

# *HASKELL*

## **KAPITEL 5**

**Rekursion**

# Die Fakultätsfunktion

$$0! = 1$$

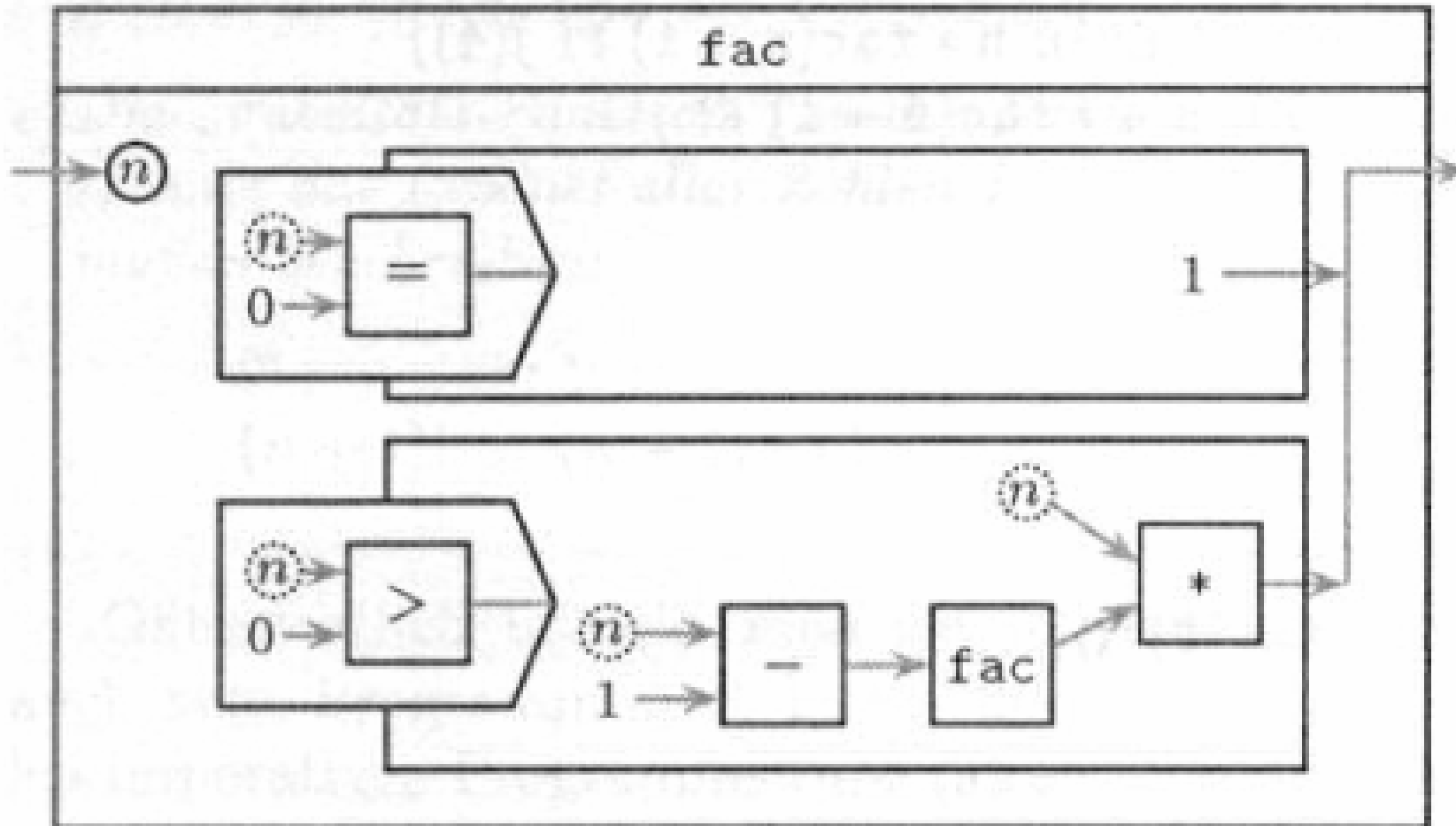
$fac :: Int \rightarrow Int$

$$n! = n * (n-1)!$$

$fac\ n = \mathbf{if}\ n = 0\ \mathbf{then}\ 1$

falls  $n > 0$

$\mathbf{else}\ n * fac(n - 1)$



# Der Binominalkoeffizient

$$\binom{n}{0} = \binom{n}{n} = 1 \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad n > 0, k > 0$$

*binom* :: Int → Int → Int

*binom* n k = **if** k = 0 ∨ k = n **then** 1

**else** *binom*(n-1, k-1) + *binom*(n-1, k)

# Wurzel ziehen durch Approximation

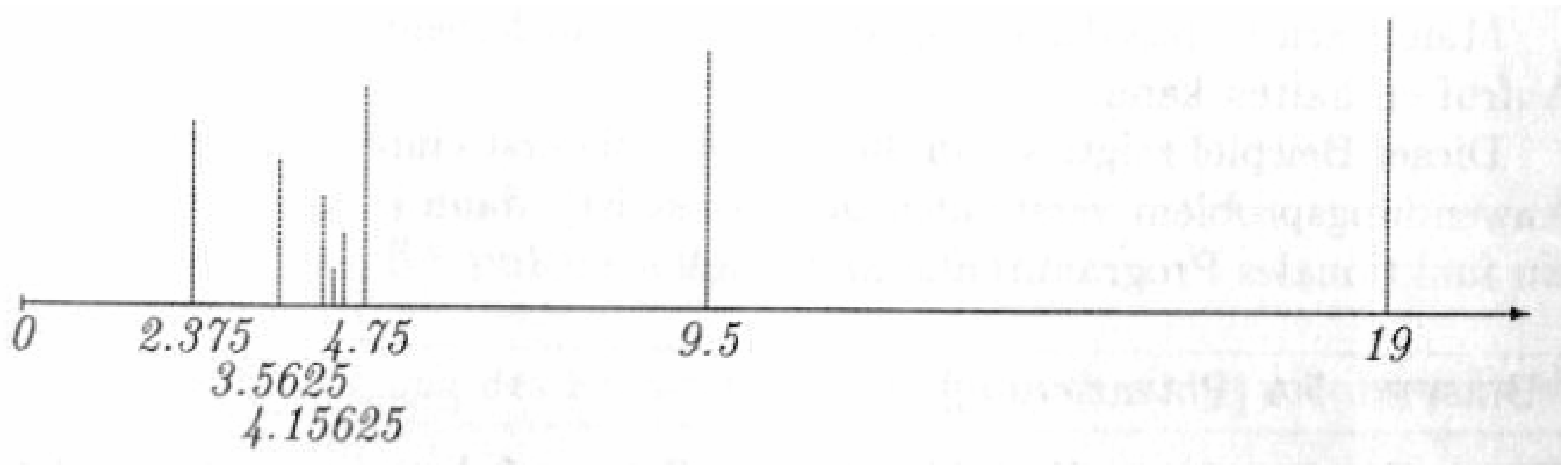
$\sqrt{19}$  liegt bestimmt im Intervall  $[0, \dots, 19]$ .

Algorithmus: halbieren (9.5), Hälfte quadrieren, zu gross.

weiter mit der linken Hälfte ...

Abbrückriterium: ( $\approx$ )  $:: float \rightarrow float \rightarrow bool$

$$x \approx y = abs(x - y) < 0.0000001$$



# Approximation

*approx* :: *float* → *float* → *float* → *float*

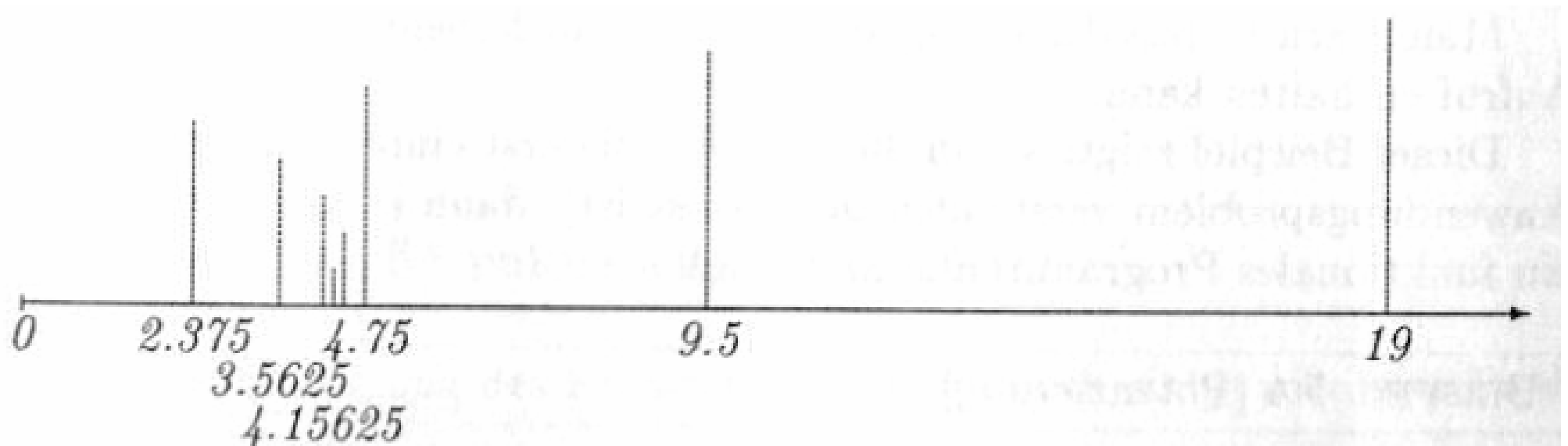
*approx* *x* *low* *high*

**let** *middle* == (*low* + *high*)/2 **in**

| *square* *middle* ≈ *x* = *middle*

| *square* *middle* > *x* = *approx* *x* *low* *middle*

| *square* *middle* < *x* = *approx* *x* *middle* *high*



# verschiedene Sorten Rekursion

## 1. Repetitive Rekursion

im Rumpf wird die Funktion in jedem Zweig höchstens ein mal, und ganz außen aufgerufen.

Beispiel bisher: *approx* und die

“*modulo a*“ Funktion:

*mod*  $:: Int \rightarrow Int \rightarrow Int$

*mod*  $a\ b =$

**if**  $b < a$  **then**  $b$  **else**  $mod\ a\ (b - a)$

## 2. Lineare Rekursion

im Rumpf wird die Funktion in jedem Zweig höchstens ein mal, aber nicht unbedingt ganz außen aufgerufen.

Beispiel: *fac* (oben:  $\dots n * fac(n - 1) \dots$ )

und die inverse Quadratsumme:

*invSqSum*  $:: Int \rightarrow Int$

*invSqSum* *n*

$$| n = 0 \quad = 0$$

$$| n > 0 \quad = 1/(n*n) + invSqSum(n - 1)$$

# 3. Baumartige Rekursion

im Rumpf wird die Funktion mehrfach nebeneinander aufgerufen.

Beispiel bisher. die *binom* – Funktion und die Fibonacci-Funktion:

*fib* :: *Int* → *Int*

*fib* *n*

| *n* = 0 = 0

| *n* = 1 = 1

| *n* ≥ 2 = *fib* (*n* – 1) + *fib* (*n* – 2)



# 4. Geschachtelte Rekursion

im Rumpf wird die Funktion mehrfach hintereinander aufgerufen.

Beispiel: die 91-Funktion:

$$f91 \quad :: \quad Int \rightarrow Int$$
$$f91 \ n$$
$$| \ n > 100 \ = \ n - 10$$
$$| \ n \leq 100 \ = \ f91 \ (f91(n + 11) )$$

... liefert 91 für jedes Argument unter 102

# 4. Geschachtelte Rekursion

noch ein Beispiel: schnelle ganzzahlige Division:

$div \quad \quad \quad :: Int \rightarrow Int \rightarrow Int$

$div \ a \ b$

$| \ a < \ b \quad \quad \quad = \ a$

$| \ b \leq a \wedge a < 2*b \quad = \ a - b$

$| \ a \geq 2*b \quad \quad \quad = \ div \ (div \ a \ 2*b) \ b$

# 4. Ausdruckskraft

geschachtelte Rekursion ist nicht reduzierbar auf ungeschachtelte.

Beispiel: Die Ackermann-Funktion

$ack \quad :: \quad Int \rightarrow Int \rightarrow Int$

$ack \quad m \quad n$

$/ m = 0 \quad = \quad n + 1$

$/ m > 0 \wedge n = 0 \quad = \quad ack \quad (m - 1) \quad 1$

$/ m > 0 \wedge n > 0 \quad = \quad ack \quad (m - 1) \quad (ack \quad (m - 1) \quad (n - 1))$

# 5. Verschränkte Rekursion

im Rumpf rufen sich mehrere Funktionen gegenseitig auf  
(kommt oft bei Interpretern vor)

Beispiel (etwas künstlich):

$even, odd \ :: \ nat \rightarrow bool$

$even \ n$

$/n = 0 \quad = \ true$

$/n > 0 \quad = \ odd \ (n - 1)$

$odd \ n$

$/n = 0 \quad = \ false$

$/n > 0 \quad = \ even \ (n - 1)$

# Interpreter einer Programmiersprache (angedeutet)

`executeStatement = ... evalExpression...`

`evalExpression = ... processMethodCall...`

`processMethodCall = ... executeStatement...`

Typ: *verschränkte Rekursion*: Mehrere Funktionen rufen sich gegenseitig

# ***HASKELL***

## **KAPITEL 6**

**Funktionen höherer Ordnung**

# 6.1

## Funktionen als Argument

# Beispiel: *min* und *max* verallg.

*extreme* :: (*Int* → *Int* → *Bool*) → *Int* → *Int* → *Int* → *Int*

*extreme before a b c*

| (*a before b*) ∧ (*a before c*) = *a*

| (*b before a*) ∧ (*b before c*) = *b*

| (*c before a*) ∧ (*c before b*) = *c*

Dann gilt:

*min* (*a*, *b*, *c*) = *extreme* ≤ *a b c*

*max* (*a*, *b*, *c*) = *extreme* ≥ *a b c*



# Beispiel: Differential

gegeben: stetig differenzierbare Funktion  $f: \mathbb{R} \rightarrow \mathbb{R}$

$x \in \mathbb{R}$

gesucht: Steigung von  $f$  an der Stelle  $x$

mathematische Notation:

$$\frac{df(x)}{dx}$$

$dif :: (Float \rightarrow Float) \rightarrow (Float \rightarrow Float)$

$dif f x = \dots$

übliche Schreibweise:  $f'$  statt  $dif f$

typische Anwendungen:

$dif \text{ square } 0.7$

*Resultat: 1.4*

# Differentialrechnung

*dif* :: (*Float* → *Float*) → (*Float* → *Float*)

*dif f x =*

**let** *h*<sub>0</sub> = 0.1

*d*<sub>0</sub> = *diffquotient f x h*<sub>0</sub>

**in**

*iterate x h*<sub>0</sub> *d*<sub>0</sub>

fehlt noch: Funktionen *diffquotient* und *iterate*

übliche Schreibweise: *f*′ statt *dif f*

# Die Funktion *iterate*

*iterate* :: *Float* → *Float* → *Float* → *Float*

*iterate* *x* *h*<sub>old</sub> *d*<sub>old</sub> =

**let** *h*<sub>new</sub> = *h*<sub>old</sub> / 2

*d*<sub>new</sub> = *diffquotient* *f* *x* *h*<sub>new</sub>

**in**

**if** *d*<sub>old</sub> ≈ *d*<sub>new</sub> **then** *d*<sub>new</sub>

**else** *iterate* *x* *h*<sub>new</sub> *d*<sub>new</sub>

fehlt noch: Funktionen *diffquotient* und ≈

# Die Funktionen *diffquotient* und $\approx$

*diffquotient* :: (Float → Float) → Float → Float → Float

*diffquotient* f x h = (f(x + h) - f(x - h)) / (2 \* h)

$\approx$  :: Float → Float → Float

$x \approx y$  = ( abs (x - y) < 0.000001 )

f :: Float → Float

f x = ... die gewünschte Funktion

# entsprechend: Integral

gegeben: stetige Funktion  $f: \mathbb{R} \rightarrow \mathbb{R}$

$a, b \in \mathbb{R}$

gesucht: Fläche zwischen der x-Achse und der Kurve  
von  $f$  zwischen  $a$  und  $b$ .

mathematische Notation:

$$\int_a^b f(x) dx$$

*integral* :: (*Float*  $\rightarrow$  *Float*)  $\rightarrow$  *Float*  $\rightarrow$  *Float*  $\rightarrow$  *Float*

*integral f a b*

typische Anwendungen:

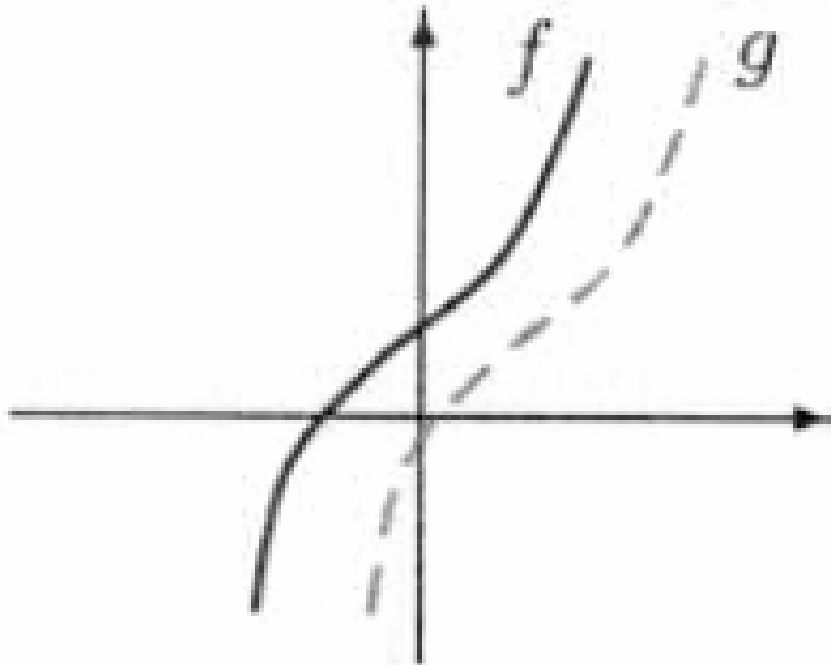
*integral sin 0  $\pi/2$*

*integral cos  $\pi/4$   $3\pi/2$*

## 6.2

# Funktionen als Argumente und Resultate

# Funktion $f$ um $dx$ verschieben



(a) *shift*

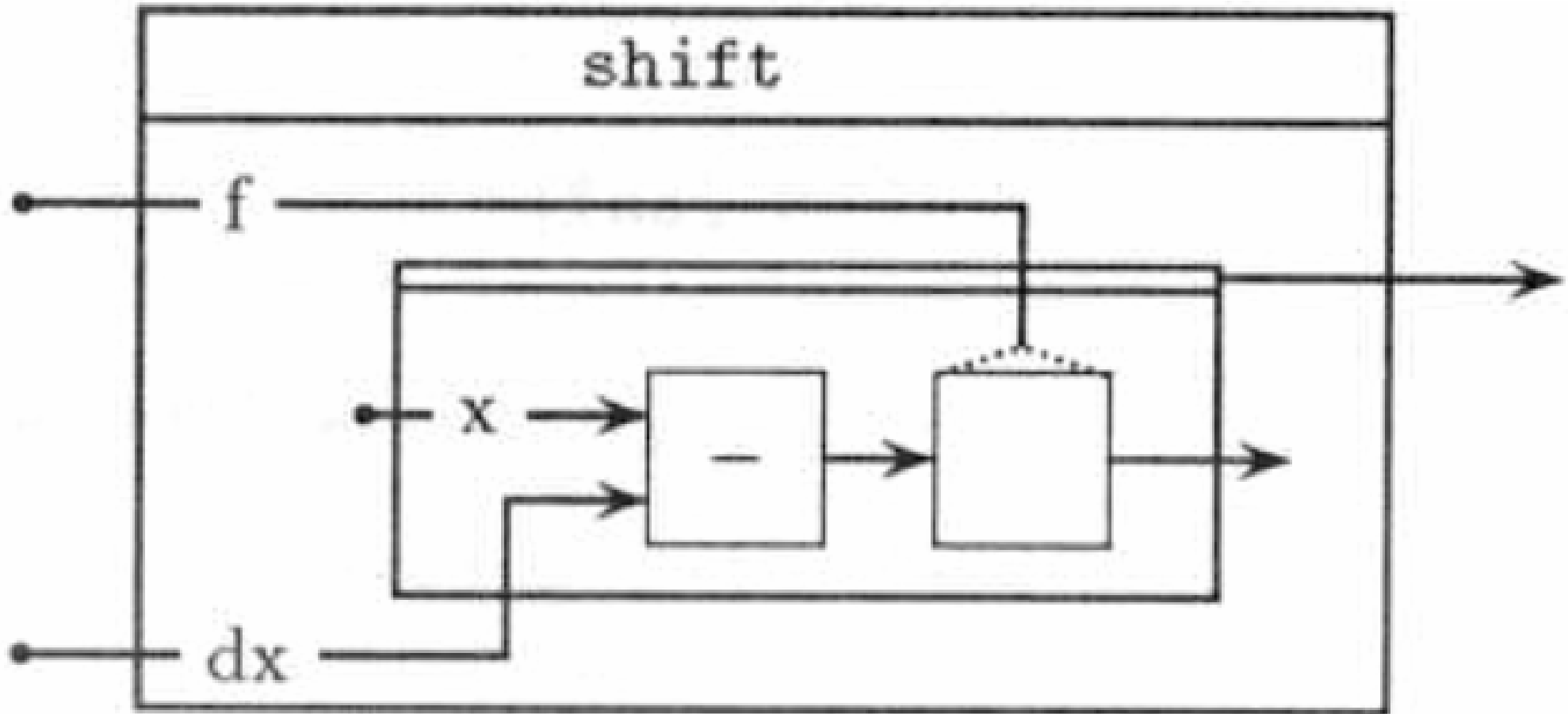
mit der bekannten Funktion  $\sin$ :  
 $\cos = \text{shift}(-\pi/2)(\sin)$

$\text{shift} \quad :: \quad \text{Float} \rightarrow (\text{Float} \rightarrow \text{Float}) \rightarrow (\text{Float} \rightarrow \text{Float})$

$\text{shift } dx \ f \ x = f \ (x - dx)$

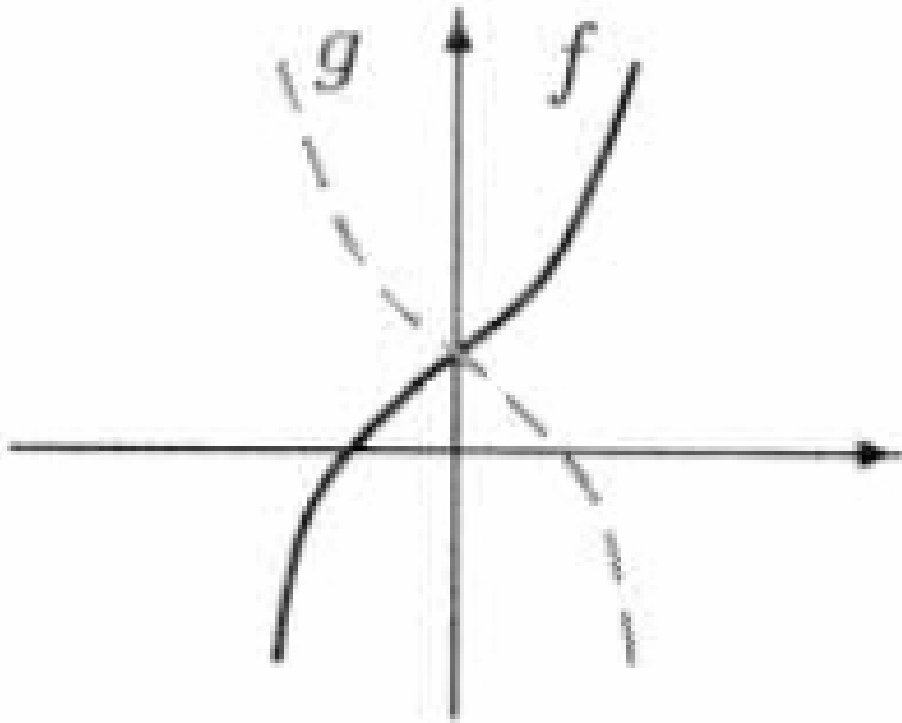
in der Graphik:  $g == \text{shift } dx \ f$

# graphisch





# Funktion $f$ an der $y$ -Achse spiegeln



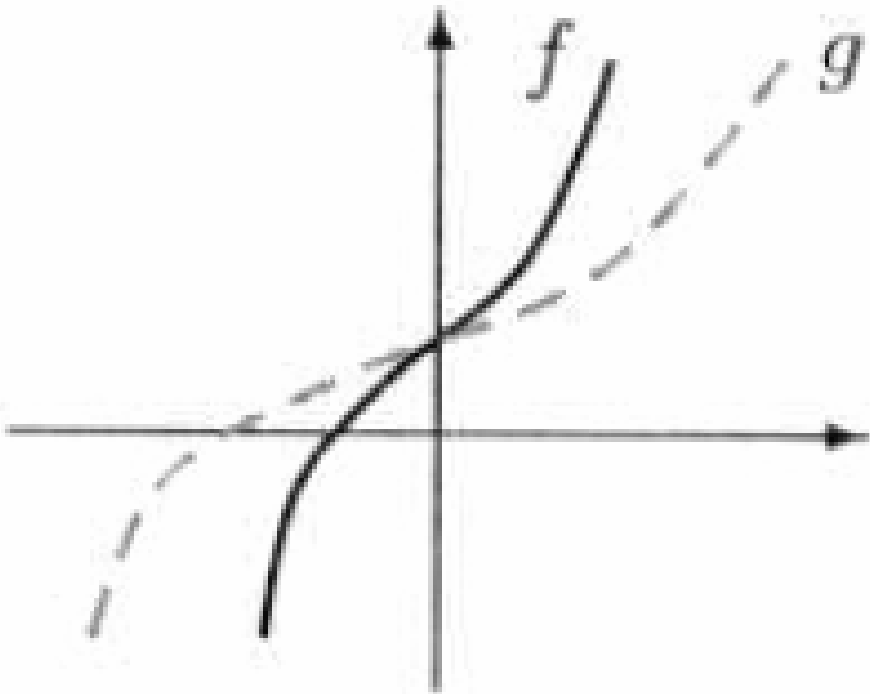
(b) *mirror*

*mirror*  $:: (Float \rightarrow Float) \rightarrow (Float \rightarrow Float)$

*mirror*  $f\ x = f\ (-x)$

in der Graphik:  $g == \textit{mirror}\ f$

# Funktion $f$ um den Faktor $r$ strecken



(c) *stretch*

*stretch* ::  $\text{Float} \rightarrow (\text{Float} \rightarrow \text{Float}) \rightarrow (\text{Float} \rightarrow \text{Float})$

$\text{stretch } r \ f \ x = f \ (x / r)$

in der Graphik:  $g == \text{stretch } r \ f$

# 6.3

## einige allgemeine Funktionale

# Verwendung symbolischer Typen

„Sei  $\alpha$  ein beliebiger Typ, z.B. *Int* oder *Float*“

typische Verwendung:

Identitätsfunktion:

$$id \quad :: \quad \alpha \rightarrow \alpha$$
$$id \ x \ = \ x$$

Konstante Funktion:

$$K \quad :: \quad (\alpha \rightarrow \beta \rightarrow \alpha)$$
$$K \ x \ y \ = \ x$$



# n – fache Iteration

$f^n$  schreiben wir als  $f^{\wedge} n$ :

$$(\wedge) \quad :: (\alpha \rightarrow \alpha) \rightarrow \text{Int} \rightarrow (\alpha \rightarrow \alpha)$$

$$f^{\wedge} n \ x =$$

$$| \ n = 0 \ = \ x$$

$$| \ n > 0 \ = \ f^{\wedge} (n - 1) \ f \ x$$

genauso gut geschrieben:

$$f^{\wedge} n \ x =$$

$$| \ n = 0 \ = \ x$$

$$| \ n > 0 \ = \ f^{\wedge} (n - 1) \ \bullet \ f \ x$$

# *while*

klassisches Programm: *while*  $p(x)$  *do*  $f(x)$

funktional geschrieben:

*while*  $\quad \quad \quad :: ((\alpha \rightarrow \text{Bool}), (\alpha \rightarrow \alpha)) \rightarrow (\alpha \rightarrow \alpha)$

*while*  $(p, f) x = \mathbf{if} \ p \ x \ \mathbf{then} \ \mathit{while} \ (p, f) \ f \ x$

**else**  $x$

# entsprechend: *until*

klassisches Programm: *do f(x) until p(x)*

funktional geschrieben:

*until* ::  $((\alpha \rightarrow \alpha), (\alpha \rightarrow \text{Bool})) \rightarrow (\alpha \rightarrow \alpha)$

*until* ( *f*, *p* ) *x* = **let** *y* = *f x* **in**  
**if** *p y* **then** *y*  
**else** *until* ( *f*, *p* ) *y*

es gilt:

$(f \text{ until } q) == \text{while } (\neg p, f) \bullet f$



# boole'sche Operationen als Funktionen

$$\neg \quad :: (\alpha \rightarrow Bool) \rightarrow (\alpha \rightarrow Bool)$$

$$(\wedge) \quad :: (\alpha \rightarrow Bool) \rightarrow (\alpha \rightarrow Bool) \rightarrow (\alpha \rightarrow Bool)$$

$$(\vee) \quad :: (\alpha \rightarrow Bool) \rightarrow (\alpha \rightarrow Bool) \rightarrow (\alpha \rightarrow Bool)$$

$$(\neg p) a = \neg (p a)$$

$$(p \wedge q) a = (p a) \wedge (q a)$$

$$(p \vee q) a = (p a) \vee (q a)$$

## 6.4

# Beispiele aus der Numerik

# Summe

häufig gegeben:

Funktionen  $f, h : \text{Float} \rightarrow \text{Float}$  und  
ein Prädikat  $p : \text{Float} \rightarrow \text{Bool}$ .

gesucht: Summe der Art

$$f(x) + f(h(x)) + f(h^2(x)) + f(h^3(x)) + \dots + f(h^n(x)).$$

wobei  $n$  die kleinste Zahl ist mit  $\neg p(f(h^{n+1}(x)))$ .

*sum* ::

$(\text{Float} \rightarrow \text{Float}) \rightarrow (\text{Float} \rightarrow \text{Float}) \rightarrow (\text{Float} \rightarrow \text{Bool}) \rightarrow$   
 $(\text{Float} \rightarrow \text{Float})$

*sum f h p x* = **if** *p x* **then** *f x* + *sum f h p h x*  
**else** 0

# Konvergenz

häufig gegeben:

Funktion  $h : \text{Float} \rightarrow \text{Float}$ , die für jedes  $x$  konvergiert  
d.h.  $\lim_{n \rightarrow \infty} h^n x$  existiert.

Aufgabe: Berechne  $\lim_{n \rightarrow \infty} h^n x$  näherungsweise  
d.h. bis  $h(x) \sim x$ .

*converge*  $:: (\text{Float} \rightarrow \text{Float}) \rightarrow \text{Float} \rightarrow \text{Float}$

*converge*  $h\ x =$  **if**  $h\ x \sim x$  **then**  $h\ x$   
**else** *converge*  $h\ h\ x$

# ***HASKELL***

## **KAPITEL 7**

**Datentypen**

# 7.1 Boole'sche Werte

Schlüsselwort:

„einen Datentyp bilden“

Alternativen einer

Aufzählung

**data** *Bool* = *False* | *True*.

*not* :: *Bool* → *Bool*

*not False* = *True*

*not True* = *False*

verwendbar als

Reduktionsregel für *not e* :

erst *e* reduzieren zu einem

Ausdruck *a* (wenn möglich).

Dann gilt: Wenn ...

-  $a == \text{False} : \text{not } a == \text{True}$

-  $a == \text{True} : \text{not } a == \text{False}$

-  $a ==$  ein Wert ungleich

*True* und *False*

(Typfehler):  $\text{not } a == \perp$

- die Berechnung von *a* nicht

terminiert:  $\text{not } a == \perp$

-  $a == \perp : \text{not } a == \perp$

Résumé: *drei* bool. Werte !!

# Boole'sche Operationen

$(\wedge), (\vee) \quad :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$

$\text{False} \wedge x = \text{False}$

$\text{True} \wedge x = x$

$\text{False} \vee x = x$

$\text{True} \vee x = \text{True}$

Motivation:

Pattern Matching arbeitet von links nach rechts.

Konsequenz:

$\text{False} \wedge \perp = \text{False}$

$\perp \wedge \text{False} = \perp$

$\text{True} \wedge \perp = \perp$

$\perp \wedge \text{True} = \perp$

$\text{False} \vee \perp = \perp$

$\perp \vee \text{False} = \perp$

$\text{True} \vee \perp = \text{True}$

$\perp \vee \text{True} = \perp$

# Bemerkung zu $\perp$

*Jeder* Datentyp hat ein implizites  $\perp$  „*undefined*“.

$\perp$  ist im Rechner nicht notwendig darstellbar.

Statt „ $\perp$ “ kann der Rechner auch

- „*Typfehler*“ sagen oder
- gar nichts sagen (weiterrechnen bis Systemabbruch)

Die Semantik von *Haskell* verlangt hier nichts spezielles.

Eine Funktion  $f$  ist *strikt*, wenn  $f(\perp) == \perp$ .

Beispiele:

*not* ist strikt,

$\wedge$  ist nicht strikt.  $False \wedge \perp = False$



# eine weitere Rolle von „ $\perp$ “

Bedeutung von „ $==$ “ : Rechner testet Gleichheit

Bedeutung von „ $=$ “ : definierend (und im üblichen mathematischen Sinn, gut geschrieben als  $=_{\text{def}}$ )

Konsequenz

mathematisch:  $\text{double} == \text{square}$  ist falsch

$\perp == \perp$  ist wahr

Haskell: beides kann das System nicht berechnen

beim Berechnen von  $f u$  kommt *immer* was raus!

... manchmal „ $\perp$ “

## 7.1.1 Gleichheit und Ungleichheit

$(==)$      $::$      $Bool \rightarrow Bool \rightarrow Bool$

$x == y$      $=$      $(x \wedge y) \vee (not\ x \wedge not\ y)$

$(\neq)$      $::$      $Bool \rightarrow Bool \rightarrow Bool$

$x \neq y$      $=$      $not\ (x == y)$

# Überladung

Manche Typen haben ein „ $==$ “ und ein „ $\neq$ “, manche haben es nicht.

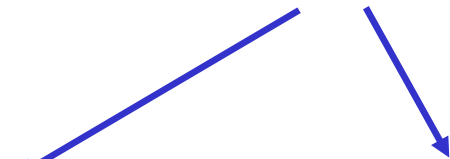
Streng genommen braucht jeder Typ sein eigenes „ $==$ “ und sein „ $\neq$ “ (falls der Typ sie überhaupt enthält).

Begründung: Sie werden jeweils anders realisiert.

Idee: Typenklasse *Eq* bilden: Sie enthalte alle Typen, die ein „ $==$ “ und ein „ $\neq$ “ haben.

# Definition einer Typ-Klasse

Schlüsselwörter



**class** *Eq*  $\alpha$  **where**

$(==), (\neq) :: \alpha \rightarrow \alpha \rightarrow Bool$

$\alpha$  ist eine Variable für Typen

Die Klasse *Eq* hat zwei Methoden,  $(==)$  und  $(\neq)$ .

Beide haben den Typ  $\alpha \rightarrow \alpha \rightarrow Bool$

auch geschrieben:

$(==), (\neq) :: Eq\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow Bool$

# Instanz einer Typ-Klasse

**class** *Eq*  $\alpha$  **where**

$(==), (\neq) \quad :: \quad \alpha \rightarrow \alpha \rightarrow \text{Bool}$

Ein Typ kann als Instanz der Klasse definiert werden.

Beispiel:

**instance** *Eq* *Bool* **where**

$x == y \quad = \quad (x \wedge y) \vee (\text{not } x \wedge \text{not } y)$

$x \neq y \quad = \quad \text{not } (x == y)$

# Die Typ-Klasse *ord*

Eine Menge kann man nur ordnen, wenn auf ihr Gleichheit definiert ist.

Wenn  $<$  gegeben ist, wird daraus  $\leq$ ,  $>$ ,  $\geq$  abgeleitet:

**class** (*Eq*  $\alpha$ )  $\Rightarrow$  *Ord*  $\alpha$  **where**

$(<), (\leq), (>), (\geq) :: \alpha \rightarrow \alpha \rightarrow \text{Bool}$

$(x \leq y) = (x < y) \vee (x == y)$

$(x > y) = \text{not } (x \leq y)$

$(x \geq y) = (x > y) \vee (x == y)$

# Eine Instanz von *ord*

zur Erinnerung:

**instance** *Eq Bool* **where**

$x == y = (x \wedge y) \vee (\text{not } x \wedge \text{not } y)$

$x \neq y = \text{not } (x == y)$

**class** (*Eq*  $\alpha$ )  $\Rightarrow$  *Ord*  $\alpha$  **where**

$(<), (\leq), (>), (\geq) :: \alpha \rightarrow \alpha \rightarrow \text{Bool}$

$(x \leq y) = (x < y) \vee (x == y)$

$(x > y) = \text{not } (x \leq y)$

$(x \geq y) = (x > y) \vee (x == y)$

**instance** *Ord Bool* **where**

$\text{False} < \text{False} = \text{False}$

$\text{False} < \text{True} = \text{True}$

$\text{True} < \text{False} = \text{False}$

$\text{True} < \text{True} = \text{False}$

damit gibt es auf *Bool*

automatisch

$\leq, >, \geq$

# Die Datentypen *Integer* und *Int*

## *Integer*

bezeichnet die ganzen Zahlen.

*Haskell* garantiert genaues Rechnen  
(... und riskiert dabei Speicherüberlauf).

## *Int*

bezeichnet eine Teilmenge von *Integer*,  
„die gewöhnlich reicht“.

*Haskell* garantiert genug Speicher  
(... und riskiert an den Grenzen ungenaues Rechnen).



## 7.1.2 Beispiel: Schaltjahr

*leapyear*  $:: Int \rightarrow Bool$

*leapyear*  $y = (y \mathbf{mod} 4 == 0) \wedge$   
 $(y \mathbf{mod} 100 \neq 0 \vee y \mathbf{mod} 400 == 0)$

alternativ:

*leapyear*  $y = \mathbf{if} (y \mathbf{mod} 100 == 0)$   
 $\quad \mathbf{then} (y \mathbf{mod} 400 == 0)$   
 $\quad \mathbf{else} (y \mathbf{mod} 4 == 0)$

## 7.1.3 Beispiel: Dreiecke

Gegeben: 3 ganze Zahlen,  $a \leq b \leq c$ .

Bekannt: Wenn  $a+b > c$ , gibt es ein Dreieck  $D$ , dessen Seiten die Länge  $a$ ,  $b$ , und  $c$  haben.

$D$  ist *gleichschenkelig*, wenn 2 Seiten gleich lang sind.

$D$  ist *gleichseitig*, wenn alle 3 Seiten gleich lang sind.

$D$  ist *unregelmäßig*, wenn  $D$  nicht gleichschenkelig ist.

Aufgabe:

ein Haskell-Programm mit einem Zahlentripel als Parameter, das auf diese Eigenschaften hin analysiert wird.

# Dreiecke in Haskell

```
data Triangle = Fehlerhaft | Gleichschenkelig |  
              Gleichseitig | Unregelmäßig
```

```
analyze      :: (Int, Int, Int) → Triangle
```

```
analyze (x, y, z)
```

```
  /x + y ≤ z           = Fehlerhaft
```

```
  /x == z             = Gleichseitig
```

```
  /x == y ∨ y == z   = Gleichschenkelig
```

```
  /otherwise          = Unregelmäßig
```

**Bem.:** klappt nur unter der Annahme  $x \leq y \leq z$ .

## 7.2 Der Datentyp *Char*

*Char* hat 256 Elemente:

**data** *Char* = *Char0* | *Char1* | ... | *Char255*

mit einer Konvention zur Bezeichnung der Elemente.

Beispiel: ‘*b*‘ für *Char98* .

Die meisten sind sichtbar, der Rest sind control-Zeichen.

Ein Element von *Char* wird in ‘...‘ notiert.

Beispiele:

‘a‘

‘7‘

newline: ‘↓‘

Leerzeichen ‘□‘

# Bezug zwischen *Char* und *Int*

*ord* :: *Char* → *Int*

*chr* :: *Int* → *Char*      ( *chr n* definiert für  $0 \leq n < 256$  )

für jedes *x* vom Typ *Char* gilt:

$$\mathit{chr} (\mathit{ord} \ x) = x$$

Beispiele:

*ord* 'b' ergibt 98

*chr* 98 ergibt 'b'

*chr* (*ord* 'b'+1) ergibt 'c'

*ord* '↓' ergibt 10

# *Char* als Aufzählungstyp

Gleichheit auf *Char*:

```
instance Eq Char where
```

```
(x == y) = (ord x == ord y)
```

Ordnung auf *Char*:

```
instance Ord Char where
```

```
(x < y) = (ord x < ord y)
```

*in Int*



Konsequenzen:

'0' < '9'

'a' < 'z'

'A' < 'Z'

'A' < 'a'

# Umgang mit *Char*

*isDigit, isLower, isUpper*  $:: Char \rightarrow Bool$

*isDigit c* = ( '0'  $\leq$  *c* )  $\wedge$  ( *c*  $\leq$  '9' )

*isLower c* = ( 'a'  $\leq$  *c* )  $\wedge$  ( *c*  $\leq$  'z' )

*isUpper c* = ( 'A'  $\leq$  *c* )  $\wedge$  ( *c*  $\leq$  'Z' )

*capitalize*  $:: Char \rightarrow Char$

*capitalize c* = **if** *isLower c*

**then** *chr (offset + ord c)*

**else** *c*

**where** *offset* = *ord 'A' - ord 'a'*

# wie man was beweisen kann

*capitalize 'a'*

= { def. „*capitalize*“ und *isLower 'a' = True* }

*chr (offset + ord 'a' )*

= { def. von *offset* }

*chr ((ord 'A' - ord 'a' ) + ord 'a' )*

= { Arithmetik }

*chr (ord 'A' )*

= { *chr (ord x) = x* }

'A'

*capitalize*  $:: Char \rightarrow Char$

*capitalize c* = **if** *isLower c*

**then** *chr (offset + ord c)*

**else** *c*

**where** *offset = ord 'A' - ord 'a'*



## 7.3 Aufzählungstypen

**Konstante:** Zeichenkette, beginnt mit Großbuchstabe

damit bisher:

**data** *Bool* = *False* | *True*.

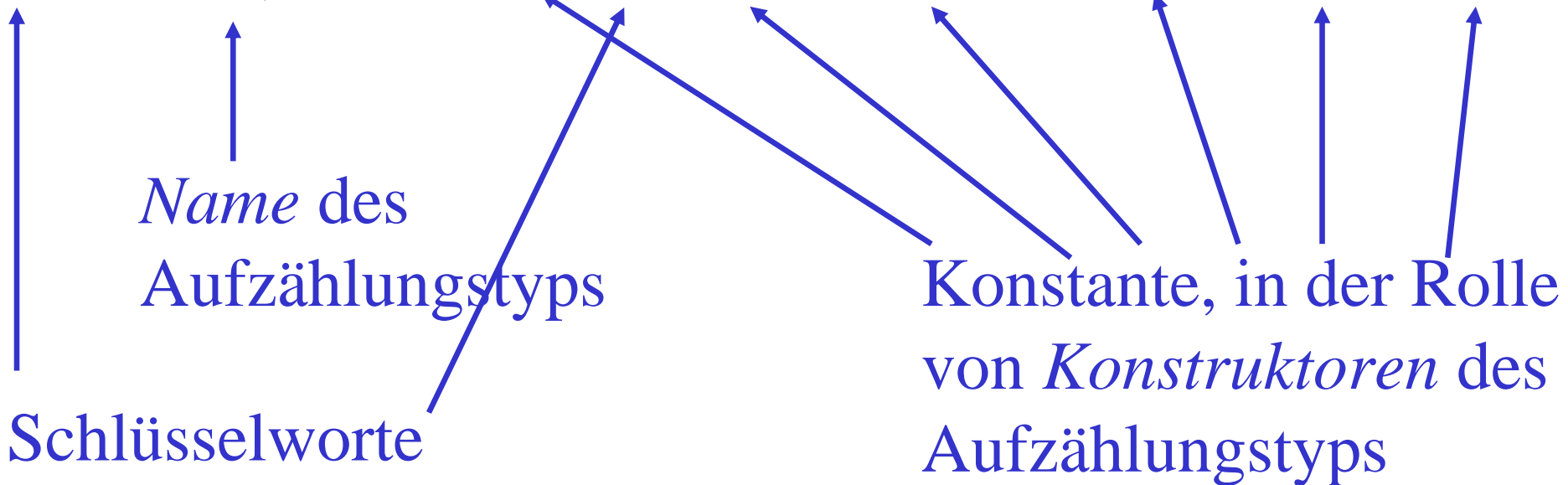
**data** *Triangle* = *Fehlerhaft* | *Gleichschenkelig* |  
*Gleichseitig* | *Unregelmäßig*

**data** *Char* = *Char0* | *Char1* | ... | *Char255*

# allg. Form für Aufzählungstypen

typisches Beispiel:

```
data Day = Sun | Mon | Thu | Wed Thu Fri
```



*implizit immer mit dabei: Konstruktor*  $\perp$

*Day* hat also 8 Elemente.

*Bool* hat 3 Elemente

# Enumerationstypen

**class** *Enum*  $\alpha$  **where**

*fromEnum*  $:: \alpha \rightarrow \text{Int}$

*toEnum*  $:: \text{Int} \rightarrow \alpha$

gewünscht, aber in *Haskell* nicht ausdrückbar:

*toEnum (fromEnum x) = x*

# *Day* als Enumerationstyp

**class** *Enum*  $\alpha$  **where**

*fromEnum*  $:: \alpha \rightarrow \text{Int}$

*toEnum*  $:: \text{Int} \rightarrow \alpha$

*fromEnum* für  $\alpha = \text{Day}$  :

**instance** *Enum* *Day* **where**

*fromEnum* *Sun* = 0

*fromEnum* *Mon* = 1

*fromEnum* *Tue* = 2

*fromEnum* *Wed* = 3

*fromEnum* *Thu* = 4

*fromEnum* *Fri* = 5

*fromEnum* *Sat* = 6

*toEnum*: später

# *Day* als geordneter Typ

**instance** *Eq Day* **where**

$$(x == y) = (fromEnum x == fromEnum y)$$

**instance** *Ord Day* **where**

$$(x < y) = (fromEnum x < fromEnum y)$$

Anwendungen:

*workday* :: *Day* → *Bool*

$$workday\ d = (Mon \leq d) \wedge (d \leq Fri)$$

*restday* :: *Day* → *Bool*

$$restday\ d = (d == Sat) \vee (d == Sun)$$

# „morgen“

*dayAfter* :: *Day* → *Day*

*dayAfter* *d* = *toEnum* ((*fromEnum* *d* + 1) **mod** 7)

insbesondere: *dayAfter* *Sun* = *Mon*

# *Char* als Enumerationstyp

**instance** *Enum Char* **where**

*fromEnum* = *ord*

*toEnum* = *chr*

# 7.3.1 Automatische Instanz-Daklaration

bisher: 3 Typklassen

*Eq, Ord, Enum*

und Instanzen davon für

*Bool, Char, Day*

Kurzschreibweise mit **deriving**:

```
data Day = Sun | Mon | Thu | Wed | Thu | Fri | Sat  
         deriving (Eq, Ord, Enum)
```



# 7.4 Tupel

Tupel: aus 2 Typen die Paare bilden

übliche Schreibweise der Mathematik:

$A \times B$  „kartesisches Produkt der Mengen A und B“

Haskell: (A, B)

Beispiel: (*Integer*, *Char*)

# Tupel als polymorpher Typ

Mit Variablen  $\alpha, \beta$  für Typen:

$(\alpha, \beta)$  ist Kurzform der Typdeklaration

**data** *Pair*  $\alpha, \beta = MkPair \ \alpha, \beta$

Der Konstruktor *MkPair* ist eine Funktion!

bisher: Konstruktoren waren Konstante:

**data** *Bool* = *False* | *True*.

**data** *Char* = *Char0* | *Char1* | ... | *Char255*

als kartesisches Produkt nicht sinnvoll darstellbar:

*MkPair*:  $\alpha \times \beta \rightarrow \alpha \times \beta$

*MkPair*( $x, y$ ) = ( $x, y$ ) ????

Typ von *MkPair* :

*MkPair*  $:: \alpha \rightarrow \beta \rightarrow Pair \ \alpha \ \beta$

# Funktionen für *Pair*

**data** *Pair*  $\alpha, \beta = \text{MkPair } \alpha, \beta$

*MkPair*  $x\ y$  wird kurz geschrieben als  $(x, y)$

zwei kanonische Funktionen:

*fst*  $:: (\alpha, \beta) \rightarrow \alpha$

*fst*  $(x, y) = x$

*snd*  $:: (\alpha, \beta) \rightarrow \beta$

*snd*  $(x, y) = y$

Berechnen: Pattern matching:

*fst*  $e$  formt  $e$  in die Form  $(x, y)$  um und gibt dann  $x$  aus.

# $\perp$ für $(\alpha, \beta)$

$undefined \ :: (\alpha, \beta)$  liefert  $\perp$ , weil nicht  
 $undefined = undefined$  reduzierbar

ist verschieden von  $(\perp, \perp)$ ,

wenn  $\perp$  das „undefined“ von  $\alpha$  oder von  $\beta$  ist.

Beweis:

$test \ :: (\alpha, \beta) \rightarrow bool$

$test(x, x) = True$

mit pattern matching folgt:

$test \perp = \perp$ , wg. Typfehler,  $test(\perp, \perp) = True$

# mehrere Funktionen als Argumente

*pair* ::  $(\alpha \rightarrow \beta, \alpha \rightarrow \gamma) \rightarrow \alpha \rightarrow (\beta, \gamma)$

*pair* (f, g) x = (f x, g x)

*cross* ::  $(\alpha \rightarrow \beta, \gamma \rightarrow \delta) \rightarrow (\alpha, \gamma) \rightarrow (\beta, \delta)$

*cross* (f, g) = *pair* (f • fst, g • snd)

# Eigenschaften von *pair* und *cross*

$$\text{fst} \bullet \text{pair} (f, g) = f$$

$$\text{snd} \bullet \text{pair} (f, g) = g$$

$$\text{pair} (f, g) \bullet h = \text{pair} (f \bullet h, g \bullet h)$$

$$\text{cross} (f, g) \bullet \text{pair} (h, k) = \text{pair} (f \bullet h, g \bullet k)$$

Beweise gehen „punktfrei“ (hier keine Einzelheiten)

... damit im Haskell-System

# Beispiel für Tupel

die beiden reellen Wurzeln einer quadratischen Gleichung  
 $ax^2 + bx + c = 0$

*roots* :: (*Float*, *Float*, *Float*) → (*Float*, *Float*)

*roots* (*a*, *b*, *c*)

*/a == 0* = *error* „not quadratic“

*/e < 0* = *error* „complex roots“

*/otherwise* = ( $(-b - r)/d$ ,  $(-b + r)/d$ )

**where** *r* = *sqrt e*

*d* =  $2*a$

*e* =  $b*b - 4*a*c$

# Operationen auf Tupel vererben

Gleichheit:

Wenn  $\alpha$  und  $\beta$  Gleichheit haben, dann hat  $(\alpha, \beta)$  eine kanonische Gleichheit:

**instance**  $(Eq\ \alpha, Eq\ \beta) \Rightarrow Eq\ (\alpha, \beta)$  **where**

$$(x, y) == (u, v) = (x == u) \wedge (y == v)$$

Ordnung:

Wenn  $\alpha$  und  $\beta$  total geordnet sind, dann hat  $(\alpha, \beta)$  eine kanonische „lexikographische“ totale Ordnung:

**instance**  $(Ord\ \alpha, Ord\ \beta) \Rightarrow Ord\ (\alpha, \beta)$  **where**

$$(x, y) < (u, v) = (x < u) \vee (x == u \wedge y < v)$$



# Konsequenz für $<$

auf Grund der Ordnung des pattern Matchig:

$(1, \perp) < (2, \perp)$  ergibt *True*

$(\perp, 1) < (\perp, 2)$  ergibt  $\perp$

Ordnung:

Wenn  $\alpha$  und  $\beta$  total geordnet sind, dann hat  $(\alpha, \beta)$  eine kanonische „lexikographische“ totale Ordnung:

**instance**  $(Ord\ \alpha, Ord\ \beta) \Rightarrow Ord\ (\alpha, \beta)$  **where**

$(x, y) < (u, v) = (x < u) \vee (x == u \wedge y < v)$

# Nullstellige Funktionen

Typ  $()$  „Nullstelliges“

hat 2 Elemente:

$\perp$  und

$()$  „leeres Argument“

damit kann man 0-stellige Funktionen  $f: \{()\} \rightarrow A$  bilden.

Beispiel:  $\text{pifun} :: () \rightarrow \text{Float}$

$\text{pifun} () = 3.14159$

„Jedes Element ist eine 0-stellige Funktion“

$A^0 \rightarrow A \cong A$

# wo sinnvoll verwendbar?

im „punktfreien Programmieren“

nur noch: Funktionen komponieren

nicht mehr: Funktion auf Argument anwenden

Beispiel:

*square • square • pifun* statt *(square • square) pi*

„Die Welt (der Funktionalen Programmierung)  
besteht aus Funktionen“

# 7.5 Typen vereinigen

„*Bool*  $\cup$  *Char*“

kann man in *Haskell* nicht schreiben.

Man braucht Konstruktoren, die aus dem zusammengesetzten Typ die Komponenten heraussortieren:

```
data Either = Left Bool | Right Char
```

Ausdruck *Left True* ist korrekt,  
nicht weiter reduzierbar

# allgemeiner

**data** *Either*  $\alpha$   $\beta$  = *Left*  $\alpha$  | *Right*  $\beta$

*Die Konstruktoren Left und Right ergeben sich kanonisch:*

*Left:*  $:: \alpha \rightarrow \text{Either } \alpha \beta$

*Right:*  $:: \beta \rightarrow \text{Either } \alpha \beta$

*Der obige Typ heißt dann*

**data** *Either* *Bool* *Char*

# Verwendung

**data** *Either*  $\alpha \beta = \text{Left } \alpha \mid \text{Right } \beta.$

*Left*  $:: \alpha \rightarrow \text{Either } a \beta$

*Right*  $:: \beta \rightarrow \text{Either } a \beta$

*case*  $:: (\alpha \rightarrow \gamma, \beta \rightarrow \gamma) \rightarrow \text{Either } \alpha \beta \rightarrow \gamma$

*case*  $(f, g) (\text{Left } x) = f x$

*case*  $(f, g) (\text{Right } y) = g y$

damit definierbar:

*plus*  $:: (\alpha \rightarrow \beta, \gamma \rightarrow \delta) \rightarrow \text{Either } \alpha \beta \rightarrow \text{Either } \gamma \delta$

*plus*  $(f, g) \text{ case}(\text{Left} \bullet f, \text{Right} \bullet g)$

# Eigenschaften von *Either* und *case*

**data** *Either*  $\alpha$   $\beta$  = *Left*  $\alpha$  | *Right*  $\beta$ .

*case*  $:: (\alpha \rightarrow \gamma, \beta \rightarrow \gamma) \rightarrow \text{Either } \alpha \beta \rightarrow \gamma$

*case* ( $f, g$ ) • *Left* =  $f$

*case* ( $f, g$ ) • *Right* =  $g$

$h \bullet \text{case } (f, g) = \text{Either } (f \bullet h, g \bullet k)$

*case* ( $f, g$ ) • *plus* ( $h, k$ ) = *Either* ( $f \bullet h, g \bullet k$ )

# kanonische Übertragung von $==$ und $<$

**instance**  $(Eq\ \alpha, Eq\ \beta) \Rightarrow Eq\ (\alpha, \beta)$  **where**

$Left\ x == Left\ y = (x == y)$

$Left\ x == Right\ y = False$

$Right\ x == Left\ y = False$

$Right\ x == Right\ y = (x == y)$

**instance**  $(Ord\ \alpha, Ord\ \beta) \Rightarrow Ord\ (\alpha, \beta)$  **where**

$Left\ x < Left\ y = (x < y)$

$Left\ x < Right\ y = True$

$Right\ x < Left\ y = False$

$Right\ x < Right\ y = (x < y)$



# kürzer

**data** *Either*  $\alpha$   $\beta$  = *Left*  $\alpha$  / *Right*  $\beta$   
**deriving** ( *Eq*, *Ord* )

*... damit:*

*die wichtigsten Konstruktionen für neue Datentypen.*

*es fehlt: Rekursion*

*... kommt später*

# 7.6 Typ-Synonomie

alternativer Name für bekannten Typ

früheres Beispiel:

die beiden Wurzeln einer quadratischen Gleichung.

Deklaration:

*roots* :: (*Float, Float, Float*) → (*Float, Float*)

*intuitiver:*

**type** *Coeffs* = (*Float, Float, Float*)

**type** *Roots* = (*Float, Float*)

*roots* :: *Coeffs* → *Roots*

# noch ein Beispiel

von einer Position  $(x, y)$  aus in einem Winkel  $a$  eine Strecke  $d$  zurücklegen; Endpunkt ausrechnen.

**type** *Position* = (*Float*, *Float*)

**type** *Angle* = *Float*

**type** *Distance* = *Float*

*move* :: *Distance*  $\rightarrow$  *Angle*  $\rightarrow$  *Position*  $\rightarrow$  *Position*

*move d a (x, y)* =  $(x + d * \cos a, y + d * \sin a)$

# Typ-Synonymien mit Typ-Variablen

sinnvolle Synonym-Deklarationen:

**type** *Pairs*  $\alpha = (\alpha, \alpha)$

**type** *Automorphism* =  $\alpha \rightarrow \alpha$

**type** *Flag*  $\alpha = (\alpha, \text{Bool})$

*rechte Seite muss bekannt sein. Ist der Fall bei*

**type** *Bools* = *Pairs Bool*

*synonym für (Bool, Bool)*

*Synonyme in Deklarationen verwenden:*

**data** *oneTwo*  $\alpha = \text{one } \alpha / \text{Two } (\text{Pairs } \alpha)$

## 7.6.1 neue Typen

Ein Typ-Synonym erbt alle Klasseninstanzen des definierenden Typs.

Das ist gelegentlich nicht erwünscht.

Beispiel:

```
type Angle      = Float  
      erbt == von Float.
```

Gewünscht: Gleichheit modulo  $2 * \pi$ .

Dafür nötig: neuen Datentyp

```
data Angle      = MkAngle Float
```

mit dem Konstruktor *MkAngle* .

# Beispiel: **data** *Angle*

**data** *Angle* = *MkAngle* *Float*

**instance** *Eq* *Angle* **where**

*MkAngle* *x* == *MkAngle* *y* = *normalize* *x* == *normalize* *y*

*normalize* :: *Float* → *Float*

*normalize* *x*

| *x* < 0 = *normalize* (*x* + *rot*)

| *x* ≥ *rot* = *normalize* (*x* − *rot*)

| otherwise = *x*

**where** *rot* = 2\* $\pi$ .

# Problem damit

**data** *Angle* ist „ganz neu“ im Vergleich mit `Float`.

**data** *Angle* hat eigenes „ $\perp$ “.

hin – und herrechnen *Angle* – *Float* braucht immer *MkAngle*

*Haskell*:

```
newtype Angle = MkAngle Float
```

steigert Effizienz.