

# Vorsemesterkurs Informatik

## Sommersemester 2011

### Grundlagen der Programmierung in Haskell

SoSe 2011

- 1 Ausdrücke und Typen
- 2 Funktionen
- 3 Rekursion

## Haskell-Programmieren:

- Im Wesentlichen formt man **Ausdrücke**
- z.B. arithmetische Ausdrücke  $17*2+5*3$
- Ausführung: Berechnet den Wert eines Ausdrucks
- Ausdrücke **zusammensetzen** durch:

**Anwendung** von Funktionen auf Argumente,  
dabei sind **Werte** die kleinsten „Bauteile“

# Programmieren in Haskell (2)


- In Haskell hat jeder Ausdruck (und Unterausdruck) einen **Typ**
- Typ = Art des Ausdrucks  
z.B. Buchstabe, Zahl, Liste von Zahlen, Funktion, ...
- Die Typen müssen zueinander passen:  
Z.B. **verboten**

`1 + "Hallo"`

Die Typen passen nicht zusammen (Zahl und Zeichenkette)




# Typen

Im GHCi Typen anzeigen lassen:

```
Prelude> :type 'C'   
'C' :: Char
```

Sprechweise: „'C' hat den Typ Char“



- Char ist der Typ in Haskell für Zeichen (engl. Character)
- Typnamen beginnen immer mit einem Großbuchstaben
- Im GHCi: `:set +t` führt dazu, dass mit jedem Ergebnis auch dessen Typ gedruckt wird.

```
*Main> :set +t   
*Main> zwei_mal_Zwei   
4  
it :: Integer  
*Main> oft_fuenf_addieren   
55  
it :: Integer
```

## Typen (2)

- Der Typ `Integer` stellt beliebig große ganze Zahlen dar
- Man kann Typen auch selbst angeben:




Schreibweise `Ausdruck :: Typ`




```
*Main> 'C' :: Char 
'C'
it :: Char
*Main> 'C' :: Integer 

<interactive>:1:0:
  Couldn't match expected type 'Integer'
  against inferred type 'Char'
  In the expression: 'C' :: Integer
  In the definition of 'it': it = 'C' :: Integer
```

# Wahrheitswerte: Der Datentyp Bool

- Werte (Datenkonstruktoren) vom Typ `Bool`:
  - `True` steht für „wahr“
  - `False` steht für „falsch“
- Operationen (Funktionen):
  - Logische Negation: `not`: liefert `True` für `False` und `False` für `True`
  - Logisches Und: `a && b`: nur `True`, wenn `a` und `b` zu `True` auswerten
  - Logisches Oder: `a || b`: `True`, sobald `a` oder `b` zu `True` auswertet.

```
*Main> not True 
False
it :: Bool
*Main> not False 
True
it :: Bool
*Main> True && True 
True
it :: Bool
```

```
*Main> False && True 
False
it :: Bool
*Main> False || False 
False
it :: Bool
*Main> True || False 
True
it :: Bool
```

# Ganze Zahlen: Int und Integer

- Der Typ **Int** umfasst ganze Zahlen **beschränkter** Größe (je nach Rechner, z.B.  $-2^{31}$  bis  $2^{31} - 1$ )
- Der Typ **Integer** umfasst ganze Zahlen **beliebiger** Größe



# Ganze Zahlen: Int und Integer

- Der Typ **Int** umfasst ganze Zahlen **beschränkter** Größe (je nach Rechner, z.B.  $-2^{31}$  bis  $2^{31} - 1$ )
- Der Typ **Integer** umfasst ganze Zahlen **beliebiger** Größe
- Darstellung der Zahlen ist identisch z.B. 1000
- Defaulting: Integer, wenn es nötig ist, sonst offenlassen:

# Ganze Zahlen: Int und Integer



- Der Typ `Int` umfasst ganze Zahlen **beschränkter** Größe (je nach Rechner, z.B.  $-2^{31}$  bis  $2^{31} - 1$ )
- Der Typ `Integer` umfasst ganze Zahlen **beliebiger** Größe
- Darstellung der Zahlen ist identisch z.B. 1000
- Defaulting: `Integer`, wenn es nötig ist, sonst offenlassen:

```
Prelude> :type 1000
1000 :: (Num t) => t
Prelude> :set +t
Prelude> 1000
1000
it :: Integer
```

- In etwa 1000 ist vom Typ `t`, wenn `t` ein **numerischer** Typ ist.
- Genauer: `(Num t) =>` ist eine sog. **Typklassenbeschränkung**
- `Int` und `Integer` sind numerische Typen (haben Instanzen für `Num`)

# Gleitkommazahlen

- Typen `Float` und `Double` (mit doppelter Genauigkeit)
- Kommastelle wird mit e. Punkt dargestellt
- Typklasse dazu: `Fractional`

```
Prelude> :type 10.5   
10.5 :: (Fractional t) => t  
Prelude> 10.5   
10.5  
it :: Double
```

- Beachte: Das Rechnen mit solchen Kommazahlen ist **ungenau!**

# Zeichen und Zeichenketten

- Der Typ `Char` repräsentiert Zeichen  
Darstellung: Zeichen in einfache Anführungszeichen, z.B. `'A'`
- Spezielle Zeichen (Auswahl):

Darstellung in Haskell	Zeichen, das sich dahinter verbirgt
<code>'\\'</code>	Backslash <code>\</code>
<code>'\''</code>	einfaches Anführungszeichen <code>'</code>
<code>'\"'</code>	doppeltes Anführungszeichen <code>"</code>
<code>'\n'</code>	Zeilenumbruch
<code>'\t'</code>	Tabulator

- Zeichenketten: Typ `String`  
Darstellung in doppelten Anführungszeichen, z.B. `"Hallo"`.
- Genauer ist der Typ `String` gleich zu `[Char]`  
d.h. eine Liste von Zeichen (Listen behandeln wir später)

# Beispiel Zeichenketten

```
> :set +t
> "Ein \n\'mehrzeiliger\'\nText mit \"Anfuhrungszeichen\""
"Ein \n\'mehrzeiliger'\nText mit \"Anfuhrungszeichen\""
it :: [Char]
> putStrLn "Ein \n\'mehrzeiliger\'\nText mit \"Anfuhrungszeichen\""
Ein
\'mehrzeiliger\'
Text mit "Anfuhrungszeichen"
it :: ()
```

# Operatoren auf Zahlen

- Operatoren auf Zahlen in Haskell:  
Addition  $+$ , Substraktion  $-$ , Multiplikation  $*$  und Division  $/$
- Beispiele:  $3 * 6$ ,  $10.0 / 2.5$ ,  $4 + 5 * 4$
- Beim  $-$  muss man aufpassen, da es auch für negative Zahlen benutzt wird


```
Prelude> 2 * -2 
```

```
<interactive>:1:0:
```

```
  Precedence parsing error
```

```
    cannot mix '*' [infixl 7] and prefix '-' [infixl 6]
```

```
    in the same infix expression
```








```
Prelude> 2 * (-2) 
```







```
-4
```

# Vergleichsoperationen

Gleichheitstest == und  
Ungleichheitstest /=

größer: >, größer oder gleich: >=  
kleiner: <, kleiner oder gleich: <=

```
Prelude> 1 == 3   
False  
Prelude> 3*10 == 6*5   
True  
Prelude> True == False   
False  
Prelude> False == False   
True  
Prelude> 2*8 /= 64   
True  
Prelude> 2+8 /= 10   
False  
Prelude> True /= False   
True
```

```
Prelude> 5 >= 5   
True  
Prelude> 5 > 5   
False  
Prelude> 6 > 5   
True  
Prelude> 4 < 5   
True  
Prelude> 4 < 4   
False  
Prelude> 4 <= 4   
True
```

# Funktionen

Funktionen verwenden, Funktionstypen, Funktionen definieren



# Funktionen






- Sei  $f$  eine Funktion, die den Rest einer Division mit Rest berechnet.
- In der Mathematik würde man z.B. schreiben:  $f(10, 3) = 1$

# Funktionen

- Sei  $f$  eine Funktion, die den Rest einer Division mit Rest berechnet.
- In der Mathematik würde man z.B. schreiben:  $f(10, 3) = 1$
- Schreibweise in Haskell: `f 10 3` oder auch `(f 10 3)`
- Grund: Man darf in Haskell **partiell anwenden**, z.B. `(f 10)`

# Funktionen



- Sei  $f$  eine Funktion, die den Rest einer Division mit Rest berechnet.
- In der Mathematik würde man z.B. schreiben:  $f(10, 3) = 1$
- Schreibweise in Haskell: `f 10 3` oder auch `(f 10 3)`
- Grund: Man darf in Haskell **partiell anwenden**, z.B. `(f 10)`
- Obige Funktion heißt in Haskell: `mod`
- Passend dazu: `div`: Ganzzahligen Anteil der Division mit Rest

```
Prelude> mod 10 3   
1  
Prelude> div 10 3   
3  
Prelude> mod 15 5   
0  
Prelude> div 15 5   
3  
Prelude> (div 15 5) + (mod 8 6)   
5
```

# Präfix und Infix

- `+`, `*`, ... werden **infix** verwendet:  
zwischen den Argumenten, z.B. `5+6`
- `mod`, `div` werden **präfix** verwendet:  
vor den Argumenten, z.B. `mod 10 3`

# Präfix und Infix

- `+`, `*`, ... werden **infix** verwendet:  
zwischen den Argumenten, z.B. `5+6`
- `mod`, `div` werden **präfix** verwendet:  
vor den Argumenten, z.B. `mod 10 3`
- Präfix-Operatoren infix verwenden:  
In Hochkommata setzen (  +  )  
z.B. `10 'mod' 3`
- Infix-Operatoren präfix verwenden:  
In runde Klammern setzen  
z.B. `(+) 5 6`




# Funktionstypen (1)

- Auch Funktionen haben einen Typ
- Funktion hat mehrere Eingaben und eine Ausgabe
- Jeder der Eingaben und die Ausgabe haben einen Typ
- Verkettung durch  $\rightarrow$

# Funktionstypen (1)

- Auch Funktionen haben einen Typ
- Funktion hat mehrere Eingaben und eine Ausgabe
- Jeder der Eingaben und die Ausgabe haben einen Typ
- Verkettung durch  $\rightarrow$

```

Prelude> :type not 
not :: Bool -> Bool
Prelude> :type (&&) 
(&&) :: Bool -> Bool -> Bool
Prelude> :type (||) 
(||) :: Bool -> Bool -> Bool
  
```

- not: Eine Eingabe vom Typ Bool und Ausgabe vom Typ Bool
- (&&): Zwei Eingaben vom Typ Bool und Ausgabe vom Typ Bool
- (||): Zwei Eingaben vom Typ Bool und Ausgabe vom Typ Bool

# Funktionstypen (2)

**Allgemein:**  $f$  erwartet  $n$  Eingaben, dann ist der Typ von  $f$ :

$$f :: \underbrace{\text{Typ}_1}_{\text{Typ des 1. Arguments}} \rightarrow \underbrace{\text{Typ}_2}_{\text{Typ des 2. Arguments}} \rightarrow \dots \rightarrow \underbrace{\text{Typ}_n}_{\text{Typ des } n. \text{ Arguments}} \rightarrow \underbrace{\text{Typ}_{n+1}}_{\text{Typ des Ergebnisses}}$$

- $\rightarrow$  in Funktionstypen ist **rechts-geklammert**
- $(\&\&) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$   
entspricht  $(\&\&) :: \text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool})$   
und **nicht**  $(\&\&) :: (\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool}$
- Das passt zur partiellen Anwendung:  
 $(\&\&) :: \text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool})$   
 $\text{True} :: \text{Bool}$ 


---

 $((\&\&) \text{True}) :: (\text{Bool} \rightarrow \text{Bool})$





# Funktionstypen (3)

Typen von mod und div:

```
mod :: Integer -> Integer -> Integer
```






```
div :: Integer -> Integer -> Integer
```

In Wirklichkeit:

```
Prelude> :type mod 
mod :: (Integral a) => a -> a -> a
Prelude> :type div 
div :: (Integral a) => a -> a -> a
```

In etwa: Für alle Typen  $a$  die Integral-Typen sind,  
hat mod den Typ  $a \rightarrow a \rightarrow a$

## Funktionstypen (4)

```
Prelude> :type (==)   
(==) :: (Eq a) => a -> a -> Bool  
Prelude> :type (<)   
(<) :: (Ord a) => a -> a -> Bool  
Prelude> :type (<=)   
(<=) :: (Ord a) => a -> a -> Bool  
Prelude> :type (+)   
(+) :: (Num a) => a -> a -> a  
Prelude> :type (-)   
(-) :: (Num a) => a -> a -> a
```

# Funktionstypen (5)

Die Typen müssen stets passen, sonst gibt's einen **Typfehler**:

```
Prelude> :type not
not :: Bool -> Bool
Prelude> not 'C'
<interactive>:1:4:
  Couldn't match expected type 'Bool' against inferred type 'Char'
  In the first argument of 'not', namely 'C'
  In the expression: not 'C'
  In the definition of 'it': it = not 'C'
```

Manchmal “merkwürdige” Fehlermeldung:

```
Prelude> not 5
<interactive>:1:4:
  No instance for (Num Bool)
    arising from the literal '5' at <interactive>:1:4
  Possible fix: add an instance declaration for (Num Bool)
  In the first argument of 'not', namely '5'
  In the expression: not 5
  In the definition of 'it': it = not 5
```

# Funktionen selbst definieren

```
verdopple x = x + x
```

In der Mathematik würde man schreiben  $\text{verdopple}(x) = x + x$

# Funktionen selbst definieren

```
verdopple x = x + x
```

In der Mathematik würde man schreiben  $\text{verdopple}(x) = x + x$

Allgemein:

*funktion\_Name*  $par_1 \dots par_n = Haskell\_Ausdruck$

wobei

- $par_i$ : Formale Parameter, z.B. Variablen  $x, y, \dots$
- Die  $par_i$  dürfen rechts im *Haskell\_Ausdruck* verwendet werden
- *funktion\_Name* muss mit Kleinbuchstaben oder einem Unterstrich beginnen

# Funktionen selbst definieren

```
verdoppeln :: Integer -> Integer  
verdoppeln x = x + x
```

In der Mathematik würde man schreiben  $\text{verdoppeln}(x) = x + x$

Allgemein:

$$\text{funktion\_Name } par_1 \dots par_n = \text{Haskell\_Ausdruck}$$

wobei

- $par_i$ : Formale Parameter, z.B. Variablen  $x, y, \dots$
- Die  $par_i$  dürfen rechts im *Haskell\_Ausdruck* verwendet werden
- *funktion\_Name* muss mit Kleinbuchstaben oder einem Unterstrich beginnen

Man darf auch den Typ angeben!

# Funktion testen

```
Prelude> :load programme/einfacheFunktionen.hs ↵  
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs)  
Ok, modules loaded: Main.  
*Main> verdopple 5 ↵  
10  
*Main> verdopple 100 ↵  
200  
*Main> verdopple (verdopple (2*3) + verdopple (6+9)) ↵  
84
```

# Fallunterscheidung: if-then-else

**Syntax:** `if b then e1 else e2`

```
verdoppleGerade :: Integer -> Integer
```

```
verdoppleGerade x = if even x then verdopple x else x
```

even testet, ob eine Zahl gerade ist:

```
even x = x 'mod' 2 == 0
```

```
*Main> :reload
```



```
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )
```

```
Ok, modules loaded: Main.
```

```
*Main> verdoppleGerade 50
```



```
100
```

```
*Main> verdoppleGerade 17
```



```
17
```




# Fallunterscheidung: if-then-else (2)

Verschachteln von if-then-else:

```
jenachdem :: Integer -> Integer
jenachdem x = if x < 100 then 2*x else
              if x <= 1000 then 3*x else x
```

Falsche Einrückung:

```
jenachdem x =
if x < 100 then 2*x else
  if x <= 1000 then 3*x else x
```

```
Prelude> :reload 
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )

programme/einfacheFunktionen.hs:9:0:
  parse error (possibly incorrect indentation)
Failed, modules loaded: none.
Prelude>
```

# Noch ein Beispiel

```
verdoppeln_oder_verdreifachen :: Bool -> Integer -> Integer
verdoppeln_oder_verdreifachen b x =
  if b then 2*x else 3*x
```

```
*Main> verdoppeln_oder_verdreifachen True 10
20
*Main> verdoppeln_oder_verdreifachen False 10
30
```

verdoppeln mithilfe von verdoppeln\_oder\_verdreifachen:


```
verdoppeln2 :: Integer -> Integer
verdoppeln2 x = verdoppeln_oder_verdreifachen True x
-- oder auch:
verdoppeln3 :: Integer -> Integer
verdoppeln3 = verdoppeln_oder_verdreifachen True
```

verdoppeln3: **keine** Eingabe, **Ausgabe ist eine Funktion**


# Higher-Order Funktionen

- D.h.: **Rückgabewerte** dürfen in Haskell auch **Funktionen** sein
- Auch **Argumente** (Eingaben) dürfen **Funktionen** sein:

```
wende_an_und_addiere f x y = (f x) + (f y)
```

```
*Main> wende_an_und_addiere verdopple 10 20 
```

```
60
```

```
*Main> wende_an_und_addiere jenachdem 150 3000 
```

```
3450
```

Daher spricht man auch von **Funktionen höherer Ordnung!**

# Nochmal Typen

## Typ von `wende_an_und_addiere`

```
wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer -> Integer
```

```
wende_an_und_addiere f x y = (f x) + (f y)
```

# Nochmal Typen

## Typ von wende\_an\_und\_addiere

`wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer -> Integer`  
⏟ ⏟ ⏟ ⏟  
Typ von f Typ von x Typ von y Typ des Ergebnisses

`wende_an_und_addiere f x y = (f x) + (f y)`

# Nochmal Typen

## Typ von `wende_an_und_addiere`

```
wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer -> Integer
```

```
wende_an_und_addiere f x y = (f x) + (f y)
```

Achtung: Im Typ

```
(Integer -> Integer) -> Integer -> Integer -> Integer
```

darf man die Klammern **nicht** weglassen:

```
Integer -> Integer -> Integer -> Integer -> Integer
```

denn das entspricht

```
Integer -> (Integer -> (Integer -> (Integer -> Integer))).
```

# Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a  
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

# Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (a -> a) -> a -> a
```



# Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

# Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a  
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B.  $a = \text{Int}$

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B.  $a = \text{Bool}$

```
zweimal_anwenden :: (a -> a) -> a -> a
```

# Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a  
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B.  $a = \text{Int}$

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B.  $a = \text{Bool}$

```
zweimal_anwenden :: (Bool -> Bool) -> Bool -> Bool
```

# Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B.  $a = \text{Int}$

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B.  $a = \text{Bool}$

```
zweimal_anwenden :: (Bool -> Bool) -> Bool -> Bool
```

z.B.  $a = \text{Char} \rightarrow \text{Char}$

```
zweimal_anwenden :: (a -> a) -> a -> a
```

# Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B.  $a = \text{Int}$

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B.  $a = \text{Bool}$

```
zweimal_anwenden :: (Bool -> Bool) -> Bool -> Bool
```

z.B.  $a = \text{Char} \rightarrow \text{Char}$

```
zweimal_anwenden :: ((Char->Char)->(Char->Char))->(Char->Char)->(Char->Char)
```

# Polymorphe Typen (2)

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Der GHCi setzt beim Anwenden automatisch die richtigen Typen ein

```
*> :type verdopple
verdopple :: Integer -> Integer
*> zweimal_anwenden verdopple 10
40
*> :type not
not :: Bool -> Bool
*> zweimal_anwenden not True
True
*> :type vergleiche True
vergleiche True :: Bool -> String
*> zweimal_anwenden (vergleiche True) True
<interactive>:1:18:
  Couldn't match expected type 'Bool' against inferred type 'String'
  In the first argument of 'zweimal_anwenden', namely '(vergleiche True)'
  In the expression: zweimal_anwenden (vergleiche True)
```

# Formale Parameter

Gesucht:

Funktion erhält zwei Eingaben und liefert  
"Die Eingaben sind gleich",  
wenn die beiden Eingaben gleich sind.

# Formale Parameter

Gesucht:

Funktion erhält zwei Eingaben und liefert  
"Die Eingaben sind gleich",  
wenn die beiden Eingaben gleich sind.

Falscher Versuch:

```
sonicht x x = "Die Eingaben sind gleich!"
```

```
Conflicting definitions for 'x'  
In the definition of 'sonicht'  
Failed, modules loaded: none.
```

Die formalen Parameter müssen **unterschiedliche** Namen haben.

```
vergleiche x y =  
  if x == y then "Die Eingaben sind gleich!" else ""
```



# Rekursion

*„Wer Rekursion verstehen will, muss Rekursion verstehen.“*

# Rekursion

Eine Funktion ist **rekursiv**, wenn sie sich selbst aufrufen kann.

$$f\ x\ y\ z = \dots (f\ a\ b\ c) \dots$$

oder z.B. auch

$$\begin{aligned} f\ x\ y\ z &= \dots (g\ a\ b) \dots \\ g\ x\ y &= \dots (f\ c\ d\ e) \dots \end{aligned}$$

# Rekursion (2)

Bei Rekursion muss man aufpassen:

```
endlos_eins_addieren x = endlos_eins_addieren (x+1)
```

endlos\_eins\_addieren a terminiert nicht!

# Rekursion (3)

So macht man es richtig:

- **Rekursionsanfang**: Der Fall, für den sich die Funktion **nicht** mehr selbst aufruft.
- **Rekursionsschritt**: Der rekursive Aufruf
- Dabei darauf achten, dass der Rekursionsanfang irgendwann sicher erreicht wird.

Beispiel:

```
erste_rekursive_Funktion x =  
  if x <= 0 then 0 -- Rekursionsanfang  
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

# Rekursion (4)

```
erste_rekursive_Funktion x =  
  if x <= 0 then 0 -- Rekursionsanfang  
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Was berechnet `erste_rekursive_Funktion`?

- `erste_rekursive_Funktion n` ergibt 0 für  $n \leq 0$
- $n > 0$ ?

Testen:

```
*Main> erste_rekursive_Funktion 5  
15  
*Main> erste_rekursive_Funktion 10  
55  
*Main> erste_rekursive_Funktion 11  
66  
*Main> erste_rekursive_Funktion 12  
78  
*Main> erste_rekursive_Funktion 1  
1
```

# Rekursion (5)

```
erste_rekursive_Funktion x =  
  if x <= 0 then 0 -- Rekursionsanfang  
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Ein Beispiel nachvollziehen:

```
erste_rekursive_Funktion 5  
= 5 + erste_rekursive_Funktion 4  
= 5 + (4 + erste_rekursive_Funktion 3)  
= 5 + (4 + (3 + erste_rekursive_Funktion 2))  
= 5 + (4 + (3 + (2 + erste_rekursive_Funktion 1)))  
= 5 + (4 + (3 + (2 + (1 + erste_rekursive_Funktion 0))))  
= 5 + (4 + (3 + (2 + (1 + 0))))  
= 15
```

# Rekursion (6)

```
erste_rekursive_Funktion x =  
  if x <= 0 then 0 -- Rekursionsanfang  
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Allgemein:

```
erste_rekursive_Funktion x  
= x + erste_rekursive_Funktion (x-1)  
= x + (x-1) + erste_rekursive_Funktion (x-2))  
= x + (x-1) + (x-2) + erste_rekursive_Funktion (x-3)))  
= ...
```

# Rekursion (6)

```

erste_rekursive_Funktion x =
  if x <= 0 then 0                -- Rekursionsanfang
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
  
```

Allgemein:

```

erste_rekursive_Funktion x
= x + erste_rekursive_Funktion (x-1)
= x + (x-1) + erste_rekursive_Funktion (x-2))
= x + (x-1) + (x-2) + erste_rekursive_Funktion (x-3))
= ...
  
```

Das ergibt  $x + (x - 1) + (x - 2) + \dots + 0 = \sum_{i=0}^x i$



# Rekursion (7)

Warum ist Rekursion nützlich?

- Man kann damit schwierige Probleme einfach lösen

Wie geht man vor?

- Rekursionsanfang:

Der einfache Fall, für den man die Lösung direkt kennt

$$\text{(z.B. } \sum_{i=0}^0 i = 0 \text{)}$$

- Rekursionsschritt:

Man löst ganz wenig selbst, bis das Problem etwas kleiner ist.

Das (immer noch große) Restproblem erledigt die Rekursion,

$$\text{z.B. } \sum_{i=0}^x i = x + \sum_{i=0}^{x-1} i \text{ (für } x > 0 \text{)}$$

# Rekursion (8)

Beispiel:  $n$ -mal Verdoppeln

- Rekursionsanfang:

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n =
```

# Rekursion (8)

Beispiel:  $n$ -mal Verdoppeln

- Rekursionsanfang:  $n = 0$ : Gar nicht verdoppeln

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n =
  if n == 0 then x
```

# Rekursion (8)

Beispiel:  $n$ -mal Verdoppeln

- Rekursionsanfang:  $n = 0$ : Gar nicht verdoppeln
- Rekursionsschritt:  $n > 0$ : Einmal selbst verdoppeln, die restlichen  $n - 1$  Verdopplungen der Rekursion überlassen

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n =
  if n == 0 then x
  else n_mal_verdoppeln (verdopple x) (n-1)
```

# Pattern matching (auf Zahlen)

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n =
  if n == 0 then x
  else n_mal_verdoppeln (verdopple x) (n-1)
```

Man darf statt Variablen auch **Pattern** in der Funktionsdefinition verwenden, und **mehrere Definitionsgleichungen** angeben. Die Pattern werden von unten nach oben abgearbeitet.

```
n_mal_verdoppeln2 :: Integer -> Integer -> Integer
n_mal_verdoppeln2 x 0 = x
n_mal_verdoppeln2 x n = n_mal_verdoppeln2 (verdopple x) (n-1)
```

## Falsch:

```
n_mal_verdoppeln2 :: Integer -> Integer -> Integer
n_mal_verdoppeln2 x n = n_mal_verdoppeln2 (verdopple x) (n-1)
n_mal_verdoppeln2 x 0 = x
```

# Guards

**Guards** (Wächter): Boolesche Ausdrücke,  
die die Definition der Funktion festlegen

```
f x1 ... xm
  | 1. Guard = e1
  ...
  | n. Guard = en
```

- Abarbeitung von oben nach unten
- Erster Guard, der zu True auswertet, bestimmt die Definition.

```
n_mal_verdoppeln3 :: Integer -> Integer -> Integer
n_mal_verdoppeln3 x n
  | n == 0      = x
  | otherwise = n_mal_verdoppeln3 (verdopple x) (n-1)
```

Vordefiniert: otherwise = True

# Die Error-Funktion

```
n_mal_verdoppeln3 :: Integer -> Integer -> Integer
n_mal_verdoppeln3 x n
  | n == 0    = x
  | otherwise = n_mal_verdoppeln3 (verdopple x) (n-1)
```

Was passiert bei negativem  $n$ ?

# Die Error-Funktion

```
n_mal_verdoppeln3 :: Integer -> Integer -> Integer
n_mal_verdoppeln3 x n
  | n == 0    = x
  | otherwise = n_mal_verdoppeln3 (verdopple x) (n-1)
```

Was passiert bei negativem n?

- `error :: String -> a`

```
n_mal_verdoppeln4 :: Integer -> Integer -> Integer
n_mal_verdoppeln4 x n
  | n < 0    = error "Negatives Verdoppeln verboten!"
  | n == 0    = x
  | otherwise = n_mal_verdoppeln4 (verdopple x) (n-1)
```

```
*Main> n_mal_verdoppeln4 10 (-10)
*** Exception:
in n_mal_verdoppeln4: negatives Verdoppeln ist verboten
```



# Beispiel

*In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab. Wieviel Rehe gibt es im Wald am 1.1. des Jahres  $n$ ?*



# Beispiel

*In einem Wald werden **am 1.1. des ersten Jahres 10 Rehe gezählt.** Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab. Wieviel Rehe gibt es im Wald am 1.1. des Jahres  $n$ ?*

Rekursionsanfang: Jahr 1, 10 Rehe

```
anzahlRehe 1 = 10
```

# Beispiel

*In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab. Wieviel Rehe gibt es im Wald am 1.1. des Jahres  $n$ ?*

Rekursionsanfang: Jahr 1, 10 Rehe

Rekursionsschritt: Sei  $k$  = Anzahl Rehe am 1.1. des Jahres  $n - 1$

```
anzahlRehe 1 = 10
```

```
anzahlRehe n =
```

```
anzahlRehe (n-1)
```

# Beispiel

In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung **verdreifacht**. In jedem Jahr schießt der Förster **17 Rehe**. In jedem **2. Jahr** gibt der Förster **die Hälfte** der verbleibenden Rehe am 31.12. an einen anderen Wald ab. Wieviel Rehe gibt es im Wald am 1.1. des Jahres  $n$ ?

Rekursionsanfang: Jahr 1, 10 Rehe

Rekursionsschritt: Sei  $k$  = Anzahl Rehe am 1.1. des Jahres  $n - 1$

$$\text{Jahr } n: \begin{cases} 3 * k - 17, & \text{falls } n - 1 \text{ kein zweites Jahr} \\ (3 * k - 17) / 2, & \text{falls } n - 1 \text{ ein zweites Jahr} \end{cases}$$

```
anzahlRehe 1 = 10
anzahlRehe n = if even (n-1)
                then 3*(anzahlRehe (n-1))-17 `div` 2
                else 3*(anzahlRehe (n-1))-17
```

# Beispiel (2)

```
*Main> anzahlRehe 1
10
*Main> anzahlRehe 2
13
*Main> anzahlRehe 3
11
*Main> anzahlRehe 4
16
*Main> anzahlRehe 5
15
*Main> anzahlRehe 6
28
*Main> anzahlRehe 7
33
*Main> anzahlRehe 8
82
*Main> anzahlRehe 9
114
*Main> anzahlRehe 10
325
*Main> anzahlRehe 50
3626347914090925
```

# Let-Ausdrücke: Lokale Definitionen

```
anzahlReihe 1 = 10
anzahlReihe n = if even (n-1)
                 then ((3*anzahlReihe (n-1))-17) 'div' 2
                 else 3*(anzahlReihe (n-1))-17
```

Mit let:

```
anzahlReihe2 1 = 10
anzahlReihe2 n = let k = (3*anzahlReihe2 (n-1))-17
                  in  if even (n-1) then k 'div' 2
                  else k
```

# Let-Ausdrücke: Lokale Definitionen (2)

Allgemeiner:

```
let  Variable1  =  Ausdruck1  
     Variable2  =  Ausdruck2  
     ...  
     VariableN  =  AusdruckN  
in  Ausdruck
```