Kapitel 2: Grundlegende Elemente der Sprache Haskell

Lernziele dieses Kapitels

1. Haskell-Syntax
2. Verstehen der häufigsten Fehlermeldungen
3. Fallunterscheidungen
4. Lokale Definitionen
5. Funktionen als Argument und Ergebnis
6. Tupel, Listen und List-Comprehensions
7. Polymorphie, Overloading
8. Algebraische Datentypen
9. Typklassen und Instanzen

Variablen, Wertmengen, Typen

Paradigma	Variable steht für
imperativ	wiederverwendbaren Speicherplatz
funktional	eindeutigen Wert, vor der Definition: $\bot$

Vordefinierte Typen in Haskell:

Typ	Wertemenge
`()` "unit"	$\{\bot, ()\}$
`Bool`	$\{\bot, \texttt{False}, \texttt{True}\}$
`Int`	$\{\bot\} \cup$ `[(minBound::Int)..(maxBound::Int)]`
`Integer`	$\{\bot\} \cup \mathbb{Z}$

Außerdem: Zeichen(ketten): `Char`/`String` , Fließpunktzahlen: [`Float`|`Double`], rationale Zahlen: `Rational`, komplexe Zahlen: `Complex` [`Float`|`Double`]

Haskell-Ausdrücke und Typangaben

testen mit dem Interpreter `ghci`

- Arithmetische Ausdrücke wie üblich: `2+3`, `7*(5-2)`, ...
- Typangaben nachgestellt mit `::`*`Typname`*
  - `(2^12)^5` $\rightsquigarrow$ `1152921504606846976` (default: `Integer`)
  - `((2^12)^5)::Int` $\rightsquigarrow$ `0` (Überlauf nicht spezifiziert, hier mglw. modulo 2^{32})
- Arithmetische Operatoren erzwingen Typgleichheit,
  Konstanten und Operatoren sind überladen (z.B. `2` $\in$ { `Int`, `Float`, ... })
  - `(2::Int)+3`
    `3` und das Ergebnis sind vom Typ `Int`
  - `((2::Int)+3)::Integer`
    Typfehler, keine implizite Typwandlung (coercion) in Haskell!
  - `(fromIntegral ((2::Int)+3))::Integer`
    OK, explizite Typwandlung (conversion) durch `fromIntegral`

Der Interpreter ghci

- Auswertung (gib den Wert von 3+5 aus)

```
Prelude> 3+5
8
```

- Definition (sei x gleich 10001; Wert von $x * (3x - 5 + x^2)$?)

```
Prelude> let x=10001
Prelude> x*(3*x-5+x*x)
1000600039999
```

- Typabfrage (was ist der Typ des Ausdrucks `(+)` ?)

```
Prelude> :t (+)
forall a. (Num a) => a -> a -> a
```

Antwort: für alle Typen `a` gilt: `a` muss ein Zahlentyp (`Num`) sein und Argumente/Ergebnis von `(+)` haben denselben Typ `a`

Gültigkeitsbereiche von Variablen im ghci

```
Prelude> let x=5      der Wert von x sei 5
Prelude> let y=x+1    der Wert von y sei x+1, also 6
Prelude> y==x+1       ist y gleich x+1 ?
True                  Antwort: True, Typ: Bool
Prelude> let x=0      neue Variable, die auch x heisst
Prelude> y==x+1       ist y gleich x+1 ?
False                 Antwort: False, Typ: Bool
Prelude> y            was ist der Wert von y ?
6                     Antwort: 6
```

- Neudefinition (`let x =`) bewirkt Verschattung des alten Namens
- Referenzielle Transparenz gilt (Gültigkeitsbereich von `x` beachten!)
- Statische Variablenbindung (`y` an den Wert 6, nicht an `x+1`)

Syntax der Funktionsanwendung (Applikation)

```
Prelude> let f x = x^2-3*x+1  definiere Funktion f
Prelude> f 2   wende f auf den Zahlwert 2 an durch
-1                          Juxtaposition (=Hintereinanderschreibung)
Prelude> f2    Fehler: f2 wird als Name angesehen
Prelude> f(2)  möglich, aber Klammern zur Tokentrennung unerwünscht
-1
Prelude> f (f 2)      wende f auf das Ergebnis von f 2 an
5
Prelude> f f 2        Typfehler, die Juxtaposition ist linksassoziativ
                      und f soll als erstes Argument nicht f haben!
Prelude> (f . f) 2    Funktionskomposition: f . f für f∘f
Prelude> f $ f 2      Rechtsassoziativer Applikationsoperator $ hat
5                     geringere Priorität als die Juxtaposition
```

Achtung: `$`/`.` ohne Abstand haben andere Bedeutung (`$f` Splice, `A.` Namespace)!

Haskell Moduldatei

Datei Mean.hs

```
module Mean where   -- Kopfzeile mit Modulnamen  Mean
-- Typdefinitionen
sum3, mean3 :: Double -> Double -> Double -> Double
{-              Arg.1     Arg.2     Arg.3     Ergebnis
Funktionsdefinitionen (ohne let) -}
sum3   x  y  z  =  x+y+z
mean3  x  y  z  =  sum3 x y z  / 3
```

Verwendung mit ghci

```
Prelude> :l Mean              Laden der Definitionen
Compiling Mean                ( Mean.hs, interpreted )
Ok, modules loaded: Mean.
Mean> mean3 3 9 1             Anwendung von mean3
4.333333333333333
```

Fehlermeldungen (1)

```
Prelude> let x = 9
Prelude> x -7
2
Prelude> id -7   Fehler
<interactive>:1:
    No instance for (Num (a -> a))
      arising from use of ‘-’ at <interactive>:1
    In the definition of ‘it’: it = id - 7
```

Parser macht keine Typanalyse: `-` wird als Infix-Operator registriert, aber `id` ist eine Funktion, keine Zahl.

stattdessen:

```
Prelude> id (-7)
-7
```

Fehlermeldungen (2)

```
Prelude> let { f::Integer->Integer; f x = x+1 }
Prelude> f 5
6
Prelude> f 4.0   Fehler
<interactive>:1:
    No instance for (Fractional Integer)
      arising from the literal ‘4.0’ at <interactive>:1
    In the first argument of ‘f’, namely ‘4.0’
    In the definition of ‘it’: it = f 4.0
```

- `f` erwartet ein Argument vom Typ `Integer`
- `4.0` gehört zur Typklasse `Fractional` (Zahlentypen mit Division ohne Rest)
- `Integer` $\notin$ `Fractional` (`No instance for (Fractional Integer)`)

Fehlermeldungen (3)

```
Prelude> let { f :: Integer -> Integer; f x y = x+y } Fehler
<interactive>:1:
    Couldn't match 'Integer' against 't -> t1'
        Expected type: Integer
        Inferred type: t -> t1
    In the definition of 'f': f x y = x + y
```

- Nach Typdefinition ist `f x` vom Typ `Integer`
- Nach Funktionsgleichung ist `f x` vom Typ `t -> t1`, d.h. eine Funktion (erwartet `y` als weiteres Argument)

Fehlermeldungen (4)

```
Prelude> let { f :: a->a; f x = x^2 }
<interactive>:1:
    Could not deduce (Num a) from the context ()
      arising from use of `^' at <interactive>:1
    Probable fix:
        Add (Num a) to the type signature(s) for `f'
    In the definition of `f': f x = x ^ 2
```

Typangabe `f :: a->a` ist zu allgemein: `^2` verlangt Einschränkung auf Zahlentypen (`a` $\in$ `Num`).

Abhilfe: Einführung des Kontextes `Num a =>`

```
Prelude> let { f :: Num a => a->a; f x = x^2 }
```

Die Typklasse der Zahlen (Num a)

```
Prelude> let { square :: Num a => a->a; square x = x^2 }
Prelude> square (2::Int)
4
Prelude>  square 2.5
6.25
Prelude> square ((Ratio.%) 2 3)
4 % 9
Prelude>  square ((Complex.:+) 0 1)
(-1.0) :+ 0.0
Prelude> square '2'
<interactive>:1:
    No instance for (Num Char)
      arising from use of 'square' at <interactive>:1
    In the definition of 'it': it = square '2'
```

rationale Zahl $\frac{2}{3}$

$\frac{4}{9}$

komplexe Zahl i (0 + 1i)

-1

Fehler:'2' ist ein Zeichen, keine Zahl

Typdefinitionen statt Typfehlersuche (1)

Kontexte können automatisch hergeleitet werden:

```
Prelude> let square x = x^2
Prelude> :t square
square :: forall a. (Num a) => a -> a
```

Warum also den Typ angeben?

```
Prelude> let sumup n = n*(n+1)/2          ohne Typangabe
Prelude> sumup (10^20)
5.0e39                                    nicht exakt
Prelude> sumup ((10^20)::Integer)         Typfehler
<interactive>:1:
    No instance for (Fractional Integer)
      arising from use of ‘sumup’ at <interactive>:1
    In the definition of ‘it’: it = sumup ((10^20) :: Integer)
```

Wie kommt die Forderung nach `Fractional` zustande?

Typdefinitionen statt Typfehlersuche (2)

Wie kommt die Forderung nach `Fractional` zustande?

```
Prelude> let sumup n = n*(n+1)/2
Prelude> :t sumup
forall a. (Fractional a) => a -> a
```

Antwort: durch die Definition von `sumup` kann man früher erkennen!

```
Prelude> let { sumup :: Integer->Integer; sumup n = n*(n+1)/2 }
<interactive>:1:
    No instance for (Fractional Integer)
      arising from use of '/' at <interactive>:1
    In the definition of 'sumup': sumup n = (n * (n + 1)) / 2
```

es liegt an /

Lösung: Ersetze `/` durch ganzzahlige Division `div` (Typklasse `Integral`)

```
Prelude> let {sumInt :: Integral a => a->a; sumInt n = n*(n+1)`div`2}
Prelude> sumInt ((10^20)::Integer)
5000000000000000005000000000000000000
```

Typdefinitionen statt Typfehlersuche (3)

1. Man überlege sich den Typ jeder Funktion, bevor man sie implementiert.
2. Vergleich mit dem Typ, den der Haskell-Interpreter liefert.

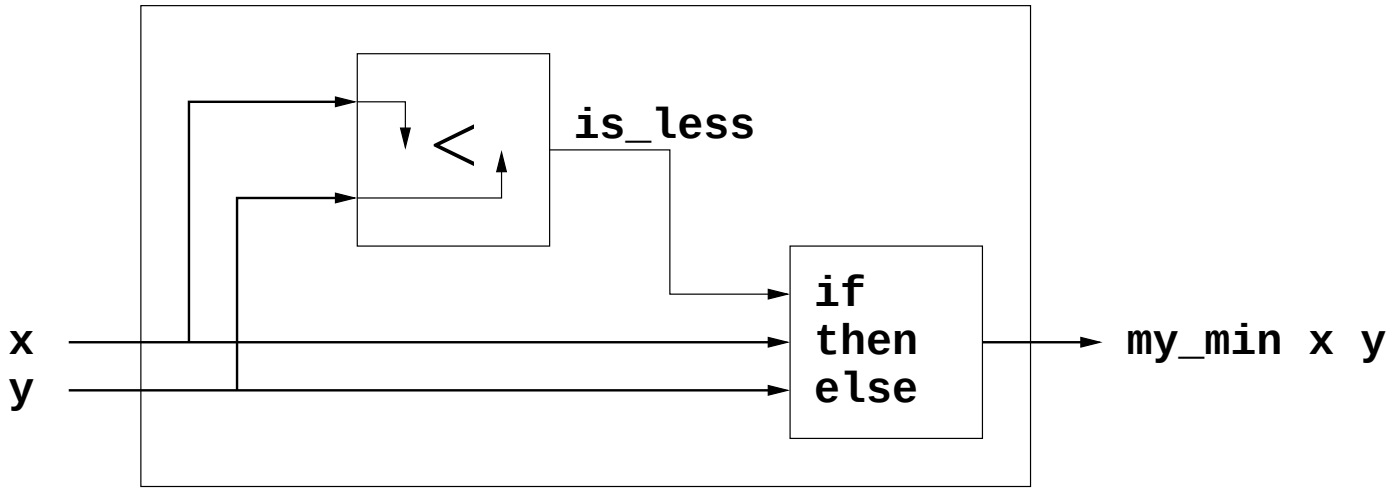
Generell: der Typ sollte möglichst allgemein sein (Wiederverwendung), aber es kann Einschränkungen durch den Algorithmus geben.

- Bereichsgrenzen: `Integer` statt Typklasse `Integral`
- Genauigkeit: `Double` statt Typklasse `Fractional`

Bei komplizierten Funktionen kann der Haskell-Interpreter u.U. nicht in der Lage sein, den allgemeinsten Typ selbst herauszufinden. In dem Fall kann man den speziellen Typ angeben, den man braucht. Typfehlermeldung z.B.: `...is not polymorphic enough`. Für dieses Kapitel nicht prüfungsrelevant und abhängig von Interpreterversion, deshalb hier kein Beispiel.

Schrittweiser Entwurf

1. Plan der Struktur (meist nur gedanklich)



2. Angabe des Typs (unmöglich? ⇒ Entwurfsfehler, zurück zu 1.)

```
my_min :: Double -> Double -> Double
```

3. Angabe der Funktionsgleichung

```
my_min x y = let is_less = x<y
             in  if is_less then x else y
```

Fallunterscheidungen

1. `if-then-else`

```
implies x y = if x then y else True
```

2. mehrere Gleichungen

```
implies True  y = y
implies False _ = True
```

3. Guards

```
implies x y | x     = y
            | not x = True
```

4. `case`

```
implies x y = case x of
                True -> y
                False -> True
```

`if` *cond* `then` x `else` y

- Syntax: Ausdruck, kann Teilausdruck sein,
  `else`-Zweig muss vorhanden sein
  Bsp.: `7 + (if doubleIt then (*2) else (*1)) x - y`
- Typen:
  Typ(*cond*)=`Bool`, Typ(x)=Typ(y)=Typ(Gesamtausdruck)
- Semantik:
  1. Falls *cond* $= \perp$, Ergebnis $\perp$ (Nichttermination)
  2. Falls *cond* = `True`, Ergebnis x
  3. Falls *cond* = `False`, Ergebnis y

  Es wird nur ein Zweig ausgewertet, der Wert des anderen darf undefiniert sein (wichtig für den Abbruch einer Rekursion!)

Lokale Definitionen

Es gibt `let` und `where`, vorzugsweise

- `let`, um die Schritte einer Berechnung zu verdeutlichen
- `where`, um einen Ausdruck näher zu spezifizieren

Manchmal gibt es nur eine Möglichkeit:

- in einem Ausdruck nur `let`

```
f x = x + 3 * (let z = x^3-5 in  z*x+z) - 2
```

- für mehrere Guards nur `where`

```
intpow :: Double -> Integer -> Double
intpow x n | n<0  = 1 / pow x (-n)
           | n>=0 = pow x n
  where pow x n = x^n
```

Bsp.: Nullstelle von aX^2+bX+c

```
root :: Double -> Double -> Double -> Bool -> Double
root a b c chooselower
 = let p = b/a
       q = c/a
       disc  = (p/2)^2-q
       rootd = sqrt disc
   in if disc < 0
         then error "root: Diskriminante kleiner Null"
         else -p/2+(if chooselower then -rootd else rootd)
```

```
root 1 1 (-2) False ⇝ 1.0
root 1 1 (-2) True ⇝ -2.0
root 1 0 5 False
 ⇝ *** Exception:  root:  Diskriminante kleiner Null
```

`rootd = sqrt disc` nur Definition, wird erst bei Verwendung berechnet.

Der Layout-Stil von Haskell

ohne Layout	mit Layout

`f x = let { y=x+1; z=y*y } in z`	`f x = let y=x+1`
	`z=y*y`
	`in z`

- Sinn: Programme leichter lesbar zu machen
- Zeilen gleicher Blöcke müssen in der gleichen Spalte beginnen
- Möglich bei `let`, `where`, `case`, etc.
- Für jeden Block separat anwendbar

Besonderheiten bei `let`-Ausdrücken

1. Die Reihenfolge der Definitionen ist unwichtig
2. Es gilt die Definition aus dem innersten umschliessenden Block
3. Es wird nur das berechnet, was benutzt wird

```
f x = let y   = 4                lokaler Zahlenwert
          g w = w+y              lokale Funktion
          z   = let a = x+y      (zu 1: y aus nächster Zeile)
                    y = g x      (g benutzt äusseres y)
                    z = y-x      (zu 2: y aus voriger Zeile)
                    b = ⊥        (zu 3: ignoriert)
                in z+x+(if g x > x then a else b)
          u = y+z                (zu 2: y=4 aus erster Zeile)
      in u*u
```

Funktionen als Argumente

Die Funktion `restricted_equal` soll prüfen, ob ihre Argumente, die Funktionen `f, g::Int->Int`, auf der Menge {0,1,2,3} identisch sind.

```
restricted_equal :: (Int->Int) -> (Int->Int) -> Bool
restricted_equal f g =     (f 0 == g 0) && (f 1 == g 1)
                        && (f 2 == g 2) && (f 3 == g 3)
```

Anmerkung: Zwei Polynome n-ten Grades sind gleich, wenn sie an $n+1$ Stellen übereinstimmen.

Wir prüfen die Gleichheit jeweils zweier Polynome dritten Grades:

```
Test> let p1 x = x^3-1
Test> let p2 x = (x-1)*(x^2+x+1)
Test> restricted_equal p1 p2
True
```

Eine Funktion als Ergebnis

Die Funktion `twice` soll eine Funktion `f::Int->Int` bekommen und sie zweimal hintereinander anwenden.

```
twice :: (Int->Int) -> (Int->Int)
twice f = f . f
```

Test:

```
Test> (+1) 6
7
Test> let f = twice (+1)     f entspricht (+2)
Test> f 6
8
Test> let g = twice f        g entspricht (+4)
Test> g 6
10
```

Bsp.: Numerischer Differenzenoperator

Typ: Funktion als Argument und als Ergebnis.

```
diff :: (Double->Double) -> ( Double->Double )
diff f = ...   ? hier müsste eine Funktion stehen
```

Lösung: Wir definieren, wie sich die Funktion auf ihrem Argument verhält (extensionale Definition, war bei `twice` nicht nötig)

```
diff f x = let h = 1.0e-10
           in (f (x+h) - f x) / h
```

Zusatzargument OK: zweites Klammernpaar im Typ unnötig.

```
Test> let f x = sin x
Test> let f' = diff f
Test> f' 0                    f' ist cos
1.0
```

Warnung: `(diff . diff)` funktional möglich, aber *numerisch* falsch!

Tupel und Listen

Struktur	#Komponenten	Komponententyp	Typbsp.
Tupel	fest	beliebig	`(Int,Bool)`
Liste	beliebig	fest	`[Double]`

Beispiele:

- `(2,True)::(Integer,Bool)`
- `[1.2, -3.456e23, 67, 7.31, 3.2E-5]::[Double]`
- `[("inc",(+1)), ("double",(*2))]::[(String,Int->Int)]`

Tupel

Sei der Typ des Ausdrucks x_i gleich α_i; dann gilt:

Ausdruck	Typ
(x_0, x_1)	(α_0, α_1)
(x_0, x_1, x_2)	$(\alpha_0, \alpha_1, \alpha_2)$
(x_0, x_1, x_2, x_3)	$(\alpha_0, \alpha_1, \alpha_2, \alpha_3)$
...	...

Verschieden geklammerte Tupel haben unterschiedliche Typen, obwohl die Mengen isomorph sind:

$$((\alpha, \beta), \gamma) \overset{Typ}{\neq} (\alpha, \beta, \gamma)$$

Pattern Matching zur Komponentenselektion

Prinzip: Struktur (hier Tupel) auf der linken Seite einer Gleichung:

```
fst (x,y) = x
snd (x,y) = y
```

Anwendung: (1) passe aktuellen und formalen Parameter
(2) weise jeder Variablen den entsprechenden Teil zu

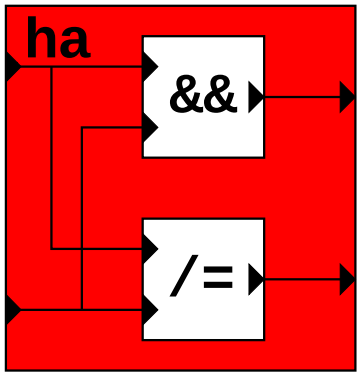
Beispiele:

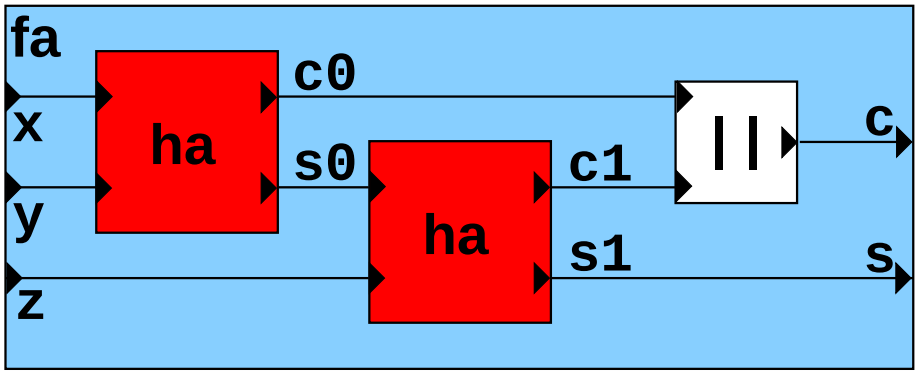
```
proj_nachname_gehalt (_,(_,nachname),_,gehalt)
                     = (nachname,gehalt)


implies (True,False) = False      Matching mit Konstanten
implies _            = True       in gegebener Reihenfolge
```

Tupel als Ergebnis von Funktionen

Bsp.: Logikschaltung






```
ha :: (Bool,Bool) -> (Bool,Bool)
ha (x,y) = (x && y, x /= y)

fa :: (Bool,Bool,Bool) -> (Bool,Bool)
fa (x,y,z) = let (c0,s0) = ha (x,y)
                 (c1,s1) = ha (s0,z)
             in (c0 || c1, s1)
```

Listen

- Dynamische Datenstruktur mit Elementen gleichen Typs
- Erzeugt durch zwei *Datenkonstruktoren*

Name	Bedeutung	Symbol	Typ
nil	leere Liste	`[]`	$\forall\alpha.[\alpha]$
cons	am Anfang anfügen	`(:)`	$\forall\alpha.\alpha \rightarrow [\alpha] \rightarrow [\alpha]$

- **cons** meist als rechtsassoziierender Infixoperator benutzt
- Syntaktischer Zucker: `[1,2,3,4]` statt `1:2:3:4:[]`
- Liste als Argument mit Pattern-Matching

```
map :: (a->b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs    Rekursion im Detail später
```

Wichtige Funktionen auf Listen

Name	Typ	Beispiel
`null`	`[a]->Bool`	`null []` ⇝`True`
`head`	`[a]->a`	`head [1,2,3,4]` ⇝`1`
`tail`	`[a]->[a]`	`tail [1,2,3,4]` ⇝`[2,3,4]`
`map`	`(a->b)->[a]->[b]`	`map (*2) [1,2,3]` ⇝`[2,4,6]`
`length`	`[a]->Int`	`length [1,3,5,7,9]` ⇝`5`
`(++)`	`[a]->[a]->[a]`	`[1,2,3]++[4,5]` ⇝`[1,2,3,4,5]`
`(!!)`	`[a]->Int->a`	`[8,5,7,3] !! 1` ⇝`5`
`take`	`Int->[a]->[a]`	`take 3 [4,5,6,7,8]` ⇝`[4,5,6]`
`drop`	`Int->[a]->[a]`	`drop 3 [4,5,6,7,8]` ⇝`[7,8]`
`concat`	`[[a]]->[a]`	`concat [[1,5],[],[3,7,2]]` ⇝`[1,5,3,7,2]`

Arithmetische Sequenzen

Ausdruck	erzeugte Liste	Schrittweite
`[1..5]`	`[1,2,3,4,5]`	ohne Angabe: `1`
`[1,3..12]`	`[1,3,5,7,9,11]`	2 (Abstand `1,3`)
`[4..2]`	`[]`	1, deshalb leere Liste
`[9,6..0]`	`[9,6,3,0]`	`-3`

- Arithmetische Ausdrücke möglich: `let x=7 in [x..2*x]`
- Auch für andere Aufzählungstypen: `[False .. True]` Abstand vor `..`
- Sogar unendliche Listen lassen sich definieren und verwenden, aber es kann nur ein endlicher Präfix ausgegeben werden (`take`):

Ausdruck	erzeugte Liste
`take 9 [1..]`	`[1,2,3,4,5,6,7,8,9]`
`take 7 [4,2..]`	`[4,2,0,-2,-4,-6,-8]`

Zeichen und Strings

- Unicode-Zeichen, Typ `Char`
  - `import Char` für viele Funktionen nötig
  - Konstanten, Bsp.: `'a'`, `'\n'` , `'\112'`
  - Konversion mit `Int`: `digitToInt`, `intToDigit`, `ord`, `chr`
  - Zeichenart: `isAscii`, `isUpper`, `isLower`, `toUpper`, `toLower`
- Strings, Listen von Zeichen

  `type String = [Char]` Typsynonym
  - Syntaktischer Zucker: `"Hallo"` statt `['H','a','l','l','o']`
  - Sequenz: `['a','c'..'o']` $\rightsquigarrow$ `"acegikmo"`
  - `show`-Funktion: `(Show a) => a->String` z.B. `Float` $\in$ `Show`

    Bsp.: `let x=27.5 in ("Preis: " ++ show x ++ " Euro")`

List Comprehensions (1)

Generierung von Listen, orientiert an Mengenkomprehension

Bsp.: Pythagoräische_Tripel $= \{ (x, y, z) \in \mathbb{N}^3 \mid x^2 + y^2 = z^2 \}$

Naive Implementierung mit Obergrenze `n` für die Zahlen:

```
ptriple :: Integer -> [(Integer,Integer,Integer)]
ptriple n =
 [ (x,y,z)            Element der Ergebnisliste
 | x <- [1..n],       nimm nacheinander ein x
   y <- [1..n],       für jedes x eine Folge von y
   z <- [1..n],       für jedes x und y eine Folge von z
   x^2+y^2==z^2       akzeptiere Wahl von x,y,z, falls Bedingung gilt
 ]
```

```
ptriple 15 ⇝ [(3,4,5),(4,3,5),(5,12,13),(6,8,10),(8,6,10),(12,5,13)]
```

List Comprehensions (2)

Ähnlichkeit zu verschachtelten `for`-Schleifen

- C:

```
count=0;
for (i=a;i<=b;i++)
    if (p(i)) { limit = f(i);
                for (j=0;j<=limit;j++)
                    result[count++] = exp(i,j); }
```

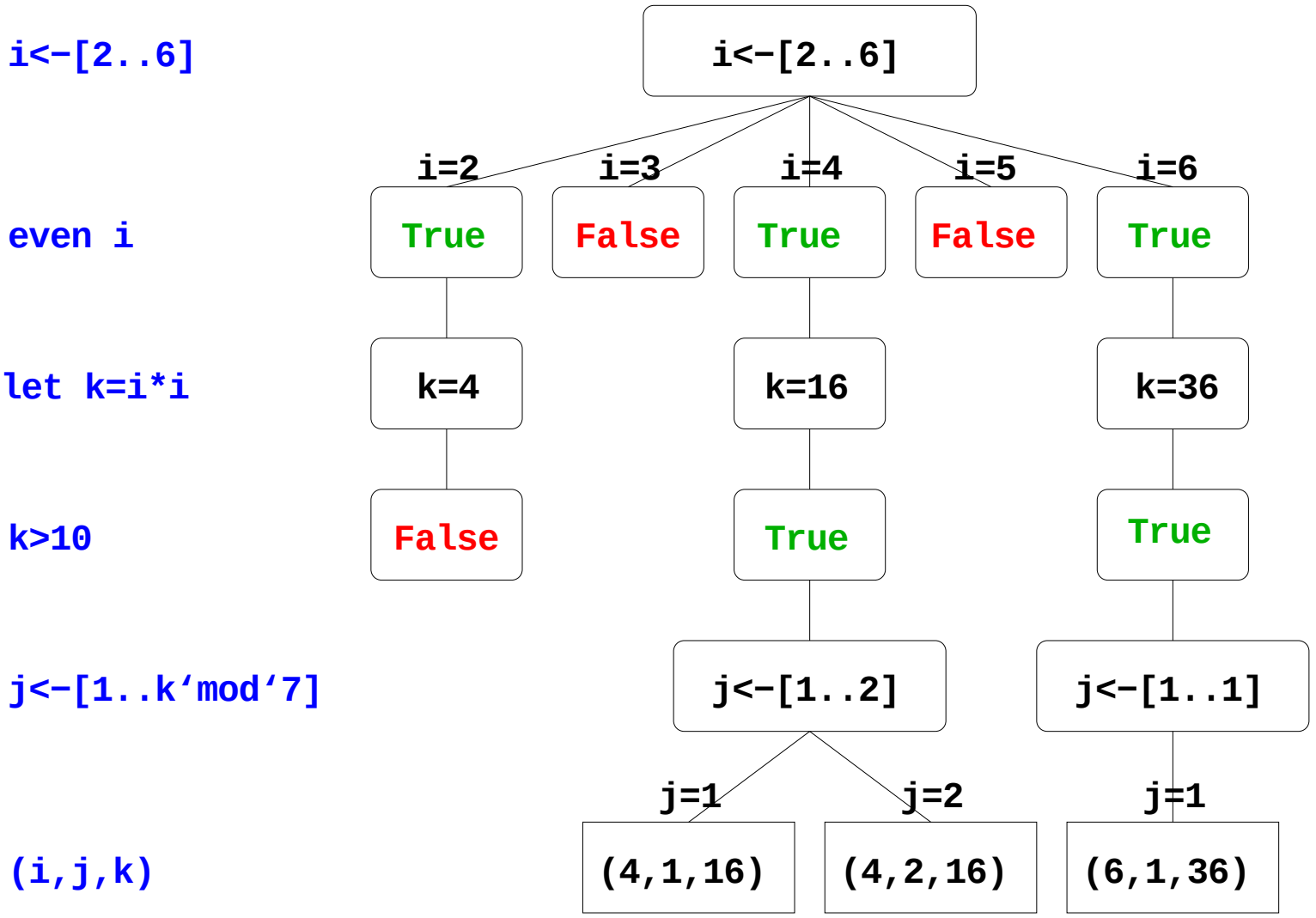
- Haskell:

```
result = [ exp (i,j) | i<-[a..b],
                       p i,
                       let limit = f i,
                       j <- [0..limit] ]
```

List Comprehensions (3)

- Syntax: `[` *exp* `|` q_0 `,` ... `,` q_n `]`
  wobei *exp* Element der Ergebnisliste, q_i Qualifier
- Semantik: Erzeugen eines Entscheidungsbaums von Variablenbindungen, die Ebenen sind q_0,q_1..q_n,*exp*;
  Ergebnis: Liste der Werte von *exp* an den Blättern
- Arten von Qualifiern
  - Generator: *Pattern* `<-` *Liste*
    für jedes Listenelement: falls Patternmatch erfolgreich: neuer Teilbaum mit Bindung der Variablen im Pattern. Bsp. `(True,x) <- [(True,4),(False,7),(True,2)]` erzeugt zwei Teilbäume mit den Bindungen `x=4` und `x=2`.
  - Guard: boolescher Ausdruck ($\rightsquigarrow$`False`: abschneiden des Teilbaums)
  - Lokale Definition: `let` *pattern* `=` *expression* (kein `in`)

Beispiel: [(i,j,k) | i<-[2..6], even i, let k=i*i, k>10, j<-[1..k`mod`7]]



Ergebnis: [(4,1,16),(4,2,16),(6,1,36)]

Semantik von List Comprehensions

Gegeben durch eine induktive Definition. `Q` steht für eine Folge von Qualifiern, `e` für einen Ausdruck, `b` für einen booleschen Ausdruck, `p` für ein Pattern, `l` für eine Liste und `decls` für eine Folge von Deklarationen.

1. Leere Liste von Qualifiern (nur intern)

   ```
   [ e | ] = [ e ]
   ```

2. Guard

   ```
   [ e | b, Q ] = if b then [ e | Q ] else []
   ```

3. Generator (`ok` sei eine frische Variable)

   ```
   [ e | p <- l, Q ] = let ok p = [ e | Q ]
                           ok _ = []
                       in concat (map ok l)
   ```

4. `let`-Zuweisung

   ```
   [ e | let decls, Q ] = let decls in [ e | Q ]
   ```

Polymorphie, Overloading und Typklassen

- Polymorphie: Typunabhängigkeit
  Bsp.: `length :: [a] -> Int` unabhängig vom Typ von `a`
- Overloading: ein Name für verschiedene Funktionen ($+^{\mathbb{Z}}$, $+^{\mathbb{Q}}$)
  Bsp: `(+) :: (Num a) => a -> a -> a`
  `(+)` ist nur auf Elementen der Typklasse `Num` definiert
- Typklasse: Zusammenfassung von Typen, die dieselben überladenen Funktionen verwenden
  Bsp.: `Num`, auf ihren Elementen ist `(+)`, `(-)` und `(*)` definiert
  Elemente von `Num`: `Int`, `Integer`, `Float`, `Double`, `Rational` ...

Polymorphe Funktion

- Mindestens eine und nur unbeschränkte Typvariablen.
- Keine Typ-spezifischen Operationen auf den Elementen mit polymorphen Typen.
- Funktion folgt manchmal bereits aus der Typdefinition:

Typ	`a->a`	`(a,b)->a`	`a->b->a`
einzige totale Funktion	`id`	`fst`	`const`

- Implementierung verwaltet polymorphe Daten nur, aber verknüpft sie nicht. Beispiel:

```
my_replicate :: Int -> a -> [a]
my_replicate n x = [ x | _ <- [1..n] ]
```

Jedes Elemente der Ergebnisliste enthält einen Verweis auf das Datenobjekt `x`.

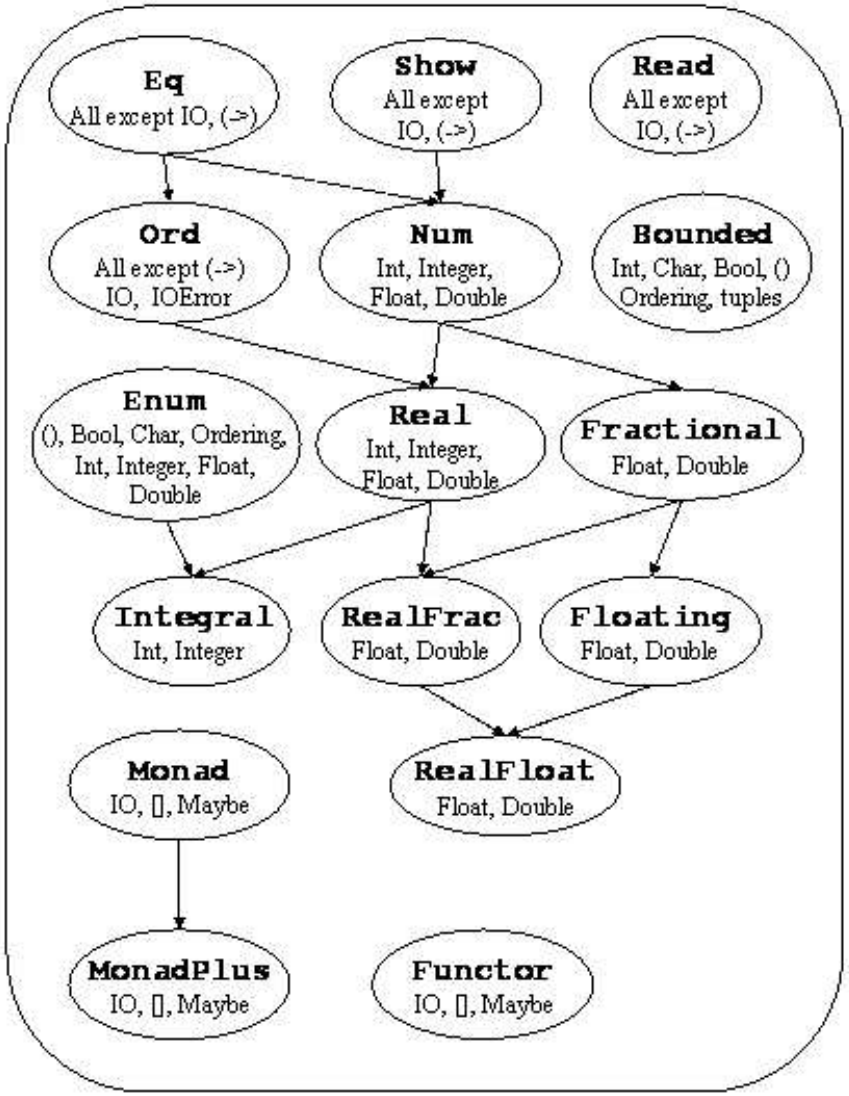
Überladene Funktion

- Mindestens eine beschränkte Typvariable vorhanden.
- Operationen auf den Elementen der beschränkten Typvariablen
  1. direkt durch Verwendung einer Operation einer niedrigeren Softwareschicht oder einer Hardware-Operation; Implementierung durch Spezialisierung
     - `(+)::Int`, z.B. Maschinenbefehl für arithmetisch-logische Einheit (ALU)
     - `(+)::Float`, z.B. Maschinenbefehl für Gleitpunkt-Prozessoreinheit
     - `(+)::Integer`, z.B. Verwendung einer Funktion der GNU-Multiple-Precision-Library
  2. indirekt durch Verwendung anderer überladener Funktionen;
     - Implementierung durch Typinformation in Laufzeitdaten und Fallunterscheidungen beim Übergang zu (1.)
     - zur Vermeidung explosiver Codeduplikation keine Spezialisierung

Einige vordefinierte Typklassen in Haskell

Name	El.-Eigenschaft	Bsp.-Fkt.	Elemente
`Show`	anzeigbar	`show`	alle ausser `IO`, `->`
`Read`	lesbar	`read`	alle ausser `IO`, `->`
`Eq`	vergleichbar	`(==)`	alle ausser `IO`, `->`
`Ord`	geordnet	`(<)`	alle ausser `IO`, `->`
`Num`	Zahl	`(+)`	`Int`, `Float`, ...
`Enum`	"aufzählbar"	`[..]`	`Bool`, `Int`, `Char`, ...
`Integral`	ganzzahlig	`mod`	`Int`, `Integer`
`Fractional`	invertierbar	`(/)`	`Float`, `Double`

Vordefiniertes Typklassensystem in Haskell



Typsynonyme (`type`)

- ein anderer Name für den gleichen Typ
- Bsp.: `type IndexValue = (Int,Int)` für Index/Wert-Paare
- Vorteil: Die Typinformation enthält impliziten Kommentar.
- Nachteil: logische Fehler werden so nicht automatisch erkannt.
  - `type Point = (Int,Int)`
    `Point` kann anstelle von `IndexValue` verwendet werden.
    Abhilfe: algebraische Datentypen (`data`)
  - Index und Wert können unerkannt vertauscht werden.
    Abhilfe: Namen für Komponenten (labelled fields)
- Sinnvolle Verwendung von `type` als Abkürzung ohne spezielle Bedeutung:
  `type DoubleV4 = (Double,Double,Double,Double)`

Algebraische Datentypen (data)

Konstruktion eines neuen Typs, ungleich aller bestehenden

Bsp.: vordefinierter Typ `Bool`

```
data Bool          Typkonstruktor
   = False         erster Datenkonstruktor
   | True          zweiter Datenkonstruktor
   deriving        automatische Herleitung von Typklasseninstanzen
   (Eq, Ord, Enum, Read, Show, Bounded)
```

- `Eq` `False==True` ⇝ `False`
- `Ord` `False<True` ⇝ `True`
- `Enum` `fromEnum True` ⇝ `1`
- `Read` `read "False" :: Bool` ⇝ `False`
- `Show` `show True` ⇝ `"True"`
- `Bounded` `maxBound::Bool` ⇝ `True`

data: Datenkonstruktoren mit Typen als Argumente

```
data Temperature  Name des neuen Typs
   = Celsius    Double    Datenkonstruktor Celsius, ein Argument
   | Kelvin     Double    Datenkonstruktor Kelvin,  ein Argument
   | Fahrenheit Double    Datenkonstruktor Fahrenheit, ein Argument
   deriving Show   Ableitung der show-Funktion

normalize :: Temperature -> Temperature
normalize (Celsius    cel) = Kelvin (cel+273.16)
normalize (Kelvin     kel) = Kelvin kel
normalize (Fahrenheit fah) = normalize (Celsius ((fah-32)*5/9))
```

Die Normalisierungsfunktion kann nur auf Elemente vom Typ `Temperature` angewendet werden und der Tag (`Celsius`/`Kelvin`/`Fahrenheit`) ist zwingend für die Implementierung ⇒ so ist immer sofort erkennbar, in welcher Einheit ein Wert angegeben ist.

`data`: Typkonstruktoren mit Typparametern

Bsp.: vordefinierter Typkonstruktor `Maybe`

```
data Maybe a = Nothing | Just a
               deriving (Eq,Ord,Read,Show)
```

```
[("Bill",Nothing),("Peter",Just 2.0),
 ("Tom",Just 1.2)]  :: [(String, Maybe Double)]
```

Datenkonstruktoren sind eine Art von Funktionen:

```
Nothing :: forall a. Maybe a
Just    :: forall a. a -> Maybe a
```

`data`: Verwendung eines Kontextes

Aufgabe: Verzeichnis als Liste von Schlüssel/Wert-Paaren.
Forderungen:

- Schlüssel müssen vergleichbar sein ⇒ Klasse `Eq`
- Schlüssel und Einträge sollen angezeigt werden können ⇒ Klasse `Show`

```
data (Eq a, Show a, Show b)                Kontext
     => Dictionary a b = Dict [(a,b)]

lookupDict :: (Eq a, Show a, Show b) => a -> Dictionary a b -> String
lookupDict key (Dict xs)
 = case lookup key xs of
    Nothing -> "key " ++ show key ++ " not found"
    Just x  -> show x
```

In Haskell vordefiniert: `lookup :: (Eq a) => a -> [(a, b)] -> Maybe b`

data: Labelled Fields

- Definition des Datentyps mit den Komponenten:

```
data Person = Customer { name::String, address::String }
            | Employee { name::String, salary::Float   }
```

- Erzeugung eines neuen Elements des Datentyps:

```
employee = Employee { name="Schmidt", salary=5000.00 }
```

- Abfragen eines Komponentenwerts: `salary employee` $\rightsquigarrow$`5000.00`
- Pattern-Matching mit Komponentenzuweisung an lokale Variablen `n` und `s`

```
printPerson (Employee {name=n, salary=s})
 = "Angestellter " ++ n ++ ", Gehalt: " ++ show s
```

- Kopie mit Änderung einer einzelnen Komponente (`@`: as-pattern)

```
changeAddress :: Person -> String -> Person
changeAddress person@(Customer {}) newaddress
 = person {address=newaddress}
```

Deklaration neuer Typklassen

```
class [ Kontext =>] Klassenname where
   Typdeklaration der Methoden
   Default-Definitionen
```

Bsp. (`Num` aus der Prelude):

```
class Eq a where                          ein Typ a gehört zur Klasse Eq, gdw.
     (==), (/=) :: a -> a -> Bool           Gleichheit ist auf a definiert

     x /= y  = not (x==y)                 Default-Definitionen
     x == y  = not (x/=y)
```

Deklaration neuer Instanzen

```
instance [ Kontext  => ] Klassenname Instanztyp where
   Implementierung der Methoden
```

Bsp.: es sei definiert: `data Color = Red | Yellow | Green`

```
instance Eq Color where          Color sei eine Instanz von Eq
 Red    == Red     = True
 Yellow == Yellow  = True
 Green  == Green   = True
 _      == _       = False
```

In einfachen Fällen automatisch mit deriving:

```
data Color = Red | Yellow | Green deriving Eq
```

Klasse Num aus der Prelude

```
class  (Eq a, Show a)                    Kontext (Superklassen)
    => Num a  where                      neue Klasse für Typ a
    (+), (-), (*)     :: a -> a -> a        Signatur
    negate            :: a -> a
    abs, signum       :: a -> a
    fromInteger       :: Integer -> a

        -- Minimal complete definition:
        --       All, except negate or (-)
    x - y             =  x + negate y      Default-Definitionen
    negate x          =  0 - x
```

Complex als Instanz von Num (Modul Complex)

Syntax einer komplexen Zahl: <Realteil>:+<Imaginärteil>

```
instance  (RealFloat a)                  a kann Float oder Double sein
 => Num (Complex a)  where               Klassenname Elementtyp
    (x:+y) + (x':+y') =  (x+x') :+ (y+y')
    (x:+y) - (x':+y') =  (x-x') :+ (y-y')
    (x:+y) * (x':+y') =  (x*x'-y*y') :+ (x*y'+y*x')
    negate (x:+y) =  negate x :+ negate y
    abs z =  magnitude z :+ 0
    signum 0 =  0
    signum z@(x:+y) =  x/r :+ y/r  where r = magnitude z
    fromInteger n =  fromInteger n :+ 0
```

Ratio als Instanz von Num (Modul Ratio)

Syntax einer rationalen Zahl: <Zähler>%<Nenner>

Infix-Konstruktor :% wird nicht automatisch exportiert

```
instance  (Integral a)                  Kontext, a muss eine
                                        ganze Zahl sein
 => Num (Ratio a)  where                Klassenname Elementtyp
    (x:%y) + (x':%y')    =  reduce (x*y' + x'*y) (y*y')
    (x:%y) * (x':%y')    =  reduce (x * x') (y * y')
    negate (x:%y)        =  (-x) :% y
    abs (x:%y)           =  abs x :% y
    signum (x:%y)        =  signum x :% 1
    fromInteger x        =  fromInteger x :% 1
```

Rekursive Instanzdeklarationen

Bsp.: Gleichheit für Listen

```
instance Eq a => Eq [a] where
   []     == []     = True
   (x:xs) == (y:ys) = x==y && xs==ys
   _      == _      = False
```

Bsp.: Gleichheit für Paare

```
instance (Eq a, Eq b) => Eq (a,b) where
   (x,y) == (x',y')  =  x==x' && y==y'
```

Arithmetische Sequenzen für boolesche Tupel

```
instance (Bounded a, Bounded b, Enum a, Enum b)
      => Enum ((,) a b) where
 fromEnum (x,y) = fromEnum x * (1+fromEnum (maxBound::b)
                                   -fromEnum (minBound::b))
                  + fromEnum y
 toEnum i = let size = 1 + fromEnum (maxBound::b)
                         - fromEnum (minBound::b)
            in (toEnum (i`div`size), toEnum (i`mod`size))
```

Test: `[(False,(False,False))..(True,(True,True))]` $\rightsquigarrow$

```
[(False,(False,False)),(False,(False,True)),
 (False,(True,False)),(False,(True,True)),
 (True,(False,False)),(True,(False,True)),
 (True,(True,False)),(True,(True,True))]
```