

Inhaltsverzeichnis

Einführung	3
Haskell-Skript.....	3
Workflow	3
Pure Funktionen	4
Evaluationsreihenfolge.....	4
Funktionale Zerlegung.....	6
Divide-and-Conquer	6
Rekursion.....	6
Typen	7
Typen-Konstruktor	7
Bool.....	7
Aufzählungstyp (Enum) selbst definieren	8
Funktionen definieren mithilfe von Pattern-Matching	8
Polymorphismus.....	9
Typing	11
Aufzählungstyp (Enum) erweitert	12
Schlüsselwort: „case“	13
Schlüsselwort „let“	13
Schlüsselwort „where“	14
Typ-Deklarationen mit Typ-Variablen	14
Tupel.....	15
Lineare Ordnung – Ord.....	15
Char (Zeichen)	16
Strings (Zeichenkette)	16
Schlüsselwort „type“	17
„Show“.....	17
Funktionen höherer Ordnung	18
Schönfinkeln/Currying.....	18
Sections/Abschnitte und der Anwendungsoperator.....	19
λ -Abstraktionen (Lambda).....	19
Schleifen	20

Parser und Type-Inference	21
Listen	22
Unveränderliche Daten	23
„Replicate“	23
Rezept für rekursive Funktionen	23
„last“-Funktion und @-Pattern.....	24
Listen mithilfe des Aufzählungstypen (Enum) erstellen.....	24
Summen und Produkte.....	25
Funktionen höherer Ordnung und Listen.....	25
Listen-Abstraktion (List Comprehensions)	26
„Fold“	26
Foldr („Fold right“)......	26
Foldl („Fold left“)	27
List-Laws	27
Module und abstrakte Datentypen.....	28
Sichtbarkeitsbereich/Geltungsbereich (Scope).....	28
Module	28
Qualifizierte Import-Deklaration (Qualified Imports)	29
Abstrakte Datentypen	29
„Newtype“	30
Effizienz/Lazy Evaluation	31
Rekursionsarten	31
Strikte Evaluation	32
Unendliche Listen („Infinite Lists“)......	32
I/O: Input and Output.....	33
I/O und das Typensystem.....	33
Do-Notation.....	34
Weitere Anmerkungen.....	34
Funktionen höherer Ordnung und I/O	35
I/O-Monade.....	35

Einführung

Funktionale Programmierung:

- Definiert Funktionen, keine Zuweisungen
- Computer evaluiert Gleichungen durch Reduktion

Vorteile:

- Intuitive Evaluationsmechanismen
- Verifikation möglich
- Expressive Language Features
- Parallelisierung möglich

Nachteile:

- State, Nebenwirkungen, und I/O schwer zu modellieren
- Nicht main-stream

Haskell-Skript

Ein Haskell-Skript besteht aus (mehreren) definierten Funktionen.

Jede Funktion besteht aus...

```
type declaration: square :: Integer -> Integer
syntax: <name> :: <type>
defining equation: square x = x * x
syntax: <name> <vars> = <exp>
evaluation from left to right
<name>s and <vars>
  • always start with lower-case letters
  • consist of letters, digits and _
```

Kommentare werden in der Evaluation ignoriert.

- single: `-- everything right of -- is a comment`
- multi: `{- comments can deactivate ... parts of script -}`

Workflow

- Funktionen werden in Skript definiert
- Skript wird geladen (Skript kompiliert oder produziert eine Error-Nachricht)
- REPL (Read-Evaluate-Print Loop)
 - Normale Form: Werte, welche nicht weiter vereinfacht werden können, z.B.: 5

- Evaluierungen mit...
 - ...integrierten/eingebauten Funktionen, definiert in „Prelude“.
 - ...vom User selbst definierte Funktionen im Skript.

Ähnlich wie der Workflow eines Taschenrechners.

Pure Funktionen

Eine Funktion ist „pur“, wenn sie beim selben Input immer dasselbe Ergebnis liefert. Ähnlich wie mathematische Funktionen. Haskell ist eine pure Programmiersprache.

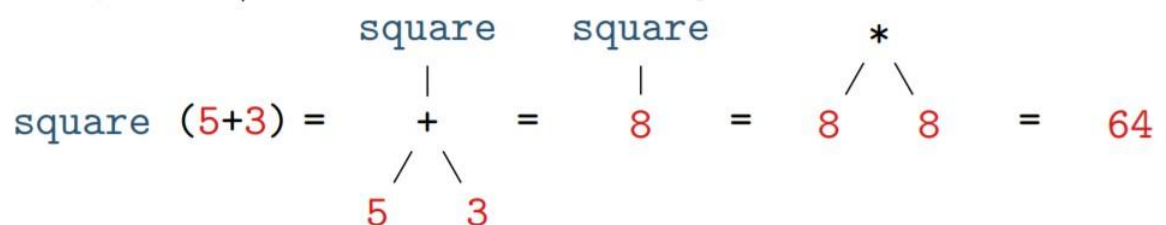
Evaluationsreihenfolge

In puren Programmiersprachen hat die Reihenfolge keine Auswirkung auf das Ergebnis der Evaluation.

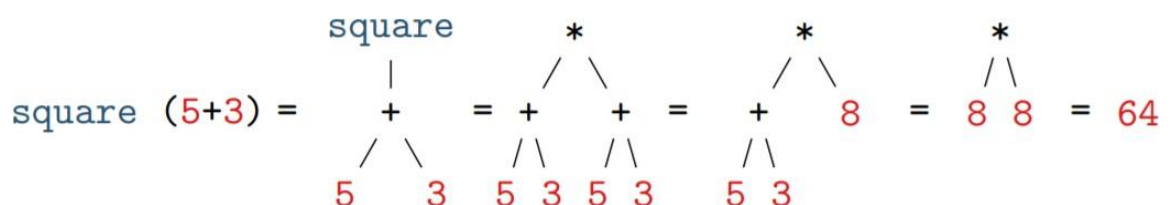
Standard-Evaluationsreihenfolgen

Jede funktionale Programmiersprache hat eine fixe Evaluationsreihenfolge. Es gibt drei prominente Strategien:

- **Call-by-Value/Strikte Evaluation**
Evaluert die Argumente zuerst.



- **Call-by-Name/Nicht-Strikte Evaluation**
Ersetzt die angewandte Funktion direkt mit ihrer rechten Seite.



- **Call-by-Need/Lazy Evaluation** (Call-by-Name mit Sharing)

$$\text{square } (5+3) = \begin{array}{c} \text{square} \\ | \\ + \\ / \quad \backslash \\ 5 \quad 3 \end{array} = \begin{array}{c} * \\ (\quad) \\ + \\ / \quad \backslash \\ 5 \quad 3 \end{array} = \begin{array}{c} * \\ (\quad) \\ 8 \end{array} = 64$$

Haskell verwendet hauptsächlich Lazy Evaluation mit einer links-nach-rechts Reihenfolge für Argumente.

Theorem: Wenn es eine berechenbare Normalform gibt, kann diese immer durch nicht-strike oder Lazy Evaluation zu dieser Normalform berechnet werden.

Die Semantik von **if-then-else** ist **nicht-strikt**.

Bottom \perp

Eine Funktion ist strikt, wenn das Ergebnis immer \perp ist, sobald eines der Argumente \perp ist.

Prioritäten

Vorrang/Priorität	Operatoren	Assoziativität
9	!!, .	links (!!), rechts (.)
8	^, ^^, **	rechts
7	*, /, `div`	links
6	+, -	links
5	:, ++	rechts
4	==, /=, <, <=, >, >=	-
3	&&	rechts
2		rechts
1	>>, >>=	links
0	`, \$!, `seq`	rechts

Parsing

Ein Parser wandelt eine Eingabe in ein für Weiterverarbeitung geeignetes Format um. Parsing ist ein Hauptbestandteil eines jeden Compilers.

Funktionale Zerlegung

Verschiedene Unteraufgaben sollen herausgesucht und dann in verschiedene Funktionen („**pro Aufgabe, eine Funktion**“) aufgeteilt werden. Hilfreich für Struktur, Übersicht und Arbeit.

Divide-and-Conquer

Das Problem wird, wenn es nicht trivial lösbar ist, in gleichartige Teilprobleme aufgeteilt. Die Teilprobleme werden rekursiv gelöst und schlussendlich zu einer Gesamtlösung vereint.

Rekursion

Eine rekursive Funktion ist eine Funktion, welche sich selbst aufruft.

Konzepte

- Rekursionsanfang:
Ohne Rekursionsanfang wäre die Rekursion unendlich (Abbruchbedingung).
- Rekursionsschritt:
Rekursiver/Erneuter Aufruf der Funktion.

Typen

Ein Typ kann als eine „Sammlung von Werten“ gedacht werden. Ein Beispiel für einen Typ:

Bool = {True, False}

Typensignatur/Typen-Beschränkung: $e :: t$ (Ausdruck „e“ ist vom Typen „t“)

Haskell hat ein **statisches Typensystem**, das bedeutet, dass zur Zeit der Kompilierung die Typen festgelegt sind und zur Laufzeit nicht mehr verändert werden können. Errors aufgrund von Typen werden zur Kompilierzeit überprüft.

Typen-Konstruktor

Ein Typen-Konstruktor erstellt einen Typen. Es kann sich dabei um einen Typ selbst handeln (Integer, Bool, etc.) und möglicherweise werden Argumente benötigt. Typen-Konstruktoren beginnen immer mit einem Großbuchstaben.

-> ist ein Typen-Konstruktor für Funktions-Typen.

Bool

Bool ist ein eingebundener Typen-Konstruktor mit genau zwei Konstruktoren, welche Wahrheitswerte repräsentieren. True und False.

True, False :: Bool

Wichtig: Konstruktoren erstellen Werte (von einem Typ). Typen-Konstruktoren erstellen Typen. Beispiel:

- Die Konstruktoren (True, False) erstellen Werte von Bool.
- Der Typ-Konstruktor (Bool) erstellt den Typen.

Funktionen für Boolesche Werte:

- `not :: Bool -> Bool` is negation, swaps `True` and `False`
- `(&&) :: Bool -> Bool -> Bool` is **conjunction**
 - `a && b` is `True` iff both `a` and `b` are `True`
 - `a && b` is non-strict in its second argument
- `(||) :: Bool -> Bool -> Bool` is **disjunction**
 - `a || b` is `False` iff both `a` and `b` are `False`
 - disjunction \neq exclusive-or
 - `a || b` is non-strict in its second argument

Aufzählungstyp (Enum) selbst definieren

Mit dem Schlüsselwort „**data**“ können neue Datentypen selbst definiert werden. Ein Beispiel:

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Variablen selbst-definierter Datentypen ausgeben: Schlüsselwort „Show“

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun deriving Show
```

```
ghci> Mon
Mon
```

Funktionen definieren mithilfe von Pattern-Matching

Ein Pattern (Muster) kann ein Konstruktor-Name, eine Variable oder `_` sein. Ein Beispiel:

```
weekend :: Weekday -> Bool
```

```
weekend Sat  = True
weekend Sun  = True
weekend _    = False
```

Die Funktion „**weekend**“ bekommt eine Variable des Datentyps **Weekday** und gibt einen **Booleschen Wert** zurück. Die Funktion überprüft mithilfe eines Musters, ob es sich bei der Variable um „**Sat**“ oder „**Sun**“, also Samstag oder Sonntag, handelt und gibt in jedem anderen Fall „**False**“ zurück.

Die Muster werden *von oben nach unten evaluiert*. Das bedeutet als Erstes „weekend Sat“, dann „weekend Sun“ usw.

Polymorphismus

Mit Typen-Variablen (a, b, ...) kann jeder Typ dargestellt werden.

Overloading

„Überladen“ wird verwendet, damit derselbe (Funktions-)Name mehrere Bedeutungen haben kann. Dabei können Funktionen oder Typen-Klassen überladen werden.

Beispiel:

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

instance Eq Bool where -- def. for Bool
    True == b = b
    False == b = not b

instance Eq Weekday where -- def. for Weekday
    Mon == Mon = True
    Tue == Tue = True
    ... == ... = True -- Wed, Thu, Fri, Sat
    Sun == Sun = True
    _ == _ = False
```

Typen-Klassen (funktionieren wie ein „Blueprint“) können mehrere Funktionen definieren.

Um diesen Entwurf zu instanziiieren, müssen alle Funktionen definiert werden. Für einen individuellen Typ, muss also eine *Instanz* erzeugt werden.

default-definitions definieren Funktionen bereits im Interface, damit nur die benötigten Funktionen definiert werden müssen. Ein Beispiel:

```
(==) :: Eq a => a -> a -> Bool
```

- `Eq` is a type-class which specifies the operator `==`
- there are multiple instances (= types) of `Eq`, where **each instance of `Eq` has its own definition** of `==` (overloading)
- `Eq a =>` is type-constraint which defines that the type-variable `a` can only be replaced by types that instantiate the `Eq` type-class

```

class Eq a where
    (==) :: a -> a -> Bool    -- equality
    (/=) :: a -> a -> Bool    -- disequality
    x == y = not (x /= y)     -- default def.
    x /= y = not (x == y)     -- default def.

```

Dank der default-definitions reicht es nun völlig aus eine der beiden Funktionen (==) oder (/=) zu definieren, um die Typen-Klasse „Eq“ zu instanziiieren.

Parameter-Polymorphismus:


Der Programmierer kann selbst polymorphe Funktionen definieren. Ein Beispiel:

```

if_equal :: Eq a => a -> a -> b -> b -> b
if_equal n m x y = if n == m then x else y

```

Nummer-Typen:

Integer (keinen Overflow), **Int** (Overflow möglich  aber schneller), **Float** und **Double** sind alles Instanzen der „Num“-Klasse.

- specification, all of:


```

      (+)      :: Num a => a -> a -> a
      (*)      :: Num a => a -> a -> a
      (-)      :: Num a => a -> a -> a
      abs      :: Num a => a -> a
      signum   :: Num a => a -> a
      fromInteger :: Num a => Integer -> a
      
```
- additional functions: `negate`, `4715 :: Num a => a, ...`

Fractional-Klasse:

- excerpt of functions


```

      (/) :: Fractional a => a -> a -> a
      
```

Voraussetzung: Num; Instanzen: Float, Double

Integral-Klasse:

- excerpt of functions

```
toInteger :: Integral a => a -> Integer
div       :: Integral a => a -> a -> a
mod       :: Integral a => a -> a -> a
```

Voraussetzungen: Num, Ord; Instanzen: Int, Integer

Ord-Klasse:

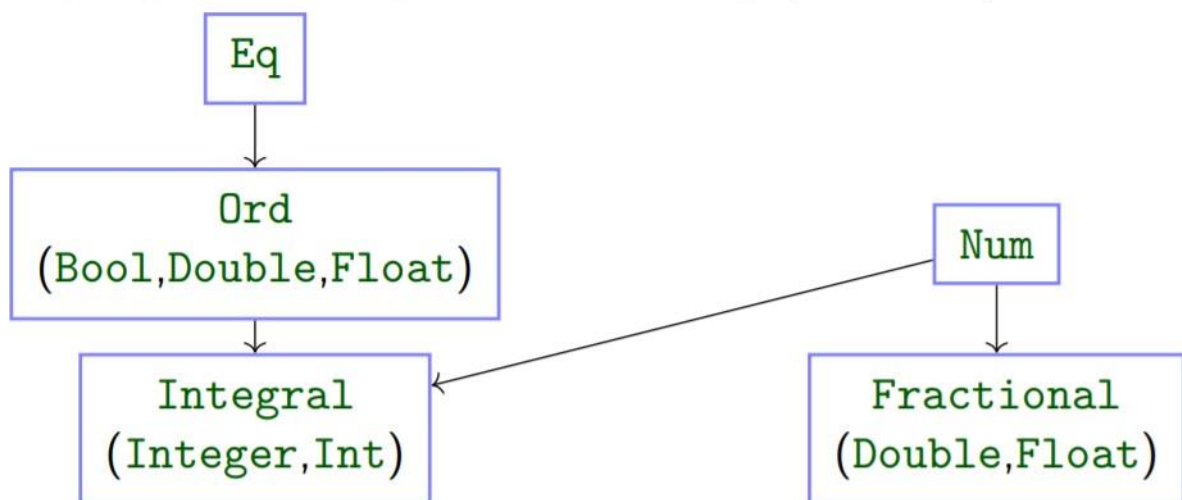
- specification, one of:


```
compare :: Ord a => a -> a -> Ordering
(<=)    :: Ord a => a -> a -> Bool
```
- where **data** Ordering = LT | EQ | GT
- additional functions: (<), (>=), (>), min, max

Voraussetzung: Eq; Instanzen: Int, Integer, Float, Double, Bool

Die Funktion **sqrt** kann auch selbst definiert werden. Eingebundene/Bereits vorhandene Funktionen können also des Öfteren auch vom User selbst definiert werden.

Typen-Klassen Hierarchie:



Typing

Type-Checking: Wird einer Funktion/einem Ausdruck ein Typ gegeben, wird die Definition überprüft.

- **fact** :: Int -> Int

Type-Inference: Deutet einen Typ für die Funktion/den Ausdruck an und schränkt den Datentyp ein, gibt allerdings keinen bestimmten Datentypen an.

- `fact :: (Ord a, Num a) => a -> a`

Explizite Typen-Angaben können die angedeuteten Datentypen weiter einschränken.

- `fact :: Integer -> Integer`

Unifikation: Funktionen mit derselben Typensignatur können in derselben Zeile aufgerufen werden. Dazu muss meistens durch **Type-Substitution** (das Abbilden von Typen-Variablen zu Typen) so generalisiert wie möglich werden.

Aufzählungstyp (Enum) erweitert

Das Schlüsselwort **data** ist grundsätzlich etwas allgemeiner definiert, als wir es bisher, rein für einen Aufzählungstypen, benutzt haben.

```
data [context =>] type tv1 ... tvi = con1  c1t1 c1t2... c1tn |
    ... | conm cmt1 ... cmtq
    [deriving]
```

© https://wiki.haskell.org/Type#Data_declarations

Jeder Konstruktor hat einen Typ und Argumente. Ein Beispiel:

- `data Temperature = Celsius Float | Fahrenheit Float`
 - `Temperature` is a new type
 - `Celsius :: Float -> Temperature` is a constructor of arity 1
 - `Celsius 19.7` is value of type `Temperature`

„**Temperature**“ ist ein neuer, selbst definierter Typ, welcher entweder den Konstruktor für „**Celsius**“ oder „**Fahrenheit**“ aufruft. `Celsius` ist ein Konstruktor, der einen Datentyp von „**Temperature**“ mit einem `Float`-Wert initialisiert und zurückgibt.

Dieser „**data**“-Typ, kann ebenfalls mit Pattern Matching verglichen werden.

Weiterführendes Beispiel:

```
data Temperature = Celsius Float | Fahrenheit Float
```

```
boiling_water (Celsius dc)    = dc >= 100
boiling_water (Fahrenheit df) = df >= 212
```

Schlüsselwort: „case“

Pattern Matching kann nicht nur auf der linken Seite, beim Definieren von Funktionen (s.o.), sondern auch schon auf der rechten Seite, mithilfe von „case“, angewendet werden.

$$\begin{array}{l} \text{case } e \text{ of } \langle pat_1 \rangle \rightarrow e_1 \\ \quad \vdots \\ \quad \langle pat_n \rangle \rightarrow e_n \end{array}$$

Der „case“ wird von oben nach unten durchgegangen und bei dem ersten „pat“ (steht für Pattern), der auf „e“ zutrifft, wird ausgeführt. Trifft auf „e“ also beispielsweise „pat₁“ zu, dann wird „e₁“ ausgeführt.

Das Layout bei „case“ ist wichtig, da Ausdrücke, welche in derselben Spalte anfangen, in eine Gruppe gepackt werden. Das kann durch die Verwendung von „{“ verhindert werden. Ein Beispiel:


```
and_1 b1 b2 = case b1 of
  True  -> case b2 of
    True  -> True
    False -> False
and_2 b1 b2 = case b1 of
  True  -> case b2 of
    True  -> True
    False -> False
```

In dem ersten Fall gehört das zweite „True“ und das „False“ zusammen in eine Gruppe. Im zweiten Fall gehört das erste „True“ und das „False“ zusammen. Das **Ergebnis**:

```
ghci> and_1 True False
False

ghci> and_2 True False
*** error: non-exhaustive patterns
```

Schlüsselwort „let“

Das Schlüsselwort „let“ wird für lokale Definitionen verwendet. Jeder „let“-Ausdruck kann mehrere Definitionen aufweisen und die Reihenfolge dieser ist egal. Außerhalb des Ausdrucks sind diese Definitionen nicht mehr sichtbar ( lokal!).

```
let
  pat      = exp  -- definitions by pattern matching
  fname args = exp  -- function definitions
in exp      -- result
```

Ein Beispiel:

```
f x
= let a = w x
  in case () of
    _ | cond1 x    -> a
      | cond2 x    -> g a
      | otherwise -> f (h x a)
```

© https://wiki.haskell.org/Let_vs._Where

Schlüsselwort „where“

Das Schlüsselwort „where“ wird ähnlich wie das Schlüsselwort „let“ verwendet. Es gelten dieselben Bestimmungen wie bei „let“, nur das „where“ eine andere Reihenfolge hat. Ein Beispiel:

```
f x
| cond1 x    = a
| cond2 x    = g a
| otherwise  = f (h x a)
where
  a = w x
```

© https://wiki.haskell.org/Let_vs._Where

Funktions-Definitionen können auch „guarded“ (geschützt) sein, bei welchen jede Kondition ein Boolescher Ausdruck ist (s.o.). Ein Syntax-Beispiel:

```
fname args
| cond_1 = exp_1
| cond_2 = exp_2
| ...
where ... -- optional where-block
```

Typ-Deklarationen mit Typ-Variablen

Beispiele:

- `data Pair a b = Pair_C a b` -- product type
- `data Either a b = Left a | Right b` -- choice/sum-type
- `data Maybe a = Just a | Nothing` -- option-type

„**Either**“ kann entweder die linke oder die rechte Seite aufrufen, während „**Maybe**“ entweder etwas zurückgibt oder eben nicht(s).

Weitere Beispiele: <http://learnyouahaskell.com/making-our-own-types-and-typeclasses>

Tupel

Ein sogenanntes Tupel besteht aus einer Liste endlich vieler, nicht notwendigerweise voneinander verschiedener Objekte mit einer bestimmten Reihenfolge.

Tupel können ebenfalls mit verschachtelten Paaren dargestellt werden. Ein Beispiel:

(8, 3.25, True) ~ (8, (3.25, True))

[Integer] [Float] [Bool]

Zusammenfassendes Beispiel:

- task: compute the sum of the roots of a quadratic polynomial
- solution with **explicit failure via option-type**

```
roots :: (Ord a, Floating a) => a -> a -> a -> Maybe (a,a)
roots a b c
  | a == 0 = Nothing
  | d < 0 = Nothing
  | otherwise = Just ((- b - r) / e, (- b + r) / e)
  where d = b * b - 4 * a * c
        e = 2 * a
        r = sqrt d

sum_roots :: (Ord a, Floating a) => a -> a -> a -> Maybe a
sum_roots a b c =
  case roots a b c of -- case for explicit error handling
    Just (x, y) ->    -- nested pattern matching
      Just (x + y)
    Nothing -> Nothing -- can't be repl. by "e -> e"! (types)
```

Lineare Ordnung – Ord

Die **lexikographische Ordnung** ist eine Methode, um aus einer linearen Ordnung für einfache Objekte, beispielsweise alphabetisch angeordnete Buchstaben, eine lineare Ordnung für zusammengesetzte Objekte, beispielsweise aus Buchstaben zusammengesetzte Wörter, zu erhalten. (Zitat Wikipedia)

Ein Beispiel für eine Implementation von „Ord“:

- obtain concise and systematic `Ord`-implementation for `T a b`

```
instance (Ord a, Ord b) => Ord (T a b) where
  compare (C_1 x1 y1) (C_1 x2 y2) = compare (x1,y1) (x2,y2)
  compare C_2 C_2                = compare () ()
  compare (C_3 z1) (C_3 z2)      = compare (z1) (z2)
  compare e f                    = compare (num_T e) (num_T f) where
    num_T (C_1 _ _) = (0 :: Int)
    num_T C_2       = 1
    num_T (C_3 _)   = 2
```

Char (Zeichen)

Char ist ein Typ für Zeichen '1', 'a', 'A', '\65' (via ASCII-Code). Mithilfe von dem Backslash werden Zeichen „escaped“ (maskiert). Beispiele für Zeichen, welche „escaped“ werden müssen:

```
'\ ': single quote
'\ ": double quote
'\n': newline
'\t': tabulator
'\\': backslash
```

Char ist ebenfalls Teil des Aufzählungstyps (Enum).

```
class Enum a where
  fromEnum :: a -> Int -- value to code
  toEnum    :: Int -> a -- code to value
  succ, pred :: a -> a -- successor and predecessor
```

Beispiel: `(toEnum 65 :: Char) = 'A'`

Nicht ASCII-Zeichen ('ß', 'ü') sollten nicht mit diesen Standard-Funktionen, welche Enum zur Verfügung stellt, aufgerufen werden.

Strings (Zeichenkette)

Strings sind Listen von Zeichen "Hallo Welt!". Es können auch leere Strings mit "" oder [] initialisiert werden. Einzelne Strings können mit (++) zusammengefügt werden. Ein Beispiel:

"Hallo" ++ " Welt!"

Standard-Funktionen für Strings:


```
head :: String -> Char
example: head "hello" = 'h'
tail :: String -> String
example: tail "hello" = "ello"
null :: String -> Bool -- test on empty string
```

String ist ein Synonym für [Char]. Listen können aber auch mit einem generischen Datentyp erstellt werden, z.B.: [a].

Schlüsselwort „type“

Mithilfe von dem Schlüsselwort „**type**“ können in Haskell Typen-Synonyme für Typen deklariert werden. Ein Beispiel:

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

„Show“

Die „Show“-Klasse konvertiert Elemente zu Strings, um sie ausgeben zu können. [Siehe oben](#).

Funktionen höherer Ordnung

Eine Funktion höherer Ordnung ist eine Funktion, welche entweder als Argument eine Funktion übergeben bekommt oder eine Funktion zurückgibt. So eine Funktion nennt man dann auch **Funktional**.

Partielle Anwendung:

using partial application we can easily define

```
dist_free_fall_earth = dist_free_fall 9.81
dist_free_fall_moon = dist_free_fall 1.62
```

Funktionen sind auch ohne Argumente ein Ausdruck. Funktionsfähige Anwendung ist nicht beschränkt durch den Funktionsnamen oder die Operatoren.

function application associates to the left: `add 1 2 = (add 1) 2`

- `add :: Int -> (Int -> Int)`
`add x y = x + y`
- the first argument is passed to `add`,
the result is a function: `add 1 :: Int -> Int`
- this function is applied on the next argument: `(add 1) 2`,
the result is an integer
- defining equations are only applied once sufficiently many arguments are available: `add 1` cannot be evaluated further,
`(add 1) 2 = add 1 2 = 1 + 2 = 3`

Der `(.)` Operator kettet Funktionen hintereinander.

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

Schönfinkeln/Currying

Currying ist die Umwandlung einer Funktion mit mehreren Argumenten in eine Sequenz von Funktionen mit jeweils einem Argument. (Zitat Wikipedia)

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

- `add` takes a single argument, which is a pair of numbers
- `add'` takes two arguments, one after the other
- functions can be specified in at least two alternatives
 - take single argument which might be a tuple (**tuple form**)
 - take arguments one by one (**curried form** – Haskell B. Curry, Moses Schönfinkel)

Anschauliches Beispiel: <https://wiki.haskell.org/Currying>

Sections/Abschnitte und der Anwendungsoperator

sections permits to write `(exp op)` or `(op exp)`

- `(5 >)` – testing for `x` whether `5 > x` holds, same as `(>) 5`
- `(> 5)` – testing for `x` whether `x > 5` holds, so here, `5` is second argument of `>`
- `(+ 5)` – function which adds 5
- one exception with subtraction: `(- 5)` is not a function which subtracts 5, but the number -5

Der Anwendungsoperator (`$`) hat die geringste Priorität, aber die Funktionsanwendung hat die höchste Priorität. Er wird verwendet, um Klammerung zu vermeiden.

```
-- as expression: order of eval. hard to parse
f x y z = 1000 - square ((x + y + z) * 2) `div` 3

-- with sections, . and $: evaluation from right to left
f x y z = (1000 -) . (`div` 3) . square . (* 2) $ x + y + z
```

Ein Beispiel für den Unterschied zwischen (`$`) und (`.`): <https://stackoverflow.com/questions/940382/what-is-the-difference-between-dot-and-dollar-sign>

λ -Abstraktionen (Lambda)

Eine λ -Abstraktion ist ein Ausdruck, welcher eine anonyme Funktion repräsentiert. Der Syntax:

- `\ vars -> exp`

Ein Beispiel: `\ x -> x + 5` ist das Gleiche wie `(+ 5)`.

Abstraktionen sind grundsätzlich dazu da, Muster im Code zu erkennen und diese zu simplifizieren, um duplizierten Code zu vermeiden. Ein Beispiel:

Aus:
$$\begin{aligned} &(9.81 \cdot 5 \cdot 5)/2 \\ &(9.81 \cdot 17.2 \cdot 17.2)/2 \\ &(1.62 \cdot 7 \cdot 7)/2 \end{aligned}$$
 Wird:
$$(x \cdot y \cdot y)/2$$

Schleifen

In imperativen Programmiersprachen sind Schleifen eingebunden/bereits vorhanden.

In der Funktionalen Programmierung, wo solche Funktionen nicht eingebunden sind, könnte eine „while“-Schleife mithilfe von Funktionen höherer Ordnung selbst definiert werden:

```
while :: (state -> Bool) -> (state -> state) ->
      (state -> state)
while cond upd s =
  if cond s then while cond upd (upd s) else s
```

Der Nachteil daran ist, dass der „state“ ein explizites Argument sein muss, welches bei imperativen Programmiersprachen selbstverständlich wegfällt/impliziert wird.

Der Vergleich von C und Haskell:

- C program for $n!$

```
int fact(int n) {
    int f = 1;
    while (n > 0)
        { f = f * n;  n = n - 1; }
    return f;
}
```
- using `while` from previous slide
with `state` instantiated as pairs of numbers `(f, n)`

```
fact :: (Num a, Ord a) => a -> a
fact n = \ (f, n) -> f $ while
  (\ (f, n) -> n > 0)
  (\ (f, n) -> (f * n, n - 1))
  (1, n)
```

Parser und Type-Inference

Parser und Type-Inference unterstützen Funktionen höherer Ordnung **nicht!**

Es müssten Anpassungen vorgenommen werden, damit Parser und Type-Inference auf Funktionen höherer Ordnung angewandt werden können, ihre normale Verwendung im vorher gegebenen Kontext (s.o.) bleibt allerdings gleich.

Listen

[a] stellt eine Liste vom Typ a dar. Ein paar Beispiele:

```
[1, 5, 7] :: Num a => [a]
[True, False] :: [Bool]
[[1, 2], []] :: Num a => [[a]]
[(+), (-)] :: Num a => [a -> a -> a]
```

Der Typ „Liste“ hat bereits **integrierte/eingebaute Funktionen**:

- **head** :: [a] -> a (gibt das erste Element zurück)
- **tail** :: [a] -> [a] (gibt den Rest der Liste (ohne das erste Element) zurück)
- **(++)** :: [a] -> [a] -> [a] (fügt zwei Listen zusammen)
- **null** :: [a] -> Bool (Wahr/Falsch, je nachdem ob die Liste leer ist oder nicht)

```
head :: [a] -> a
head (x : _) = x
head [] = error "..."
```

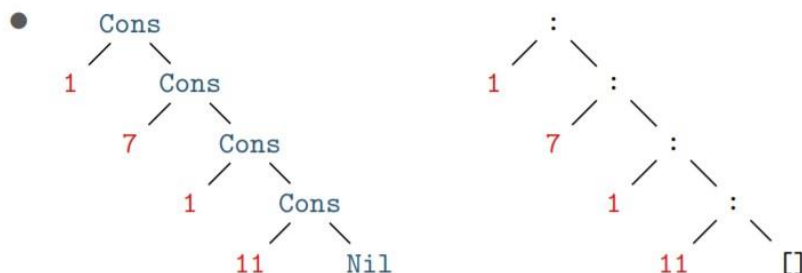
```
tail :: [a] -> [a]
tail (_ : xs) = xs
tail [] = error "..."
```

```
null :: [a] -> Bool
null [] = True
null (_ : _) = False
```

Ein darstellendes Beispiel für „**Listen als Datentyp**“ in Haskell:

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

- representations of [1,7,1,11]
- Cons 1 (Cons 7 (Cons 1 (Cons 11 Nil)))



[1, 7, 1, 11] is valid Haskell expression
(syntactic sugar for 1 : 7 : 1 : 11 : [])
[5] is same as 5 : [], etc.

Syntaktischer Zucker sind Syntaxerweiterungen in Programmiersprachen, welche der Vereinfachung von Schreibweisen dienen. (Zitat Wikipedia)

Unveränderliche Daten

In reinen, funktionalen Programmiersprachen können Daten nicht verändert, sondern nur neu kreiert werden. Eine imperative Programmiersprache lässt User die Daten in den meisten Fällen ändern.

„Replicate“

„**replicate** n x“ erstellt eine Liste der Länge n mit dem Element x wiederholt. Das bedeutet, dass der Befehl „**replicate** 2 3“ eine Liste [3, 3] erstellt.

In Haskell sind Listenpositionen, Längen, etc. im Typ „Int“ gespeichert.

Rezept für rekursive Funktionen

1. think about specification via textual description or examples (e.g., `length [1,4,2] = 3`)
2. define type (e.g., `length :: [Int] -> Int`)
3. enumerate cases (e.g., [] and `x : xs`)
4. define simple cases (e.g., `length [] = 0`)
5. define other cases (e.g., `length (x : xs) = 1 + length xs`), might use recursive calls on **smaller arguments**
 - shorter list
 - shorter distance to bound (e.g., from `i` to `i+1` with cond. `i < n`)
6. generalize and simplify (e.g., `length :: [a] -> Int` and `length (_ : xs) = 1 + length xs`)

result:

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

Beispiel: Auf ein bestimmtes Element mithilfe der Position in einer Liste zugreifen.

1. Beispiel: `[4,7,2,1] !! 1 = 7`

2. Typ: `(!!) :: [Int] -> Int -> Int`

3. Die Liste ist `[]` oder `x : xs`, Position ist **0** oder **nicht 0**.

4. Einfache Fälle:

a. `[] !! 0 = error "index out of bounds"`

b. `(x : xs) !! 0 = x` (Aufruf bei Position „0“)

c. `[] !! n = error "index out of bounds"`

5. Andere Fälle: Aufruf bei Position „nicht 0“

a. `(x : xs) !! n = xs !! (n - 1)`

Ist meine Position nicht 0, so wird die Liste erneut übergeben, dieses Mal nur der „tail“ (das bedeutet die Liste aber ohne das erste Element). Die Position (n-1) ist auch um 1 weniger geworden. Argumente in einem rekursiven Aufruf sind kleiner, weil die Liste dann kürzer ist.

6. Laufzeit O(n):

```
(!!) :: [a] -> Int -> a
[] !! n = error "index out of bounds"
(x : xs) !! n
  | n == 0 = x
  | otherwise = xs !! (n - 1)
```

„last“-Funktion und @-Pattern

Die „last“-Funktion gibt das letzte Element einer Liste zurück.

```
last :: [a] -> a
last (_ : x : xs) = last (x : xs)      -- >= 2 elements
last [x] = x                          -- 1 element
last [] = error "last on empty list"  -- 0 elements
```

Mit dem as-Pattern wird dem Symbol vor dem „@“ das Muster auf der rechten Seite des „@“ zugeordnet. Das Symbol/Die Variable kann somit als Synonym für die rechte Seite verwendet werden. Allgemeine Schreibweise: **var @ pat**

Listen mithilfe des Aufzählungstypen (Enum) erstellen

„enumFromTo x y“ erstellt eine Liste mit den Elementen x bis y. In Haskell kann auch die Schreibweise `[x .. y]` verwendet werden. Ein paar Beispiele:


```
[3 .. 7] = [3, 4, 5, 6, 7]
[7 .. 3] = []
reverse [3 .. 7] = [7, 6, 5, 4, 3]
['a' .. 'z'] = "abcdefghijklmnopqrstuvwxyz"
```

Die „**reverse**“-Funktion dreht die Reihenfolge einer Liste um.

Summen und Produkte

- Die „**sum**“-Funktion berechnet die Summe (Addition) aller Elemente einer Liste vom Typ „**Num**“.
- Die „**product**“-Funktion berechnet das Produkt (Multiplikation) aller Elemente einer Liste vom Typ „**Num**“.

Funktionen höherer Ordnung und Listen

- Die „**all**“-Funktion gibt zurück, ob alle Elemente der Liste die Kondition erfüllen.
- Die „**any**“-Funktion gibt zurück, ob mindestens ein Element die Kondition erfüllt.
- Die „**elem**“-Funktion gibt zurück, ob das gegebene Element in der Liste vorhanden ist.

Die „**filter**“-Funktion wählt alle Elemente einer Liste aus, welche eine bestimmte Kondition erfüllen. Zwei Beispiele:

Nur Großbuchstaben (Kondition):

```
filter (\x -> elem x ['A' .. 'Z']) "Hello World" = "HW"
```

Nur Zahlen größer als 5 (Kondition):

```
filter (> 5) [1, 7, 3, 8, 8] = [7, 8, 8]
```

Die „**map**“-Funktion wendet eine Funktion auf alle Elemente der Liste an. Zwei Beispiele:

- `map to_upper "Hello" = "HELLO" -- to_upper from sl. 3:58`
- `map (>5) [1, 7, 3, 8] = [False, True, False, True]`

- „**take**“-Funktion: Das erste Argument bestimmt, wie viele Elemente aus der Liste (zweites Argument) entnommen werden sollen.
- „**drop**“-Funktion: Das erste Argument bestimmt, wie viele Elemente aus der Liste (zweites Argument) entfernt werden sollen.

- `take 3 "hello" = "hel"`
- `drop 2 "hello" = "llo"`

- Die „**takeWhile**“-Funktion entnimmt solange Elemente aus der Liste (zweites Argument), wie die Kondition erfüllt ist. Beispiel:

takeWhile (<4) [1,2,3,4,5]

Output: [1,2, 3]

- Die „**dropWhile**“-Funktion entfernt solange Elemente aus der Liste (zweites Argument), wie die Kondition erfüllt ist. Beispiel:

dropWhile even [2,4,6,7,9,11,12,13,14]

Output: [7,9,11,12,13,14]

Zwei Listen zu kombinieren, kann man mithilfe der „**zipWith**“-Funktion oder der „**zip**“-Funktion (Spezialfall, erstellt Paare). Die Gegenfunktion von „**zip**“ ist die „**unzip**“-Funktion.

Listen-Abstraktion (List Comprehensions)

Bei List Comprehensions oder Listen-Abstraktionen geht es darum, wie vorhandene Listen verarbeitet werden sollen, um aus ihnen neue Listen zu erstellen.

Listen-Abstraktionen in Haskell kommen mithilfe der eingebauten „**concatMap**“-Funktion durchgeführt werden. Diese Funktion erstellt eine Liste von einer Listen-generierenden Funktion indem sie diese Funktion auf alle Elemente der Liste (zweites Argument) anwendet.

Gute Beispiele: http://zvon.org/other/haskell/Outputprelude/concatMap_f.html

„Fold“

In der funktionalen Programmierung bezieht sich „Fold“ auf Funktionen höherer Ordnung, die eine rekursive Datenstruktur analysieren und durch Verwendung einer bestimmten Kombinationsoperation die Ergebnisse der rekursiven Verarbeitung ihrer Bestandteile rekombinieren und einen Rückgabewert aufbauen.

Foldr („Fold right“)

Die „**foldr**“-Funktion nimmt das zweite Argument und das letzte Element der Liste (drittes Argument) und wendet die Funktion (erstes Argument) an. Dann nimmt es das vorletzte Element und so weiter. Ein Beispiel:

foldr (+) 5 [1,2,3,4]

- Zwischenergebnisse:
5, 5+4, 9+3, 12+2, 14+1
[15,14,12,9,5]

Output: 15

Foldl („Fold left“)

Die „**foldl**“-Funktion nimmt das zweite Argument und das erste Element der Liste (drittes Argument) und wendet die Funktion (erstes Argument) an. Dann nimmt es das zweite Element und so weiter. Ein Beispiel:

foldl (/) 64 [4,2,4]

- Zwischenergebnisse:

$$64/4 = 16$$

$$16/2 = 8$$

$$8/4 = 2$$

$$[64.0, 16.0, 8.0, 2.0]$$

Output: 2.0

List-Laws

Praktische Beispiele:

- `map f . map g = map (f . g)`
rhs is better since only one list traversal is required
- `filter p . filter q = filter (\ x -> q x && p x)`
note that the order of arguments to && matters
- `concatMap (\ x -> [e]) = map (\ x -> e)`
post-processing of list-comprehensions
- `filter p . concat = concat . map (filter p)`
filter first to concatenate shorter lists
- `map f xs ++ map f ys = map f (xs ++ ys)`
- `tail . map f = map f . tail`
- `head . map f = f . head` if `f` is strict
- `concat . map (map f) = map f . concat`
- `foldr f e = foldl f e` if `f` is associative with neutral element `e`

Wichtig: Listen sind keine Arrays!

Module und abstrakte Datentypen

Sichtbarkeitsbereich/Geltungsbereich (Scope)

Der Sichtbarkeitsbereich einer Variablen ist der Programmabschnitt, in dem die Variable nutzbar und sichtbar ist. **Lokale Variablen** können nur in einem bestimmten Abschnitt genutzt werden, während **globale Variablen** überall im Programm sichtbar sind.

Vorsicht besonders bei Funktionen, welche bereits in der „Prelude“ vordefiniert sind, z.B.: „length“, da diese zur Mehrdeutigkeit führen können.

Module

```
-- first line of file Module_Name.hs
module Module_Name(export_list) where
-- standard Haskell type and function definitions
```

Jeder Modul-Name muss mit einem Großbuchstaben anfangen. Module werden in extra Dateien abgespeichert (Module_Name.hs). Wenn eine Haskell-Datei keine Modul-Deklaration beinhaltet, fügt ghci selbst das Modul „main“ ein.

Die „**export_list**“ ist eine mit Kommas getrennte Liste von Funktionsnamen und Typennamen, welche dann in dieser Datei aufgerufen werden können. Es gibt verschiedene Möglichkeiten:

- **module Name (Type)** exportiert den Typen, aber nicht die Konstruktoren des Typen.
- **module Name(Type(..))** exportiert den Typen und die Konstruktoren.

Ein Beispiel:

```
module Pi_Approx(pi_approx, Rat) where
-- Prelude is implicitly imported

-- import everything that is exported by module Rat
import Rat

-- or only import certain parts
import Rat(Rat, create_Rat)
```

Die „**Prelude**“ wird automatisch importiert. In dem ersten „**import**“-Statement wird alles aus dem Modul „**Rat**“ importiert und beim zweiten nur bestimmte Teile („**Rat**“ und „**create_Rat**“).

Es kann mehrere „**import**“-Statements in einer Datei geben, aber nicht alles, was importiert wird, wird auch automatisch wieder exportiert.

Qualifizierte Import-Deklaration (Qualified Imports)

Wenn Bestandteile von Modulen dieselben Namen aufweisen, ist es sinnvoll, eine qualifizierte Import-Deklaration zu verwenden. Ein Beispiel:

Die Variable **pi** ist in „**Prelude**“, „**Foo**“ und „**Problem**“ definiert. Beim Aufruf ist es also sinnvoll die Schreibweise „**Module_name.name**“ anstatt von nur „**name**“ zu werden.

```
module Foo where pi = 3.1415
module Some_Long_Module_Name where fun x = x + x

module Example_Qualified_Imports where

-- all imports of Foo have to use qualifier
import qualified Foo
-- result: no ambiguity on unqualified "pi"

import qualified Some_Long_Module_Name as S
-- "as"-syntax changes name of qualifier

area r = pi * r^2
myfun x = S.fun (x * x)
```

Bei „**myfun**“ ist also sofort klar, dass es sich um die Funktion „**fun**“ von Modul „**S**“ handelt.

Abstrakte Datentypen

Konkrete Datentypen:

- definiert mithilfe von „**data**“, welche die möglichen Werte des Typen definiert
- User-definierte eigenen Operationen auf dem Typen mithilfe von Pattern Matching
- Keine primitiven Operationen notwendig
- Beispiele: Listen, [Temperature](#), **Bool**, etc.

Abstrakte Datentypen:

- Definiert mit ihren primitiven Operationen

- Keinen Zugriff auf die interne Struktur der Darstellung von Werten
- Pattern Matching nur mit „[Eq](#)“ möglich
- Abstraktionsschicht („abstraction barrier“): interne Struktur kann einfach verändert werden
- Beispiele: Char, Integer, Double, etc.

„Newtype“

Eine „**newtype**“-Deklaration können neue Typen gleich wie „[data](#)“ deklariert werden. Das Schlüsselwort „**newtype**“ kann sogar problemlos gegen „**data**“ ausgetauscht werden und in den meisten Fällen werden die gleichen Ergebnisse dabei entstehen.

```
newtype TName tvars = CName typ
```

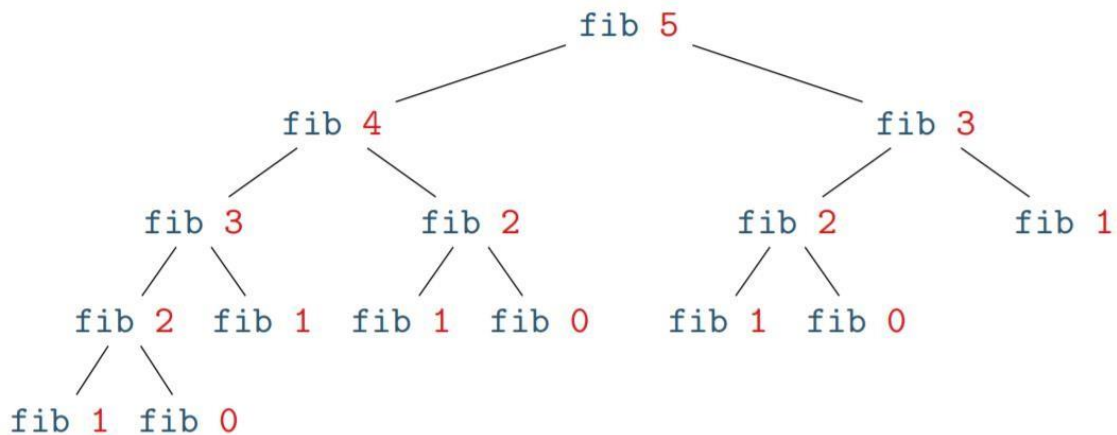
Im Gegensatz dazu, kann „**data**“ nicht einfach gegen „**newtype**“ ausgetauscht werden, sondern nur, wenn der Typ genau einen Konstruktor und ein Element hat. „**newtype**“ ist schneller, da „**CName**“ nichterst zur Laufzeit kreiert wird.

Effizienz/Lazy Evaluation

- direct translation into Haskell function

```
fib n | n <= 1      = n
      | otherwise   = fib (n - 1) + fib (n - 2)
```

- recursive calls overlap (recomputation),
leads to exponential number of recursive calls



Tipp: Wenn nur ein Wert zurückgegeben werden soll, man aber mehrere Werte zurückgeben möchte, kann man Daten in einem Tupel zurückgeben. Diesen Vorgang nennt man „**Tupling**“.

Rekursionsarten

Lineare Rekursion (linear recursion):

- Maximal ein rekursiver Aufruf.

Mehrfache Rekursion (multiple/nested recursion):

- Mehrere rekursive Aufrufe.

Endrekursion (tail recursion):

- Nichts folgt nach dem rekursiven Aufruf.

Wechselseitige Rekursion (mutual recursion):

- Mehrere Funktionen rufen sich derart gegenseitig auf, dass von mindestens einer davon mehrere Instanzen gleichzeitig auf dem Aufrufstapel aktiv sind.

Strikte Evaluation

Mithilfe von der Funktion ``seq``, strikten integrierten Library-Funktionen, „Bang Patterns“ oder strikten Datentypen möglich.

Guarded Recursion (tail recursion modulo cons):

- Wie der Name schon sagt, gilt eine Rekursion als „guarded“, wenn die einzige Operation, die nach einem rekursiven Aufruf noch ausgeführt werden muss, darin besteht, einen bekannten Wert vorne in eine ihm zurückgegebene Liste zu stellen.
- Oder im Allgemeinen eine konstante Anzahl einfacher Datenkonstruktionsoperationen auszuführen.

Unendliche Listen („Infinite Lists“)

Eine unendliche Liste, oder eine Sequenz von Elementen, kann programmiert werden, indem die Elemente nacheinander „produziert“ und „konsumiert“ werden, beispielsweise durch „Guarded Recursion“.

Weil Haskell [Lazy Evaluation](#) verwendet, führt eine unendliche Liste nicht immer zum Abbruch.

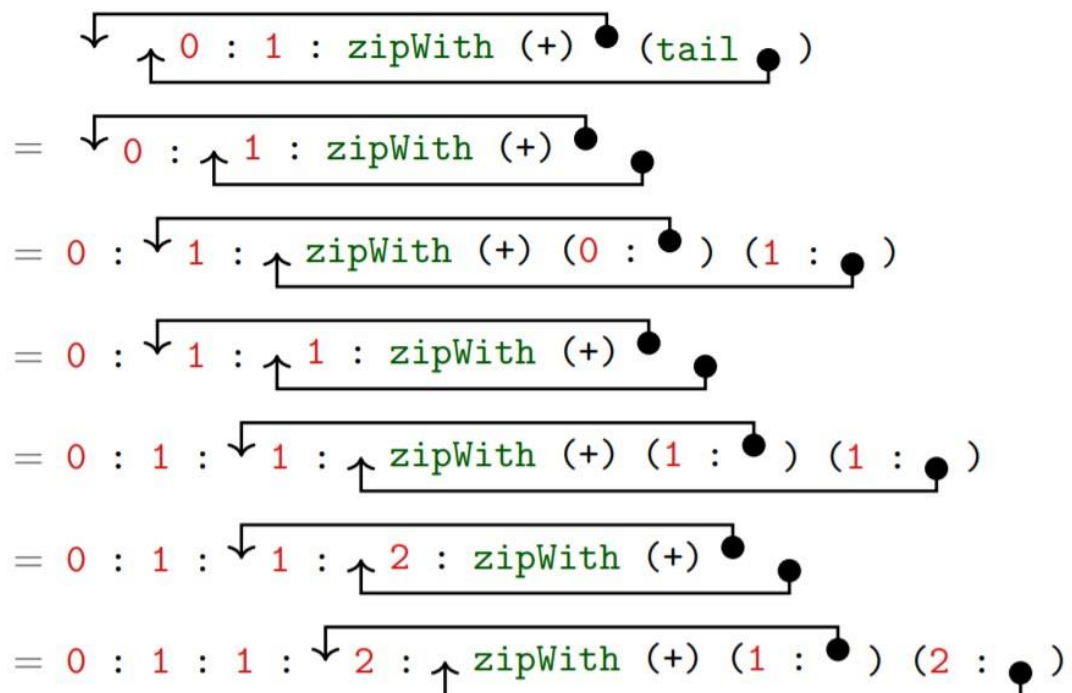
Fibonacci Numbers in Haskell

- implementation was given in first lecture (slide 8 of part 2)

```
fibs :: [Integer]
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- cyclic definition of list, evaluation:



I/O: Input and Output

Unter Input/Output versteht man...

- Lesen und Schreiben von Dateien
(z. B. der Compiler übersetzt .hs in .exe oder .tex in .pdf)
- Lesen und Schreiben im Speicher
(veränderlicher Zustand, Arrays)
- Lesen und Schreiben von Netzwerkkanälen
(z. B. Webserver und Internetbrowser)
- andere Programme starten und mit ihnen kommunizieren
Ton abspielen / aufnehmen, Mausbewegungen, ...

Ein Beispiel:

```
main = do
  putStrLn "Greetings! Please tell me your name."
  name <- getLine
  putStrLn $
    "Welcome to Haskell's IO, " ++ name ++ "!"

  putStrLn – prints string followed by newline
  getLine – reads line from standard input
  new syntax: do and <-
```

I/O und das Typensystem

`String -> IO ()` – after supplying a string, we obtain an I/O action
(in case of `putStrLn`, “printing”)
`IO ()` – just I/O (in case of `main`, run our program)
`IO String` – do some I/O and deliver a string
(in case of `getLine`, user-input)

I/O-Actions können kombiniert werden. Um diese Actions aneinander zu binden, verwendet man folgende Syntax: „>>=“ (= bind).

Ein Beispiel: `act_1 >>= \x -> act_2`

on evaluation, this expressions first performs action `act_1`
the result of action `act_1` is stored in `x`
afterwards action `act_2` is performed (which may depend on `x`)
in total, both actions are performed and the result is that of `act_2`

Ein ausführliches Beispiel:

```

putStrLn "hello" >>=
\ _ -> getLine >>=
\ name -> let answer = "hi " ++ name ++ "!" in
putStrLn answer

```

Das Ergebnis des ersten „putStrLn“ wird ignoriert. Die Antwort im „let“-Ausdruck wird komplett funktional berechnet. Der Typ ist „IO ()“ (der der letzten I/O-Aktion „putStrLn answer“). Die Ausführung dieser Actions erfolgt nach Reihenfolge, genauso wie bei einer imperativer Programmiersprache.

Do-Notation

Es gibt eine eigene Notation dafür, wenn „>>=“ (binds), „\ vars -> exp“ (Lambdas) und „let“ kombiniert werden.

do x <- act block	=	act >>= \ x -> do block
do act block	=	act >>= \ _ -> do block
do let x = e block	=	let x = e in do block

Das obere Beispiel, kann demnach auch folgendermaßen geschrieben werden:

```

do putStrLn "hello"
  name <- getLine
  let answer = "hi " ++ name ++ "."      -- no "in"!
  putStrLn answer

```

Weitere Anmerkungen

Das Ergebnis in einem do-Block ist das Ergebnis des letzten Ausdrucks.

„x <- a“ kann außerhalb von I/O-Ausdrücken nicht verwendet werden. Es gibt also eine strikte Trennung zwischen funktionalen Code und I/O.

„main :: IO ()“ ist die I/O-Aktion, welche ausgeführt wird, wenn wir ein kompiliertes Haskell-Programm ausführen.

„return“, „putChar“, „putStr“, „putStrLn“, „getChar“, „getLine“, „interact“ (String  Output), „readFile“, „writeFile“ und „appendFile“ sind integrierte/eingebaute Funktionen von Haskell.

Funktionaler Code kann in einer I/O-Aktion aufgerufen werden, umgekehrt ist das aber nicht möglich. Ein Beispiel dazu:

```

-- purely functional code
reply :: String -> String
reply name =
    "Pleased to meet you, " ++ name ++ ".\n" ++
    "Your name contains " ++ n ++ " characters."
  where
    n = show $ length name

-- invoked from I/O-part
main :: IO ()
main = do
    putStrLn "Greetings again. What's your name"
    name <- getLine
    let niceReply = reply name
    putStrLn niceReply

```

Anmerkung: Die Lazy Evaluation kann in manchen Fällen bei I/O zu Problemen führen.

Funktionen höherer Ordnung und I/O

```

better cat.hs
main = do
    files <- getArgs
    if null files then interact id else do
        foreach files readAndPrint
    where readAndPrint file = do
        s <- readFile file
        putStr s

```

I/O-Monade

In der funktionalen Programmierung sind Monaden ein abstrakter Datentyp. Wesentliche Eigenschaft von Monaden ist die Fähigkeit der Übertragung von Werten und Berechnungen eines „einfacheren“ Typs zu Berechnungen eines „höheren“ Typs, der mittels eines Typkonstruktors aus dem einfacheren Typ hervorgeht, sowie die Verknüpfung mehrerer solcher Übertragungen zu einer einzigen. (Zitat Wikipedia)