) :+: Num 1)) ⇝ Nothing
```

Dazu definieren wir nun die Funktion `eval`:

```
eval :: Expr -> Maybe Float
eval (Num n) = Just n
eval (e1 :+: e2) = case eval e1 of
  Nothing -> Nothing
  Just n1 -> case eval e2 of
    Nothing -> Nothing
    Just n2 -> Just (n1 + n2)
eval (e1 :/: e2) = case eval e1 of
  Nothing -> Nothing
  Just n1 -> case eval e2 of
    Nothing -> Nothing
    Just 0  -> Nothing
    Just n2 -> Just (n1 / n2)
```

Dies geht einfacher und schöner mit `Maybe` als Monade, wobei ein Fehler „durchschlägt“:

```
instance Monad Maybe where
  Nothing >>= k = Nothing
  Just x  >>= k = k x
  return        = Just
  fail _        = Nothing
```

Nun kann man den Auswerter einfacher definieren:

```
eval (Num n) = return n
eval (e1 :+: e2) = do
  n1 ← eval e1
  n2 ← eval e2
  return (n1 + n2)
eval (e1 :/: e2) = do
  n1 ← eval e1
```

```haskell
  n2 ← eval e2
  if (n2 == 0)
    then fail "division by zero!"
    else return (n1 / n2)
```

Eine **andere Sicht** auf den Datentyp `Maybe` ist, dass `Maybe` eine Struktur ist, die höchstens ein Element aufnehmen kann. Wenn man dies auf Listen als Container verallgemeinert, erhält man für Listen ebenso eine Monade:

```haskell
instance Monad [] where
  return x = [x]
  (x:xs) >>= f = f x ++ (xs >>= f)
  []     >>= f = []
  fail _       = []
```

Dann ist

```haskell
[1,2,3] >>= \x -> [4,5] >>= \y -> return (x, y)
⇝ [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Dies ist übersetzt in die `do`-Notation:

```haskell
do x ← [1,2,3]
   y ← [4,5]
   return (x, y)
```

Dies lässt sich noch einfacher mittels List Comprehensions darstellen:

```haskell
[ (x, y) | x ← [1,2,3], y ← [4,5] ]
```

Damit sind die List Comprehensions nur **syntaktischer Zucker** für Listen-Monaden!

Weitere Eigenschaft der Listenmonade ist, dass die leere Liste die *Null* der Struktur ist, da gilt:

```haskell
m >>= \x -> [] = []
[] >>= \x -> m = []
```

Desweiteren gibt es eine ausgezeichnete Funktion `(++)` mit

```haskell
(m ++ n) ++ o = m ++ (n ++ o)
```

Zusammengefasst ergibt sich die Klasse `MonadPlus`:

```haskell
class Monad m => MonadPlus m where
  zero :: m a
  (++) :: m a -> m a -> m a
```

Für Listen ergibt sich dann:

```
instance MonadPlus [] where
  zero = []
  [] ++ ys = ys
  (x:xs) ++ ys = x:(xs ++ ys)
```

Viele Funktionen sind auf `Monad` oder `MonadPlus` definiert, die für konkrete Instanzen verwendbar sind.

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [])
  where mcons p q = p >>= \x ->
                    q >>= \ys ->
                    return (x:ys)

sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Nun kann man beispielsweise benutzen:

```
sequence [getLine, getLine] >>= print
sequence [[1,2],[3,4]]
 ⇝ [[1,3],[1,4],[2,3],[2,4]]
```

Wir können nun `map` auf Monaden definieren:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f a = sequence_ (map f as)
```

Nun verwenden wir:

```
mapM_ putStr (intersperse " " ["Hallo", "Leute"]
 ⇝ Hallo Leute
mapM (\str -> putStr (str ++ ": ") >>
              getLine)
     ["Vorname", "Name"] >>= print
 ⇝ Vorname: Frank
    Name: Huch
    ["Frank","Huch"]
```

Wichtig ist noch folgende Funktion:

```haskell
guard :: MonadPlus m => Bool -> m ()
guard b = if b then return () else Zero
```

3.3 Implementierung der IO-Monade

Frage: Kann auch die IO-Monade in Haskell implementiert werden? Die Antwort ist ja! Die IO-Aktionen nehmen „die Welt“ und liefern die veränderte Welt in Kombination mit dem Ergebnis zurück. Wenn wir dies allerdings selbst implementieren, reicht es aus, eine „Dummy-Welt“ zu verwenden, um die Arbeit der IO-Monade zu simulieren:

```haskell
type IO a = World -> (a, World)
type World = ()
```

Dazu können wir die Monaden-Funktionen definieren als:

```haskell
return :: a -> IO a
return x w = (x, w)
(>>=) :: IO a -> (a -> IO b) -> IO b
(a >>= k) w = case a w of
    (r, w') -> k r w'
```

„Die Welt“ wird hier durchgeschleift, so dass `a` ausgeführt werden muß, bevor `k` ausgeführt werden kann! **Beachte** bei diesem Ansatz: Die Welt darf nicht dupliziert werden!

Anderer Ansatz: Die Sprache Clean stellt über das *Uniqueness*-Typsystem sicher, daß die Welt nicht dupliziert werden kann und gewährleistet so die Sequentialisierung.

Bei unserem Ansatz muß in den primitiven Funktionen die Umwandlung in und von von C-Datenstrukturen durchgeführt werden, bevor die Welt zurückgegeben wird. Das Starten unserer IO-Monade geschieht durch Applikation auf `()`:

```haskell
runIO :: IO() -> Int
runIO a = case a () of
  ((), ()) -> 42
```

Dabei ist der Rückgabetyp von `runIO` eigentlich egal, er ist im Laufzeitsystem verborgen.

Beachte: Für unsere eigene IO-Monade können wir auch folgende Operation definieren:

```haskell
unsafePerformIO :: IO a -> a
unsafePerformIO a = case a () of
                      (r, _) -> r
```

Diese Funktion ist aber **unsicher**, da sie nicht *referentiell transparent* ist:

```haskell
const :: String
const = unsafePerformIO getLine
```

Diese Konstante `c` kann bei jeder Programmausführung einen anderen Wert haben!

3.4 Erweiterung der IO-Monade um Zustände

Die hier vorgestellten Erweiterungen gehen über den Haskell 98-Standard hinaus, sind aber in den meisten Haskell-Implementierungen vorhanden. Im Modul `IOExts` ist der abstrakter Datentyp `data IORef a` definiert, der **polymorphe Speicherzellen** bietet. Das Interface:

```haskell
-- generiert neue IORef
newIORef    :: a -> IO (IORef a)
-- liest aktuellen Wert
readIORef   :: IORef a -> IO a
-- schreibt Wert
writeIORef  :: IORef a -> a -> IO ()
```

Beachte: Die Aktionen werden in der IO-Monade sequentialisiert, aber die Werte in `IORefs` werden lazy berechnet!

Beispiel: Wir implementieren Graphen als Zeigerstrukturen. Zunächst die Datenstruktur:

```haskell
data Node = Node ColorRef [NodeRef]
type NodeRef = IORef Node
type ColorRef = IORef Bool
```

Nun schreiben wir eine Funktion, die den Graphen ausgehend von einem Knoten färbt:

```haskell
paint (Node colorRef succRefs) = do
  color ← readIORef colorRef
  if color then return()
     else do
       writeIORef colorRef True
```

```haskell
        mapM_ (\nodeRef -> do
                  node ← readIORef nodeRef
                  paint node)
              succRefs
```

Beachte: Die Farbe wird destruktiv verändert, und jede Kante des Graphen wird nur einmal besucht. Dies ist ein Beispiel für eine direkte effiziente Implementietung von sequentiellen Algorithmen in einer funktionalen Sprache!

Nachteil ist hier, dass das sequentielle Programm in der funktionalen Schreibweise durch Seiteneffekte schwer verständlich wird. Oft ist es besser, den Graphen als Datenstruktur zu kodieren und eine funktionale Implementierung des Algorithmus' zu wählen (gilt auch für imperative Sprachen).

3.5 Zustandsmonaden

Frage: Ist ein globaler Zustand nur in der IO-Monade möglich?

Beispiel: Nummeriere alle Blätter eines Baumes! Mit `IORefs` ist dies über einen globalen Zustand möglich, rein funktional ist dies auch möglich.

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

number :: Tree a -> Tree (Int, a)
number tree = fst (number' tree 1)
  where number' :: Tree a -> Int ->
                             (Tree (Int, a), Int)
        number' (Leaf x) c = (Leaf (c, x), c+1)
        number' (Node tl tr) c = (Node tl' tr', c2)
          where (tl', c1) = number' tl c1
                (tr', c2) = number' tr c2
```

Dieses Durchreichen des Zählers kann aber unübersichtlich werden, schöner ist es, diesen Zustand in einer Monade zu verstecken, die ihn durchschleift.

```haskell
data State s a = ST (s -> (a, s))
```

Dabei ist `s` der Typ des Zustands und `a` das Monadenergebnis, `s -> (a, s)` ist die Zustandstransition. Nun können wir die Monadenfunktionen definieren:

```haskell
instance Monad (State s) where
-- Zustand unverändert
   return :: a -> State s a
   return r = ST (\s -> (r, s))
```

```haskell
  (>>=) :: State s a -> (a -> State s b) -> State s b
  m >>= f = ST (\s1 ->
                  let (ST strans1) = m
                      (r1, s2)     = strans1 s1
                      (ST strans2) = f r1
                      (r2, s3)     = strans2 s2
                   in (r2, s3))
```

Außerdem:

```haskell
runState :: s -> State s a -> a
runState start (ST trans) = fst (trans start)

update :: (s -> s) -> State s s
update f = ST (\s -> (s, f s))

get :: State s s
get = update id

set :: s -> State s s
set newState = update (const newState)
```

Anwendung:

```haskell
number :: Tree a -> Tree (Int, a)
number tree = runState 1 (number' tree)
  where number' :: Tree a -> State Int (Tree (Int, a))
        number' (Leaf x) = do
            c <- update (+1)
            return (Leaf (c, x))
        number' (Node tl tr) = do
            tl' <- number' tl
            tr' <- number' tr
            return (Node tl' tr')
```

Beachte aber: Die Reihenfolge ist hier (im Gegensatz zu oben!) wichtig, da der Zustand durchgereicht wird (implizites `(>>=)`).

4 Theoretische Grundlagen: Der λ-Kalkül

Grundlage funktionaler Sprachen ist der λ-Kalkül, das 1941 von Church entwickelt wurde. Motivation war, eine Grundlage der Mathematik und mathematischen Logik zu schaffen; in diesem Zusammenhang entstand der Begriff der „Berechenbarkeit". Es zeigte sich eine Äquivalenz zur Turing-Maschine, aus der die **Church'sche These**.

Wichtige Aspekte:

- Funktionen als Objekte
- gebundene und freie Variablen
- Auswertungsstrategien

4.1 Syntax des λ-Kalküls

Wir definieren zunächst die *Signatur* des λ-Kalküls: Var sei eine abzählbare Menge von Variablen, Exp die Menge der Ausdrücke des reinen λ-Kalküls, die definiert werden durch:

$$\begin{array}{rlll} \text{Exp} ::= & v & (\text{mit } v \in \text{Var}) & \text{Variable} \\ | & (ee') & (\text{mit } e, e' \in \text{Exp}) & \text{Applikation} \\ | & \lambda v.e & (\text{mit } v \in \text{Var}, e \in \text{Exp}) & \text{Abstraktion} \end{array}$$

Wir verwenden folgende **Konvention** zur Klammervermeidung:

- Applikation ist linksassoziativ – d.h. xyz statt $((xy)z)$
- Wirkungsbereich von λ so weit wie möglich – d.h. $\lambda x.xy$ statt $\lambda x.(xy)$ und nicht $((\lambda x.x)y)$
- Listen von Parametern: $\lambda xy.e$ statt $\lambda x.\lambda y.e$

Beachte: Es gibt keine Konstanten (vordefinierte Funktionen) oder If-Then-Else-Strukturen, diese sind später im reinen λ-Kalkül definierbar. Der reine λ-Kalkül ist damit minimal, aber universell!

4.2 Substitution

Die **Semantik** des reinen λ-Kalküls ist wie in einer funktionalen Sprache: $\lambda x.e$ ist eine anonyme Funktion wie `\x -> e`, somit gilt $(\lambda x.x)z \rightsquigarrow z$. Damit können Namenskonflikte auftreten:

$$\begin{array}{cccc} (\lambda f.\lambda x.fx)x & \not\rightsquigarrow & \lambda x.xx & \text{Konflikt} \\ (\lambda f.\lambda x.fx)x & \rightsquigarrow & \lambda y.xy & \text{kein Konflikt} \end{array}$$

Definiere dafür zunächst *freie* bzw. *gebundene Variablen* durch:

$$\begin{array}{ll}
\text{free}(v) = \{v\} & \text{bound}(v) = \emptyset \\
\text{free}((ee')) = \text{free}(e) \cup \text{free}(e') & \text{bound}((ee')) = \text{bound}(e) \cup \text{bound}(e') \\
\text{free}(\lambda v.e) = \text{free}(e) \setminus \{v\} & \text{bound}(\lambda v.e) = \text{bound}(e) \cup \{v\}
\end{array}$$

Ein Ausdruck e heißt *geschlossen (Kombinator)*, wenn $\text{free}(e) = \emptyset$ ist.

Wichtig: Bei der Applikation nur Ersetzung der freien Vorkommen der Parameter, wobei Namenskonflikte vermieden werden müssen. Präzisiert wird dies durch die Definition der *Substitution*:

Seien $e, f \in \text{Exp}$, $v \in \text{Var}$. Dann ist $e[v/f]$ („ersetze v durch f in e") definiert durch:

$$\begin{array}{rll}
v[v/f] & = f & \\
x[v/f] & = x & \text{für } x \neq v \\
(ee')[v/f] & = ((e[v/f])(e'[v/f])) & \\
\lambda v.e[v/f] & = \lambda v.e & \\
\lambda x.e[v/f] & = \lambda x.(e[v/f]) & \text{für } x \neq v, x \notin \text{free}(f) \\
\lambda x.e[v/f] & = \lambda y.(e[x/y][v/f]) & \text{für } x \neq v, x \in \text{free}(f), \\
& & \quad y \notin \text{free}(e) \cup \text{free}(f)
\end{array}$$

Somit gilt: Falls $v \notin \text{free}(e)$, so ist $e[v/f] = e$.

Damit ist eine **Präzisierung** des Ausrechnens der Applikation als Reduktionsrelation erreicht.

4.3 Reduktionsregeln

Definiere die *Beta-Reduktion (β-Reduktion)* durch folgende Relation:

$$\rightarrow_\beta \subseteq \text{Exp} \times \text{Exp} \ \text{ mit } \ (\lambda v.e)f \ \rightarrow_\beta \ e[v/f]$$

Beispiel: Nun gilt

$$(\lambda f.\lambda x.fx)x \rightarrow_\beta \begin{cases} \lambda y.xy \\ \lambda z.xz \end{cases}$$

Somit ist $\rightarrow_\beta$ nicht *konfluent*[6]*!*

Aber: Die Namen der Parameter spielen keine Rolle bezüglich der Bedeutung einer Funktion, d.h. die syntaktisch verschiedenen Terme $\lambda x.x$ und $\lambda y.y$ sind semantisch gleichwertig.

[6] Eine Relation $\rightarrow^*$ ist *konfluent*, falls für alle u, v, w mit $u \rightarrow^* v$ und $u \rightarrow^* w$ auch ein z existiert mit $v \rightarrow^* z$ und $w \rightarrow^* z$.

Lösung: Die *Alpha-Reduktion (α-Reduktion)* ist eine Relation

$$\to_\alpha \subseteq \text{Exp} \times \text{Exp} \;\text{ mit }\; \lambda x.e \to_\alpha \lambda y.(e[x/y]) \,(\text{falls } y \notin \text{free}(e))$$

Beispiele:

$$\lambda x.x \to_\alpha \lambda y.y, \quad \lambda y.xy \to_\alpha \lambda z.xz, \quad \lambda y.xy \not\to_\alpha \lambda x.xx$$

Somit ist gilt $e \leftrightarrow_\alpha^* e'$ („e und e' sind α-äquivalent") genau dann, wenn sich e und e' nur durch Namen der Parameter unterscheiden. **Konvention im Folgenden:** Bertrachte α-äquivalente Ausdrücke als gleich, d.h. rechne auf α-Äquivalenzklassen statt auf Ausdrücken.

Die β-Reduktion kann bisher Ausdrücke ausrechnen, allerdings nur außen – deswegen setzen wir die β-Reduktion auf beliebige Stellen in Ausdrücken fort (analog für α-Reduktion):

$$e \to_\beta e' \implies \begin{array}{rrl} & ef & \to_\beta e'f \\ \wedge & fe & \to_\beta fe' \\ \wedge & \lambda x.e & \to_\beta \lambda x.e' \end{array}$$

Eigenschaften der β-Reduktion:

- Die Relation $\to_\beta$ ist konfluent.
- Jeder Ausdruck besitzt höchstens eine Normalform bezüglich $\to_\beta$ (bis auf α-Äquivalenzklassen).

Es gibt allerdings auch Ausdrücke ohne Normalform, dies ist **wichtig** für die Äquivalenz zu Turing-Maschinen! Betrachte:

$$(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx)$$

Die darin enthaltene *Selbstapplikation* $(\lambda x.xx)$ würde in Programmiersprachen zu Typfehlern führen.

Frage: Wie findet man die Normalform, falls sie existiert?

Ein *β-Redex* sei ein Teilausdruck der Form $(\lambda x.e)f$. Dann ist eine *Reduktionsstrategie* eine Funktion von der Menge der Ausdrücke in die Menge der Redexe: Sie gibt an, welcher Redex als nächster reduziert wird. **Wichtige Reduktionsstrategien** sind:

- *Leftmost-Outermost-Strategie (LO, nicht strikt, Normal-Order-Reduction)*: wähle linkesten, äußersten Redex

- *Leftmost-Innermost-Strategie (LI, strikt)*: wähle linkesten, innersten Redex

$$\begin{array}{ll}(\lambda x.z)((\lambda x.xx)(\lambda x.xx)) & \rightarrow_{\text{LI}} (\lambda x.z)((\lambda x.xx)(\lambda x.xx)) \\ (\lambda x.z)((\lambda x.xx)(\lambda x.xx)) & \rightarrow_{\text{LO}} z\end{array}$$

Satz: Ist e' eine Normalform von e, d.h. $e \rightarrow_\beta^* e' \not\rightarrow_\beta e''$, dann existiert eine LO-Ableitung von e nach e'.

Damit berechnet LO jede Normalform, LI manche nicht! LO ist somit *berechnungsstärker* als LI.

Ein wichtiger Aspekt (für Optimierungen, Transformation, Verifikation) ist die **Äquivalenz von Ausdrücken**[7] Intuitiv sind e und e' genau dann äquivalent, wenn e überall für e' eingesetzt werden kann, ohne das Ergebnis zu verändern.
Beispiele:

- $\lambda x.x$ ist äquivalent zu $\lambda y.y$
- $\lambda f.\lambda x.((\lambda y.y)f)((\lambda z.z)x)$ ist äquivalent zu $\lambda f.\lambda x.fx$

Wünschenswert wäre es, die Äquivalenz durch syntaktische Transformationen nachweisen zu können. α- und β-Äquivalenz sind dafür aber nicht ausreichend, betrachte folgendes Beispiel (in Haskell-Notation): $(+1)$ ist äquivalent zu $\lambda x.(+)1x$, denn bei der Anwendung auf ein Argument z gilt:

$$(+1)z = (+)1z = (\lambda x.(+)1x)z$$

Doch beide Terme sind weder α- noch β-äquivalent. Daher definieren wir noch eine *Eta-Reduktion (η-Reduktion)* als Relation:

$$\rightarrow_\eta \subseteq \text{Exp} \times \text{Exp} \;\text{ mit }\; \lambda x.ex \rightarrow_\eta e \,(\text{falls } x \notin \text{free}(e))$$

Weitere Sichtweisen der η-Reduktion:

- Die η-Reduktion ist eine vorweggenommene β-Reduktion – $(\lambda x.ex)f \rightarrow_\beta ef$, falls $x \notin \text{free}(e)$ ist.
- Extensionalität: Funktionen sind äquivalent, falls sie gleiche *Funktionsgraphen* (Menge der Argument-Wert-Paare) haben. Beachte: Falls $fx \leftrightarrow_\beta^* gx$[8] gilt, dann ist $f \leftrightarrow_{\beta,\eta}^* g$ (mit $x \in \text{free}(f) \cap \text{free}(g)$), da gilt:

$$f \leftarrow_\eta \lambda x.fx \leftrightarrow_\beta^* \lambda x.gx \rightarrow_\eta g$$

[7] Wobei wir im Folgenden nur Äquivalenz auf terminierenden Ausdrücken betrachten, sonst ist nicht klar, was man unter „Ergebnis“ versteht.

[8] Unter $u \leftrightarrow_\beta^* v$ verstehen wir: es gibt ein w, so dass $u \rightarrow_\beta^* w$ und $v \rightarrow_\beta^* w$.

Es gibt auch noch eine weitere Reduktion, die *Delta-Reduktion (δ-Reduktion)*, die das Rechnen mit vordefinierten Funktionen ermöglicht, wie zum Beispiel $(+)\ 1\ 2 \rightarrow_\delta 3$. Diese vordefinierten Funktionen sind nicht notwendig, da sie im reinen λ-Kalkül darstellbar sind.

Zusammenfassung der Reduktionen im λ-Kalkül:

- α-Reduktion: Umbenennung von Parametern
- β-Reduktion: Funktionsanwendung
- η-Reduktion: Elimination redundanter λ-Abstraktionen
- δ-Reduktion: Rechen mit vordefinierten Funktionen

Beachte: Die „Semantik" des Ausdrücke des Lambda-Kalküls ist *nur* durch Relationen definiert!

4.4 Datenobjekte im reinen λ-Kalkül

Datentypen sind Objekte mit Operationen darauf, Idee hier: Stelle die Objekte durch geschlossene λ-Ausdrücke dar und definiere passende Operationen.

Betrachte den Datentyp der **Wahrheitswerte**: Objekte sind `True` und `False`, die wichtigste Operation ist die `If-Then-Else`-Funktion:

$$\texttt{if_then_else}(b, e_1, e_2) = \begin{cases} e_1 & \text{falls} \quad b = \texttt{True} \\ e_2 & \text{falls} \quad b = \texttt{False} \end{cases}$$

Daher sind Wahrheitswerte nur Projektionsfunktionen:

$$\begin{aligned} \texttt{True} \equiv \quad & \lambda x.\lambda y.x \quad \text{erstes Argument nehmen} \\ \texttt{False} \equiv \quad & \lambda x.\lambda y.y \quad \text{zweites Argument nehmen} \end{aligned}$$

Nun lässt sich eine `If-Then-Else`-Funktion definieren durch

$$\texttt{Cond} \equiv \lambda b.\lambda x.\lambda y.bxy$$

Beispiel:

$$\begin{aligned} \texttt{Cond True } e_1e_2 &\equiv (\lambda bxy.bxy)(\lambda xy.x)e_1e_2 \rightarrow^3_\beta (\lambda xy.xe_1e_2) \rightarrow^2_\beta e_1 \\ \texttt{Cond False } e_1e_2 &\equiv (\lambda bxy.bxy)(\lambda xy.y)e_1e_2 \rightarrow^3_\beta (\lambda xy.ye_1e_2) \rightarrow^2_\beta e_2 \end{aligned}$$

Nun kodieren wir die **natürlichen Zahlen**: Betrachte die *Church-Numerals*, die jede Zahl $n \in \mathbb{N}$ als Funktional darstellen, das eine Funktion f genau n

mal auf ein Argument anwendet:

$$\begin{aligned}
0 &\equiv \lambda f.\lambda x.x \\
1 &\equiv \lambda f.\lambda x.fx \\
2 &\equiv \lambda f.\lambda x.f(fx) \\
3 &\equiv \lambda f.\lambda x.f(f(fx)) \\
n &\equiv \lambda f.\lambda x.f^n x
\end{aligned}$$

Nun definieren wir Operationen:

- Die wichtigste Operation ist die Nachfolgefunktion `succ`:

  $$\texttt{succ} \equiv \lambda n.\lambda f.\lambda x.nf(fx)$$

  Beispiel: Nachfolger von Eins:

  $$\begin{aligned}
  \texttt{succ}\ 1 &\equiv (\lambda n.\lambda f.\lambda x.nf(fx))(\lambda f.\lambda x.fx) \\
  &\rightarrow_\beta \lambda f.\lambda x.(\lambda f.\lambda x.fx)f(fx) \\
  &\rightarrow_\beta \lambda f.\lambda x.(\lambda x.fx)(fx) \\
  &\rightarrow_\beta \lambda f.\lambda x.f(fx) \equiv 2
  \end{aligned}$$

- Test auf Null:

  $$\texttt{is_Null} \equiv \lambda n.n(\lambda x.\texttt{False})\texttt{True}$$

  Beispiel: Teste Null auf Null:

  $$\begin{aligned}
  \texttt{is_Null}\ 0 &= (\lambda n.n(\lambda x.\texttt{False})\texttt{True})(\lambda f.\lambda x.x) \\
  &\rightarrow_\beta (\lambda f.\lambda x.x)(\lambda x.\texttt{False})\texttt{True} \\
  &\rightarrow_\beta (\lambda x.x)\texttt{True} \\
  &\rightarrow_\beta \texttt{True}
  \end{aligned}$$

4.5 Mächtigkeit des Kalküls

Ist der reine λ-Kalkül also berechnungsuniversell? Ja, denn auch Rekursion ist darstellbar, beachte den **Fixpunktsatz**:

Satz: Zu jedem $F \in \text{Exp}$ gibt es einen Ausdruck X mit $FX \leftrightarrow_\beta X$.

Beweis: Wähle zum Beispiel $X = YF$ mit *Fixpunktkombinator* Y:

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Man kann noch zeigen, daß gilt: $YF \leftrightarrow_\beta F(YF)$, siehe Übung.

Der reine λ-Kalkül ist minimal, aber berechnungsuniversell – er besitzt aber eher theoretische Relevanz (Barendregt 1984).

4.6 Angereicherter λ-Kalkül

Für funktionale Programmiung ist aber eher ein angereicherter λ-Kalkül relevant, bei dem Konstanten (vordefinierte Objekte und Funktionen), `let`, `if-then-else` und ein Fixpunktkombinator hinzugenommen werden.

Die **Syntax** des angereicherten λ-Kalküls:

Exp ::=	v	Variable
|	k	Konstantensymbol
|	(ee')	Applikation
|	`let` $v = e$ `in` e'	lokale Definitionen
|	`if` e `then` e_1 `else` e_2	Alternative (manchmal auch `case`)
|	$\mu v.e$	Fixpunktkombinator

Als operationale **Semantik** benutzen wir zusätzlich zur α-, β- und η-Reduktion noch folgende Regeln:

- δ-Reduktion für Konstanten, z.B. $(+)\,21\,21 \to_\delta 42$
- `let` $v = e$ `in` $e' \to e'[v/e]$
- `if True then` e_1 `else` $e_2 \to e_1$
- $\mu v.e \to e[v/\mu v.e]$

Beispiel: Fakultätsfunktion:

$$\texttt{fac} \equiv \mu f.\lambda x.\ \texttt{if}\ ((==)\ x\ 0)\ \texttt{then}\ 1\ \texttt{else}\ ((*)\ x\ (f\ ((-)\ x\ 1)))$$

Der angereicherte λ-Kalkül bildet die Basis der **Implementierung funktionaler Sprachen** (Peyton, Jones 1987):

1. Übersetze Programme in diesen Kalkül (Core-Haskell im `ghc`):
   - Pattern-Matching wird in `if-then-else` (oder `case`) übersetzt,
   - $f\,x_1\ \ldots\ x_n = e$ wird übersetzt zu $f = \lambda x_1 \ldots x_n.e$,
   - für rekursive Funktionen führt man Fixpunktkombinatoren ein,
   - ein Programm entspricht einer Folge von `let`-Deklarationen und einem auszuwertenden Ausdruck.
2. Implementiere den Kalkül durch eine spezielle abstrakte Maschine, z.B. durch die SECD-Maschine von Landin in `ghc` oder durch eine Graphreduktionsmaschine.

Weitere Anwendungen des erweiterten λ-Kalküls sind die denotationelle Semantik und die Typisierung (getypte λ-Kalküle).

5 Parserkombinatoren

Sei eine *kontextfreie Grammatik* $G = (N, \Sigma, P, S)$ gegeben. Ein Programm, welches für alle $w \in \Sigma^*$ entscheidet, ob $w \in L(G)$ ist, heißt *Parser* für G. Zusätzlich sollte ein Parser noch eine Ausgabe liefern, die abstrakte Informationen über die Eingabe enthält (z.B. Linksableitung des Wortes, einen abstrakten Syntaxbaum (AST) etc.).

Zum **Implementieren von Parsern** gibt es unterschiedliche Ansätze:

- Tools wie YACC (oder Happy für Haskell) generieren einen Parser aus einer Grammatik.
- *rekursive Abstiegsparser* wurden für PASCAL von Wirth umgesetzt, auch der XML-Parser im Projekt zur Vorlesung war ein solcher Parser
- *Parserkombinatoren* erlauben es, den Parser gleichzeitig mit der Grammatik aufzustellen

5.1 Kombinatorbibliothek für Parser

Abstrakte Idee der Kombinatoren:

```haskell
type Parser s a = [s] -> [(a,[s])]
```

Dabei ist `[s]` links die Eingabetokenfolge, `a` rechts das Ergebnis und `[s]` die Resttokenfolge.

Wir definieren jetzt zunächst einige elementare Parser:

- `pSucceed`: Liefert einen Parser, der immer matcht, Eingabe wird nicht verbraucht, v ist Ergebnis.

  ```haskell
  pSucceed :: a -> Parser s a
  pSucceed v ts = [(v, ts)]
  ```

- `pFail`: Ein Parser, der immer fehlschlägt:

  ```haskell
  pFail :: Parser s a
  pFail ts = []
  ```

- `pPred`: Liefert einen Parser, der aktuelles Token testet und es als Ergebnis liefert:

```haskell
pPred :: (s -> Bool) -> Parser s s
pPred pred (t:ts) | pred t     = [(t, ts)]
                  | otherwise = []
pPred pred []                  = []
```

- `pSym`: Test auf Symbol:

```haskell
pSym :: Eq s => s -> Parser s s
pSym t = pPred (t ==)
```

Nun können wir **Operatoren** zum Kombinieren von (u.a. obigen) Parsern definieren. Zunächst legen wir die Präzedenz der Operatoren fest (dies muß am Modulanfang geschehen!):

```haskell
infixl 3 <|>
infixl 4 <*>
```

Nun definieren wir unsere ersten *Kombinatoren*:

- Der Operator `(<|>)` testet zuerst mit Parser `p`, wenn dieser fehlschlägt, dann mit Parser `q` auf der gleichen Eingabe (das Parsen geschieht also lazy).

```haskell
(<|>) :: Parser s a -> Parser s a -> Parser s a
p <|> q ts = p ts ++ q ts
```

- Der Operator `(<*>)` führt die Parser `p` und `q` hintereinander aus und wendet Ergebnis von `p` auf Ergebnis von `q` an.

```haskell
(<*>) :: Parser s (b -> a) -> Parser s b
                           -> Parser s a
(p <*> q) ts = [ (pv qv, ts2) |
                 (pv, ts1) ← p ts,
                 (qv, ts2) ← q ts1]
```

Beispiel: Parser für die Sprache: $\{a^n b^n \mid n \in \mathbb{N}\}$ mit Grammatik $S \rightarrow aSb \mid \varepsilon$:

```haskell
anbn :: Parser Char Int
anbn = pSucceed (\_ n _ -> n + 1)
       <*> pSym 'a' <*> anbn <*> pSym 'b'
       <|> pSucceed 0
```

Dieser Parser wertet dann folgendermaßen aus:

```
anbn "aabb"  ⇝  [(2, ""), (0, "aabb")]
anbn "abb"   ⇝  [(1, "b"), (0, "abb")]
anbn "aab"   ⇝  [(0, "aab")]
```

Das *korrekte* Ergebnis ist dasjenige, wo die Eingabe vollständig verbraucht ist. **Beachte** dabei: Kombinatoren liefern immer den längstmöglichen Parse zuerst.

Vorteile dieser Technik:

- Die Grammatik wird direkt als Programm geschrieben.
- Es ist eine sehr große Sprachklasse erkennbar – LL(∞), d.h. die Sprachen, die mit unendlichem Look-Ahead entscheidbar sind. [9]
- Die Grammatik kann mit Hilfe von Haskell-Funktionen aufgebaut werden (z.B. `foldr` auf Parsern etc.).
- Die Grammatik kann dynamisch (z.B. in Abhängigkeit vom Parse-Ergebnis) aufgebaut werden.
- Es können neue (spezielle) Kombinatoren definiert werden.

Weitere Kombinatoren:

```
infixl 4 <$>, <*, *>
infixl 2 `opt`

(<$>) :: (b -> a) -> Parser s b -> Parser s a
f <$> p = pSucceed f <*> p

(<*) :: Parser s a -> Parser s b -> Parser s a
p <* q = (\ x _ -> x) <$> p <*> q

(*>) :: Parser s a -> Parser s b -> Parser s b
p *> q = (\ _ x -> x) <$> p <*> q

(<**>) :: Parser s a -> Parser s (a -> b)
                                        -> Parser s b
p <**> q = (\ x f -> f x) <$> p <*> q
```

[9] Dabei gilt $\bigcup_{K \in \mathbb{N}} \mathrm{LL}(K) \subsetneq \mathrm{LL}(\infty)$.

```
pFoldr :: (a -> b -> b) -> b -> Parser s a
                                      -> Parser s b
pFoldr op e p = op <$> p <*> (pFoldr op e p) `opt` e

`opt` :: Parser s a -> a -> Parser s a
p `opt` v = p <|> pSucceed v

pList :: Parser s a -> Parser s [a]
pList p = pFoldr (:) []

-- Parser für Separator
pListSep :: Parse s b -> Parser s a -> Parser s [a]
```

Jetzt lässt sich unser Beispiel wie folgt schreiben:

```
anbn = (+1) <$>
        (pSym 'a' *> anbn <* pSym 'b')
        <|> pSucceed 0
```

Zusätzlich ist es wünschenswert, **kontext-sensitive** Eigenschaften testen zu können. **Beispiel:** Parsen von zwei gleichen Zeichen: $L = \{xx \mid x \in \Sigma\}$. Das Aufzählen aller Möglichkeiten ist hier nicht praktikabel, besser ist die Hinzunahme eines Tests:

```
check :: (a -> Bool) -> Parser s a -> Parser s a
check pred p ts = filter (pred.fst) (p ts)

doubleChar :: Parser Char Char
doubleChar = fst <$>
             check (uncurry (==))
             ((,) <$> pAnySym <*> pAnySym)

pAnySym :: Parser Char Char
pAnySym = pPred (const True)
```

Alternativer Ansatz: Wir verwenden nun das Ergebnis eines Parsers als Parameter für den nächsten Parser:

```
(<->>) :: Parser s a -> (a -> Parser s b)
                                    -> Parser s b
(p <->> qf) ts = [res | (pv, ts1) ← p ts
                        res       ← qf pv ts1]

doubleChar = pAnySym <->> pSym
```

Beispiel: Wir entwerfen wieder einen Parser für XML:

```
data XML = Tag String [XML]
         | Text String

pXMLs = (:) <$> pXML <*> pXMLs
       <|> pSucceed []

pOTag  = pSym '<' *> pIdent <* pSym '>'
pOCTag = pSym '<' *> pIdent <* pSym '/' <* pSym '>'
pCTag tag = const () <$> (pSym '<' *>
              pSym '/' *> pCheckIdent tag <* pSym '>')
```

Die Funktion `pCTag` soll wie folgt verwendet werden können:

```
> pCTag "a" "</a>bc"
[((),"bc")]
> pCTag "a" "</b>bc"
[]
```

Dazu definieren wir:

```
pIdent = (:) <$> pPred (/= '>') <*> pIdent
        <|> pSucceed ""

pCheckIdent :: String -> Parser Char ()
pCheckIdent ""     = pSucceed ()
pCheckIdent (c:cs) = pSym c *> pCheckIdent cs

pText = (:) <$> pPred ('<' /=) <*> pText
       <|> (:[]) <$> pPred ('<' /=)

pXML = flip Tag [] <$> pOCTag
      <|> pOTag
          <->> \ tag -> (\ xmls -> Tag tag xmls)
          <$> pXMLs <* pCTag tag
      <|> Text <$> pText
```

5.2 Monadische Parserkombinatoren

Frage: Lassen sich Parser als Monaden darstellen? Ja! Bisher war `Parser s a` jedoch ein Typsynonym, für eine Instanz der Klasse `Monad` benötigen wir einen Typkonstruktor:

```
data Parser s a = Parser ([s] -> [(a,[s])])

instance Monad (Parser s) where
  (Parser p) >>= qf =
     Parser (\ts -> [res | (pv,ts1) ← p ts,
                           let (Parser q) = qf pv,
                           res ← q ts1])
  return v          = Parser (\ts -> [(v,ts)])
```

Nun definieren wir `pPred` für die neue Parser-Definition:

```
pPred :: (s -> Bool) -> Parser s s
pPred pred = Parser (\ts ->
   case ts of t:ts' -> if pred t
                       then [(t,ts')]
                       else []        )
```

Parser lassen sich aber sogar der Klasse `MonadPlus` zuordnen:

```
instance MonadPlus (Parser s) where
  (Parser p) ++ (Parser q) =
         Parser (\ts -> p ts ++ q ts)
  zero = Parser (\ts -> [])

runParser :: Parser a -> [s] -> [(a,[s])]
runParser (Parser p) ts = p ts
```

Nun läßt sich beispielsweise der Parser für die Sprache $a^n b^n$ folgendermaßen definieren:

```
anbn = do
  pSym 'a'
  x ← anbn
  pSym 'b'
  return (x+1)
  ++
  return 0
```

Diese Implementierung von Parsern ist in der Parsec-Bibliothek[10] vorhanden.

[10] auf `www.haskell.org` unter „Libraries and Tools for Haskell“ ist Parsec zu finden, oder direkt auf `www.cs.uu.nl/ daan/parsec.html` – Parsec ist standardmässig in den ghc-Libraries enthalten

5.3 Anmerkung zum Parsen

Ein **Problem** beim Top-Down-Parsen ist die *Linksrekursion* in Grammatiken wie zum Beispiel

$$s \rightarrow s\,a \mid \epsilon$$

Die folgende Lösung mit Parserkombinatoren würde nicht terminieren:

```
s = (+1) <$> s <* pSym 'a'
    <|> pSucceed 0
```

Lösung: Übersetze die Linkrekursion in eine Rechtsrekursion:

$$s \rightarrow a\,s \mid \epsilon$$

Das allgemeine Vorgehen wird in der *Compilerbau-Vorlesung* behandelt!

Zur **Optimierung** bei Parsern der Operator `<|>` ist oft ineffizient, da die Alternativen jeweils Speicher benötigen. Wir definieren daher eine deterministische Variante `<||>`:

```
<||>: Parser s a -> Parser s a -> Parser s a
(p <||> q) ts = case p ts of [] -> q ts
                             rs -> rs
```

Analoges kann man für monadische Parserkombinatoren definieren.

Weitere Features der Parserkombinatoren in Kurzform:

- Exceptions zur Fehlerbehandlung
- Zustandsmonaden („globaler“ Zustand während des Parsens, Parsec)
- Zeileninformationen zur Fehlergenerierung
- Korrektur der Eingabe (Swierstra, 1999)

6 Debugging

Haskell hat viele Vorteile (z.B. Laziness), die aber das Finden von Fehlern erschweren. In imperativen Sprachen ist ein einfaches **Debugging** möglich mittels `printf`-Anweisungen; in Haskell ist dies möglich mit der unsicheren Funktion[11]

```
import IOExts
trace :: String -> a -> a
trace str x = unsafePerformIO $ do
      putStr str
      return x
```

Semantik ist die Identität, aber sobald die Berechnung von `trace` angestossen wird, wird der String als Seiteneffekt ausgegeben!

Beispiel:

```
> trace "Hallo" (3+4)
Hallo 7

> take 4 (map (trace "*") [1..])
[*1,*2,*3,*4]
```

Oft ist man auch an **Werten** interessiert:

```
traceAndShow :: Show a => a -> a
traceAndShow x = trace (show x) x

> take 4 (map traceAndShow [1..])
[1 1,2 2,3 3,4 4]
```

Probleme dabei sind die Vermischung der Debug-Ausgabe mit der Ergebnisausgabe (Lösung: Trace-Datei) und die Abhängigkeit der Ausgabe von der Auswertungsreihenfolge (beachte Laziness!)

Beispiel:

```
x = let l = map traceAndShow
                [1..] in sum           (take 4 l)
y = let l = map traceAndShow
                [1..] in sum (reverse (take 4 l))
> x
```

[11]Dabei ist $ ein Operator, der die niedrigste Präzedenz hat und somit die Präzedenz der Funktionsanwendung umkehrt: `f (g x) = f $ g x`

```
1 2 3 4 10
> y
4 3 2 1 10
```

Weitere Probleme:

- Die Funktion `traceAndShow` zerstört zum Teil die Laziness: `traceAndShow [1..]` terminiert nicht, sobald es angestossen wird! Durch die dann strikte Auswertung entsteht zum einen ein Effizienzverlust, zum anderen setzen z.B. einige Parserkombinatoren die Laziness voraus.
- Man erhält keine Informationen über den Fortschritt der Auswertung.
- Es ist nur die Beobachtung von Werten der Klasse `Show` möglich, z.B. keine Funktionen!

6.1 Debuggen mit Observations (Hood)

Idee ist hier, Werte wie mit `traceAndShow` zu beobachten, aber „`_`" für nicht benötigte (d.h. auch noch nicht ausgewertete) Teilstrukturen anzuzeigen. Dazu wird die Ausgabe bis zum Programmende (auch Strg-C) verzögert, ein zusätzlicher String-Parameter wird benutzt zur Unterscheidung der Beobachtung.

Beispiel:

```
> :l Observe
> take 2 (observe "List" [1..])
[1,2]
>>>>>>> Observations <<<<<<
List (1 : 2 : _)

> observe "List" [1,2,3,4,5]!!3
4
>>>>>>> Observations <<<<<<
List (_ : _ : _ : 4 : _ : [])
```

Verwendung: In einem Programm kann man mittels `import Observe` das Modul importieren und dann `(Observe name e)` wie `traceAndShow` benutzen.

Alle Datentypen können observiert werden. Funktionen werden durch den „benutzten Teil" des Funktionsgraphen dargestellt. **Beispiel:**

```
> map (observe "inc" (+1)) [1..3]
[2, 3, 4]
>>> Observations <<<
inc
  {\ 3 -> 4,
   \ 2 -> 3,
   \ 1 -> 2
   }
```

Observations von Funktionen sind meist wichtiger als die der Datenstrukturen, da sie die Funktionalität wiederspiegeln.

Schreibt man

```
f = observe "f" f'
f' p11 .. p1n = e1
  ⋮
f' pm1 .. pmn = em
```

So werden auch alle rekursiven Aufrufe observiert. Besser ist daher:

```
f = observe "f" f'
f' p11 .. p1n = e1[f/f']
  ⋮
f' pm1 .. pmn = em[f/f']
```

Außerdem möglich sind:

- Observations von IO-/Zustandsmonaden
- Definition von zusammenhängenden Observations.

6.2 Implementierung von Observations für Daten

Zunächst: Definition einer Datenstruktur zur Repräsentation (teil-)ausgewerteter Daten.

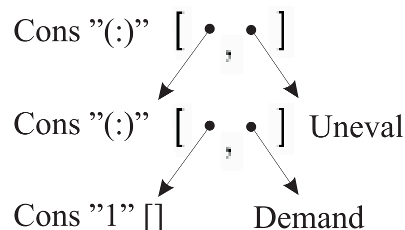
```
data EvalTree = Cons String [EvalRef]
              | Uneval      -- entspricht '_'
              | Demand      -- entspricht '!',
                            -- abgebrochene Berechnung

type EvalRef = IORef EvalTree
```

Beispiel für abgebrochene Berechnung:

```
> observe "List" [1, fac (-1)]
[1,
<< Ctrl-C >>
>>> Observations <<<
List (1:!:_)
```

Der Wert `_:1:!` wird repräsentiert durch



Zuerst implementieren wir die nötigen Funktionen für eine Datenstruktur:

```
data Tree = Empty | Node Tree Tree

oTree :: Tree -> EvalRef -> Tree
oTree Empty ref = unsavePerformIO $ do
      mkEvalTreeCons "Empty" ref 0
      return Empty
oTree (Node tl tr) ref = unsafePerformIO $ do
      [tlRef, trRef] ← mkEvalTreeCons "Node" ref 2
      return (Node (oTree tl tlRef)               -- (*)
                   (oTree tr trRef))              -- (*)

mkEvalTreeCons :: String -> IORef -> Int
                                    -> IO [EvalRef]
mkEvalTreeCons consName ref n = do
      refs ← mapM (const (newIORef Uneval)) [1..n]
      writeIORef ref (Cons consName refs)
      return refs
```

Es soll nun wie folgt ausgewertet werden:

```
isNode (Node _ _) = True
isNode _          = False

> isNode (observe "tree" (Node Empty Empty))
 ⇝ isNode (oTree ref (Node Empty Empty))
```

```
  -- wobei ref auf folgendes zeigt:
  -- Cons "Node" [ref1 ⇝ Uneval, ref2 ⇝ Uneval]
```

Verallgemeinerung auf beliebige Datentypen:

```
class Observe a where
      obs :: a -> EvalRef -> a

instance Observe Tree where
      obs = oTree

observer :: Observe a => a -> EvalRef -> a
observer x ref = unsafePerformIO $ do
      writeIORef ref Demand
      return (obs x ref)
```

Zusätzlich muss `oTree` in (*) oben durch `observer` ersetzt werden. Es wird `Demand` geschrieben, bevor die Berechnung gestartet wird. Die Reihenfolge ist insgesamt wie folgt:

$$\text{Uneval} \xrightarrow{\text{Anfrage des Werts}} \text{Demand} \xrightarrow{\text{Kopfnormalform}} \text{Cons}$$

Das **Speichern** aller Observations geschieht in einer globalen Variablen:

```
global :: IORef [IO ()]
global = unsafePerformIO (newIORef [])

observe :: Observe a => String -> a -> a
observe label x = unsafePerformIO $ do
      ref ← newIORef Uneval
      modifyIORef global
        (putStrln (label ++ "\n"
             ++ (replicate (length label '_'))
         >> showEvalTreeRef ref >>= putStrLn) :)
      return (observer x ref)

runO :: IO () -> IO ()
runO io = do
      writeIORef global []
      io     -- alternativ catch io (\_ -> printObs)
             -- aus Modul Control_Exception
      printObs
```

```
printObs = do
      putStrLn ">>> Observations <<<"
      obs ← readIORef global
      sequence obs
```

Wie definiert man nun komfortabel Instanzen der Klasse `Observe`? **Beispiel:**

```
instance Observe a => Observe [a] where
      obs (x:xs) = o2 (:) "(:)" x xs
      obs []     = o0 []  "()"
```

Hierbei sind folgende Funktionen definiert:

```
o0 :: a -> String -> EvalRef -> a
o0 cons consName ref = unsafePerformIO $ do
      mkEvalTreeCons consName ref 0
      return cons

o2 :: (Observe a, Observe b) => (a -> b -> c)
                  -> String -> a -> b -> EvalRef -> c
o2 cons consName vA vB ref = unsafePerformIO $ do
      [aRef, bRef] ← mkEvalTreeCons consName ref 2
      return (cons (observer vA aRef)
                   (observer vB bRef))
```

6.3 Implementierung von Observations für Funktionen

Problem: Betrachte die folgende Auswertung:

```
main let f = observe "inc" (+1) in
     f 3 + f 1

> main
6
>>> Observations <<<
inc
  {\3 -> 4
  ,\1 -> 2
  }
```

Beachte: `f` ist hier eine Konstante, der zugehörige Observer wird nur einmal berechnet. Eine Umsetzung wie bei Konstruktoren würde zum Überschreiben der letzten Funktionsanwendung führen!

Lösung: Behandle „->“ als dreistelligen Konstruktor im `EvalTree`:

```
Cons "->" [Cons "3" [], Cons "4" [],
           Cons "->" [Cons "1" [], Cons "2" [],
                      Uneval]]
```

```
instance (Observe a, Observe b) =>
                            Observe (a -> b) where
  obs f ref x = unsafePerformIO $ do
    nextRef <- nextFunRef ref
    [aRef, bRef, cRef] <-
                     mkEvalTreeCons "->" nextRef 3
    return (observer (f (observer x aRef)) bRef)
  where nextFunRef ref = do
    evalTreeNode <- readIORef ref
    case evalTreeNode of
      Cons _ [_, _, nextRef] -> nextFunRef nextRef
      _                      -> return ref

showEvalTree :: EvalTree -> IO String
showEvalTree (Cons cons []) = return cons
showEvalTree (Cons "->" ts) = do
  args <- mapM showEvalTreeRef ts
  case args of
    [aa, ab, nextFuns] ->
      return ("{" ++ aa ++ "->" ++ ab ++ "}" ++
        if nextFuns == "_" then ""
                           else '\n':nextFuns)
```

Statt der Verwendung einer Datenstruktur ist auch eine Observation-Datei möglich. Die ist leider weniger dynamisch, und die Anzeige muss aus einer Datei generiert werden, dies ist aber auch später möglich. Vorteil ist aber, daß der zeitliche Ablauf erkennbar ist (was aufgrund der Lazyness jedoch nicht unbedingt hilfreich ist).

Der vorgestellte Ansatz geht zurück auf Hood – er ist in Hugs fest eingebaut, aber zum Teil fehlerhaft). *Vorteile* des Ansatzes von Hood sind, daß er einfach zu benutzen ist, und sowohl die Laziness von Programmen erhält als auch bei Spracherweiterungen (keine Programmtransformation) funktioniert.

6.4 Andere Ansätze

Andere Ansätze arbeiten meist Trace-basiert, d.h. die Berechnung wird aufgezeichnet. Ein Trace wird später mit speziellen „Viewers“ analysiert. Wichtigstes Tool ist Hat.

Beispiel: Fehlerhaftes Insertionsort:

```
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x <= y     = x:ys
                | otherwise = x:insert x y s

main = putStrLn (sort "program")

> main
"agop"
```

Eine Möglichkeit ist nun `hat-observe` zur Beobachtung von Top-level-Funktionen:

```
> hat-observe sort
sort "program" = "agop"
```

```
  ⋮
sort "am" = "a"
sort "m" = "m"
sort "" = ""

> hat-observe insert
insert 'a' "m" = "a" -- Fehler
```

Vorteile: Es sind hier keine Observe-Annotationen notwendig! Zusätzlich ist ein Pattern-Matching auf Argumente/Ergebnisse möglich!

Einige weitere Views sind:

- `hat-trace` mit einer Bottom-up-Sicht der Berechnung: Man kann erfragen, wo die Werte (insbesondere auch Fehler) herkommen:

  ```
  agop \n
  ← putStrLn "agop"
  ← insert "p" "agor" | False -- Fehler
  ```

- `hat-detect` (deklaratives Debugging wie Freja)

  ```
  sort [3, 2, 1]
    sort (3:2:1:[]) = 3:3:3:[]? n
    insert 1 [] = 1:[] ? y
    insert 2 (1:[]) = 2:2:[]? n
    insert 2 [] = 2:[] ? y
  Error located!
  Bug found: "insert 2 (1:[]) = 2:2:[]"
  ```

  Nachteil: man verliert oft den Überblick, und falsche Antworten (bei großen Datenstrukturen) führen zu falschen Fehlerpositionen. Manchmal ist dies wenig Zielgerichtet, oft kann man besser mit seiner Intuition arbeieten.

- `hat-stack` erlaubt die Anzeige des Laufzeitkellers für eine strikte Berechnung zu einem Laufzeitfehler/Ctrl-C – dies entspricht der Java-Exception-Sicht.

- `hat-explore` (neu) ist ein Standarddebugger, der Innermost-Abstieg in beliebigen Argumenten ermöglicht (zumindest so weit, wie die Berechnung ausgeführt wurde). Zusätzlich ist eine Verkleinerung des möglichen Fehlercodes durch Slicing möglich.

In hat können alle Views parallel verwendet werden.[12]

Vorteil dieses Ansatzes gegenüber Hood sind neben der Möglichkeit der Analyse der Beziehungen zwischen unterschiedlichen Berechnungen auch die besseren Tools (gute Browser, die aber Übung benötigen).

Nachteile:

- Langsamere Programmausführung, manchmal langsamer View.
- Programmtransformationen auf Haskell 98 beschränkt.
- Aufwändigere Installation/Benutzung.

[12]Neue *Views* sind potentielle Themen für Diplomarbeiten!

7 Rekursion

In Programmen haben wir unterschiedliche Formen von Rekursion kennengelernt. Hierbei werden meist Datenstrukturen (Bäume) durchlaufen.

Beispiel: Selbst Rekursion über `Int` kann als Baumdurchlauf gesehen werden – man definiere etwa `data Nat = 0 | Succ Nat`.

7.1 Attributierte Grammatiken

Idee: Wir notieren nicht-kontextfreie Eigenschaften als *Attributierung* einer kontextfreien Grammatik.

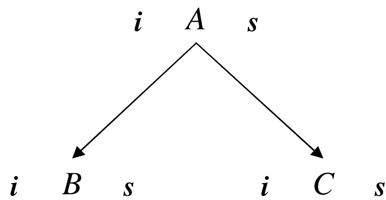
Definition: Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik (CFG). Dann kann G um eine *Attributierung* erweitert werden:

- ordne jedem Symbol $X \in N \cup \Sigma$ eine Menge von *synthetisierten Attributen* $\mathrm{Syn}(X)$ und eine Menge von *vererbten (inheriten) Attributen* $\mathrm{Inh}(X)$ zu (wobei $\mathrm{Inh}(a) = \emptyset$ für alle $a \in \Sigma$ gilt)
- ordne jeder Regel $X_0 \to X_1 \dots X_n \in P$ eine Menge von *semantischen Regeln (Attributgleichungen)* der Form $a.i = f(a_1.j_1, \dots, a_k.j_k)$ zu mit folgenden Eigenschaften:
  - $a \in \mathrm{Syn}(X_i)$ falls $i = 0$
  - $a \in \mathrm{Inh}(X_i)$ falls $i > 0$
  - für alle $1 \leq i \leq k$ gilt:
    - $a_i \in \mathrm{Syn}(X_{j_i})$ falls $j_i > 0$
    - $a_i \in \mathrm{Inh}(X_{j_i})$ falls $j_i = 0$

Die *Attributierung* einer Regel enthält eine Attributgleichung für alle synthetischen Attribute von X_0 und für alle inheriten Attribute von $X_1, \dots, X_n$.

Idee: Synthetische Attribute werden hochgereicht, inherite Attribute werden runtergereicht, und f wird interpretiert, z.B. durch ein Haskell-Programm.

Beispiel: Sei eine Regel $A \to B\,C \in P$ gegeben, Attribute seien $\mathrm{Syn}(A) = \mathrm{Syn}(B) = \mathrm{Syn}(C) = \{s\}$ und $\mathrm{Inh}(A) = \mathrm{Inh}(B) = \mathrm{Inh}(C) = \{i\}$.
Der Ableitungsbaum und die Attributte lassen sich dann folgendermaßen darstellen (wobei wir synthetische Attribute rechts, inherite Attribute links neben die Knoten schreiben):



Mögliche Regeln sind jetzt:

$$s.0 = f(s.1, s.2, i.0)$$
$$i.1 = g(s.1, s.2, i.0)$$

Beachte: Es ist kein Zykel innerhalb einer Regel definierbar!

Beispiel: Beschreibe Binärzahlen in folgender Form (Knuth 1968):

$$\begin{aligned} G\colon N &\to L \mid L.L \\ L &\to B \mid LB \\ B &\to 0 \mid 1 \end{aligned}$$

Zu der Zahl $1101.01 \in L(G)$ erhält man dann den Ableitungsbaum aus Abbildung 2.

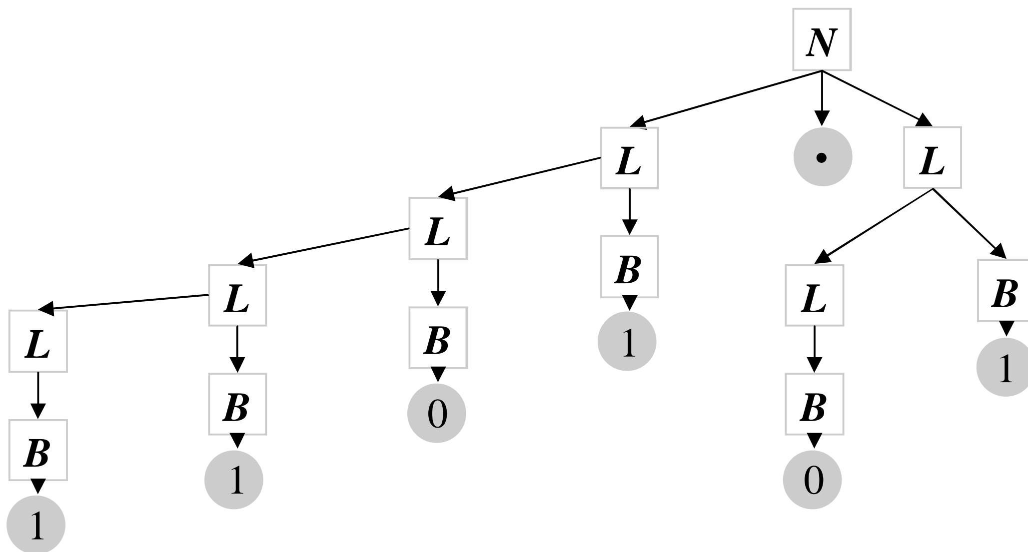


Abbildung 2: Ableitungsbaum für $1101.01 \in L(G)$

Die *Attributierung* eines Ableitungsbaumes ist nun die Zuordnung von Attributen zu den Knoten gemäß der Attributabhängigkeiten der Regeln; hier im

Beispiel wollen wir den Dezimalwert einer Binärzahl zunächst rein synthetisch ermitteln. Dazu sei

$$\begin{aligned} \text{Syn}(N) = \text{Syn}(B) &= \{v\} \\ \text{Syn}(L) &= \{v, l\} \\ \text{Syn}(0) = \text{Syn}(1) = \text{Syn}(.) &= \emptyset \end{aligned}$$

Nun können wir unsere Regeln attributieren:

$$\begin{array}{lcll} B \to 0 & : & v.0 = 0 & \\ B \to 1 & : & v.0 = 1 & \\ L \to B & : & v.0 = v.1, & l.0 = 1 \\ L \to LB & : & v.0 = v.2 + 2 \cdot v.1, & l.0 = l.1 + 1 \\ N \to L & : & v.0 = v.1 & \\ N \to L.L & : & v.0 = v.1 + \frac{v.3}{2^{l.3}} & \end{array}$$

Damit ergibt sich der attributierte Baum aus Abbildung 3.

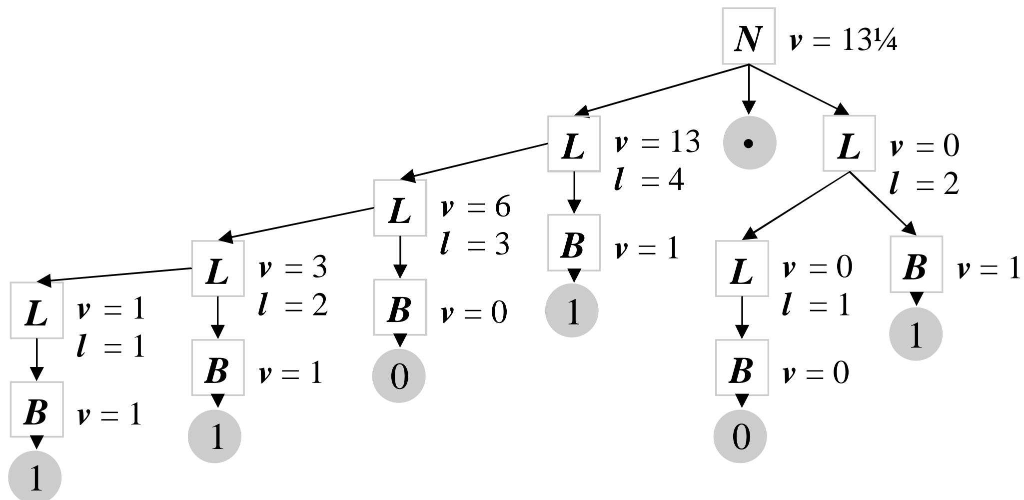


Abbildung 3: attributierter Ableitungsbaum (1) für $1101.01 \in L(G)$

Erweitertes **Beispiel:** Wir fügen zusätzlich ein inherites Attribut für die Stelle

des Bits hinzu (d.h. $\text{Inh}(L) = \text{Inh}(B) = \{n\}$):

$$
\begin{array}{lcll}
B \to 0 & : & v.0 = 0 & \\
B \to 1 & : & v.0 = 2^{n.0} & \\
L \to B & : & v.0 = v.1, & l.0 = 1 \\
 & & n.1 = n.0 & \\
L \to LB & : & v.0 = v.1 + v.2, & l.0 = l.1 + 1 \\
 & & n.1 = n.0 + 1, & n.2 = n.0 \\
N \to L & : & v.0 = v.1, & n.1 = 0 \\
N \to L.L & : & v.0 = v.1 + v.3 & \\
 & & n.1 = 0, & n.3 = -l.3
\end{array}
$$

Nun gibt es praktisch drei Durchläufe durch den Baum: Zunächst wird die Länge (l) von unten synthetisch berechnet, dann werden die Stellen (n) von oben nach unten weitergegeben, und schließlich wird der Wert (v) vo unten nach oben berechnet.

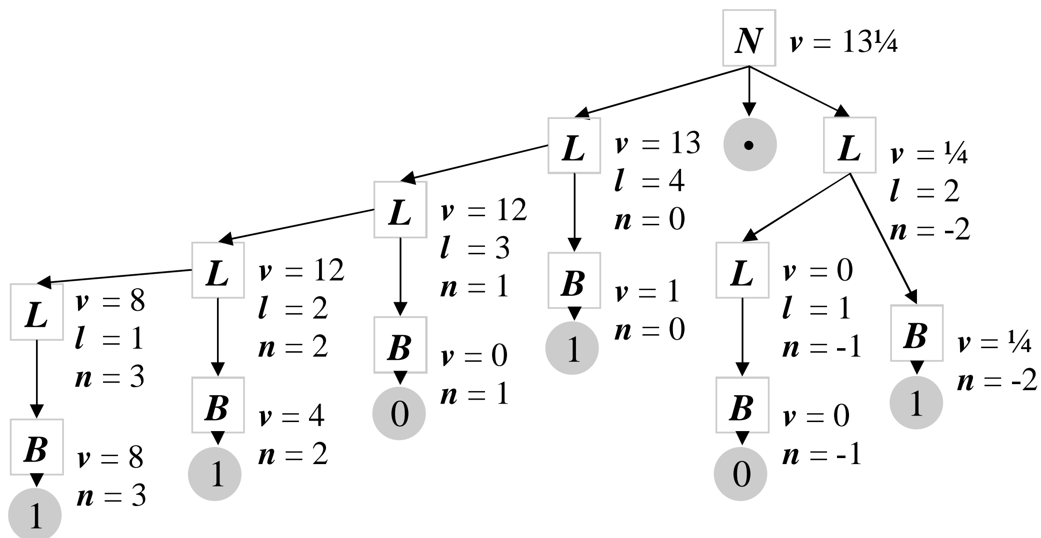


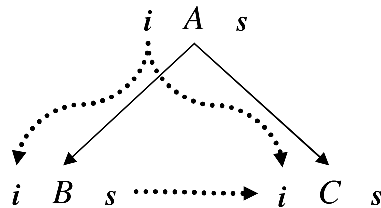
Abbildung 4: attributierter Ableitungsbaum (2) für $1101.01 \in L(G)$

Damit ergibt sich der Baum aus Abbildung 4.

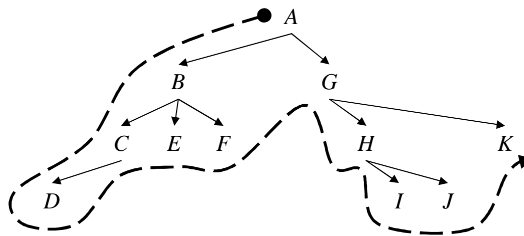
Spezialfälle von Attributierungen sind:

- Bei einer *reinen S-Attributierung* benutzen wir nur synthetische Attribute.

- Bei einer *L-Attributierung* gilt für alle Attributgleichungen der Form $a.i = f(a_1.i_1, \ldots, a_k.j_n)$ mit $i > 0$, daß $i > j_k$ ist. D.h. zur Definition eines inheriten Attributs dürfen nur synthetische Attribute von weiter links und inherite Attribute der linken Seite verwendet werden:

  Dann ist die Attributauswertung mittels eines Links-Rechts-Durchlaufs (L-Durchlauf) möglich:

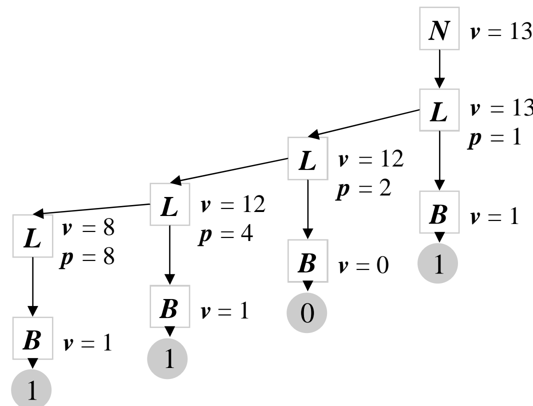
  Damit wird jeder Knoten maximal zweimal besucht (während des Abstiegs und während der Rückgabe eines Wertes).

- *Zyklische Attributierungen* sind unerwünscht – es ist dabei entscheidbar, ob eine gegebene attributierte Grammatik zykelfrei ist. Wir gehen im Folgenden davon aus, daß eine zykelfreie attributierte Grammatik vorliegt.

Beispiel: Betrachte wieder Binärzahlen, diesmal ohne Nachkomma-Anteil; und berechne den Wert als Summe der Zweier-Potenzen – dafür sei $\text{Inh}(L) = \{p\}$ ein Attribut für die jeweilige Zweier-Potenz.

$$\begin{array}{lcll}
B \to 0 & : & v.0 = 0 & \\
B \to 1 & : & v.0 = 1 & \\
L \to B & : & v.0 = p.0 \cdot v.1 & \\
L \to LB & : & v.0 = v.1 + v.2 \cdot p.0, & p.1 = 2 \cdot p.0 \\
N \to L & : & v.0 = v.1, & p.1 = 1
\end{array}$$

Als Beispielzahl betrachten wir 1101:



Frage: Wie kann die Attributierung eines Baumes berechnet werden? Dazu wird der Ableitungsbaum als Haskell-Datenstruktur repräsentiert. Jedes Nichtterminalsymbol ist ein Datentyp, und die Regeln zu einem Nichtterminalsymbol werden durch unterschiedliche Konstruktoren unterschieden.

Um die Grammatik für Dezimalzahlen (mit Nachkomma-Anteil) umzusetzen, benutze folgende Datenstruktur:

```
data N = N1 L | N2 L L -- Terminal '.' entfällt
data L = L1 B | L2 L B
data B = O | I

testBinary = N2 (L2 (L2 (L2 (L1 I) I) O) I)
               (L2 (L1 O) I)
```

Der Übergang von Wort zur Datenstruktur kann mittels eines Parser erfolgen (z.B. mit Parserkombinatoren). Danach kann die S-Attributierung wie folgt als Haskell-Programm implementiert werden (wobei die Attribute jeweils die Ergebnisse des Baumdurchlaufs sind, ggf. als Tupel bei mehreren Attributen). Wir implementieren folgende S-attributierte Grammatik von oben:

$$
\begin{array}{lll}
B \to 0 & : & v.0 = 0 \\
B \to 1 & : & v.0 = 1 \\
L \to B & : & v.0 = v.1, \qquad l.0 = 1 \\
L \to LB & : & v.0 = v.2 + 2 \cdot v.1, \quad l.0 = l.1 + 1 \\
N \to L & : & v.0 = v.1 \\
N \to L.L & : & v.0 = v.1 + \frac{v.3}{2^{l.3}}
\end{array}
$$

Diese wird umgesetzt in drei Funktionen:

```
be :: B -> Float
be O = 0
```

```
be I = 1

le :: L -> (Float, Int)
le (L1 b)    = let v1 = be b
                in (v1, 1)
le (L2 l b) = let (v1, l1) = le l
                    v2 = be b
                in (2*v1+v2, l1+1)

ne :: N -> Float
ne (N1 l)      = let (v1, _) = le l
                  in  v1
ne (N2 l lx) = let (v1, _)  = le l
                   (v3, l3) = le lx
                in  v1 + v3/2 ** l3
```

Dann ist `ne testBinary` gleich `13.25`.

Nun betrachten wir zusätzlich auch inherite Attribute – diese können als Parameter an die Funktionen für den Baumdurchlauf übergeben werden! Betrachte dazu folgende L-attributierte Grammatik von oben:

$$
\begin{array}{lcll}
B \to 0 & : & v.0 = 0 & \\
B \to 1 & : & v.0 = 1 & \\
L \to B & : & v.0 = p.0 \cdot v.1 & \\
L \to LB & : & v.0 = v.1 + v.2 \cdot p.0, & p.1 = 2 \cdot p.0 \\
N \to L & : & v.0 = v.1, & p.1 = 1
\end{array}
$$

Diese wird wieder umgesetzt in drei Funktionen:

```
be :: B -> Int
be O = 0
be I = 1

le :: L -> Int -> Int -- p, v
le (L1 b)   p0 = let v1 = be b
                 in  p0 * v1
le (L2 l b) p0 = let v1 = le l (2 * p0)
                     v2 = be b
                 in v1+p0*v2


ne :: N -> Int
ne (N1 l) = le l 1
```

Nun ergibt sich: `ne (N1 (L2 (L2 (L2 (L1 I)O)I)I))` ergibt 11.

Betrachte als letztes folgende nicht L-attributierte Grammatik von oben:

$$
\begin{array}{lcll}
B \to 0 & : & v.0 = 0 & \\
B \to 1 & : & v.0 = 2^{n.0} & \\
L \to B & : & v.0 = v.1, & l.0 = 1 \\
 & & n.1 = n.0 & \\
L \to LB & : & v.0 = v.1 + v.2, & l.0 = l.1 + 1 \\
 & & n.1 = n.0 + 1, & n.2 = n.0 \\
N \to L & : & v.0 = v.1, & n.1 = 0 \\
N \to L.L & : & v.0 = v.1 + v.3 & \\
 & & n.1 = 0, & n.3 = -l.3
\end{array}
$$

Da diese nicht L-attributiert ist, ist zunächst die Auswertungsstrategie nicht klar. Wir schreiben dennoch nach obigem Schema folgendes Programm:

```
be :: B -> Int -> Float -- n, v
be O n0 = 0
be I n0 = 2**n0

le :: L -> Int -> (Float, Int) -- n, (v, l)
le (L1 b)    n0 = let v1 = be b n0
                  in  (v1, 1)
le (L2 l b) n0 = let (v1, l1) = le l (n0+1)
                     v2 = be n0
                  in  (v1+v2, l1+1)

ne :: N -> Float
ne (N1 l)     = let (v1, _) = le l 0
                in  v1
ne (N2 l lx) = let (v1, l1) = le l 0
                   (v3, l3) = le lx (-l3)
                in  v1+v3
```

Beachte dabei leider, daß in der vorletzten Zeile `l3` auf beiden Seiten der Gleichung vorkommt: `(v3, l3)= le l' (-l3)`. Dies wird also vermutlich nicht terminieren.

Überraschung (!): Es terminiert doch und funktioniert sogar sehr gut: Haskells Lazy Evaluation zerlegt die Rekursion über Tupel (die in `le` vorkommt) in mehrere Baumdurchläufe (jedoch ohne mehrfaches Pattern Matching). Betrachte als entsprechendes Beispiel:

```
loop = loop    -- nicht-terminierend!
snd (le (L2 (L1 I) O) loop) ⇝ 2
```

Dies funktioniert, da das zweite Argument von `le` für die Berechnung der Länge als Ergebnis nicht benötigt wird!

Allgemein gilt: Für jede zykelfreie attributierte Grammatik stellt obige Implementierung einen effizienten Attributauswerter dar. Für S- und L-Attributierung stellt diese Technik auch eine Implementierung in strikten oder imperativen Sprachen dar; bei imperativen Sprachen führt nur eventuell die Rekursion zu einem Stacküberlauf.

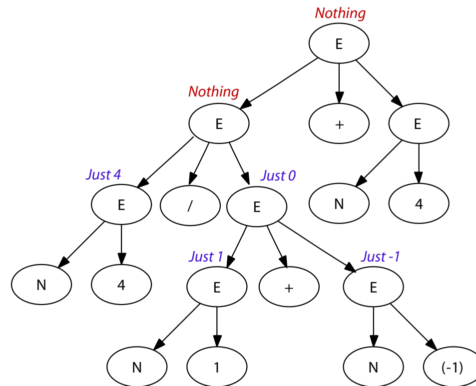
7.2 Continuation Passing Style

Im folgenden wollen wir uns nocheinmal mit Abbrüchen während eines Baumdurchlaufs beschäftigen. Betrachte hierzu wiederum:

```
data Expr = Expr :+: Expr
          | Expr :/: Expr
          | Num Float

eval :: Expr -> Maybe Float -- mit Maybe-Monade
```

Untersuche nun folgenden Baumdurchlauf:



Hier wird nach Auftreten des Fehles (`Nothing`) der Baumdurchlauf nicht völlig abgebrochen, sondern nur der Wert `Nothing` hochgereicht, d.h. dies wird an jedem Knoten oberhalb der Fehlerstelle erneut getestet (per Pattern Matching gegen `Nothing`).

Idee: Wir möchten den Abbruch direkt an der Fehlerstelle durchführen. Stelle dazu die Berechnung (den Baumdurchlauf) als sogenannten *Continuation (Fortsetzung)* dar. Dann kann im Fehlerfall die Continuation verworfen werden und der Fehler direkt zurückgegeben werden.

Dieser Programmierstil wird mit *Continuation Passing Style (CPS)* bezeichnet.

Im **Beispiel:**

```
eval :: Expr -> (Float -> Maybe Float) -> Maybe Float
eval (Num x) c = c x
eval (e1 :+: e2) c =
  eval e1 (\v1 -> eval e2 (\v2 -> c (v1+v2)))
eval (e1 :/: e2) c =
```

```
  eval e1 (\v1 -> eval e2 (\v2 ->
                if v2 == 0 then Nothing
                           else c (v1/v2)))

eval (Num 1 :/: (Num 1 :+: Num (-1)) Just ⇝ Nothing
```

Möglich wäre es auch, zuerst den Divisor zu berechnen:

```
eval (e1 :/: e2) c =
  eval e2 (\v2 -> if v2 == 0
                  then Nothing
                  else eval e1 (\v1 -> c (v1/v2)))
```

Die **Vorteile**, die weitere Berechnung als Argument zu übergeben, sind:

- Die weitere Berechnung kann „modifiziert“ werden, man kann also den Kontext der Auswertung im Baum sehen und benutzen.
- Die Aufrufe beispielsweise von `eval` sind endrekursiv, dies verbraucht keinen Platz auf dem Stack!
- Für Exceptions ist diese Berechnungsart vorteilhaft, da mehrfache Baumdurchläufe gespart werden (Coroutining).

Weiteres **Beispiel:** S-Attributierung im CPS:

```
be :: B -> (Float -> a) -> a
be I c = c 1
be O c = c 0

le :: L -> (Float -> Float -> a) -> a
le (L1 b)   c = be b (\v1 -> c v1 1)
le (L2 l b) c = le l (\v1 l1 ->
                 be b (\v2 ->
                    c (2*v1*v2) (l1+1)))

ne :: N -> (Float -> a) -> a
ne (N1 l)     c = le l (\v1 -> c v1)
ne (N2 l1 l2) c = le l1 (\v1 _ ->
    le l2 (\v3 l3 -> c (v1 + v3/2**l3)
```

Dann wird `ne testBinary id` ausgewertet zu `13.25`.

8 Algorithmen und Datenstrukturen

Bisher haben wir Listen und (unbalancierte) Suchbäume benutzt. Was sind geeignete Algorithmen und Datenstrukturen für häufige Fragestellungen?

8.1 Listen

Listen eignen sich gut, wenn Daten gesammelt werden und anschließend jedes Element verwendet wird.

Beispiel: `fac n = foldr (*)1 [0..n]`

Problem: Konkatenation von Listen ist linear im ersten Argument, deshalb ist es teuer, ein Element hinten anzuhängen: `xs++[x]`.

Aber manchmal läßt sich dies nur schwer vermeiden (z.B. bei Rekursion):

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

treeToList :: Tree -> [Int]
treeToList (Leaf n) = [n]
treeToList (Node tl tr) =
                treeToList tl ++ treeToList tr
```

Hier ist der Aufwand für `treeToList t` nicht linear in Anzahl der Blätter von `t`, da das erste Argument von `++` wächst. Bei einem entarteten Baum ergibt sich eine quadratische Laufzeit in Anzahl der Knoten bzw. Blätter.

Durch CPS läßt sich dies verbessern:

```
treeToList :: Tree -> (Int -> [Int]) -> [Int]
treeToList (Leaf n) c = c n
treeToList (Node tl tr) c =
                treeToList tl (: (treeToList tr c))
```

Dann kann man `treeToList (:[])` auf einen Baum anwenden, dies ist linear in Anzahl der Knoten bzw. Blätter.

Dies ist aber schwer zu komponieren, wie das Beispiel `Show` zeigt:

```
show :: Show a => a -> String  -- [Char]
```

Bei geschachtelten Typen müssen also unterschiedliche Implementierungen von `show` (für die einzelnen Typen) komponiert werden, für `show [Tree]` also etwa `showList`, `showInt` und `showTree`.

Die Continuation müßte dann durch alle Listen gereicht werden:

```haskell
class ShowC a where -- nicht in Haskell implementiert
  showC :: a -> (String -> String) -> String

instance showC Tree where
  showC (Leaf n) c =
    showC n (\nStr -> c ("Leaf ++ nStr))
  showC (Node tl tr) c =
    showC tl (++ (showC tr c))
```

Wir verwenden zwar `(++)`, aber dies wird nur auf kurze Listen, wie `"Leaf 42"` angewendet.

```haskell
instance ShowC a => showC [a] where
  showC [] c = c "[]"
  showC (x:xs) c =
    showC x (++ ':':showC xs c)
```

Dann kann die Umwandlung der Datenstruktur in einen String in linearer Zeit erfolgen.

Als bessere Alternative nutzen wir die Akkumulatortechnik:

```haskell
treeToList :: Tree -> [Int] -> [Int]
treeToList (Leaf n) ns = n:ns
treeToList (Node tl tr) ns =
  treeToList tl (TreeToList tr ns)
```

Dies ist scheinbar ähnlich wenig kompositionell wie mit CPS. Wir können dies aber noch anders definieren:

```haskell
treeToList (Leaf n) = (n:)
treeToList (Node tl tr) =
  treeToList tl . treeToList tr
```

Dann entspricht die Funktionskomposition dem `(++)`. Solche Listen nennt man funktionale Listen. Um die Typsignatur anzupassen kann man definieren:

```haskell
type IntListF = [Int] -> [Int]
treeToList :: Tree -> IntListF
```

Wie funktioniert dies für `Show`?

```haskell
type ShowS = String -> String

class Show a where
```

```haskell
  shows :: a -> ShowS
  show :: a -> String
  show x = shows x ""

instance Show Tree where
  shows (Leaf n) =
    showString "Leaf" . shows n . showChar ','
  shows (Node tl tr) =
    showString "Node (" . shows tl .
       showString ") (" . shows tr . showChar ')'
```

Dabei sind folgende Funktionen definiert:

```haskell
showChar :: Char -> ShowS
showChar = (:)

showString :: String -> ShowS
showString = (++)
```

Nun können wir ein allgemeines `shows` auf Listen definieren:

```haskell
instance Show a => Show [a] where
  shows [] = showString "[]"
  shows (x:xs) = shows x . showChar ':' . shows xs
```

In Haskell ist diese Implementierung noch um einen zusätzlichen Präzendenzparameter erweitert (z.B. um Klammern zu sparen), der meist 0 ist:

```haskell
showsPrec :: Int -> a -> ShowS
shows = showsPrec 0
```

Zudem ist `showsPrec` mittels `show` vordefiniert, es ist zudem auch `showList` vordefiniert (z.B. als `[1,2,3]`), kann aber überschrieben werden (z.B. für String).

8.2 Stacks (LIFO)

Zur Implementierung von Stacks eignen sich Listen ausgezeichnet:

```haskell
push = (:)
pop = tail
top = head
```

8.3 Queues (FIFO)

Zunächst implementieren wir Queues mittels Listen:

```
enter x q = q++[x]
remove q = tail
top = head
```

Für kleine Queues ist dies gut. Für große Queues ergibt sich das Problem, daß `enter` linear in Queuegröße ist. Wie kann das verbessert werden? Wir verwenden zwei Listen:

```
data Queue a = Queue [a] [a]

enter :: a -> Queue a -> Queue a
enter x (Queue out in) = Queue out (x:in)

top :: Queue a -> Maybe a
top (Queue [] []) = Nothing
top (Queue [] in) = Just (last in)
top (Queue (a:as) in) = Just a

remove :: Queue a -> Queue a
remove (Queue [] []) = Queue [] []
remove (Queue [] in) = remove (Queue (reverse in) [])
remove (Queue (a:as) in) = Queue as in
```

Die Laufzeit ist nun in den meisten Fällen konstant, nur wenn die `out`-Liste leer ist, dann ist die Operation linear in Queuegröße. Aber: je größer die Queue ist, desto seltener muß `reverse` ausgeführt werden. Die amortiesierte Laufzeit ist konstant!

8.4 Arrays

Haskell stellt Arrays im Modul `Array` zur Verfügung, wobei als Array-Index jeder Typ dienen kann, der auf `Int` abgebildet werden kann (mittels der Typklasse `Ix`). Das Interface sieht wie folgt aus:

```
data Array a b -- abstrakter Typ
array :: Ix a => (a, a) -> [(a, b)] -> Array a b
listArray :: Ix a => (a, a) -> [b] -> Array a b
(!) :: Ix a => Array a b -> a -> b
(//) :: Ix a => Array a b -> [(a, b)] -> Array a b
```

Dies kann man dann wie folgt benutzen:

```
array (1,5) [ (i,i+1) | i ← [1..5] ]
⇝ array (1,5) [(1,2),(2,3),(3,4),(4,5),(5,6)]

(listArray (1,5) [2..]) ! 2 ⇝ 3
(listArray (1,5) [2..] // [(2,4)]) ! 2 ⇝ 4
```

Laufzeiten:

- `array` und `listArray` sind linear in Arraygröße,
- `(!)` ist konstant,
- aber `(//)` ist linear in Arraygröße, denn beim Ändern wird das gesamte Array kopiert.

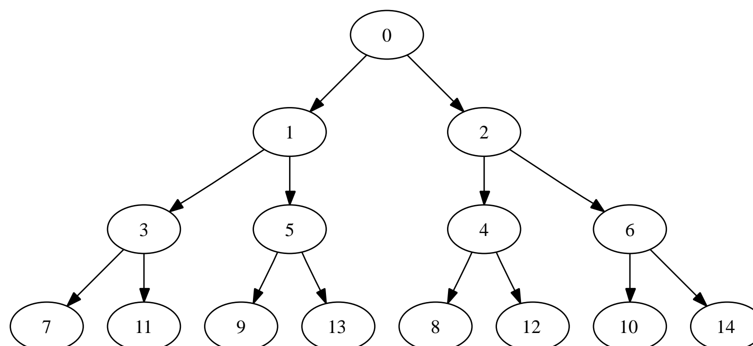
Dies ist gut für Datenstrukturen zum Nachschlagen, z.B. bei einem initialen Konstruktionsalgorithmus mit späterem Lesen des Arrays ohne Veränderung. Bei häufigen kleineren Änderungen ist diese Implementierung jedoch schlecht!

Beachte: Destruktive Updates sind in funktionalen Sprachen nicht möglich, da das alte Array an anderer Stelle auch noch verwendet werden kann (referentielle Transparenz).

Eine **alternative Implementierung** nutzt Braunbäume (hier nur mit Integers als Indizes, kann auch auf die Klasse `Ix` erweitert werden). Als Idee werden dabei die Indizes auf gerade und ungerade aufgeteilt und beim Abstieg durch zwei dividiert (und eins abgezogen bei geraden Indizes). Die Null ist dabei die Wurzel.

```
data ArrayB b = Entry b (ArrayB b) (ArrayB b)
```

Folgende Graphik zeigt den Baum beschriftet mit den Indizes:



Die Selektion eines Wertes in diesem Baum können wir nun wie folgt definieren:

```haskell
(!) :: ArrayB b -> Int -> b
(Entry v al ar) ! n
  | n == 0     = v
  | even n     = ar ! (n `div` 2 - 1)
  | otherwise = al ! (n `div` 2)

update :: ArrayB b -> Int -> b -> ArrayB b
update (Entry v' al ar) n v
  | n == 0     = Entry v al ar
  | even n     = Entry v' al
                  (update ar (n `div` 2 - 1) v)
  | otherwise = Entry v'
                  (update al (n `div` 2) v) ar
```

Dann sind sowohl `(!)` als auch `update` logarithmisch im Index.

Beachte: Bei `update` wird nicht die gesamte Datenstruktur kopiert, sondern nur der Pfad zum veränderten Knoten.

```haskell
emptyArrayB :: ArrayB a
emptyArrayB =
  Entry (error "access to non-instantiated index")
        emptyArrayB emptyArrayB
```

`ArrayB`s können beliebig groß sein, es gibt keine Dimensionsbeschränkung – aber mit größeren Indizes wird die Datenstruktur auch ineffizienter, beachte allerdings $\log_2 1000000 \approx 20$.

Bei vollständiger Verwendung des Indexbereichs `[0..n]` ergibt sich ein ausgeglichener binärer Baum.

Ein **Vorteil** gegenüber destruktiven Updates ist, daß alte und neue Variante des Arrays verfügbar sind (mit logarithmischem Zeit- und Platzverbrauch!), während man in imperativen Sprachen oft eine Kopie eines Array anlegt (linearer Zeit- und Platzverbrauch).

Das Füllen eines Arrays mit Werten aus einer Liste ist mit `update` möglich in $\mathcal{O}(n \cdot \log n)$ (falls n die Länge der Liste ist), jedoch mit einem großen konstanten Anteil. Besser ist folgende Lösung:

```haskell
split :: [a] -> ([a], [a])
split []  = ([], [])
split [x] = ([x],[])
split (x:y:xys) =
```

```
    let (xs, ys) = split xys in (x:xs, y:ys)

listToArrayB :: [b] -> ArrayB b
listToArrayB [] = emptyArrayB
listToArrayB (x:xs) =
   let (ls, rs) = split xs
    in Entry x (listToArrayB ls) (listToArrayB rs)
```

Die Laufzeit ist zwar noch in $\mathcal{O}(n \cdot \log n)$, aber sie hat bessere Konstanten. Eine lineare Implementierung ist möglich, siehe Okasaki 1997.

8.5 Höhenbalancierte Suchbäume

Nachteil von Arrays: Indizes müssen auf „kleine" Ints abgebildet werden können. Was macht man aber bei komplizierten Schlüsseln? Oft ist Hashing auf möglichst kleine Int-Werte relativ schwer (nicht alle Werte werden genutzt, Kollisionen bei zu kleiner Hashtable).

Lösung sind Suchbäume, dabei muß der Schlüssel aus der Klasse `Ord` sein (was meist „derived" werden kann), dies gibt Probleme bei Funktionen oder Suchbäumen als Schlüssel (da Suchbäume schlecht vergleichbar sind, offene Frage: wie implementiert man Mengen von Mengen?).

Effizient ist dies für zufällige Schlüsselverteilung beim Einfügen (die Operationen `insert`, `delete` und `lookup` in $\mathcal{O}(\log n)$ mit n als Anzahl der Einträge), ineffizient im Worst-Case: $\mathcal{O}(n)$.

Verbesserung sind hier *höhenbalancierte Bäume* (z.B. Spreizbäume, Rot-Schwarz-Bäume oder AVL-Bäume), die beinahe balancierte Suchbäume durch Umbalancierungen gewährleisten – die genauere gewährleistete Eigenschaft ist beispielsweise, daß die Höhe zweier benachbarter Teilbäume sich um maximal eins unterscheidet. Somit ist die Tiefe des Baumes maximal $2 \cdot \lfloor \log(n+1) \rfloor$ und `insert`, `delete` und `lookup` laufen in $\mathcal{O}(\log n)$.

Zur **Implementierung** siehe Informatik 2, in funktionalen Programmiersprachen ist die Implementierung jedoch eleganter.

Vergleich mit Braunbäumen:

- `insert`, `lookup` und `delete`:
  - höhenbalancierte Bäume: $\mathcal{O}(\log n)$ mit n als Anzahl Einträge
  - Braunbäume: $\mathcal{O}(\log n)$ mit n als Index
- Schlüssel:

- höhenbalancierte Bäume: Instanz der Klasse `Ord`
  - Braunbäume: `Int`

- Baumtiefe:
  - höhenbalancierte Bäume: $2 \cdot \lfloor \log_2(n+1) \rfloor$
  - Braunbäume: $\lfloor \log_2(n+1) \rfloor$ mit n größtem Index

- Lazyness:
  - höhenbalancierte Bäume: wenig, da für Umbalancieren alle Werte eingetragen sein müssen. In ghc sogar zusätzliche Striktheitsannotation um Speicher für suspendierte Berechnungen zu sparen.
  - Braunbäume: nur benötigte Einträge werden gemacht

- unendliche Strukturen:
  - höhenbalancierte Bäume: wegen Balancierung nicht möglich
  - Braunbäume: möglich, z.B. `emptyArrayB`

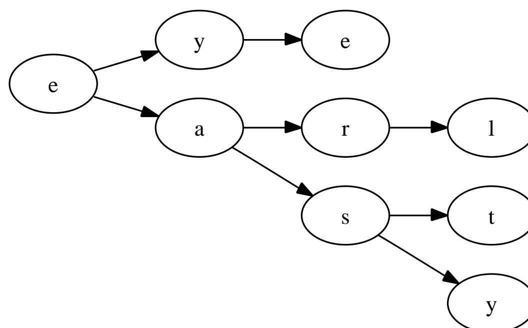
8.6 Tries

Ein alternativer Ansatz zu höhenbalancierten Bäumen sind die *Tries*, deren Idee von Thue (1912) zur Repräsentation von Strings stammt. In Haskell wurde dies verallgemeinert zur Ausnutzung der Baumstruktur von Schlüsseln.

Beispiel: Sei die folgende String-Schlüssel-Menge gegeben:

```
["ear","earl","east","easy","eye"]
```

Dann kann man dies in folgendem Baum repräsentieren:



Das Nachschlagen ist nun linear in Größe des Schlüssels (bei konstanter Alphabet-Größe). Als Datenstruktur für String-Tries nutzen wir:

```haskell
type String = [Char]
data MapStr v = Trie (Maybe v) (MapChar (MapStr v))
type MapChar v = [(Char,v)]
```

Für `MapChar v` verwenden wir eine Assoziationsliste, es ginge auch effizienter mit einem `ArrayB` über die ASCII-Werte der Zeichen.

```haskell
lookupChar :: Char -> MapChar v -> Maybe v
lookupChar = lookup

lookupStr :: String -> MapStr v -> Maybe v
lookupStr    "" (TrieStr tn tc) = tn
lookupStr (c:cs) (TrieStr tn tc) = do
  tcs <- lookupChar c tc
  lookupStr cs tcs

insertChar :: Char -> a -> MapChar a -> MapChar a
insertChar c v [] = [(c,v)]
insertChar c v ((c',v'):cvs)
    | c == c'   = (c,v):cvs
    | otherwise = (c',v'):insertChar c v cvs

emptyTrieStr :: MapStr a
emptyTrieStr = TrieStr Nothing []
```

Beachte: In MapChar werden keine Werte für v eingetragen, sondern komplette Tries!

Laufzeit: `lookup` und `insert` sind linear in Schlüsselgröße, `delete` ist etwas schwieriger, da der Baum aufgeräumt werden muß, um keine leeren Äste zu hinterlassen.

```haskell
deleteStr :: String -> MapStr a -> MapStr a
deleteStr    "" (TrieStr _  tc) = TrieStr Nothing tc
deleteStr (c:cs) (TrieStr tn tc) =
  case lookup c tc of
   Nothing -> TrieStr tn tc
   Just tcs ->
     case deleteStr cs tcs of
       TrieStr Nothing [] ->
            TrieStr tn (filter ((/=c).fst) tc)
       tcs' -> TrieStr tn (insertChar c tcs' tc)
```

```haskell
insertStr :: String -> a -> MapStr a -> MapStr a
insertStr "" v (TrieStr _ tc) =
  TrieStr (Just v) tc
insertStr (c:cs) v (TrieStr tn tc) = TrieStr tn
  (case lookup c tc of
    Nothing -> (c, insertStr cs v emptyTrieStr):tc
    Just tcs -> insertChar c (insertStr cs v tcs) tc
```

Beachte: Wenn `filter` die leere Liste liefert und `tn` gleich `Nothing`, dann wird der Knoten im übergeordneten `deleteStr` gelöscht.

Beachte: Bei der Trie-Implementierung für `String`-Schlüssel haben wir eine Trie-Implementierung für `Char`-Schlüssel verwendet. `MapChar` ist als Assoziationsliste dargestellt, also `[(Char, a)]`. Im Fall, dass keine Verlängerung eines Wortes im Trie ist, haben wir hier `[]`, was analog zu `Nothing` in dem ersten Argument von `TrieStr` ist.[13]

Erweiterung: Im nächsten Schritt verwenden wir Binärzahlen als Schlüssel:

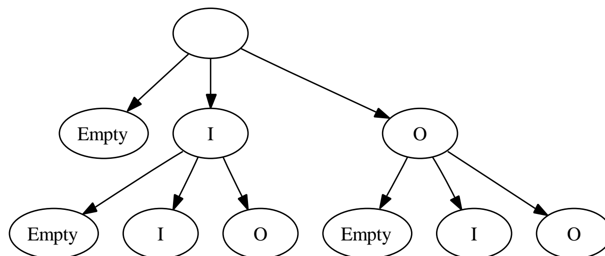
```haskell
data No = Empty | I No | O No
```

Nun können wir zunächst Zahlen in unsere Binärdarstellung umwandeln:

```haskell
intToNo :: Int -> No
intToNo 0 = Empty
intToNo n =
  (if even n then O else I) (intToNo (n `div` 2))

map intToNo [0..4] ⇝
  [Empty, I Empty, O (I Empty),
          I(I Empty), O(O(I Empty))]
```

Wie kann nun der zugehörige Trie aussehen?



Als Datenstruktur für einen Trie verwenden wir nun:

[13]Dieser Teil ist – wie auch vorher schon einige Teile – aus einer Mitschrift von Björn Duderstadt, vielen Dank!

```haskell
data MapNo a = TrieNo (Maybe a)          -- for Empty
                      (Maybe (MapNo a)) -- for I
                      (Maybe (MapNo a)) -- for O

emptyTrieNo :: MapNo a
emptyTrieNo = TrieNo Nothing Nothing Nothing
```

Nun können wir einfügen, auslesen und löschen:

```haskell
lookupNo :: No -> MapNo a -> Maybe a
lookupNo Empty (TrieNo te ti to) = te
lookupNo (I n) (TrieNo te ti to) =
  case ti of
    Nothing -> Nothing
    Just ti' -> lookupNo n ti'

insertNo :: No -> a -> MapNo a -> MapNo a
insertNo Empty v (TrieNo _ ti to) =
  TrieNo (Just v) ti to
insertNo (I n) v (TrieNo te Nothing to) =
  TrieNo te (Just (insertNo n v emptyTrieNo))
insertNo (I n) v (TrieNo te Nothing to) =
  TrieNo te (Just (insertNo n v ti)) to

deleteNo :: No -> MapNo a -> MapNo a
deleteNo Empty (TrieNo _ ti to) =
  TrieNo Nothing ti to
deleteNo (I n) t@(Trie te Nothing to) = t
deleteNo (I n) (TrieNo te (Just ti) to) =
  case deleteNo n ti of
    TrieNo Nothing Nothing Nothing ->
      TrieNo te Nothing to
    t1' -> TrieNo te (Just ti') to
deleteNo (O n) -- analog
```

Beachte: `deleteNo` räumt den Baum auf, so dass ein leerer `TrieNo` immer gleich `EmptyTrieNo` ist.

Allgemein: Die Aufteilung des Tries hat starke Ähnlichkeit mit Braunbäumen. Unter Berücksichtigung von `Empty` und Vernachlässigung redundanter Binärzahlen (z.B. 001 statt 1) sind die Zahlen gleich einsortiert. Damit können Tries auch als Verallgemeinerung von Brauntrees gesehen werden.

Als weitere Verallgemeinerung können wir nun binäre Bäume als Schlüssel zulassen:

```
data Bin = Leaf String
         | Node Bin Bin

data MapBin a =
  TrieBin (Maybe (MapStr a))            -- für "Leaf"
          (Maybe (MapBin (MapBin a))) -- für "Node",
                        -- MapBin für Unterscheidung
                        -- links/rechts in "Bin Bin"

emptyTrieBin :: MapBin a
emptyTrieBin = TrieBin Nothing Nothing
```

Allgemein: Ein Trie für einen monomorphen Typen τ hat einen Konstruktor Trieτ und für jeden Konstruktor c von τ hat Trieτ ein Argument, welches sukzessive über den Argumenten von c entsprechende Tries verzweigt. Da zu einem Schlüssel möglicherweise auch kein Eintrag im Trie steht, wird jedes Argument von Trieτ mit `Maybe` gekapselt.

8.6.1 Nested Datatypes

Beachte: `MapBin` wird auf `(MapBin a)` „appliziert“. Dies ist ein sogenannter *Nested Datatype*: Ein Typkonstruktor wird auf der rechten Seite auf einen „neuen“ Typen angewandt:

```
data T a = ... T (X a)
```

Die **Implementierung** sieht nun folgendermaßen aus:

```
lookupBin :: Bin -> MapBin a -> Maybe a
lookupBin (Leaf s) (TrieBin Nothing _) = Nothing
lookupBin (Leaf s) (TrieBin (Just tl) _ ) =
  lookupStr s tl
lookupBin (Node l r) (TrieBin _ Nothing) = Nothing
lookupBin (Node l r) (TrieBin _ (Just tn)) = do
  tn' <- lookupBin l tn
  lookupBin r tn'

insertBin :: Bin -> a -> MapBin a -> MapBin a
insertBin (Leaf s) v (TrieBin Nothing tn) =
  TrieBin (Just (insertStr s emptyTrieStr)) tn
```

```
insertBin (Leaf s) v (TrieBin(Just tl) tn) =
  TrieBin (Just (insertStr s tl)) tn
insertBin (Node l r) v (TrieBin tl Nothing) =
  TrieBin tl (Just (insertBin l
    (insertBin r v emptyTrieBin)
     emptyTrieBin))
insertBin (Node l r) v (TrieBin tl (Just tn)) =
  TrieBin tl (Just $
    case lookup l tn of
      Nothing ->
        insertBin l
           (insertBin r v emptyTrieBin) tn
      Just tn' ->
        insertBin l (insertBin r v tn') tn)

deleteBin :: Bin -> MapBin a -> MapBin a
deleteBin (Leaf s) t@(TrieBin Nothing tn) = t
deleteBin (Leaf s) (TrieBin (Just tl) tn) =
  case deleteStr s tl of
    TrieStr Nothing [] -> TrieBin Nothing tn
    tt'                -> TrieBin (Just tl') tn
deleteBin (Node _ _) t@(TrieBin tl Nothing) = t
deleteBin (Node l r) (TrieBin tl (Just tn)) =
  TrieBin tl $
    case lookupBin l tn of
      Nothing  -> Just tn
      Just tn1 ->
        case deleteBin r tn1 of
          TreeBin Nothing Nothing ->
            case deleteBin l tn of
              TrieBin Nothing Nothing -> Nothing
              tn2                     -> Just tn2
          tn2 -> Just (insertBin l tn2 tn)
```

Beachte: `deleteBin` schneidet alle leeren Teile aus dem Trie heraus, so dass diese Garbage werden können. Der leere Trie wird immer als `TrieBin Nothing Nothing` dargestellt, und nicht z.B. als

```
TrieBin Nothing (Just (TrieBin Nothing Nothing))
```

Laufzeiten: `lookup`, `insert`, `delete` erfolgen linear in der Schlüsselgröße. Besser geht es aus Komplexitätstheoretischer Sicht nicht. Tries verhalten sich

auch aus Laziness-Sicht gutmütig, da nur benötigte Teile berechnet werden müssen.

Bei der Programmierung mit *Nested Datatypes* wird meist *polymorphe Rekursion* benötigt, d.h. eine Funktion ruft (ggf. indirekt) sich selbst für einen anderen Typen auf: In `lookupBin` für `MapBin a` wird `lookupBin` zuerst auf `MapBin a` angewendet.

Problem: Bei polymorpher Rekursion ist Typinferenz im allgemeinen unentscheidbar. Deshalb erlaubt Haskell 98 polymorphe Rekursion nur bei gegebener Typsignatur der Funktion. Es wird also nur eine Typprüfung durchgeführt – diese ist entscheidbar. Der ghc versucht einen Typ zu inferieren, was meist gelingt, aber nach einigen Iterationen auch fehlschlagen kann.

9 Nebenläufige Programmierung

Moderne Programmiersprachen stellen Konzepte zur nebenläufigen Programmierung zur Verfügung – dies ist nützlich z.B. bei Echtzeitsystemen, bei GUI-Programmierung oder verteilter Programmierung, aber auch zur eleganten Formulierung von Algorithmen.

Vorteile: *Threads* (oder Prozesse) sind ein orthogonales Modularisierungskonzept zu Funktionen/Prozeduren, die quasi-parallele Ausführung von Berechnungen/Aktionen erlauben.

Aber: Durch Nebenläufigkeit alleine meist keine Effizienzsteigerung erreicht (nur bei Mehrprozessorsystemen, diese werden aber oft nicht unterstützt).

Idee: In nebenläufigen Systemen werden mehrere Programme, sogenannte *Threads*, „gleichzeitig“ ausgeführt. **Wichtige Aspekte** sind neben der Generierung und Terminierung von Threads auch Synchronisierung und Kommunikation.

9.1 ConcurrentHaskell

ConcurrentHaskell ist eine Erweiterung von HASKELL 98 zur nebenläufigen Programmierung. Dabei sind Threads Haskell-Programme vom Typ `IO ()`.

- **Generierung** neuer Threads: `forkIO` spaltet Thread ab und liefert sofort seine `ThreadId`

  ```
  forkIO :: IO () -> IO ThreadId   -- ghc
  forkIO :: IO () -> IO ()         -- Hugs
  ```

- Terminierung anderer Threads:

  ```
  killThread :: ThreadId -> IO ()
  ```

  Alternativ terminieren Threads, wenn ihr Code abgearbeitet wurde.

- Abruf der eigenen `ThreadId`:

  ```
  myThreadId :: IO ThreadId
  ```

9.2 Mutable Variables

Synchronisation und Kommunikation erfolgen in ConcurrentHaskell mittels *veränderbarer Variablen* (*mutable variables*, `MVars`). Diese funktionieren ähnlich wie `IORef`, dienen aber zusätzlich zur Synchronisation: eine `MVar` kann leer oder gefüllt sein (ähnlich zu `IORef (Maybe a)`).

- Generierung einer `MVar`:

  ```
  newEmtpyMVar :: IO (MVar a)
  newMVar :: a -> IO (Mvar a)
  ```

- Lesen einer `MVar`:

  ```
  takeMVar :: MVar a -> IO a
  ```

  Die Funktion `takeMVar` liest den Wert aus einer gefüllten `MVar`, anschließend ist die `MVar` „leer". Der ausführende Thread suspendiert, falls die `MVar` leer ist und wird wiedererweckt, sobald die `MVar` gefüllt wird. Die Aktion erfolgt atomar!

- Schreiben in die `MVar`:

  ```
  putMVar :: MVar a -> a -> IO ()
  ```

  Der ausführende Thread suspendiert, falls die `MVar` gefüllt ist. Er wird wiedererweckt, wenn die `MVar` geleert wird. Auch diese Aktion erfolgt atomar.

Beispiel: Dinierende Philosophen:

```
phil :: MVar () -> MVar () -> IO ()
phil l r = do
      takeMVar l
      takeMVar r
      --eat
      putMVar l ()
      putMVar r ()
      --think
      phil l r
```

Gnerierung der Philosophen:

```
main = startPhils 5
startPhils :: Int -> IO ()
```

```
startPhils n = do
  sticks ← mapM (const (newMVar ())) [1..n]
 mapM_ (\(l,r) -> forkIO (phil l r))
       (zip sticks (tail sticks))
  phil (last sticks) (head sticks)
```

Diese Version enthält jedoch einen Deadlock! Wir brauchen also Mechanismen zur Koordination etc.

9.3 Scheduling

Es gibt zwei wichtige Arten von **Scheduling**:

- *präemptives Multitasking:* Der Scheduler teilt die Rechenzeit fair zwischen den Threads auf (z.B. Round-Robin), dies ist in `ghc` implementiert.
- *kooperatives Multitasking:* Ein Wechsel zwischen Threads findet nur nach Suspension (z.B. `takeMVar` oder `putMVar`) oder explizit durch `yield :: IO ()` statt, dies ist in `Hugs` implementiert.

Beispiel:

```
main = do
  forkIO (mapM_ putChar (repeat 'a'))
 mapM_ putChar (repeat 'b')
```

Bei präemptivem Scheduling werden im Mittel dann gleich viele `a` und `b` erzeugt, beim kooperativen Multitasking nur `a` oder `b`.

Frage: Wann verhält sich ein nebenläufiges Programm *korrekt?* Wenn es sich unter *allen* möglichen Schedules korrekt verhält – manchmal wird dies auch eingeschränkt auf alle fairen Schedules (d.h. auf präemptives Multitasking).

Weitere Funktionen auf `mVars`:

- Test auf Leerheit ohne Suspension:

  ```
  isEmptyMVar :: MVar -> IO Bool
  ```

- Lesen einer `MVar`, ohne sie zu leeren; suspendiert ebenfalls, falls die `MVar` leer ist:

  ```
  readMVar :: MVar a -> IO a
  ```

- Verändert atomar den Wert in der `MVar`, falls leer, wird suspendiert, bis diese voll ist.

```
swapMVar :: MVar a -> a -> IO a
```

- Lesen und Schreiben ohne Suspendieren:

  ```
  tryTakeMVar :: MVar a -> IO (Maybe a)
  tryPutMVar :: MVar a -> a -> IO Bool
  ```

- Einige weitere Funktionen sind im Zusammenhang mit ghc-Exceptions definiert.

9.4 Semaphoren

Frage: Sind `MVar`s adäquat zur Synchronisation geeignet? Meist sind sie wie binäre Semaphoren zu verwenden, aber es ist auch möglich, allgemeine Semaphoren zu simulieren:

```
data QSem = QSem (MVar (Int, [MVar ()]))

newQSem :: Int -> IO QSem
newQSem init = do
  sem ← newMVar (init,[])
  return (QSem sem)

waitQSem :: QSem -> IO ()
waitQSem (QSem sem) = do
  (avail,blocked) ← takeMVar sem
  if avail > 0
     then putMVar sem (avail-1, [])
     else do
       block ← newEmptyMVar
       putMVar sem (0, blocked++[block])
       takeMVar block

signalQSem :: QSem -> IO ()
signalQSem (QSem sem) = do
  (avail,blocked) ← takeMVar sem
  case blocked of
    [] -> putMVar sem (avail+1, [])
    (block:blocked') -> do
      putMVar sem (0, blocked')
      putMVar block ()
```

Diese sind in ConcurrentHaskell so vordefiniert. Dabei wird die Queue verwendet, um die Reihenfolge der wartenden Prozesse zu berücksichtigen – ohne dies wäre es auch einfacher zu implementieren.

9.5 Message Passing

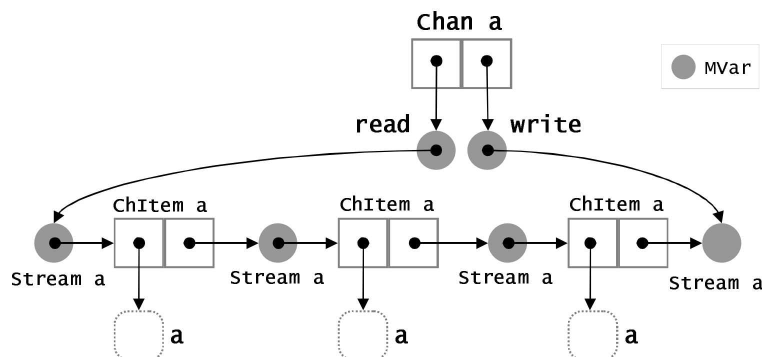
Alternative Sichtweise: `MVars` können auch wie Kanäle bzw. Puffer mit Kapazität Eins gesehen werden. Somit ist *Message Passing* als Programmierstil möglich: Ein System kann zerteilt werden in Teilkomponenten wie Server und Clients, die wechselseitige Anfragen stellen und Daten verwalten.

ConcurrentHaskell ermöglicht dies mittels des Datentyps `Chan a`. Eine mögliche Implementierung ist eine `MVar`, die eine Queue der Werte enthält. Nachteil wäre, daß gleichzeitiges Lesen und Schreiben in nicht-leeren Kanälen nicht möglich wäre; es wäre eine Liste der suspendierten Threads nötig.

Beachte: `MVar`s können wie Zeiger implementiert werden, somit bessere Implementierung möglich.

```
data Chan a = Chan (MVar (Stream a))
                   (MVar (Stream a))
type Stream a = MVar (ChItem a)
data ChItem a = ChItem a (Stream a)
```

Die Idee hinter dieser Datenstruktur ist eine verkettete Liste (jeweils über `MVar`s):

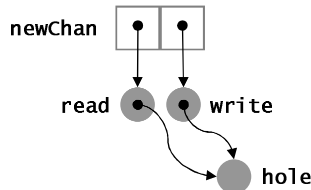


Nun können wir einen neuen leeren Kanal anlegen:

```
newChan :: IO (Chan a)
newChan = do
    hole  ← newEmptyMVar
    read  ← newMVar hole
    write ← newMVar hole
```

```
  return (Chan read write)
```

Dieser sieht folgendermaßen aus:



Nun können wir lesen und schreiben für Kanäle implementieren:

```
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ write) v = do
  newHole ← newEmptyMVar
  oldHole ← takeMVar write
  putMVar write newHole
  putMVar oldHole (ChItem v newHole)
  return newHole

readChan :: Chan a -> IO a
readChan (Chan read ) = do
  readEnd ← takeMVar read
  ChItem v newReadEnd ← readMVar readEnd
  putMVar read newReadEnd

isEmptyChan :: Chan a -> IO Bool
isEmptyChan (Chan read write) = do
  readEnd ← takeMVar read
  writeEnd ← readMVar write
  putMVar read readEnd
  return (readEnd == writeEnd)
```

Beachte: Falls ein Thread mit `readChan` auf leerem `Chan` suspendiert, blockiert diese Implementierung von `isEmptyChan`.

Weitere Funktionen:

- Duplizieren eines Kanals:

  ```
  dupChan :: Chan a -> IO (Chan a)
  ```

  Es entsteht eine Art Broadcast-Channel. Zunächst ist die Kopie leer, aber alle Werte, die nach Ausführen von `dupChan` in `Chan` geschrieben werden, landen in beiden `Chans`.

- Einfügen eines Werts am read-Ende in `Chan`:

```
ungetChan :: Chan a -> a -> IO ()
```

- Ein unendlicher Strom aller Nachrichten:

```
getChanContent :: Chan a -> IO [a]
```

  Beachte: Die Liste ist auch außerhalb der IO-Monade verwendbar, es wird ggf. suspendiert, falls noch kein Wert in `Chan` vorhanden.

- Schreiben von Listen:

```
writeListToChan :: Chan a -> [a] -> IO ()
writeListToChan = mapM_ writeChan
```

Nachteile von ConcurrentHaskell:

- Synchronisation erfolgt lock-basiert, dies kann in komplexeren Systemen eine Komposition von Teilsystemen erschweren.
- Schwer durchschaubarer Code, geringe Skalierbarkeit.
- Zur Komposition müssen Kommunikationsinterna bekannt sein.

9.6 Memory Transactions

Aus dem Datenbankbereich bekannt ist das **Transaktionskonzept**: Eine Reihe von Datenbank-Transaktionen werden definiert und dann atomar durchgeführt oder, falls ein Konflikt auftritt, verworfen bzw. neu gestartet. Es gibt keine expliziten Locks.

Frage: Kann dieses Konzept zur nebenläufigen Programmierung eingesetzt werden? Ja, es ist als *Software Transactional Memory* bekannt und in ghc (ab Version 6.4, vielleicht 6.2) im Modul `Control.Concurrent.STM` implementiert.

Interface: Transaktionen „leben" in einer Monade:

```
data STM a
instance Monad STM
```

Hier sind nur STM-Aktionen möglich, aber keine Veränderung der Welt (also kein `IO`). Die wichtigsten Funktionen sind:

- Atomare Ausführung einer Transaktion:

  ```
  atomically :: STM a -> IO a
  ```

  Ergebnis der Transaktion wird in die IO-Monade geliftet.

- Fehlschlag in der aktuellen Transaktion, erneuter Start der gesamten Transaktion:

  ```
  retry :: STM a
  ```

Analog zu `IORef` definiert man in der STM-Monade

```
data TVar a

newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM a
```

Beachte: In der STM-Monade können keine IO-Funktionen ausgeführt werden, d.h. es gibt keine Seiteneffekte. Somit ist das Verwerfen einer angefangenen Transaktion möglich, unter „Reparatur“ der veränderten `TVars`.

Beispiel: Anwendung auf die dinierenden Philosophen:

```
typeStick = TVar Bool
takeStick :: Stick -> STM ()
take Stick s = do
  b ← readTVar
  if b then writeTVar s False
       else retry

putStick :: Stick -> STM ()
putStick s = writeTVar s True

phil:: Int -> Stick -> Stick -> IO ()
phil n l r = do
  atomically $ do
    takeStick l
    takeStick r
  putStrLn (show n ++ ". Phil is eating\n")
  atomically $ do
    putStick l
    putStick r
  phil n l r
```

Diese Implementierung hat keinen Deadlock.

Transaktionen können sequentiell komponiert werden ohne zu wissen, wie ihr eigentliches Kommunikationsverhalten ist (z.B. muss `takeStick` für Komposition in `atomically` nicht bekannt sein). Der Programmierer weiß nur, dass sie ganz ausgeführt werden oder über ein `retry` fehlschlagen und somit ganz wiedergestartet werden. Es ergibt sich eine hohe Kompositionalität.

Weitere Möglichkeit der Komposition sind **Alternativen**:

```
orElse :: STM a -> STM a
```

`t1 `orElse` t2` führt Transaktion `t1` aus. Wenn `t1` fehlschlägt (mit `retry`), wird `t2` ausgeführt. Falls auch `t2` fehlschlägt, wird die gesamte Transaktion erneut gestartet. Wieder verhält sich die Komposition unabhängig von den Interna der einzelnen Transaktionen.

Beispiel: Implementieren von `MVars`

```
type MVar a = TVar (Maybe a)

newEmptyMVar :: STM (MVar a)
newEmptyMVar = newTVar Nothing

takeMVar :: MVar a -> STM a
takeMVar mv = do
  v ← readTVar mv
  case v of
    Nothing  -> retry
    Just val -> do
      writeTVar mv Nothing
      return val

putMVar :: MVar a -> a -> STM ()
putMVar mv val = do
  v ← readTVar mv
  case v of
    Nothing -> writeTVar mv (Just val)
    Just _  -> retry

tryPutMVar :: MVar a -> a -> STM Bool
tryPutMVar mv val = do
    putMVar mv val
    return true
```

```haskell
  `orElse`
    return False
```

`retry` sieht nach busy-waiting aus, aber während dem Ausführen einer Transaktion werden verwendete Resourcen (`TVars`) protokolliert und Transaktion nur wiedergestartet, falls die Resourcen verändert wurden.

Beispiel: Kanäle unbeschränkter Kapazität

```haskell
data MChan a = MChan (TVar (Stream a))
                     (TVar (Stream a))

type Stream a = TVar (ChItem a)

data ChItem a = Empty
              | Full a (Stream a)

newChan :: STM (MChan a)
newChan = do
  end   <- newTVar Empty
  read  <- newTVar end
  write <- newTVar end
  return (MChan read write)

writeMChan :: MChan a -> a -> STM ()
writeMChan (MChan _ write) v = do
  newEnd <- newTVar Empty
  writeEnd <- readTVar write
  writeTVar writeEnd (Full v newEnd)
  writeTVar write newEnd

readMChan :: MChan a -> STM a
readMChan (MChan read _) = do
  readEnd <- readTVar read
  item    <- readTVar readEnd
  case item of
    Empty -> retry
    Full v newReadEnd -> do
      writeTVar read newReadEnd
      return v

isEmtpyChan :: MChan a -> STM Bool
```

```haskell
isEmptyChan (MChan read write) = do
  readEnd  ← readTVar read
  writeEnd ← readTVar write
  return (readEnd == writeEnd)
```

10 Erlang

Die Sprache Erlang[14] ist eine funktionale Sprache, die von Ericsson entwickelt wurde. Infos gibt es unter `www.erlang.org`. Die Sprache ist ungetypt (bis auf ein schwaches Typsystem für Basistypen) und hat eine strikte Auswertung. Sie hat die Syntax zum Teil von ihrem „Vorbild" Prolog übernommen.

Dateien sind in Erlang Module, wobei der Dateiname gleich dem Modulnamen plus `.erl` sein muß. Jede Datei muß entsprechend mit einer Moduldeklaration und einer Export-Liste beginnen:

```
-module(fac).
-export([fac/1]).
```

Nun kann man Funktionen definieren:

```
fac(0) -> 1;
fac(N) when N>0 -> N * fac(N-1).
```

Verwendung des Erlang-Systems[15]

```
erl
: ...
>c(fac).          // compile
ok
>fac:fac(5).
120
```

Ausdrücke in Erlang sind Terme mit Applikationsklammerung, z.B. `f(e`$_1$`, ..., e`$_n$`)`. Statt `let`-Ausdrücken kann man Bind-Once-Variablen und Sequenzen benutzen:

```
fac(N) when N>0 -> N1 = N-1,
                   F1 = fac(N1),
                   N*F1.
```

Datenstrukturen (bzw. Konstruktoren) sind:

- **Atome** sind Zahlen oder beginnen mit Kleinbuchstaben (mit Großbuchstaben beginnen Variablennamen, soll ein mit einem Großbuchstaben beginnender String als Atom angegeben werden, so wird er in Hochkommata eingeschlossen: `a`, `b`, `c`, ..., `0`, `1`, ..., `hallo`, `'Hallo'`

[14] Abkürzung für Ericsson Language bzw. Name des dänischen Mathematikers Agner Krarup Erlang

[15] Erlangist auf den Instituts-Rechnern unter ~`erlang/bin` zu finden.

- **Listen**: `[e|l]` statt `e:l` (in Haskell), leere Liste wie gewohnt `[]`. Dabei können `e` und `l` aber – aufgrund des fehlenden Typsystems – beliebige Datenstrukturen sein, nur vordefinierte Funktionen wie `++` erwarten, daß `l` wieder eine Liste ist. Abkürzung: `[`e_1`, ..., `e_n`]` statt `[`e_1`|[...|[`e_n`|[]]...]]`.

- **Tupel**: `{`e_1`, ..., `e_n`}`, auch zur Kodierung beliebiger Datenstrukturen möglich, z.B. sind Bäume definierbar als `{node, {node, empty, 1, empty}, 2, empty}`.

Pattern Matching erfolgt mittels `pat = e`, wobei `pat` ein Konstruktorterm über Variablen ist.

Zur Semantik: `e` wird zu einer *Normalform (NF)* `v` reduziert; eine Normalform ist dabei ein Term ohne Funktionsaufrufe ohne Variablen, d.h. ein reiner Konstruktorterm. Machte dann `pat` gegen `v` mittels Substitution ϱ, die auf den gesamten zu reduzierenden Ausdruck angewendet wird – Substitutionen werden hier mittels `[x/y]` beschrieben.

$$\begin{aligned} \text{match}(c(p_1,\dots,p_n), c(v_1,\dots,v_n)) &= \text{match}(p_1,v_1) \uplus \dots \\ &\quad \uplus \text{match}(p_n,v_n) \uplus \texttt{[]} \\ \text{match}(X,v) &= \texttt{[}X/v\texttt{]} \\ \text{match}(_,_) &= \texttt{Fail} \end{aligned}$$

Dabei ist die Vereinigung von Substitutionen ϱ_1 und ϱ_2 definiert als:

$$\begin{aligned} \texttt{Fail} \uplus _ &= \texttt{Fail} \\ _ \uplus \texttt{Fail} &= \texttt{Fail} \\ \varrho_1 \uplus \varrho_2 &= \begin{cases} \varrho_1 \cup \varrho_2 & \text{falls} \quad \forall X \in \text{dom}(\varrho_1) \cap \varrho_2 \colon \varrho_1(X) = \varrho_2(X) \\ \texttt{Fail} & \text{sonst} \end{cases} \end{aligned}$$

Beispiel:

```
N = {value, 7} ⇝ [N/{value, 7}]
{value, N} = {value, 7} ⇝ [N/7]
{N, N} = {value, 7} ⇝ Fail
{N, N} = {7, 7} ⇝ [N/7]
[X|L] = [1,2,3] ⇝ [X/1, L/[2,3]]
```

Verzweigung ist möglich mittels `case`:

```
case e of
  pat1 -> e1;
```

```
  ...
  patn -> en
end
```

10.1 Nebenläufige Programmierung

Eine Erlang-Umgebung wird als Erlang-Knoten bezeichnet, auf dem mehrere Prozesse ausgeführt werden können.

Das **Starten neuer Prozesse** geschieht mittels `spawn(module, func, [`v_1`, ..., `v_n`]`. Dies generiert einen neuen Prozess, welcher mit der Berechnung von `module:func([`v_1`, ..., `v_n`])` beginnt. Das funktionale Ergebnis von spawn ist der Prozess-Identifier des neuen Prozesses. Dabei muß aber die Funktion `func` vom Modul `module` exportiert werden, andernfalls wird der Prozess ohne Fehlermeldung nicht gestartet!

Das **Senden von Nachrichten** an einen Prozess geschieht mit `pid ! v` – dabei wird allgemein bei e_1 `!` e_2 zunächst e_1 reduziert, dann e_2, und dann wird die Nachricht verschickt, d.h. `v` wird an die Mailbox (Queue) des Prozesses `pid` angefügt.

Beachte: Auch Prozess-IDs sind Werte, die ebenfalls verschickt werden können. Der Term `self()` ist eine „Konstante" mit der Prozess-ID des eigenen Prozesses.

Der **Empfang** von Nachrichten geschieht nun per Pattern-Matching auf jede Nachricht der Mailbox:

```
receive
  pat1 -> e1;
  ...
  patn -> en
after t -> e  // optional
end
```

Semantik:

- Matche die Pattern der Reihe nach gegen ersten (ältesten) Eintrag der Mailbox, dann den zweiten usw. Das erste erfolgreiche Matching ϱ von pat_i führt dazu, daß der matchende Wert aus der Mailbox entfernt wird und mit der Berechnung von $\varrho(e_i)$ fortgefahren wird.
- Falls kein erfolgreiches Matching stattfindet, suspendiert der Prozess, bis eine neue Nachricht in der Mailbox gelandet ist, dann wird wieder gematcht.

- Falls die Zeit `t` verstrichen ist (in Millisekunden), so reduziere zu `e`.
- Im Spezialfall `t` = 0 wird die komplette Mailbox genau einmal gematcht, danach aber im Fehlerfall nicht suspendiert.

Beispiel: Datenbank-Server zum Speichern von Schlüssel-Wert-Paaren:

```
-module(dataBase).
-export([start/0]).

start() -> dataBase([]).

dataBase(Es) -> receive

    {allocate, Key, P} ->
      case lookup(Key, Es) of
        nothing -> P ! free,
                   receive
                     {value, V, P} ->
                       dataBase([{Key, V}|Es])
                   end;
        {just, V} -> P ! allocated,
                     dataBase(Es)
      end;

    {lookup, Key, P} ->
      P ! lookup(Key, Es),
      dataBase(Es)

  end

lookup(K, []) -> nothing;
lookup(K, [{K, V}|_]) -> {just, V};
lookup(K, [_|KVs]) -> lookup(K, KVs).
```

Der zugehörige Client:

```
-module(dBClient).
-export([client/1,main/0]).

main() -> DB = spawn(dataBase, start, []),
          client(DB).
```

```
client(DB) -> case read ('(l)ookup, (i)nsert') of
    ... // kommt noch
```

10.2 Verteilte Programmierung

Viele Anwendungen haben verteilten Charakter, d.h. sie laufen nicht nur auf einem System – etwa Chat-Programme oder Anwendungen aus dem Bereich Banken oder Telekommunikation. Erlang bietet ein einfaches Konzept zum verteilen Programmieren. Starte hierzu mehrere Erlang-Knoten z.B. im Internet (oder z.B. zum Testen auf einem Rechner) mittels:

```
erl -name ⟨name⟩
```

So startet zum Beispiel `erl -name abc` einen Knoten mit dem (in Erlang atomaren) Namen `'abc@hostname'`. Dann kann man auf diesem Knoten Prozesse starten mittels:

```
spawn(Knoten, Module, Function, [v1, ..., vn]
```

Nun wird ein neuer Prozess auf einem anderen Knoten gestartet. Dabei ist die generierte Prozess-ID im gesamten Knoten-System eindeutig (da der Knotenname mit in der Prozess-ID einbezogen ist).

Beispiel: Auslagern des Datenbankservers z.B. zur Lastverteilung:

```
main() -> DB = spawn('server@schnellerRechner',
                     dataBase, start, []),
          client(DB).
```

Weiteres Problem: Für offene verteilte Systeme, in denen sich die Kommunikationspartner vorab zunächst nicht kennen müssen (z.B. bei einem Chat), muß eine Möglichkeit bestehen, zur Laufzeit eine Verbindung zu einem anderen Knoten herzustellen. Die Registrierung einer „Kontaktstelle“ (d.h. das Starten eines Server-Sockets) geschieht mittels:

```
register(Name, Pid)
```

Nun kann man an einen registrierten Prozess (d.h. einem Server auf einem Knoten) eine Nachricht senden mittles

```
{Name, Knoten} ! v
```

Die Nachricht `v` wird dann in die Mailbox des auf dem Knoten unter `Name` registrierten Prozesses eingetragen. Das Senden an registrierte Prozesse sollte nur für den „Erstkontakt“ verwendet werden, danach sollten die Prozess-IDs ausgetauscht werde, um Skalierbarkeit zu gewährleisten.

Beispiel: Angenommen, der Datenbank-Server wurde gestartet auf einem Knoten mit Bezeichner `'dbServer@sauron.informatik.uni-kiel.de'`.

```
-module(dataBase).
-export([start/0, dataBase/1]).

start() ->
  register(dbServer, spawn(dataBase, dataBase, [[]])).

dataBase(Es) -> receive

    {allocate, Key, P} -> ...
    {lookup, Key, P} -> ...

    {connect, P} ->
      P ! {connected, self()},
      dataBase(Es)

  end
```

Beim Start des Cleints muß nun eine Verbindung zum Datenbankserver hergestellt werden:

```
-module(dBClient).
-export([client/1,main/0]).

main ->
  {dbServer, 'dbServer@sauron...'} !
                                    {connect, self()},
  receive
    {connected, DB} -> client(DB)
  end.
```

Beachte: Es sind nur wenige Änderungen beim Übergang von der nebenläufigen zur verteilten Programmierung notwendig – im Gegensatz zu Java, wo verteilte Programmierung als Erweiterung der sequentiellen Programmierung angesehen wird (Remote Method Invocation, RMI).

Beispiel: Es soll ein Chat in Erlang programmiert werden. Zunächst einige Basisfunktionen:

```
-module(base).
-export([lookup/2,remove/2,fst/1,snd/1]).

lookup(Key,[]) -> nothing;
lookup(Key,[{Key,Value}|_]) -> {just,Value};
```

```
lookup(Key,[_|Xs]) -> lookup(Key,Xs).

remove(Key,[]) -> [];
remove(Key,[{Key,Value}|Xs]) -> remove(Key,Xs);
remove(Key,[X|Xs]) -> [X|remove(Key,Xs)].

fst({X,Y}) -> X.

snd({X,Y}) -> Y.
```

Der Server ist nun folgendermaßen implementiert:

```
-module(chatServer).
-export([startChat/0]).
-import(base,[lookup/2]).

startChat() ->
  process_flag(trap_exit,true),
  register(chat,self()),
  chatServer([]).

chatServer(Clients) ->
  receive
    {message,P,M} -> case lookup(P,Clients) of
                       {just,Name} -> sendAll(Clients,{message,Name,M});
                       nothing     ->
                         io:format("Unknown client message from ~w~n",[P])
                     end,
                     chatServer(Clients);
    {connect,P,Name} -> link(P),
                        P!{connected,self(),
                           lists:map({base,snd},Clients)},
                        sendAll(Clients,{connect,Name}),
                        chatServer([{P,Name}|Clients]);
    {disconnect,P} -> case lookup(P,Clients) of
                       {just,Name} ->
                         NewClients=lists:filter(fun({P1,_}) -> P1/=P end,
                                                 Clients),
                         sendAll(NewClients,{disconnect,Name}),
                         chatServer(NewClients);
                       nothing     ->
                         io:format("Unknown disconnect from ~w~n",[P]),
                         chatServer(Clients)
                     end;
    {'EXIT',P,_} -> self()!{disconnect,P},
                    chatServer(Clients);
    Msg            -> io:format("Unknown Message: ~w",[Msg]),
                        chatServer(Clients)
  end.

sendAll(Clients,M) -> lists:map(fun({P,_}) -> P!M end,Clients).
```

Der Client:

```
-module(chatClient).
-export([joinChat/2,inputLoop/1]).

joinChat(Node,Name) ->
   {chat,Node}!{connect,self(),Name},
   receive
     {connected,Server,Names} ->
        io:format("Welcome in chat room. You are chating with:~n",[]),
```

```
        lists:map(fun(N)->io:format("~s~n",[N]) end,Names),
        spawn(chatClient,inputLoop,[self()]),
        chatLoop(Server)
  after 200000 -> io:format("Specified Server not reachable~n",[])
  end.

inputLoop(ComP) ->
  case io:get_line('') of
    "bye\n" -> ComP!bye;
    Str     -> ComP!{own,Str},
               inputLoop(ComP)
  end.

chatLoop(Server) ->
  receive
    {disconnect,N} -> io:format("~s left the chat.~n",[N]);
    {connect,N}    -> io:format("~s joined the chat.~n",[N]);
    {message,N,M}  -> io:format("~s: ~s",[N,M]);
    bye            -> Server!{disconnect,self()},
                      exit(bye);
    {own,M}        -> Server!{message,self(),M};
    Other          -> io:format("Unknown message: ~w~n",[Other])
  end,
  chatLoop(Server).
```

10.3 Robuste Programmierung

Verteilte Systeme müssen sich robus gegenüber Ausfällen von Komponenten verhalten. In Erlang wird dies ermöglicht mittels Linking:

```
link(Pid)
```

Damit etabliert man eine bidirektionale Verbindung zwischen dem ausführenden und dem mittels `Pid` angegebenen Prozess. Wird ein Prozess beendet (Terminierung oder Absturz), so gibt es zwei mögliche Verhalten des anderen Prozesses:

1. Er wird auch beendet (Standard).
2. Er bekommt eine Fehlernachricht geschickt mit dem Grund des Beendens.

Im obigen Chat-Beispiel ist letzte Variante gewählt, durch die Anweiseung

```
process_flag(trap_exit,true),
```

erhält der Server eine Nachricht `{'EXIT',P,...}`, sobald z.B. ein Teilnehmer „abstürzt“.

Weitere Features:

- Sicherung der Kommunikation durch „Cookies“,

- verteilte Datenbank MNESIA
- Funktionen höherer Ordnung sind möglich,
- Verschicken von Funktionen ist möglich,
- man kann Code während der Laufzeit austauschen,
- generische Prozessarchitekturen (z.B. genServer)