Deskriptive Programmierung

SS 2016

Jun.-Prof. Dr. Janis Voigtländer
Institut für Informatik III
Universität Bonn

Zeiten im SS 2016

	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
8					
9					
10	Vorlesung				Vorlesung
11					
12					Übung
13					
14					
15					
16					

Übungen und Kommunikation

- Übungen: **Fr 12:15 – 13:45** (Beginn: vorauss. **22.04.2016**)
  - Kriterien für erfolgreiche Teilnahme (und damit Zulassung zur Prüfung):
    - regelmäßige Einreichung von Lösungen für gekennzeichnete Aufgaben
    - 60% der bei diesen erreichbaren Punkte (zu zwei Stichtagen)
    - außerdem 30% pro Übungsblatt
    - Programmierprojekt gegen Ende des Semesters
    - Genaues/Details, siehe Aushang vor Prüfungsamt!

- Webseite(n) zur Vorlesung als Hauptkommunikationsmedium:
  - „News of the Day" (Bitte regelmäßig checken!)
  - Folien (als PDF-Dateien) zum Download (jeweils nach der Vorlesung)
  - weiterführende Literaturangaben, Links etc.
  - Links zu benötigter Software

http://www.iai.uni-bonn.de/~jv/teaching/dp/

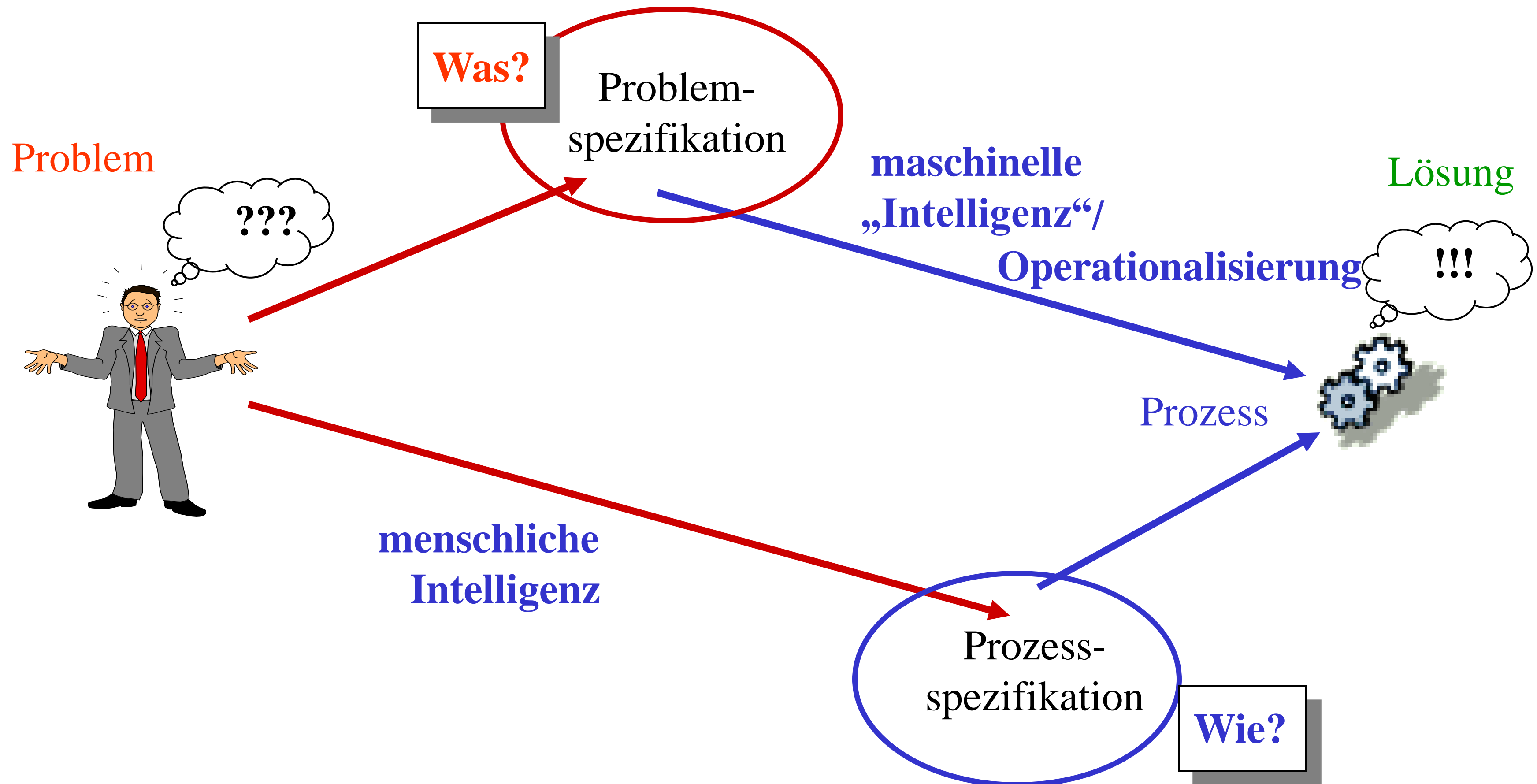
https://ecampus.uni-bonn.de/goto_ecampus_crs_794521.html

Deskriptive Programmierung

Einführung und Motivation

Ideal (und ein Stück weit, Historie) der deskriptiven Programmierung

„Befreiung" des Menschen von der Notwendigkeit, zur Problemlösung führende Rechenprozesse explizit zu planen und zu spezifizieren: **„Was statt Wie"**



© 2016 J. Voigtländer

Deklarative Programmierung als Programmierparadigma

Die deklarative Programmierung ist ein Programmierparadigma, welches auf mathematischer, rechnerunabhängiger Theorie beruht.

Zu den deklarativen Programmiersprachen gehören:

- funktionale Sprachen (u.a. LISP, ML, Miranda, Gofer, Haskell)
- logische Sprachen (u.a. Prolog)
- funktional-logische Sprachen (u.a. Babel, Escher, Curry, Oz)
- Datenflusssprachen (wie Val oder Linda)

(aus Wikipedia, 07.04.08)

- In der Regel erlauben deklarative Sprachen in irgendeiner Form die Einbettung imperativer Programmteile, mehr oder weniger direkt und/oder „diszipliniert".

- Andere Programmiersprachenkategorien, einigermaßen orthogonal zu dekl./imp.:
  - Objektorientierte oder ereignisorientierte Sprachen
  - Parallelverarbeitende/nebenläufige Sprachen
  - Stark oder schwach, statisch oder dynamisch, oder gar nicht getypte Sprachen

Charakteristika deskriptiver Spezifikationen (vs. imperativer Programme)

- Deskriptive Programme (Spezifikationen) sind oft
  - signifikant kürzer
  - signifikant lesbarer
  - signifikant wartbarer (und zuverlässiger)

  als ihre imperativen „Gegenstücke“.

- Insbesondere funktionale Sprachen betonen Abstraktionen, die Seiteneffekte für Programmteile ausschließen oder gezielt (und flexibel) unter Kontrolle halten.
  (S. Peyton Jones: „Haskell is the world’s finest imperative programming language.“)

- Deskriptive Konzepte eignen sich besonders gut zur Realisierung/Einbettung domänenspezifischer Sprachen (DSLs).

- aber:
  - Deskriptive Sprachen sind noch weniger verbreitet als imperative Sprachen.
  - Die Produktentwicklung für das Arbeiten mit deskriptiven Sprachen ist nicht so weit fortgeschritten.
  - Beschränkungen in der Anwendung liegen oft (Annahmen über, oder tatsächlich) mangelhaft effiziente Operationalisierungsmethoden zu Grunde.

Deklarative Programmierung „in the Real World“

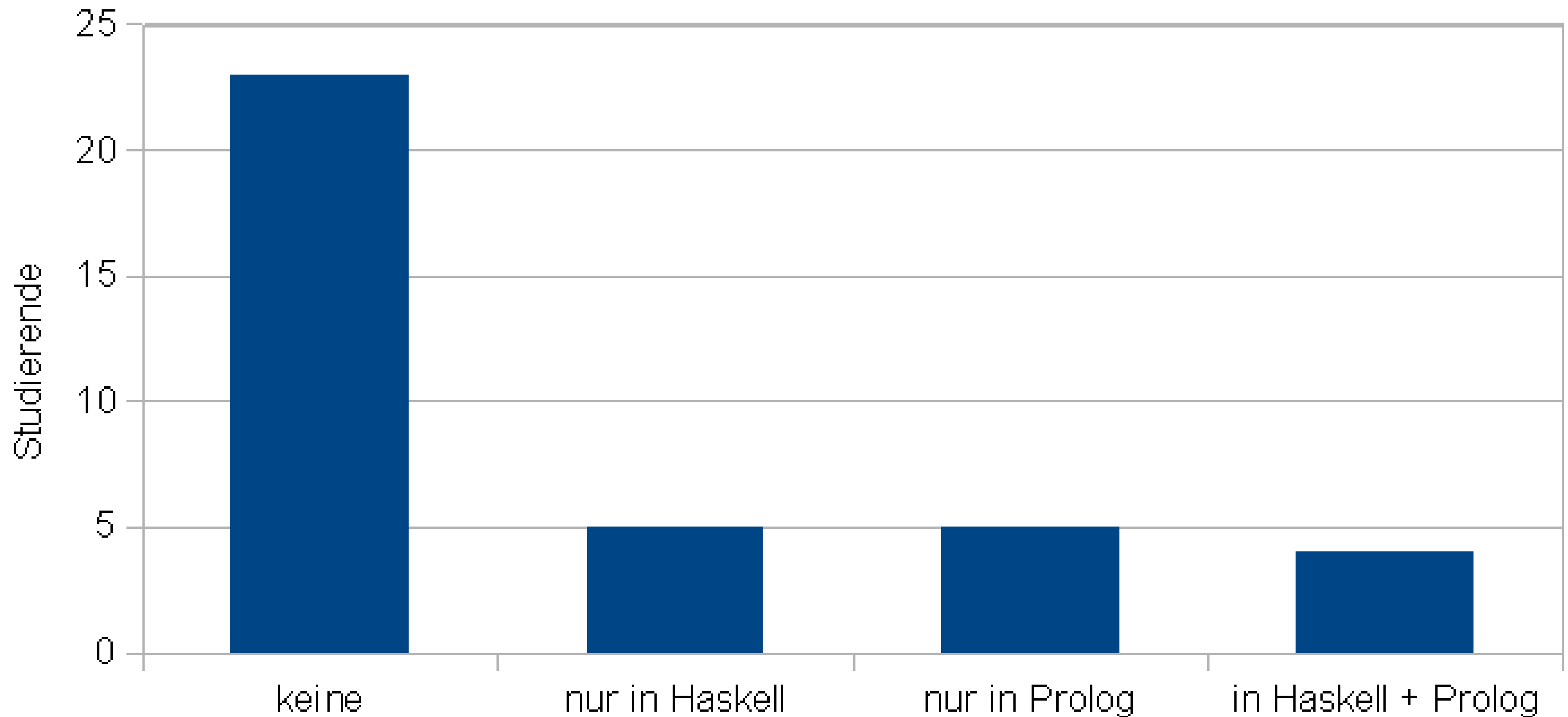
- Kommerzielle Anwender:
  - im Bankensektor (Trading, Quantitative Analysis), z.B. Barclays Capital, Jane Street Capital, Standard Chartered Bank, McGraw Hill Financial, …
  - im Bereich Communication/Web Services, z.B. Ericsson, Facebook, Google
  - Hardware-Design/Verification, z.B. Intel, Bluespec, Antiope
  - System-Level Development, z.B. Microsoft
  - High Assurance Software, z.B. Galois

http://cufp.org/

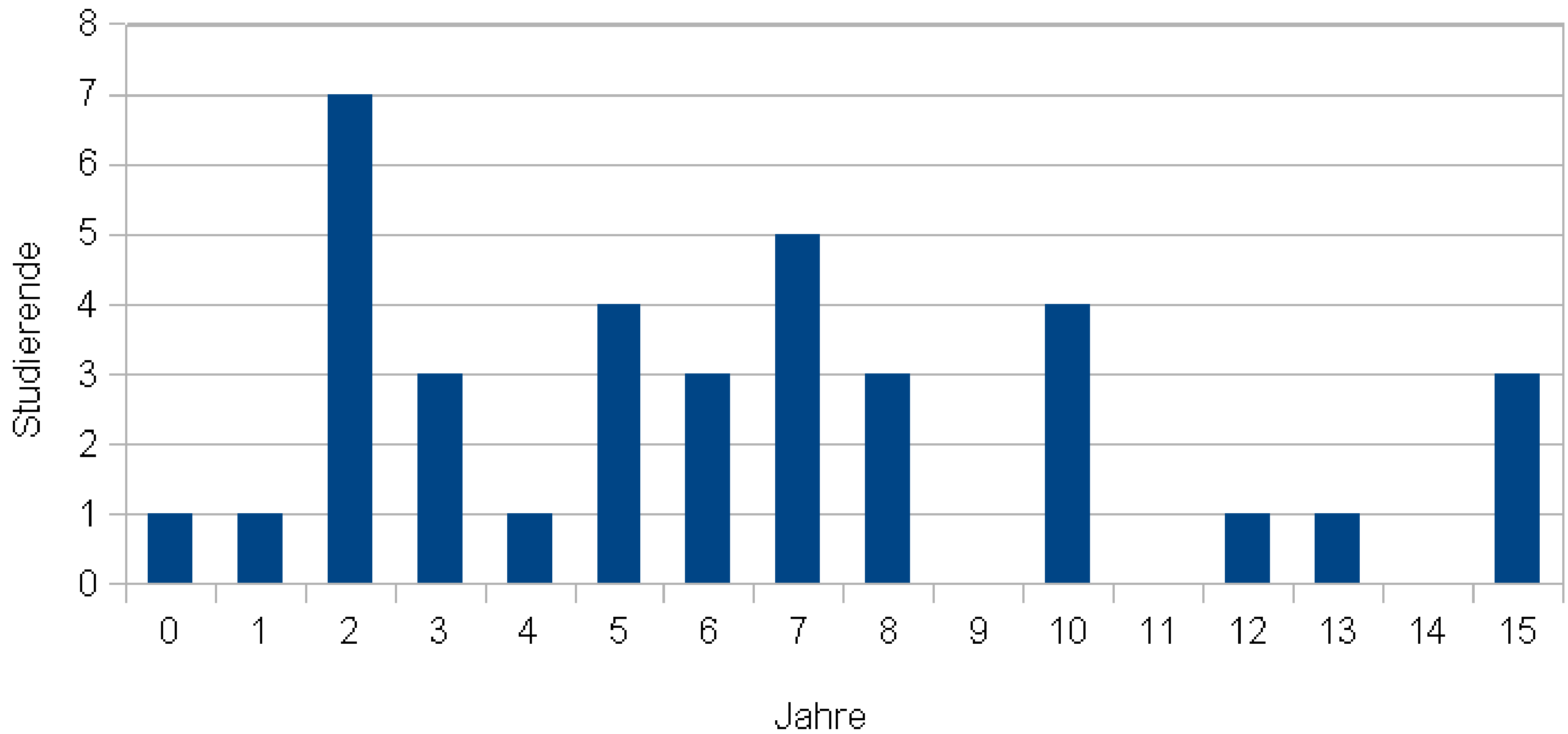
http://groups.google.co.uk/group/cu-lp

- „nicht-akademische“ Sprachen:
  - für spezielle Anwendungsgebiete, z.B. Erlang (Ericsson), reFLect (Intel)
  - für allgemeine Anwendungen, z.B. F# (Microsoft)
  - Einfluss auf Mainstream-Sprachen, z.B. Java, C#, und „sogar“ Visual Basic (allgemein: LINQ-Framework)

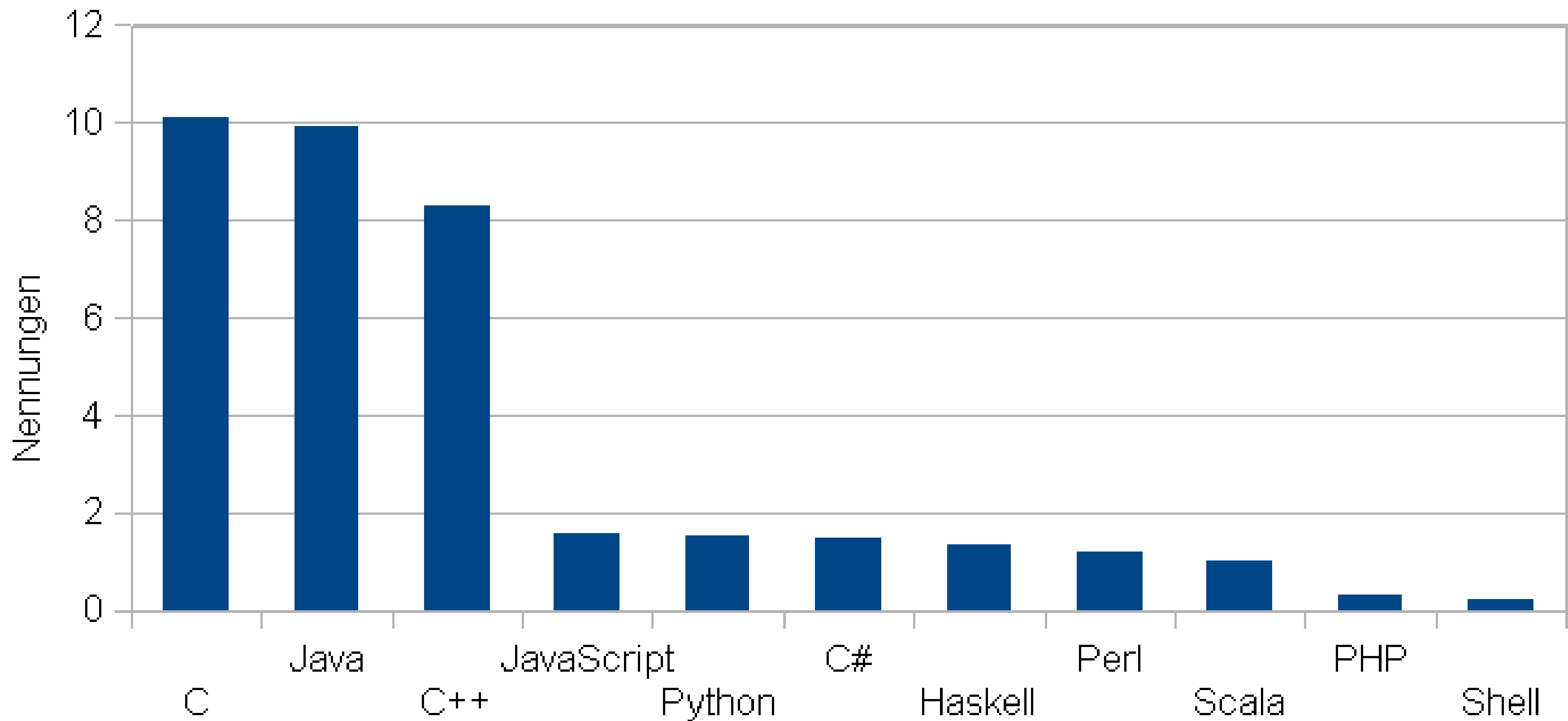
Umfrage zu Beginn der Vorlesung im SS 2013 (1)



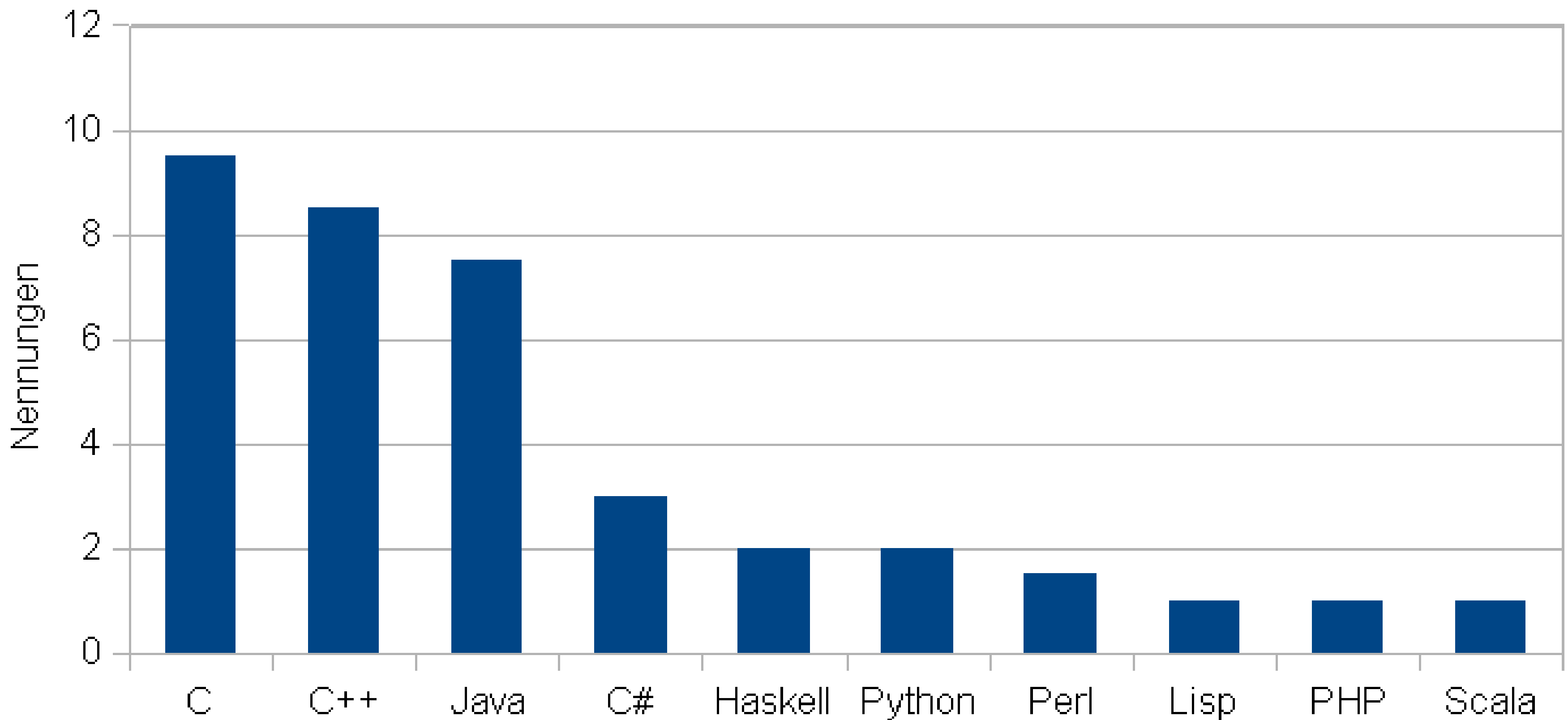
Umfrage zu Beginn der Vorlesung im SS 2013 (2)



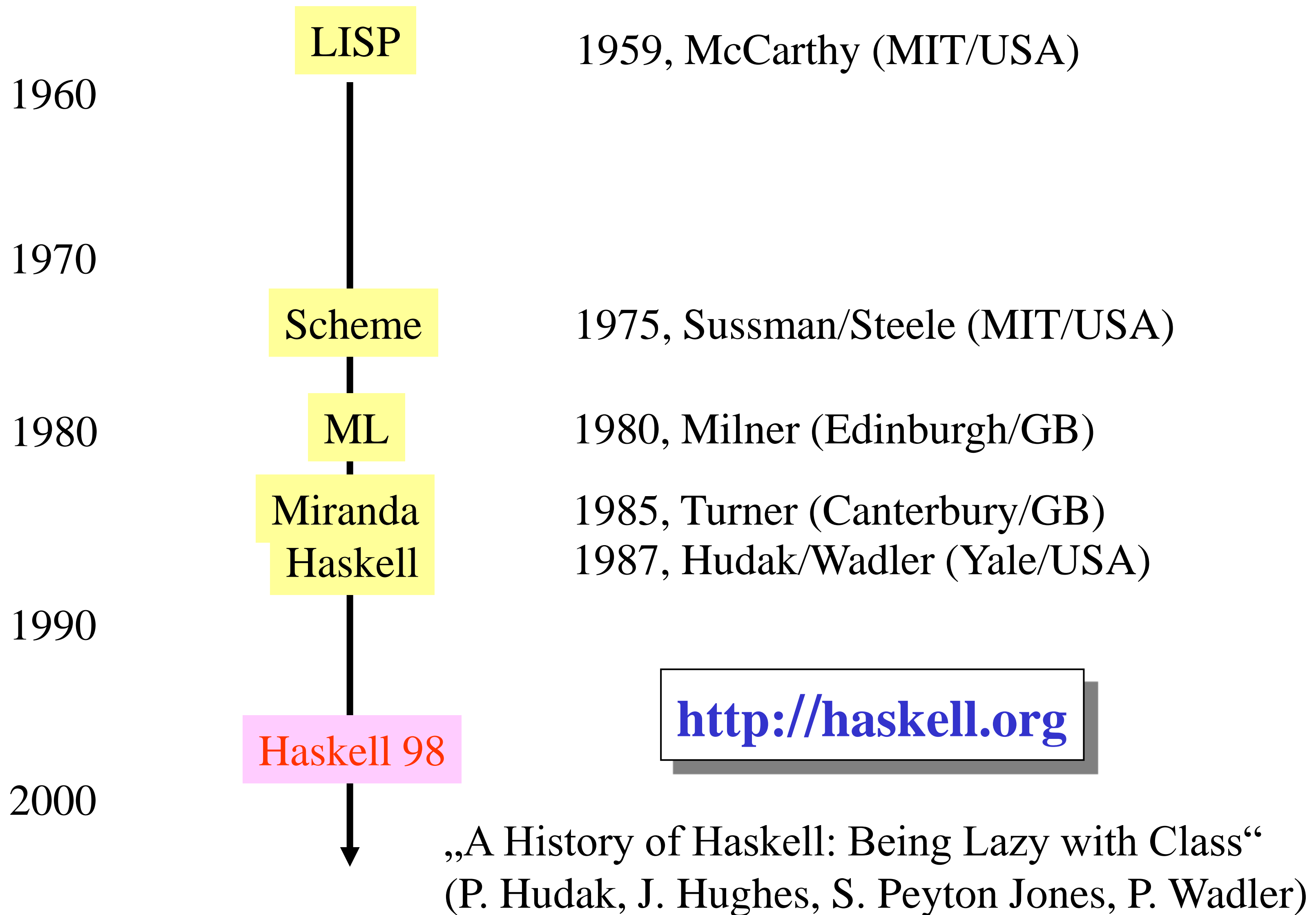
Umfrage zu Beginn der Vorlesung im SS 2013 (3)



Umfrage zu Beginn der Vorlesung im SS 2013 (4)



Wichtige funktionale Sprachen im historischen Überblick



Wofür steht „Haskell“?

- Namen von Programmiersprachen sind oft Akronyme
  (z.B. COBOL, FORTRAN, BASIC, …)

- Der Name „Haskell“ dagegen leitet sich von einer Person her:

Haskell Brooks Curry
(1900 - 1982)

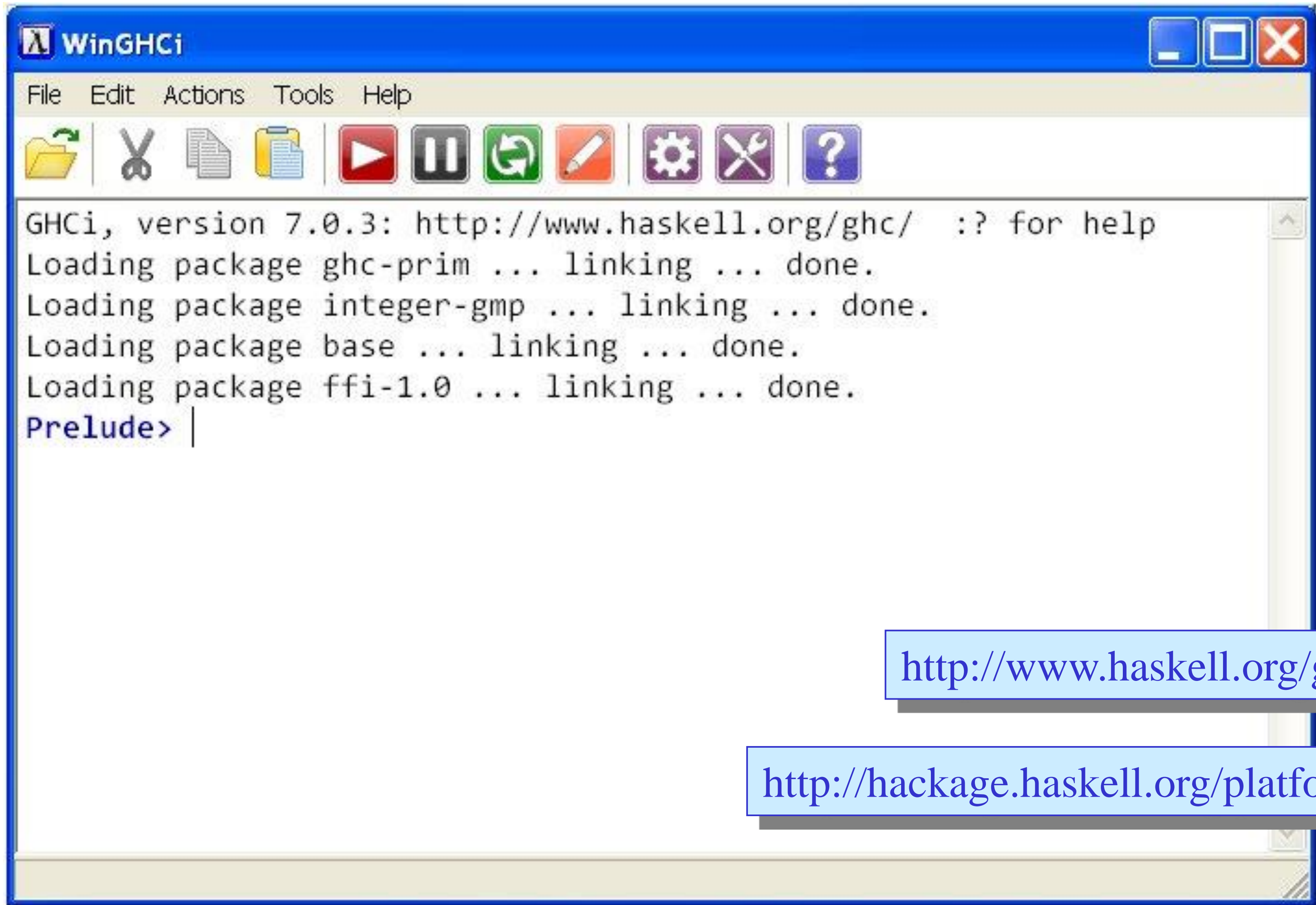
amerikanischer Logiker

Literatur zu Haskell

- Lehrbücher (zum Beispiel):
  - R. Bird:
    Introduction to Functional Programming using Haskell
    Prentice Hall, 1998
  - M. Block, A. Neumann:
    Haskell-Intensivkurs
    Springer-Verlag, 2011
  - P. Hudak:
    The Haskell School of Expression
    Cambridge University Press, 2000
  - G. Hutton:
    Programming in Haskell
    Cambridge University Press, 2007
  - S. Thompson:
    Haskell - The Craft of Functional Programming
    Addison Wesley, 2011

- einführender Artikel:
  P. Hudak, J. Peterson, J. Fasel: A Gentle Introduction to Haskell
  (haskell.org, 1999)

Verwendete Implementierung: GHC(i)



http://www.haskell.org/ghc/

http://hackage.haskell.org/platform/

Deskriptive Programmierung

Beispiele in Haskell eingebetteter DSLs

Beschreibung von Grafiken mittels „gloss“

- eine einfache Bibliothek (siehe Installationsanleitung auf Übungsblatt)
- Grundkonzepte:

```
Float, String, Path, Color, Picture

text      :: String → Picture
line      :: Path → Picture
polygon   :: Path → Picture
arc       :: Float → Float → Float → Picture
circle    :: Float → Picture
…

color     :: Color → Picture → Picture
translate :: Float → Float → Picture → Picture
rotate    :: Float → Picture → Picture
scale     :: Float → Float → Picture → Picture

pictures  :: [ Picture ] → Picture
```

Beschreibung von Grafiken mittels „gloss“

- Verwendung in konkretem „Programm“:

```
module Main (main) where

import Graphics.Gloss

main = display (InWindow "Bsp" (100, 100) (0, 0)) white scene

scene = pictures
        [
          circleSolid 20
        , translate 25 0 (color red (polygon [(0, 0), (10, –5), (10, 5)]))
        ]
```

- Let‘s play a bit. …

Eine ganz andere Domäne: Verarbeitung arithmetischer Ausdrücke

- Nehmen wir an, wir wollen arithmetische Ausdrücke in „Maschinencode“ compilieren, also zum Beispiel:

"2+3*5"	$\mapsto$	"LIT 2; LIT 3; LIT 5; MUL; ADD; "
"2*3+5"	$\mapsto$	"LIT 2; LIT 3; MUL; LIT 5; ADD; "

- Zunächst müssen wir erstmal die Struktur von (gültigen) Ausdrücken beschreiben.

- Zum Beispiel mittels einer formalen Grammatik:

```
expr    ::=  term + expr | term
term    ::=  factor * term | factor
factor  ::=  nat | (expr)
```

- … und jetzt könnten wir (in einer „konventionellen“ Programmiersprache) einen Algorithmus zum Parsen entsprechend einer/dieser Grammatik entwickeln/umsetzen.

Eine ganz andere Domäne: Verarbeitung arithmetischer Ausdrücke

- Attraktiver wäre, möglichst direkt die vorhandene Beschreibung

```
expr   ::= term + expr | term
term   ::= factor * term | factor
factor ::= nat | (expr)
```

zu verwenden, und diese selbst als „Programm“ zu lesen.

- Immerhin nah dran:

```
expr   = ( ADD <$> term <* char '+' <*> expr ) ||| term

term   = ( MUL <$> factor <* char '*' <*> term ) ||| factor

factor = ( LIT <$> nat ) ||| ( char '(' *> expr <* char ')' )
```

- Schonmal ausprobieren:

```
> parse expr "2*3+5"
ADD (MUL (LIT 2) (LIT 3)) (LIT 5)
```

Eine ganz andere Domäne: Verarbeitung arithmetischer Ausdrücke

- Um die eigentlich gewünschte Ausgabe zu erhalten:

```
data Expr  =  LIT Int  |  ADD Expr Expr  |  MUL Expr Expr

instance Show Expr where
  show (LIT n)           =  "LIT "  ++  show n  ++  "; "
  show (ADD e1 e2)  =  show e1  ++  show e2  ++  "ADD; "
  show (MUL e1 e2)  =  show e1  ++  show e2  ++  "MUL; "
```

- Dann tatsächlich:

```
>  parse expr "2*3+5"
LIT 2; LIT 3; MUL; LIT 5; ADD;
```

- Alternativ zum Beispiel auch direkte Berechnung des Ergebnisses möglich:

```
eval  (LIT n)          =  n
eval  (ADD e1 e2)  =  eval e1  +  eval e2
eval  (MUL e1 e2)  =  eval e1  *  eval e2
```

Eine ganz andere Domäne: Verarbeitung arithmetischer Ausdrücke

- Alternativ zum Beispiel auch direkte Berechnung des Ergebnisses möglich:

```
eval  (LIT n)         =  n
eval  (ADD e1 e2)  =  eval e1  +  eval e2
eval  (MUL e1 e2)  =  eval e1  *  eval e2
```

- Dann zum Beispiel:

```
> eval  (parse expr "2*3+5")
11
```

- Oder sogar Auswertung direkt beim Parsen:

```
expr   =  ( (+)  <$>  term  <*  char '+'  <*>  expr )  |||  term

term   =  ( (*)  <$>  factor  <*  char '*'  <*>  term )  |||  factor

factor  =  nat  |||  ( char '('  *>  expr  <*  char ')' )
```

- Dann nämlich:

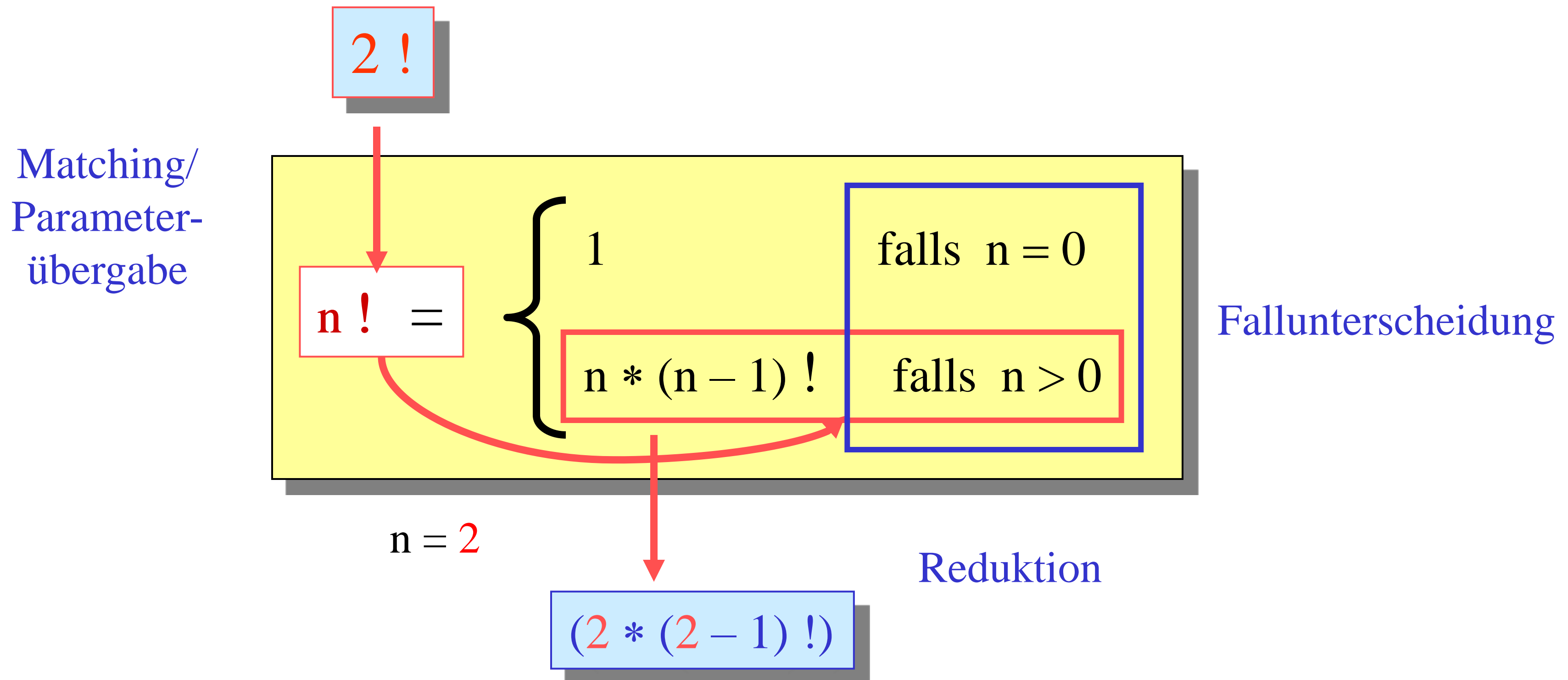
```
> parse expr "2*3+5"
11
```

Deskriptive Programmierung

Haskell-Grundlagen/Syntax

Prinzip des funktionalen Programmierens

Spezifikationen: Funktionsdefinitionen
Operationalisierung: Auswertung von Ausdrücken (syntaktische Reduktion)



Prinzip des funktionalen Programmierens

„Let the symbols do the work.“
Leibniz/Dijkstra

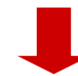
Spezifikation („Programm“) ≡ Funktionsdefinition(en)

$$n\,! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)\,! & \text{falls } n > 0 \end{cases}$$

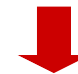
vordefinierte Operatoren

$2\,!$
$\Rightarrow (2 * (2 - 1)\,!)$
$\Rightarrow (2 * 1!\,)$
$\Rightarrow (2 * (1 * (1 - 1)\,!))$
$\Rightarrow (2 * (1 * 0\,!))$
$\Rightarrow (2 * (1 * 1))$
$\Rightarrow (2 * 1)$
$\Rightarrow 2$

Eingabe: auszuwertender Term/Ausdruck



(wiederholte) Funktionsanwendung



Ausgabe: resultierender Funktionswert

GHCi als „Taschenrechner“

```
WinGHCi
File  Edit  Actions  Tools  Help

GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> 3*(4+5)
27
Prelude> 
```

Einfache Typen, Operatoren und Funktionen

- Int, Integer:
  - ganze Zahlen (–12, 0, 42, ...)
  - Operatoren: +, –, *, ^
  - Funktionen: div, mod, min, max, ...
  - Vergleiche: ==, /=, <, <=, >, >=
- Float, Double:
  - Gleitkommazahlen (–3.7, pi, ...)
  - Operatoren: +, –, *, /
  - Funktionen: sqrt, log, sin, min, max, ...
  - Vergleiche: …
- Bool:
  - Boolesche Werte (True, False)
  - Operatoren: &&, ||
  - Funktionen: not; Vergleiche: …
- Char:
  - einzelne Zeichen ('a', 'b', '\n', ...)
  - Funktionen: succ, pred; Vergleiche: …

Auswertung einfacher Ausdrücke

```
> 5+7
12
```

```
> div 17 3
5
```

nicht: div(17,3) dafür:

```
> 17 `div` 3
5
```

```
> pi/1.5
2.0943951023932
```

```
> min (sqrt 4.5) (1.5^3)
2.1213203435596
```

nicht: min(sqrt(4.5),1.5^3)

```
> 'a' <= 'c'
True
```

```
> if 12 < 3 || 17.5 /= sqrt 5 then 17 – 3 else 6
14
```

nie ohne else-Zweig!

Komplexere Typen, Ausdrücke und Werte

- Listen:
  - [Int] für [] oder [–12, 0, 42]
  - [Bool] für [] oder [False, True, False]
  - [[Int]] für [[3, 4], [], [6, –2]]
  - …
  - Operatoren: :, ++, !!
  - Funktionen: head, tail, last, null, …

```
> 3 : [–12, 0, 42]
[3, -12, 0, 42]
```

```
> [1.5, 3.7] ++ [4.5, 2.3]
[1.5, 3.7, 4.5, 2.3]
```

```
> [False, True, False] !! 1
True
```

- Zeichenketten:
  - String = [Char]
  - spezielle Notation: "" für [] und "abcd" für ['a', 'b', 'c', 'd']
- Tupel:
  - (Int, Int) für (3, 5) und (0, –4)
  - (Int, String, Bool) für (3, "abc", False)
  - ((Int, Int), Bool, [Int]) für ((0, –4), True, [1, 2, 3])
  - [(Bool, Int)] für [(False, 3), (True, –4), (True, 42)]
  - …
  - Funktionen: fst und snd auf Paaren

```
> (3 – 4, snd (head [('a', 17), ('c', 3)]))
(–1, 17)
```

Deklaration von Werten

- in Datei:

```
x = 7
y = 2 * x
z = (mod y (x + 2), tail [1 .. y])

a = b – c
b = fst z
c = head (snd z)

d = (a, e)
e = [fst d, f]
f = head e
```

All dies sind Deklarationen, keine wert-ändernden Zuweisungen!

```
x = x + 1
```

ergibt keinen Sinn!

- nach dem Laden:

```
> z
(5, [2,3,4,5,6,7,8,9,10,11,12,13,14])
```

```
> a
3
```

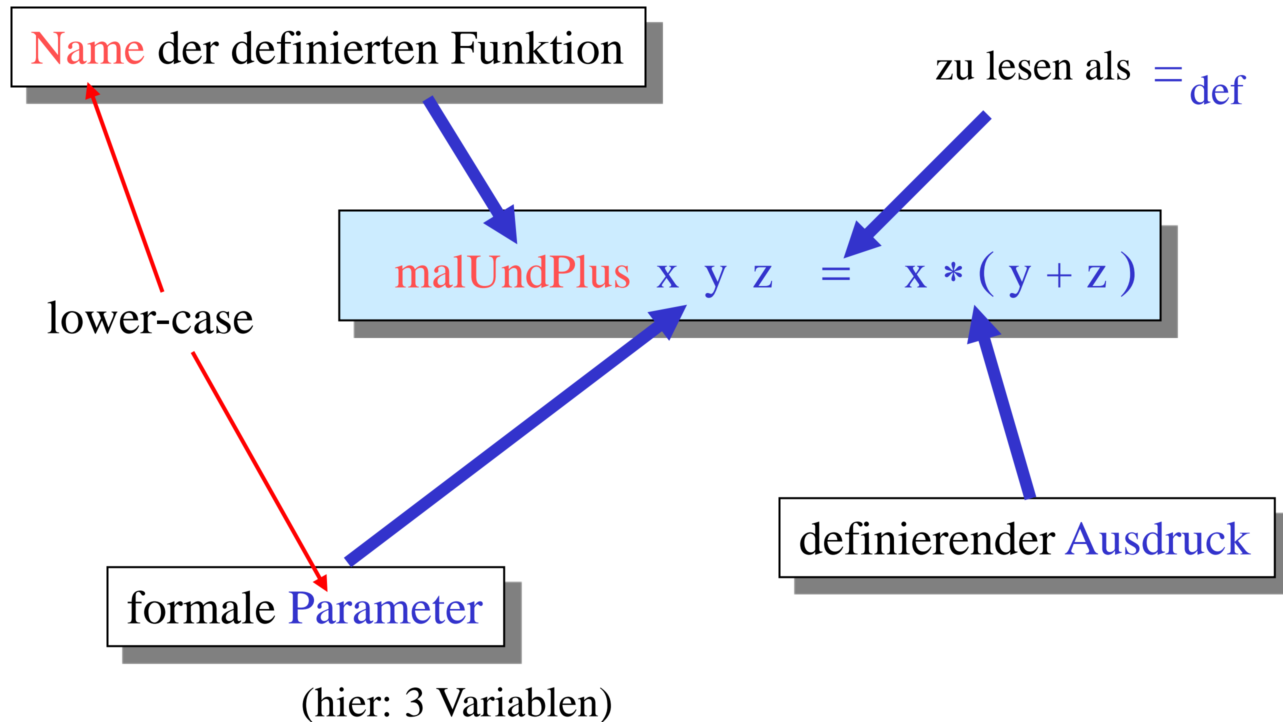
```
> d
(3, [3, 3])
```

Optionale Typangaben

```
x , y :: Int
x = 7
y = 2 * x

z :: (Int, [Int])
z = (mod y (x + 2), tail [1 .. y])

a , b , c :: Int
a = b – c
b = fst z
c = head (snd z)

d :: (Int, [Int])
d = (a, e)


...
```

Funktionsdefinitionen in Haskell

Prinzipieller Aufbau einer (sehr einfachen) Funktionsdefinition:



Deklaration von Funktionen (mit Typangaben)

zur Erinnerung: „if-then“ in Haskell immer mit explizitem „else“!

```
min3 :: Int → Int → Int → Int
min3 x y z  =  if x<y then (if x<z then x else z)
                      else (if y<z then y else z)
```

```
> min3 5 4 6
4
```

```
min3' :: (Int, Int, Int) → Int
min3' (x, y, z)  =  if x<y then (if x<z then x else z)
                           else (if y<z then y else z)
```

```
> min3' (5, 4, 6)
4
```

```
min3'' :: Int → Int → Int → Int
min3'' x y z  =  min (min x y) z
```

```
> min3'' 5 4 6
4
```

```
isEven :: Int → Bool
isEven n  =  (n `mod` 2) == 0
```

```
> isEven 12
True
```

Gleichheits<u>test</u>!

Beispiele Syntax für Funktionsapplikationen

Mathe-üblich	Haskell-üblich
f(x)	f x
f(x,y)	f x y
f(g(x))	f (g x)
f(x,g(y))	f x (g y)
f(x) + g(y)	f x + g y
f(a+b)	f (a + b)
f(a) + b	f a + b

Deskriptive Programmierung

Haskell-Auswertungssemantik

Bedarfs-Auswerten von (rekursiven) Funktionen

```
fac :: Int → Int
fac n  =  if n == 0 then 1 else n * fac (n – 1)
```

```
> fac 5
120
```

```
sumsquare :: Int → Int
sumsquare i  =  if i == 0 then 0 else i * i + sumsquare (i – 1)
```

```
> sumsquare 4
30
```

Berechnung durch schrittweises Auswerten:

```
> sumsquare 2
= if 2 == 0 then 0 else 2 * 2 + sumsquare (2 – 1)
= 2 * 2 + sumsquare (2 – 1)
= 4 + sumsquare (2 – 1)
= 4 + if (2 – 1) == 0 then 0 else …
= 4 + (1 * 1 + sumsquare (1 – 1))
= 4 + (1 + sumsquare (1 – 1))
= 4 + (1 + if (1 – 1) == 0 then 0 else …)
= 4 + (1 + 0)
= 5
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

```
> d
= (a, e)
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

```
> d
= (a, e)
= (3, e)
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
= (3, [fst (3, [fst d, f]), f])
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
= (3, [fst (3, [fst d, f]), f])
= (3, [3, f])
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
= (3, [fst (3, [fst d, f]), f])
= (3, [3, f])
= (3, [3, head e])
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
= (3, [fst (3, [fst d, f]), f])
= (3, [3, f])
= (3, [3, head e])
= (3, [3, head [3, head e]])
```

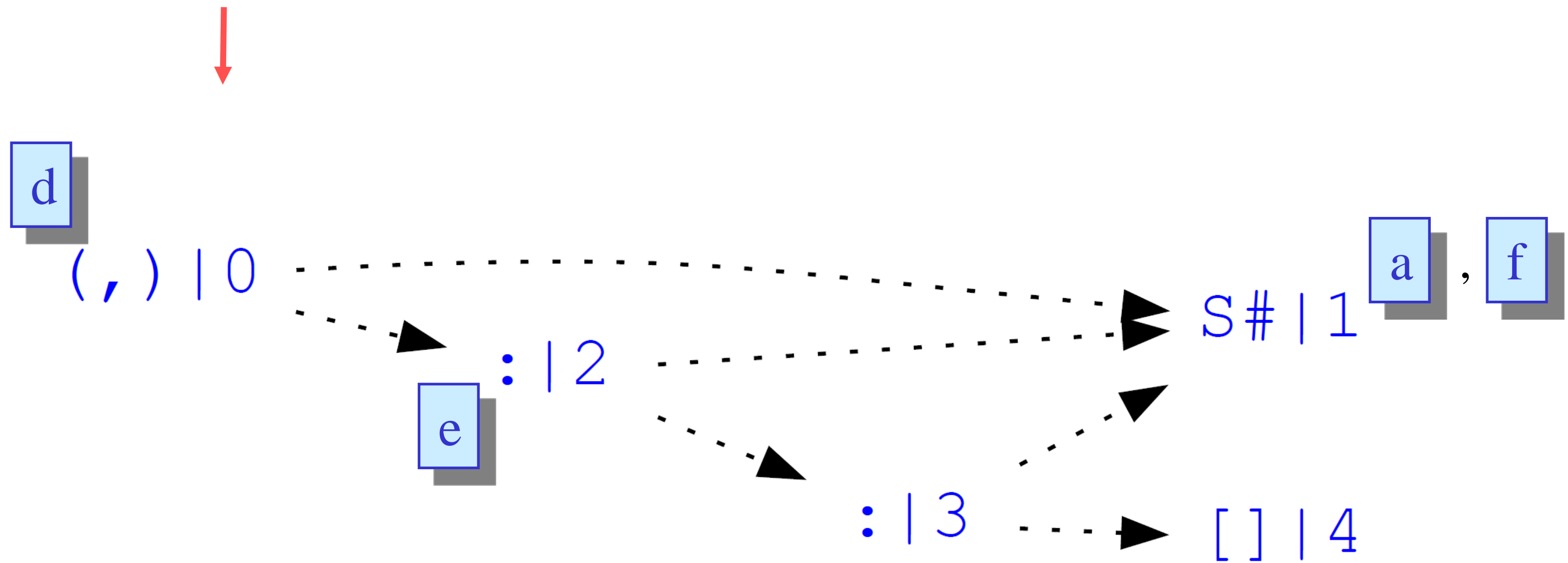
Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

```
a = 3
d = (a, e)
e = [fst d, f]
f = head e
```

```
> d
= (a, e)
= (3, e)
= (3, [fst d, f])
= (3, [fst (3, [fst d, f]), f])
= (3, [3, f])
= (3, [3, head e])
= (3, [3, head [3, head e]])
= (3, [3, 3])
```

Komplexere rekursive Abhängigkeiten: „Cleveres“ Auswerten

a = 3
d = (a, e)
e = [fst d, f]
f = head e



Deskriptive Programmierung

Mehr zur Syntax von Funktionsdefinitionen

Definierende Gleichungen: Grundregeln

- Auf der linken Seite einer definierenden Gleichung in Haskell dürfen
  - keine noch auszuwertenden Ausdrücke, sondern …
  - nur Variablen und Konstanten (sowie Pattern, siehe später …)

  vorkommen:

```
f x (2 * y) = x * y
```

unzulässig!

```
f x 1 = x * 2
```

okay

- Auf der rechten Seite einer definierenden Gleichung dürfen
  - beliebige Ausdrücke, (natürlich) auch auszuwertende, aber . . .
  - nur Variablen von der linken Seite (also keine „frischen“ Variablen)

  vorkommen:

```
f x = x * y
```

unzulässig!

```
f x 1 = x * 2
```

okay

Definierende Gleichungen: Grundregeln

- In der Liste von formalen Parametern einer Funktionsdefinition darf jede Variable nur genau einmal vorkommen:

```
f n 0 n = n^2
```

unzulässig!

stattdessen:

```
f n 0 m | n == m = n^2
```

Funktionsdefinitionen: Fallunterscheidung (1)

Komplexere Funktionsdefinitionen sind aus mehreren Alternativen zusammengesetzt. Jede der Alternativen definiert einen Fall der Funktion:

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)! & \text{falls } n > 0 \end{cases}$$

In Haskell, schon gesehen:

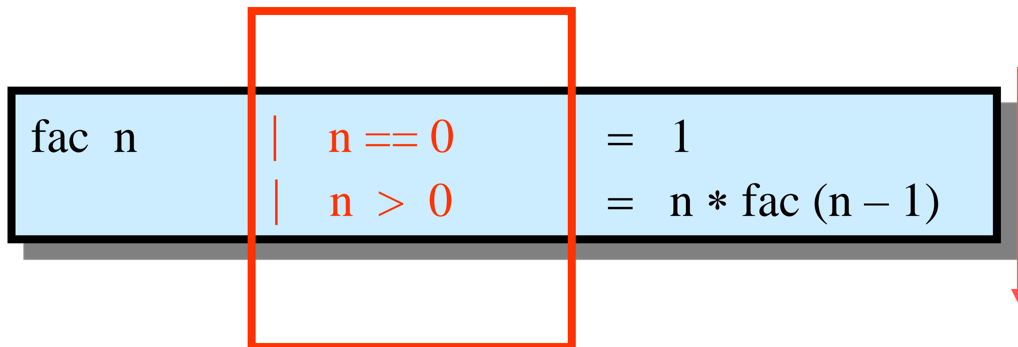
```
fac n  =  if n == 0 then 1 else n * fac (n – 1)
```

In Haskell kann auch der obere Stil imitiert werden, allerdings stehen die Bedingungen vor dem Gleichheitszeichen:

```
fac n   | n == 0   =  1
        | n >  0   =  n * fac (n – 1)
```

Funktionsdefinitionen: Fallunterscheidung (2)

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)\,! & \text{falls } n > 0 \end{cases}$$

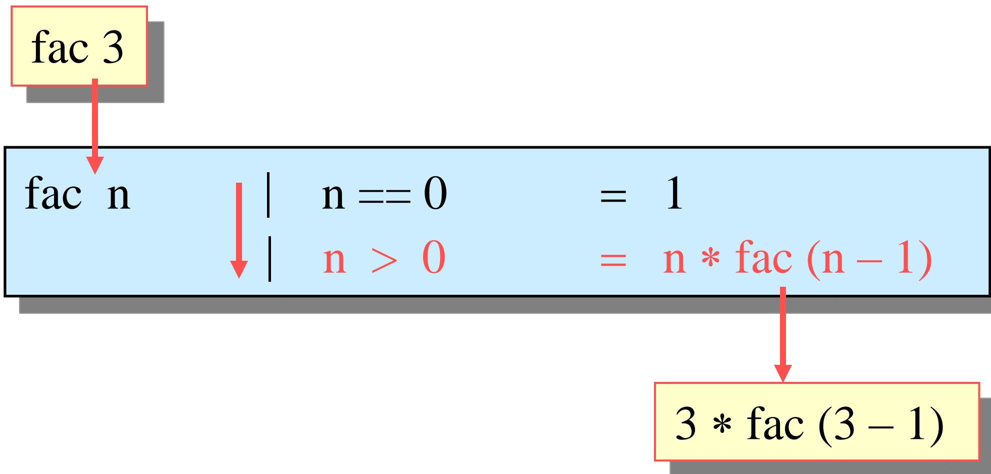


„Wächter“
(engl.: „guards“)

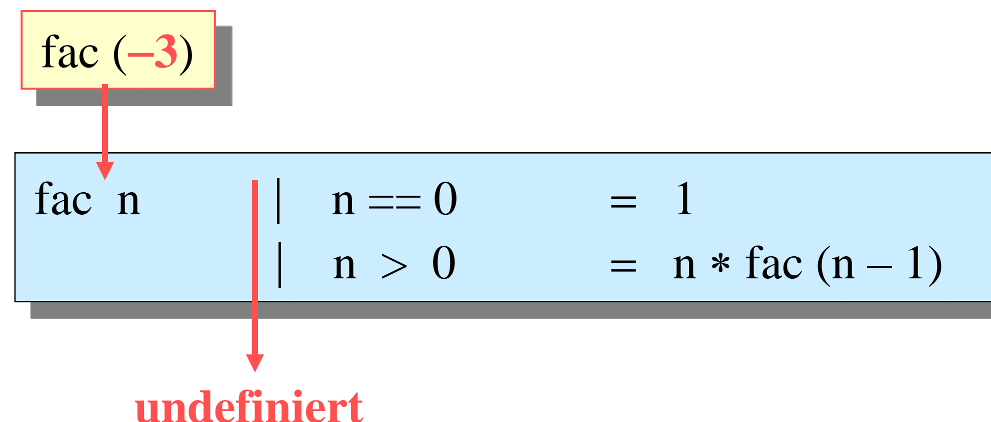
Boolesche Ausdrücke

Wie in der mathematischen Notation werden die „Wächter“ beim Auswerten von oben nach unten durchlaufen, bis zum ersten Mal eine Bedingung erfüllt ist.

Dieser Fall wird dann zum Reduzieren herangezogen.

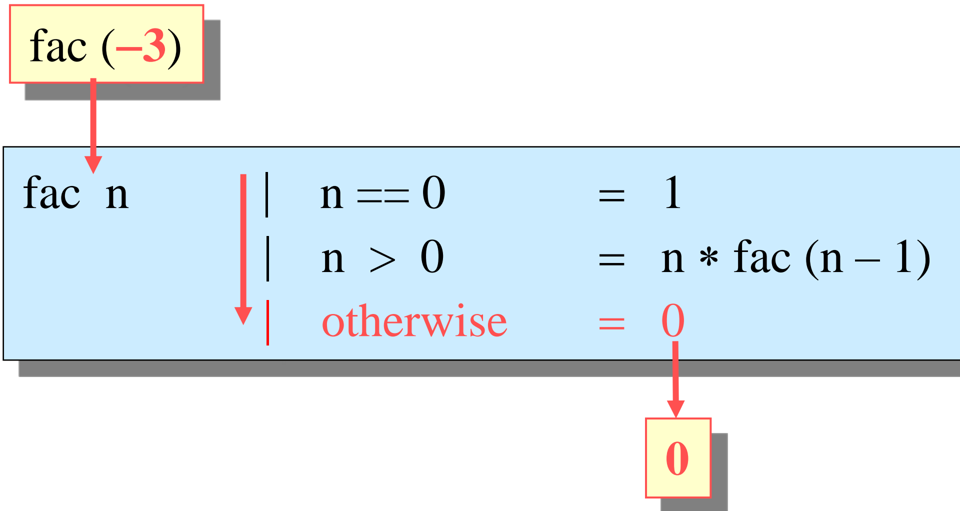


Die Fakultätsfunktion ist nur partiell definiert: für negativen Inputparameter wird kein „passender“ Fall gefunden, so dass das Resultat undefiniert ist.

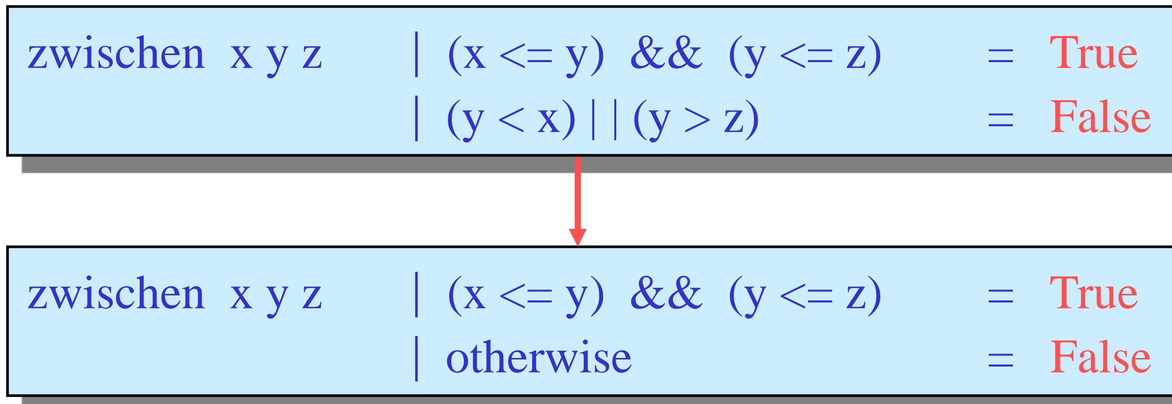


Funktionsdefinitionen: Fallunterscheidung (4)

Überführung in eine total definierte Funktion durch Anfügen eines „catch all"-Falls mit der Pseudo-Bedingung otherwise:



Mitunter hilfreich auch zur Abkürzung:



Funktionsdefinitionen: Fallunterscheidung (5)

Variationen:

```
fac n     | n == 0  =  1
          | n > 0   =  n * fac (n – 1)
```

ist „eigentlich“ nur eine Abkürzung für die Notation

```
fac n     | n == 0  =  1
fac n     | n > 0   =  n * fac (n – 1)
```

Noch eine Notationsvariante, in der die erste Bedingung durch einen Konstantenparameter ausgedrückt wird:

```
fac 0               =  1
fac n     | n > 0   =  n * fac (n – 1)
```

Funktionsdefinitionen: Fallunterscheidung (6)

- offenbar wichtige Grundtechnik:
  Auswahl eines „passenden“ Definitionsfalls für eine auszuwertende Funktionsapplikation

- zwei Auswahlkriterien (in dieser Reihenfolge!):
  - „pattern matching“: dt. ≈ Mustervergleich
  - Auswertung der „Wächterbedingung“

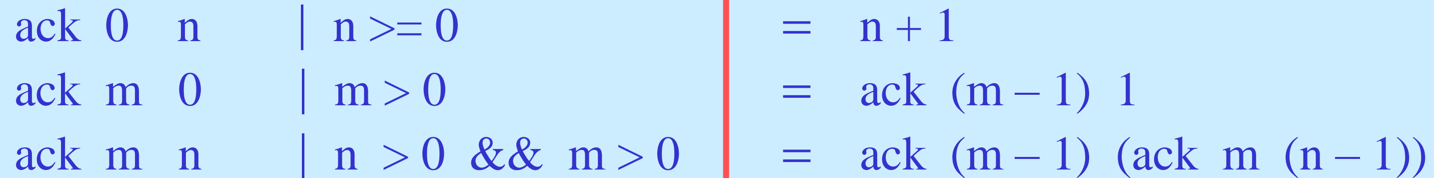
```
(1)   ack 0 n  | n >= 0          = n + 1
(2)   ack m 0  | m > 0           = ack (m – 1) 1
(3)   ack m n  | n > 0 && m > 0  = ack (m – 1) (ack m (n – 1))
```

Ackermann-Funktion

ack 0 0	passt zu (1)
ack 2 0	passt zu (2)
ack 2 1	passt zu (3)

Reihenfolge bei der Abarbeitung von Fällen in Funktionsdefinitionen

- Bei der Auswertung der Applikation ack 0 0 würden alle drei linken Seiten „matchen“!



- Der definierende Fall ist der (von oben nach unten durchlaufen) erste matchende Fall, dessen Wächter erfüllt ist.

- Auf diese Weise ist sichergestellt, dass es immer einen eindeutigen Funktionswert gibt. (… wenn es überhaupt einen gibt!)

- Bei der Ackermann-Funktion liefert jede Reihenfolge der drei Gleichungen dieselbe Funktion. Das ist aber nicht immer so! fac 0 verhält sich hier verschieden:

```
fac 0  =  1
fac n  =  n * fac (n – 1)
```

} 1

```
fac n  =  n * fac (n – 1)
fac 0  =  1
```

} undefiniert

Deskriptive Programmierung

Pattern Matching

Pattern Matching: Prinzip

konkrete Applikation

```
> ack 0 (ack 2 1)
```

pattern matching

linke Seite einer Definition

```
ack 0 n | ... = ...
```

Regeln des Pattern Matching:

- Voraussetzung: identischer Funktionsname
- Konstanten „matchen“
  - sich selbst (z.B.: 1 ↔ 1)
  - jede Variable (z.B.: 1 ↔ n)
- komplexe Ausdrücke matchen
  - jede Variable (z.B.: (fib 3) ↔ x)
  - diejenige Konstante, die ihren Funktionswert bezeichnet (z.B.: (fib 4) ↔ 5)
- Tupel matchen
  - jede Variable, und Tupel gleicher Länge bei komponentenweisem Match (z.B.: (1, False, fib 4) ↔ (1, x, 5))
- …

erzwingt Auswertung!

Auswirkung von Pattern-Matching-„Strategien“

- Beispiele auf Booleschen Werten:

```
not False = True
not True  = False
```

```
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

- etwas kompakter:

```
not False = True
not _     = False
```

```
True && True = True
_    && _    = False
```

anonyme Variablen

- aber effizienter? ja, für manche Eingaben sehr drastisch!

```
False && (ack 4 2 > 0)
```

Auswirkung von Pattern-Matching-„Strategien"

- Beispiele auf Booleschen Werten:

```
not False = True
not True  = False
```

```
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

- etwas kompakter:

```
not False = True
not _     = False
```

```
True && True = True
_    && _    = False
```

- aber effizienter? ja, für manche Eingaben!

andere Variante: ⟶

```
b && True = b
_ && _    = False
```

Matching von links nach rechts!

- nicht möglich:

```
b && b = b
_ && _ = False
```

Alternative Syntax (und Scoping!)

- Sogenannte case-Ausdrücke, zum Beispiel:

```
ifThenElse i t e  =  case i of
                         True  → t
                         False → e
```

- Oder, zum Beispiel:

```
f x y  =  case (x + y, x – y) of
             (z, _)  | z > 0 → y
             (0, x)          → x + y
```

- Was ergibt sich dann wohl aus folgendem Aufruf dieser Funktion?

```
> f  10 (–10)
```

Deskriptive Programmierung

Elementares Arbeiten mit Listen

Damit Pattern Matching so richtig interessant wird: Verarbeitung von Listen

- Haskell-Liste: Folge von Elementen gleichen Typs (homogene Struktur)

- Syntax: Listenelemente in eckige Klammern eingeschlossen.

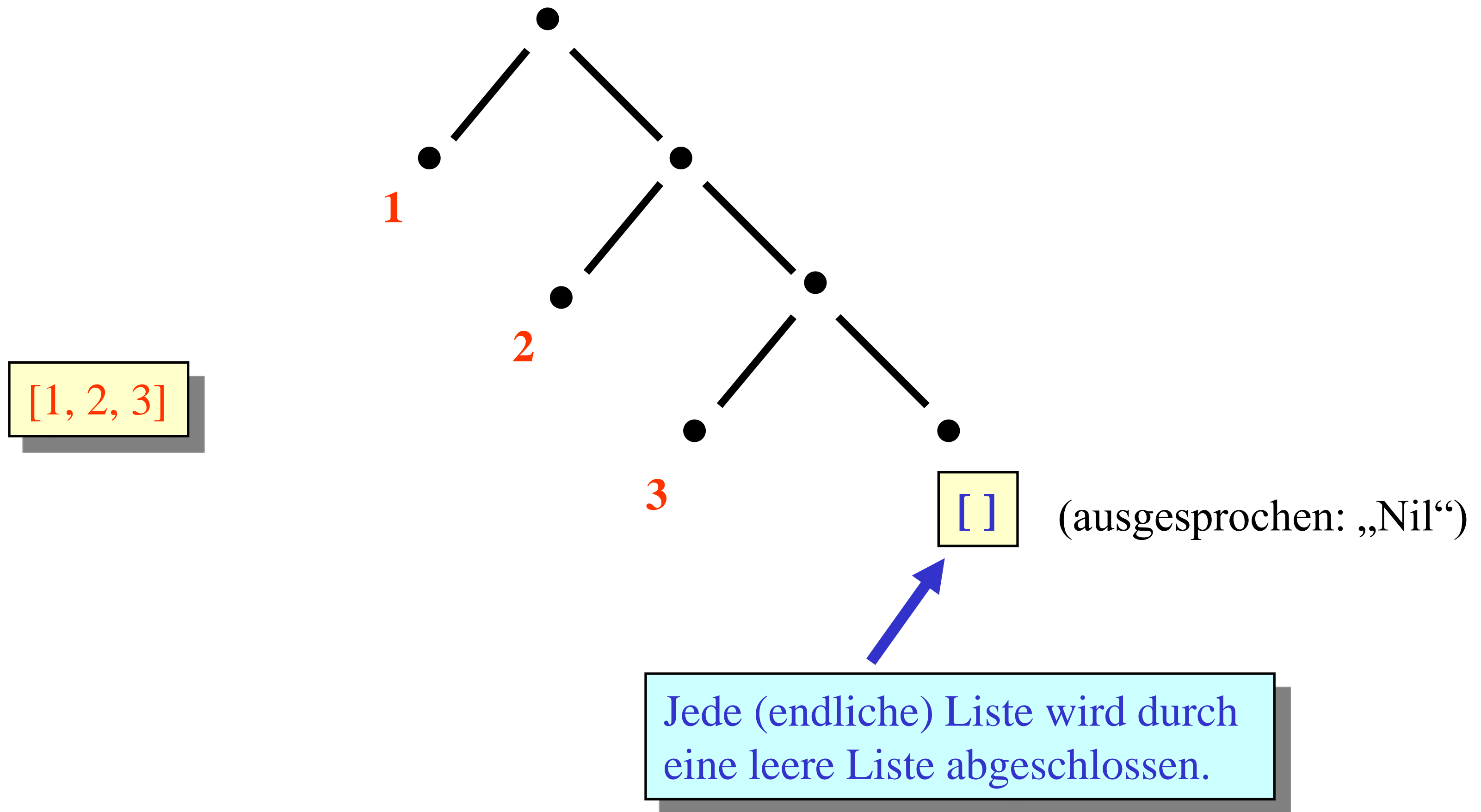
[1, 2, 3]	Liste von ganzen Zahlen (Typ: Int)
['a', 'b', 'c']	Liste von Buchstaben (Typ: Char)
[ ]	leere Liste (beliebigen Typs)
[[1,2], [ ], [2]]	Liste von Int-Listen

[[1,2], 'a', 3]	<u>keine</u> gültige Liste (verschiedene Elementtypen)

- Im Gegensatz zu dem, was viele Beispiele in der Vorlesung suggerieren mögen, sind Listen in der Praxis oft <u>nicht</u> die Datenstruktur, die man verwenden sollte! (Stattdessen nutzerdefinierte Datentypen, oder Typen aus Bibliotheken wie Data.ByteString, Data.Array, Data.Map, …)

Baumdarstellung von Listen

Listen werden intern als bestimmte Binärbäume dargestellt, deren Blätter mit den einzelnen Listenelementen markiert sind:

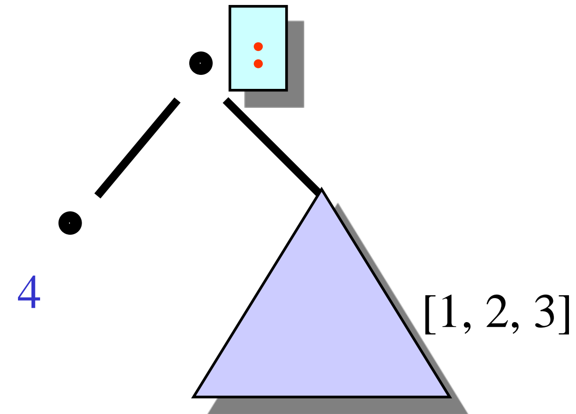


Listenkonstruktor

- Elementarer Konstruktor („Operator“ zum Konstruieren) für Listen:

- Der Konstruktor „:“ dient zum Erweitern einer gegebenen Liste um ein Element, das am Listenkopf eingefügt wird:

```
> 4 : [1, 2, 3]
[4, 1, 2, 3]
```

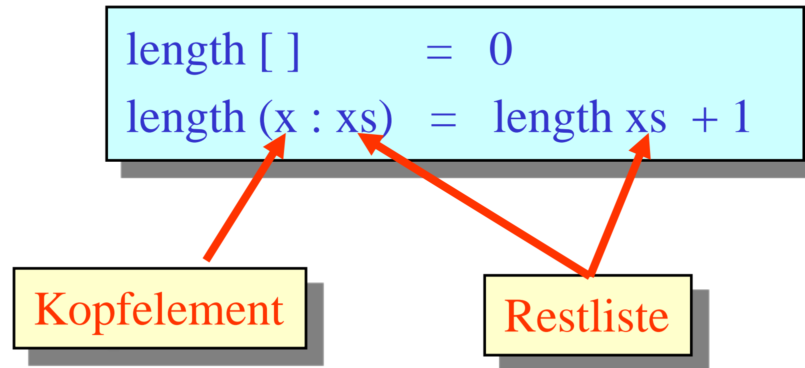


- Alternative Notation für Listen (analog zur Baumdarstellung):

```
4 : 1 : 2 : 3 : [ ]
```

Länge einer Liste

- Funktion zur Bestimmung der Länge einer Liste (vordefiniert):

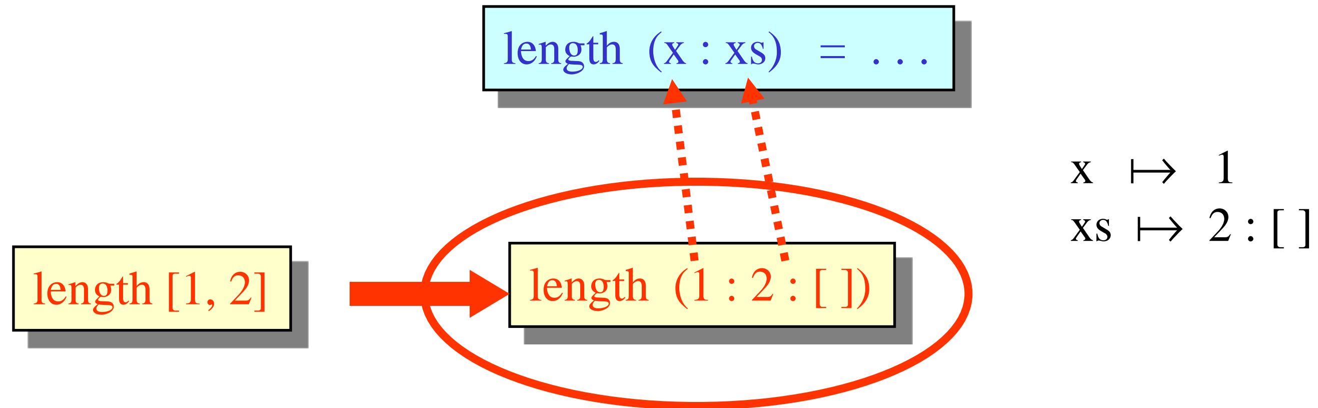


- Beispiel für die Anwendung von length:

```
> length [1, 2]
= length [2]  + 1
= (length [ ]  + 1)  + 1
= (      0         + 1)  + 1
= 1 + 1
= 2
```

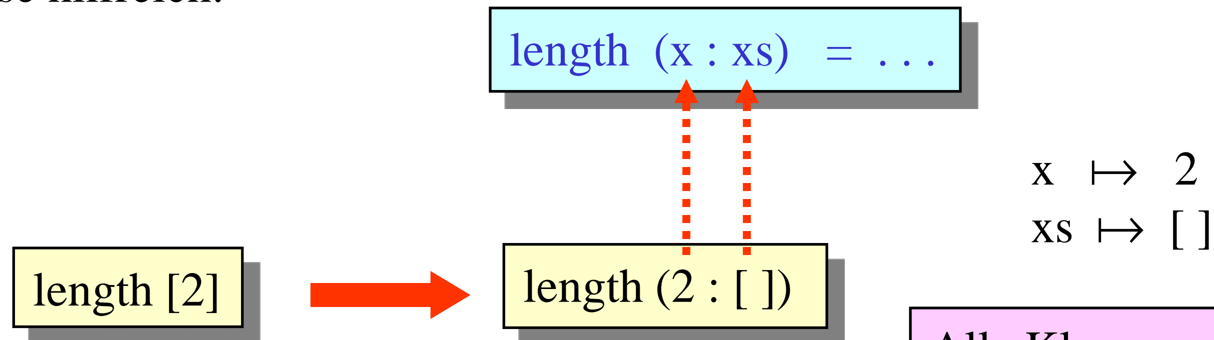
Pattern Matching mit Listenkonstruktoren

- Pattern Matching zwischen Listen und Konstruktorausdrücken ist nur zu verstehen, wenn man beide Ausdrücke in Konstruktorform sieht:



x ↦ 1
xs ↦ 2 : [ ]

- Auch beim „rekursiven Abbauen“ von einelementigen Listen ist diese Sichtweise hilfreich:



x ↦ 2
xs ↦ [ ]

Alle Klammern auf dieser Folie zwingend!

Konkatenation von Listen

- wichtige Grundoperation für alle Listen: Konkatenieren zweier Listen
  (= Aneinanderhängen)

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

- Beispielanwendung:

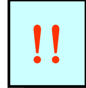
```
>  concatenation [1, 2]  [3, 4]
[1, 2, 3, 4]
```

- Als Infixoperator vordefiniert:

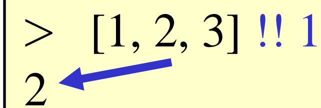
```
> [1, 2] ++ [3, 4]
[1, 2, 3, 4]
```

Zugriff auf einzelne Listenelemente und Teillisten

- gezielter Zugriff auf einzelne Elemente einer Liste durch weiteren vordefinierten Infixoperator:



- Zählung der Listenelemente beginnt mit 0 !



- Zugriff per (x : xs)-Pattern natürlich nur auf nichtleere Listen:

tail (x : xs) = xs

> tail [ ]
ERROR - Pattern match failure: tail []

head (x : xs) = x

> head [ ]
ERROR - Pattern match failure: head []

(Leider ist der Fehlerursprung in solchen Fällen nicht immer so einfach identifizierbar.)

Komplexeres Pattern Matching

```
f :: [Int] → [[Int]]
f [ ]               =  [ ]
f [x]               =  [[x]]
f (x : y : zs)      =  if x <= y then (x : s) : ts else [x] : s : ts
     where s : ts = f (y : zs)
```

lokale Definition + Match

Komplexeres Pattern Matching

```
f :: [Int] → [[Int]]
f [ ]               = [ ]
f [x]               = [[x]]
f (x : y : zs)      = if x <= y then (x : s) : ts else [x] : s : ts
    where s : ts = f (y : zs)
```

Berechnung durch schrittweise Auswertung:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts     where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = f (2 : [0])
```

Komplexeres Pattern Matching

```
f :: [Int] → [[Int]]
f [ ]                 =  [ ]
f [x]                 =  [[x]]
f (x : y : zs)        =  if x <= y then (x : s) : ts else [x] : s : ts
      where s : ts = f (y : zs)
```

Berechnung durch schrittweise Auswertung:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts       where s : ts = f (2 : [0])
= (1 : s) : ts                                        where s : ts = f (2 : [0])
= (1 : s) : ts                                        where s : ts = [2] : s' : ts'
                                                            where s' : ts' = f (0 : [ ])
```

Komplexeres Pattern Matching

```
f :: [Int] → [[Int]]
f [ ]               =  [ ]
f [x]               =  [[x]]
f (x : y : zs)      =  if x <= y then (x : s) : ts else [x] : s : ts
     where s : ts = f (y : zs)
```

Berechnung durch schrittweise Auswertung:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts     where s : ts = f (2 : [0])
= (1 : s) : ts                                      where s : ts = f (2 : [0])
= (1 : s) : ts                                      where s : ts = [2] : s' : ts'
                                                          where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                              where s' : ts' = f (0 : [ ])
```

Komplexeres Pattern Matching

```
f :: [Int] → [[Int]]
f [ ]              =  [ ]
f [x]              =  [[x]]
f (x : y : zs)     =  if x <= y then (x : s) : ts else [x] : s : ts
      where s : ts = f (y : zs)
```

Berechnung durch schrittweise Auswertung:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts     where s : ts = f (2 : [0])
= (1 : s) : ts                                      where s : ts = f (2 : [0])
= (1 : s) : ts                                      where s : ts = [2] : s' : ts'
                                                          where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                              where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                              where s' : ts' = [[0]]
```

Komplexeres Pattern Matching

```
f :: [Int] → [[Int]]
f [ ]              = [ ]
f [x]              = [[x]]
f (x : y : zs)     = if x <= y then (x : s) : ts else [x] : s : ts
    where s : ts = f (y : zs)
```

Berechnung durch schrittweise Auswertung:

```
> f [1, 2, 0]
= if 1 <= 2 then (1 : s) : ts else [1] : s : ts     where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = f (2 : [0])
= (1 : s) : ts                                       where s : ts = [2] : s' : ts'
                                                         where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                               where s' : ts' = f (0 : [ ])
= (1 : [2]) : s' : ts'                               where s' : ts' = [[0]]
= (1 : [2]) : [0] : [ ] = [[1, 2], [0]]
```

Komplexeres Pattern Matching

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]            =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

Variante für lokale Definition

Berechnung durch schrittweise Auswertung:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
```

Komplexeres Pattern Matching

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]             =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

Variante für lokale Definition

Berechnung durch schrittweise Auswertung:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
= let (xs', ys') = unzip [ ] in (1 : 3 : xs', 2 : 4 : ys')
```

Komplexeres Pattern Matching

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]            =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

Variante für lokale Definition

Berechnung durch schrittweise Auswertung:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
= let (xs', ys') = unzip [ ] in (1 : 3 : xs', 2 : 4 : ys')
= let (xs', ys') = ([ ], [ ]) in (1 : 3 : xs', 2 : 4 : ys')
```

```
unzip :: [(Int, Int)] → ([Int], [Int])
unzip [ ]             =  ([ ], [ ])
unzip ((x, y) : zs) =  let (xs, ys) = unzip zs in (x : xs, y : ys)
```

Variante für lokale Definition

Berechnung durch schrittweise Auswertung:

```
> unzip [(1, 2), (3, 4)]
= let (xs, ys) = unzip [(3, 4)] in (1 : xs, 2 : ys)
= let (xs, ys) = (let (xs', ys') = unzip [ ] in (3 : xs', 4 : ys')) in (1 : xs, 2 : ys)
= let (xs', ys') = unzip [ ] in (1 : 3 : xs', 2 : 4 : ys')
= let (xs', ys') = ([ ], [ ]) in (1 : 3 : xs', 2 : 4 : ys')
= ([1, 3], [2, 4])
```

Zwischendurch bemerkt: Layout in Haskell

```
let  y = a * b
     f x = (x + y) / y
in  f c + f d
```

implizites Layout
(„offside rule“)

```
let  { y = a * b; f x = (x + y) / y }
in  f c + f d
```

äquivalent, explizites Layout

```
let  y = a * b
              f x = (x + y) / y
in  f c + f d
```

nicht äquivalent,
inkorrekt

```
let   y = a * b
  f x = (x + y) / y
in  f c + f d
```

(analog für andere Sprachkonstrukte, z.B. where, case)

Pattern Matching über mehreren Argumenten (und „veraltete“ (n + k)-Pattern)

```
drop :: Int → [Int] → [Int]
drop 0       xs       = xs
drop n       [ ]      = [ ]
drop (n + 1) (x : xs) = drop n xs
```

in Haskell 98 erlaubt, in Haskell 2010 nicht mehr!

```
> drop 0 [1, 2, 3]
[1, 2, 3]
```

```
> drop 5 [1, 2, 3]
[ ]
```

```
> drop 3 [1, 2, 3, 4, 5]
[4, 5]
```

Reihenfolge beim Pattern Matching

- Nochmal als Warnung,

```
zip :: [Int] → [Int] → [(Int, Int)]
zip (x : xs) (y : ys)   =  (x, y) : zip xs ys
zip xs       ys         =  [ ]
```

ist okay:

```
> zip [1 .. 3] [10 .. 15]
[(1, 10), (2, 11), (3, 12)]
```

- aber

```
zip :: [Int] → [Int] → [(Int, Int)]
zip xs       ys         =  [ ]
zip (x : xs) (y : ys)   =  (x, y) : zip xs ys
```

ist problematisch:

```
> zip [1 .. 3] [10 .. 15]
[ ]
```

Deskriptive Programmierung

List Comprehensions

Arithmetische Sequenzen

- nützliche Notationsform für Listen von Zahlen:

arithmetische Sequenzen

- abkürzende Schreibweise für Listen von Zahlen mit identischer Schrittweite:

```
>   [ 1 .. 10 ]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Höhere Schrittweite als 1 wird durch Angabe eines zweiten Elements festgelegt:

```
> [ 1, 3 .. 10 ]
[1, 3, 5, 7, 9]
```

- alternative Definition der Fakultätsfunktion (ohne explizite Rekursion):

```
fac n  =  prod  [ 1 .. n ]
```

List comprehensions (1)

- mächtiges und elegantes Sprachkonzept in Haskell:

list comprehension

engl. von „comprehensive“: umfassend

- Vorbild: implizite Mengennotation der Mathematik (Menge aller x, so dass ...), z.B.

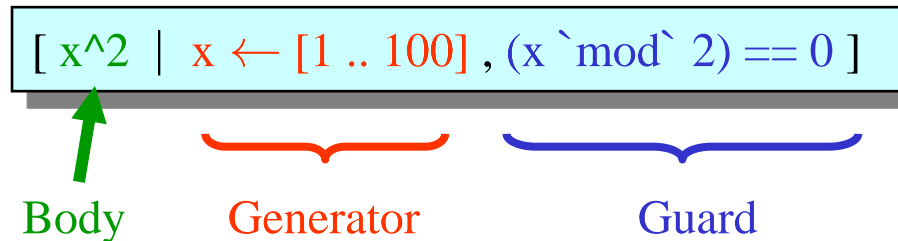
$$\{\ x^2 \mid x \in \{1, \ldots, 100\} \land (x \bmod 2) = 0\ \}$$

- in Haskell analoges Konzept für Listen:

```
[ x^2 | x ← [1 .. 100] , (x `mod` 2) == 0 ]
```

List comprehensions (2)

- Eine *list comprehension* besteht im Prinzip aus drei „Zutaten“:



- Der Body für die Listenelemente ist ein Ausdruck, der in der Regel mindestens eine Variable enthält, deren mögliche Werte durch den Generator erzeugt werden.

- Der Generator ist ein Ausdruck der Form Variable ← Liste, der die Variable sukzessive an alle Elemente der Liste bindet (in der Listenreihenfolge).

- Der Guard ist ein Boolescher Ausdruck, der die generierten Werte auf diejenigen beschränkt, für die der Ausdruck den Wert True liefert.

- Zusätzlich möglich: lokale Definitionen mittels let.

List comprehensions (3)

- Teile sind optional, z.B.:

```
[ x^2 | x ← [1 .. 10] ]
```

- Eine list comprehension darf auch mehrere Variablen mit mehreren Generatoren enthalten, z.B.:

```
>  [ (x, y) | x ← [ 1, 2, 3 ], y ← [1 .. x] ]
[ (1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3) ]
```

- Jede (nicht aus Kontext bekannte) Variable braucht einen Generator:

```
[ (x * y) | x ← [ 1, 2, 3 ], y ← [ 1, 2, 3 ] ]
```

aber auch

```
[ x ++ y | (x, y) ← [ ("a", "b"), ("c", "d") ] ]
```

List comprehensions (4)

- Reihenfolge der Generatoren beeinflusst Ausgabereihenfolge:

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

vs.

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

(wie verschachtelte Schleifen)

List comprehensions (5)

- „Spätere“ Generatoren können von „früheren“ abhängen, z.B.:

```
> [ (x, y) | x ← [ 1, 2, 3 ],  y ← [1 .. x] ]
[ (1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3) ]
```

- Insbesondere kann eine per Generator gebundene Variable selbst als „Erzeugerliste“ dienen:

```
concat :: [[Int]] → [Int]
concat xss  =  [ x | xs ← xss, x ← xs ]
```

```
> concat [ [ 1, 2, 3 ], [ 4, 5 ], [ 6 ], [ ] ]
[ 1, 2, 3, 4, 5, 6 ]
```

List comprehensions (6)

- Guards können (auch) nur von früheren Generatoren abhängen, z.B.:

```
> [ x | x ← [1 .. 10], even x ]
[ 2, 4, 6, 8, 10 ]
```

- Noch ein Beispiel:

```
factors :: Int → [Int]
factors n = [ x | x ← [1 .. n], n `mod` x == 0 ]
```

```
> factors 15
[ 1, 3, 5, 15 ]
```

List comprehensions (7)

- Kann man folgende Funktion mit list comprehensions realisieren?

$$
\begin{array}{lcl}
\text{zip} :: [\text{Int}] \to [\text{Int}] \to [(\text{Int}, \text{Int})] & & \\
\text{zip}\ (x : xs)\ (y : ys) & = & (x, y) : \text{zip}\ xs\ ys \\
\text{zip}\ xs\ \quad ys & = & [\ ]
\end{array}
$$

> zip [1 .. 3] [10 .. 15]
[(1, 10), (2, 11), (3, 12)]

- Nur mit „parallel list comprehensions“:

$$
\begin{array}{l}
\text{zip} :: [\text{Int}] \to [\text{Int}] \to [(\text{Int}, \text{Int})] \\
\text{zip}\ xs\ ys\ = [\ (x, y)\ \mid\ x \leftarrow xs\ \mid\ y \leftarrow ys\ ]
\end{array}
$$

weder in Haskell 98,
noch in Haskell 2010,
aber als GHC extension

Unendliche Listen

- Es gibt in Haskell sogar abkürzende Notationen für unendliche Listen.

```
[ 1, 3 .. ]
```

steht für [1, 3, 5, 7, 9, ...]

- Zum Beispiel:

```
naturals,evens,odds :: [Integer]
naturals   =   [ 1 .. ]
evens      =   [ 2, 4 .. ]
odds       =   [ 1, 3 .. ]
```

- Damit lassen sich unendliche Folgen als Listen darstellen, z.B.:

```
squares   =   [  n^2  |  n ← naturals ]
facs      =   [ fac n  |  n ← naturals ]
primes    =   2 : [ n  |  n ← odds, factors n == [1, n] ]
```

„Endliches Arbeiten“ mit unendlichen Listen

- Eingabe eines Ausdrucks, der eine unendliche Liste bezeichnet, führt zu einer nicht-terminierenden Ausgabe (muss „per Hand“ unterbrochen werden!)

- Praktikabel ist aber das Arbeiten mit endlichen Teillisten von unendlichen Listen, z.B.:

```
> take 5 primes
[2, 3, 5, 7, 11]
```

```
> primes !! 5
13
```

- Dass so etwas möglich ist, ist nicht selbstverständlich, sondern liegt an Haskells spezieller Bedarfsauswertungs-Strategie, die den Wert eines Ausdrucks nur dann berechnet, wenn er unbedingt nötig ist („lazy evaluation“).

- Folgender Ausdruck bezeichnet zwar „intuitiv“ eine endliche Liste, die Berechnung terminiert aber nicht:

Warum ?

```
> [ x | x ← squares, x < 100 ]
[1, 4, 9, 16, 25, 36, 49, 64, 81,
```

Varianten zur Primzahlgenerierung

- Statt:

```
odds       =  [ 1, 3 .. ]
factors n  =  [ x | x ← [1 .. n], n `mod` x == 0 ]
primes     =  2 : [ n | n ← odds, factors n == [ 1, n ] ]
```

- Zum Beispiel:

```
primes     =  2 : [ n | n ← [ 3, 5 .. ], isPrime n ]
isPrime n  =  and [ n `mod` t > 0 | t ← candidates primes ]
  where  candidates (p : ps) | p * p > n   = [ ]
                             | otherwise   = p : candidates ps
```

- Oder auch:

```
primes          =  sieve [ 2 .. ]
sieve (p : xs)  =  p : sieve [ x | x ← xs, x `mod` p > 0 ]
```

Deskriptive Programmierung

Die Rolle (bzw. Arten) von Rekursion

Pattern Matching + Rekursion vs. List Comprehensions

Wir hatten gesehen:

```
sumsquare :: Int → Int
sumsquare i  =  if i == 0 then 0 else i * i + sumsquare (i – 1)
```

```
> sumsquare 4
30
```

Aber auch möglich:

```
sumsquare :: Int → Int
sumsquare n  =  sum [ i * i  |  i ← [0 .. n] ]
```

```
> sumsquare 4
30
```

Welche Form ist nun „besser“?

Lässt sich nicht so ohne Weiteres beantworten. Was wären Kriterien?
Vielleicht:

- Effizienz
- Lesbarkeit
- „Beweisbarkeit“

Fakt: Auch sum, [0 .. n], … sind letztlich selbst rekursiv definiert.

Die Rolle von Rekursion, bzw. zwei Arten von Rekursion

Strukturelle Rekursion:

```
sum :: [Int] → Int
sum [ ]        =  0
sum (x : xs)  =  x + sum xs
```

Letztlich auch „strukturell" oder zumindest einfach induktiv:

```
sumsquare :: Int → Int
sumsquare i  =  if i == 0 then 0 else i * i + sumsquare (i – 1)
```

Allgemeine Rekursion:

```
quersumme :: Int → Int
quersumme n  | n < 10       =  n
             | otherwise    =  let (d, m) = n `divMod` 10  in  m + quersumme d
```

Auch: ack, …, Quicksort, …

Vergleich allgemeine und strukturelle Rekursion

- Allgemeine Rekursion ist deutlich flexibler!
  - algorithmische Prinzipien wie „Divide and Conquer“ direkt umsetzbar
  - manche Funktionen sind beweisbar nicht durch strukturelle Rekursion realisierbar

- Strukturelle Rekursion:
  - liefert ein sehr nützliches „Rezept“ zur Definition von Funktionen
  - garantiert Termination (auf endlichen Strukturen)
  - ermöglicht sehr direkt Beweise per Induktion
  - lässt sich als wiederverwendbares Programmschema „verpacken“

Deskriptive Programmierung

kurz Allgemeines zu Typen in Haskell

Typen

- Wichtiges Grundkonzept von Haskell, bisher eher beiläufig betrachtet:

  Jeder Ausdruck und jede Funktion hat einen Typ.

- Notation für Typzuordnung: doppelter Doppelpunkt

  z.B.: `1 :: Int`

- Grundlage: vordefinierte Basistypen für Konstanten

  - diverse numerische Typen, z.B. Integer, Rational, Float, Double
  - Buchstaben: Char
  - Wahrheitswerte: Bool

- zusätzlich: diverse Typkonstruktoren für komplexe Typen

Typisierung, „type checking“, „type inference“

- Jeder Ausdruck hat genau einen Typ, der noch vor der Laufzeit bestimmbar ist:

  Haskell ist eine stark und statisch getypte Sprache.

- Funktionsdefinition und -anwendungen werden auf Typkonsistenz geprüft:

  Typprüfung (engl.: „type checking“)

- Haskell bietet darüber hinaus Typherleitung (engl.: „type inference“) , d.h., Typen müssen nicht unbedingt explizit angegeben werden.

- Es gibt kein (implizites oder explizites) Casting zwischen Typen.

Besonderheiten zur Typisierung von Zahlen

- Es wurden bereits verschiedene Zahlentypen erwähnt: Int, Integer, Float (und es gibt noch eine ganze Reihe weiterer, zum Beispiel Rational).

- Zahlenliterale können je nach Kontext verschiedenen Typ haben (zum Beispiel, 3 :: Int, 3 :: Integer, 3 :: Float, 3.0 :: Float, 3.5 :: Float, 3.5 :: Double).

- Für allgemeine Ausdrücke gibt es überladene Konversionsfunktionen, zum Beispiel:
  - fromIntegral :: Int → Integer, fromIntegral :: Integer → Int, fromIntegral :: Int → Rational, fromIntegral :: Integer → Float, …
  - truncate :: Float → Int, truncate :: Double → Int, truncate :: Float → Integer, …, round :: …, ceiling :: …, floor :: …

- Konversionen sind nicht nötig in zum Beispiel 3 + 4.5 oder in:

```
f x = 2 * x + 3.5
g y = f 4 / y
```

,

aber zum Beispiel in:

```
f :: Int → Float
f x = 2 * (fromIntegral x) + 3.5
```

oder in:

```
f x = 2 * x + 3.5
g y = f (fromIntegral (length "abcd")) / y
```

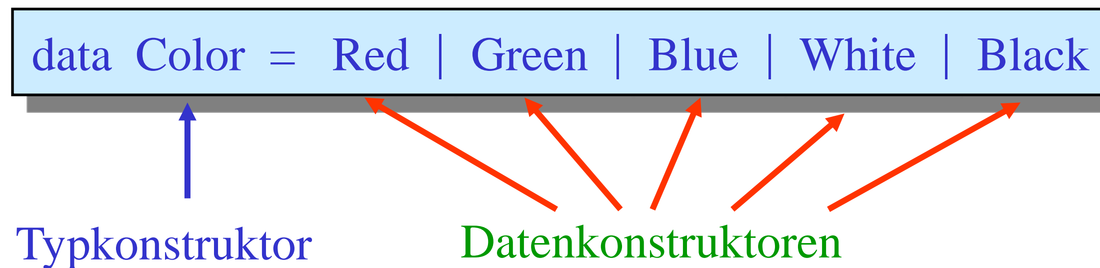
Deskriptive Programmierung

Algebraische Datentypen

Deklaration von (algebraischen) Datentypen

- Ein wesentlicher Aspekt typischer Haskell-Programme ist die Definition problemspezifischer Datentypen (statt alles aus Listen zu bauen o.ä.).

- Dazu dienen in erster Linie Datentypdeklarationen:



- Konstruktoren in Haskell (Daten- wie Typkonstruktoren) beginnen grundsätzlich mit Großbuchstaben (Ausnahme: bestimmte symbolische Formen wie bei Listen).

- Der hier neu definierte Typ Color ist ein Aufzählungstyp, der aus genau den fünf aufgeführten Elementen besteht.

Deklaration von (algebraischen) Datentypen

- Selbst deklarierte Datentypen:

```
data  Color  =  Red | Green | Blue | White | Black
```

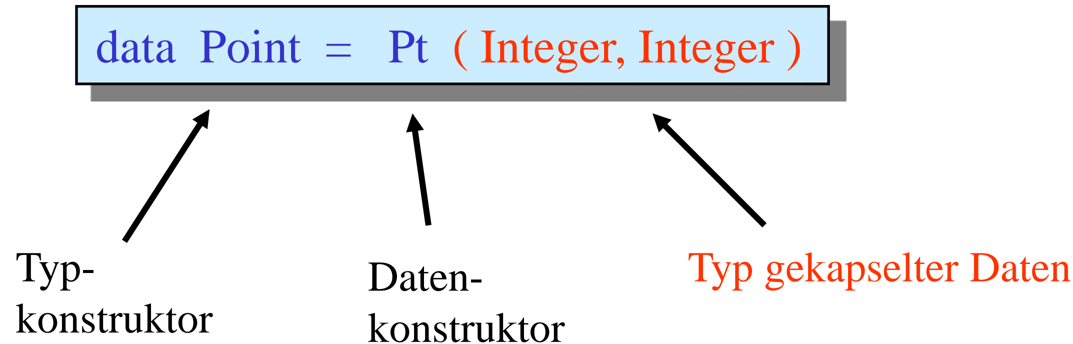
… können beliebig als Komponenten in anderen Typen auftreten, etwa [(Color, Int)] mit Werten z.B. [ ], [(Red, –5)] und [(Red, –5), (Blue, 2), (Red, 0)].

- Berechnung mittels Pattern Matching möglich:

```
primaryCol :: Color → Bool
primaryCol Red       =  True
primaryCol Green     =  True
primaryCol Blue      =  True
primaryCol _         =  False
```

Selbstdefinierte strukturierte Typen

- Man kann auch eigene strukturierte Typen deklarieren, indem man einen Datenkonstruktor mit Parametern einsetzt:



- Mit einem solchen selbstdefinierten Datenkonstruktor lassen sich dann strukturierte Werte eines selbstdefinierten Typs konstruieren:

Pt (1, 2) :: Point

- Es ist zulässig, für die Bezeichnung (irgend-)eines Typs denselben Konstruktor zu verwenden wie zur Konstruktion von Datenelementen (etwa hier beide Male Pt).

Selbstdefinierte strukturierte Typen

- Ein etwas komplexeres Beispiel:

```
data Date  = Date Int Int Int
data Time = Hour Int
data Connection  =  Train Date Time Time |
                    Flight String Date Time Time
```

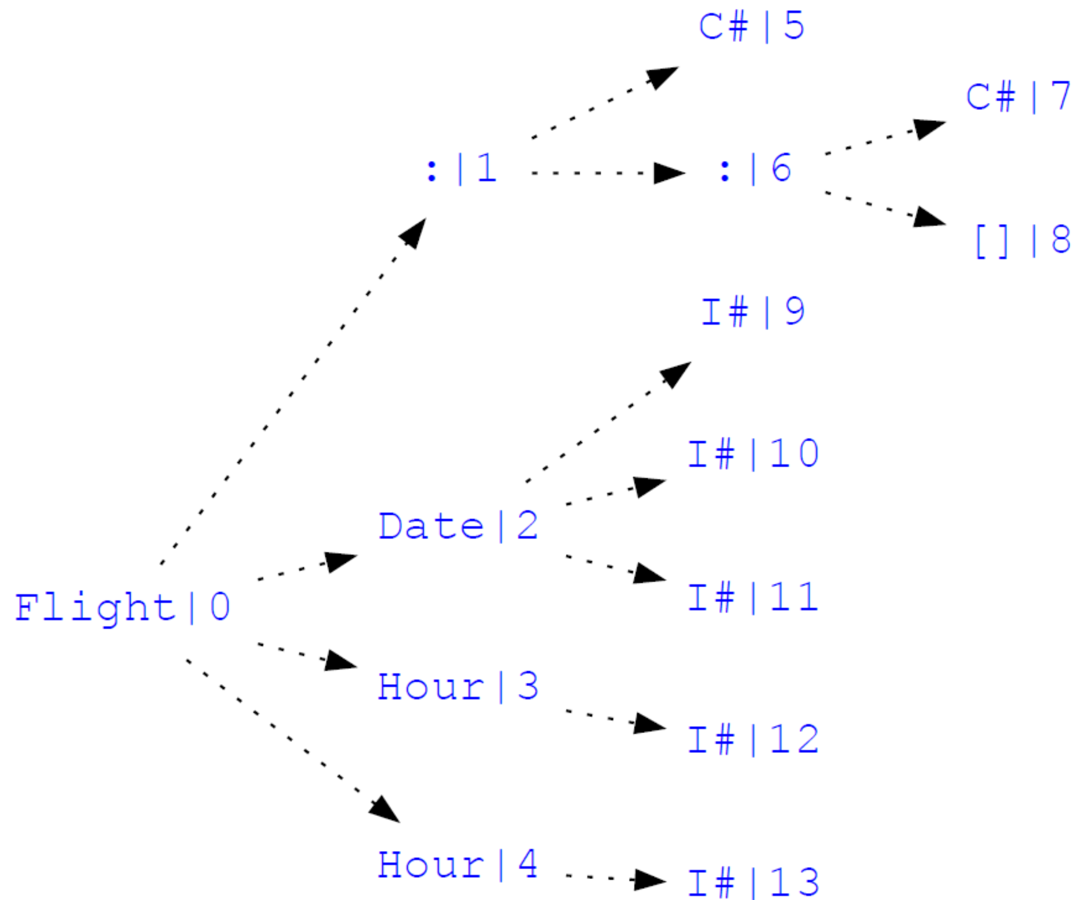
- mögliche Werte für Connection:
  - Train (Date 20 04 2011) (Hour 14) (Hour 11)
  - Flight "LH" (Date 20 04 2011) (Hour 16) (Hour 30)
  - …

- Berechnung mittels Pattern Matching:

```
travelTime :: Connection → Int
travelTime (Flight _ _ (Hour d) (Hour a)) = a – d + 2
travelTime (Train _ (Hour d) (Hour a))    = a – d + 1
```

Selbstdefinierte strukturierte Typen

- interne Repräsentation für: Flight "LH" (Date 20 04 2011) (Hour 16) (Hour 30)



Datenkonstruktoren als spezielle Funktionen

Für:

```
data Date       = Date Int Int Int
data Time       = Hour Int
data Connection = Train Date Time Time |
                  Flight String Date Time Time
```

erhalten wir:

```
> :t Date
Date :: Int → Int → Int → Date
> :t Hour
Hour :: Int → Time
> :t Train
Train :: Date → Time → Time → Connection
> :t Flight
Flight :: String → Date → Time → Time → Connection
```

Rekursive Datentypen

- Wie Funktionsdefinitionen können auch Datentypdeklarationen rekursiv sein.

- Vielleicht das einfachste Beispiel:

```
data Nat = Zero | Succ Nat
```

- Werte von Typ Nat:

  Zero, Succ Zero, Succ (Succ Zero), …

- Berechnungen darauf mittels Pattern Matching:

```
add :: Nat → Nat → Nat
add Zero       m = m
add (Succ n)  m = Succ (add n m)
```

Rekursive Datentypen

- Die Definition:

```
add :: Nat → Nat → Nat
add Zero      m = m
add (Succ n)  m = Succ (add n m)
```

erinnert vielleicht an:

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

- Tatsächlich sind Listen intern definiert als, im Prinzip:

```
data [Bool]  =  [ ]  |  (:) Bool [Bool]
```

Rekursive Datentypen

- Ein etwas komplexeres Beispiel (so ähnlich schon gesehen):

```
data Expr  =  Lit Int | Add Expr Expr | Mul Expr Expr
```

- Mögliche Werte:

  Lit 42 , Add (Lit 2) (Lit 7) , Mul (Lit 3) (Add (Lit 4) (Lit 0)) , …

- Ein „Mini-Interpreter“:

```
eval :: Expr → Int
eval  (Lit n)          =  n
eval  (Add e1 e2)  =  eval e1 + eval e2
eval  (Mul e1 e2)  =  eval e1 * eval e2
```

Rekursive Datentypen

- Oder auch allgemeine Binärbäume:

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

- mit wie folgt getypten Datenkonstruktoren:

```
> :t Leaf
Leaf :: Int → Tree
> :t Node
Node :: Tree → Int → Tree → Tree
```

- und (zu definierenden) Funktionen für „Flattening“, Prefix-Traversal, Postfix-Traversal, …

Simultan-rekursive Datentypen

- Schließlich, ein etwas künstliches Beispiel:

```
data T1  =  A T2 | E
data T2  =  B T1
```

- Mögliche Werte für T1:

  E , A (B E) , A (B (A (B E))) , A (B (A (B (A (B E))))) , …

- Mögliche Werte für T2:

  B E , B (A (B E)) , B (A (B (A (B E)))) , …

- Berechnung:

```
as :: T1 → Int
as (A t) = 1 + as' t
as E     = 0

as' :: T2 → Int
as' (B t) = as t
```

Typsynonyme

- Typsynonyme vergeben neue Namen für schon existierende Typen:

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

  – im Unterschied zu data keine Konstruktoren, keine Alternativen; außerdem wirklich nur ein neuer Name, kein neuer Typ

  – können verschachtelt sein:

```
type Pos   = (Int, Int)
type Trans = Pos → Pos
```

  aber nicht rekursiv!

Typisomorphe

- mit newtype Erzeugung einer unterscheidbaren Kopie eines vorhandenen Typs:

```
newtype Rat = Rat (Int, Int)
```

  – genau ein Datenkonstruktor mit genau einem Argument, keine Alternativen; wirklich ein neuer Typ!

  – typischer Anwendungsfall:

```
newtype Rat = Rat (Int, Int)
newtype Pos = Pos (Int, Int)
```

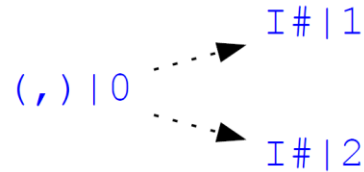
  – Rekursion erlaubt:

```
newtype InfList = Cons (Int, InfList)
```

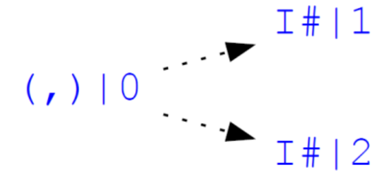
  – per Anschein nur ein Spezialfall von data, tatsächlich aber Unterschiede hinsichtlich Effizienz und Termination

Typsynonyme vs. Typisomorphe vs. „data“

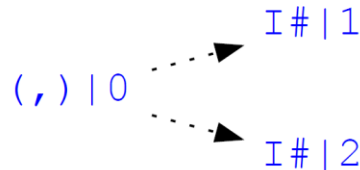
type Pos = (Int, Int)
p = (3, 4)



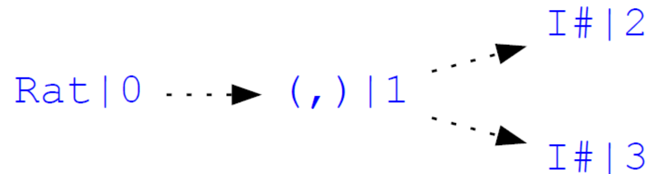
newtype Pos = Pos (Int, Int)
p = Pos (3, 4)



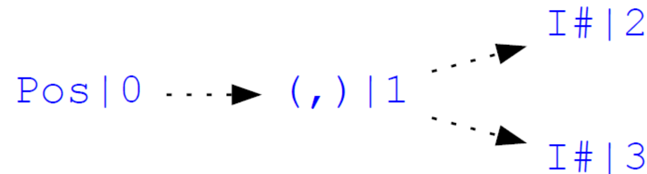
newtype Rat = Rat (Int, Int)
r = Rat (3, 4)



data Rat = Rat (Int, Int)
r = Rat (3, 4)

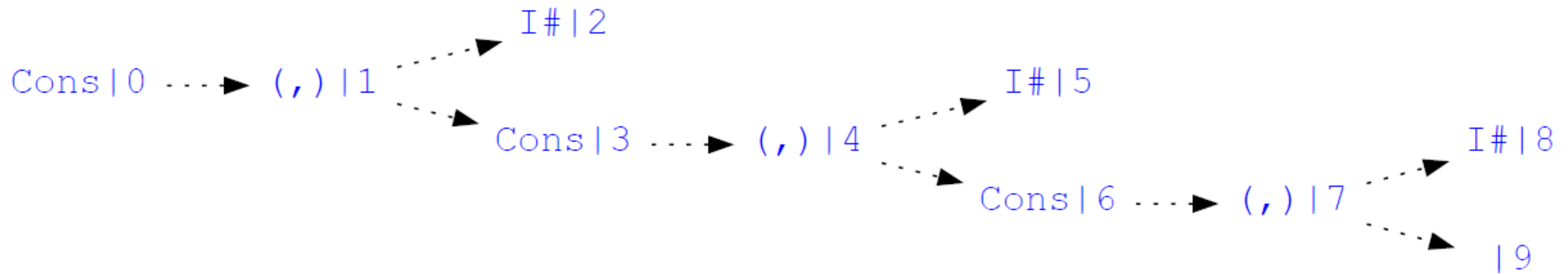


data Pos = Pos (Int, Int)
p = Pos (3, 4)

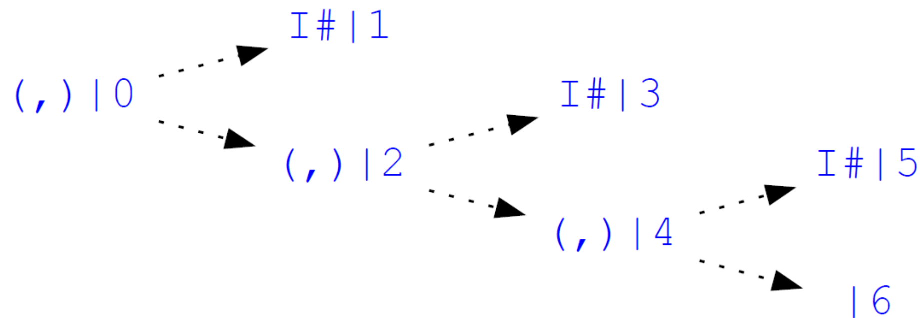


Typsynonyme vs. Typisomorphe vs. „data“

```
data InfList = Cons (Int, InfList)
xs = Cons (1, Cons (2, Cons (3, xs)))
```




```
newtype InfList = Cons (Int, InfList)
xs = Cons (1, Cons (2, Cons (3, xs)))
```

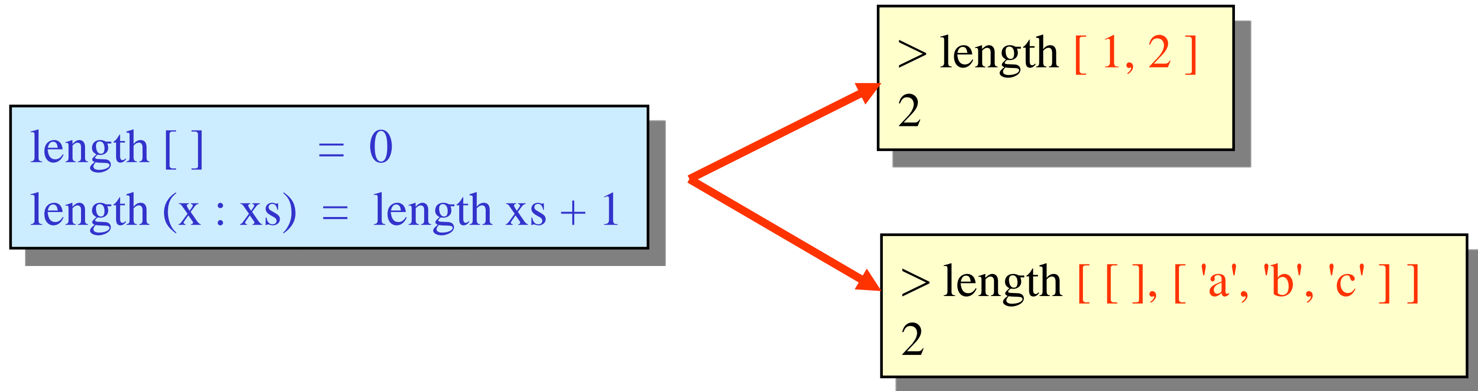


~~type InfList = (Int, InfList)~~
~~xs = (1, (2, (3, xs)))~~

Deskriptive Programmierung

Parametrische Polymorphie

- Viele schon gesehene/vorhandene Listenoperatoren sind für Listen aus beliebigen Elementtypen gedacht, z.B.:



- Wie für Standardfunktionen hat man natürlich auch für selbstdefinierte Funktionen gern solche Flexibilität:

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

Parametrisch polymorphe Funktionen

- Statt mehrerer Varianten:

$$
\begin{array}{l}
\text{concatenation} :: [\text{Int}] \to [\text{Int}] \to [\text{Int}] \\
\text{concatenation}\ [\,]\quad \text{ys} = \text{ys} \\
\text{concatenation}\ (\text{x} : \text{xs})\ \text{ys} = \text{x} : \text{concatenation xs ys}
\end{array}
$$

$$
\begin{array}{l}
\text{concatenation}' :: [\text{Bool}] \to [\text{Bool}] \to [\text{Bool}] \\
\text{concatenation}'\ [\,]\quad \text{ys} = \text{ys} \\
\text{concatenation}'\ (\text{x} : \text{xs})\ \text{ys} = \text{x} : \text{concatenation}'\ \text{xs ys}
\end{array}
$$

$$
\begin{array}{l}
\text{concatenation}'' :: \text{String} \to \text{String} \to \text{String} \\
\text{concatenation}''\ [\,]\quad \text{ys} = \text{ys} \\
\text{concatenation}''\ (\text{x} : \text{xs})\ \text{ys} = \text{x} : \text{concatenation}''\ \text{xs ys}
\end{array}
$$

- nur eine Definition:

$$
\begin{array}{l}
\text{concatenation} :: [\text{a}] \to [\text{a}] \to [\text{a}] \\
\text{concatenation}\ [\,]\quad \text{ys} = \text{ys} \\
\text{concatenation}\ (\text{x} : \text{xs})\ \text{ys} = \text{x} : \text{concatenation xs ys}
\end{array}
$$

Typvariablen und parametrisierte Typen

- Um polymorphen Funktionen einen Typ zuordnen zu können, werden Variablen verwendet, die als Platzhalter für beliebige Typen stehen:

Typvariablen

- Mit Typvariablen können für polymorphe Funktionen parametrisierte Typen gebildet werden:

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

- Ist auch der Resultattyp mittels einer Typvariable beschrieben, dann bestimmt natürlich der Typ der aktuellen Parameter den Typ des Resultats:

```
> :t last
last :: [a] → a
```

```
> :t last [ True, False ]
last [ True, False ] :: Bool
```

Sichere Verwendung polymorpher Funktionen

```
concatenation :: [a] → [a] → [a]
concatenation [ ]      ys  =  ys
concatenation (x : xs) ys  =  x : concatenation xs ys
```

```
> concatenation [ True ] [ False, True, False ]
[True, False, True, False]

> concatenation "abc" "def"
"abcdef"

> concatenation "abc" [True]
  Couldn't match 'Char' against 'Bool'
    Expected type: Char
    Inferred type: Bool
  In the list element: True
  In the second argument of 'concatenation', namely '[True]'
```

Weitere Beispiele

```
drop :: Int → [Int] → [Int]
drop 0       xs       = xs
drop n       [ ]      = [ ]
drop (n + 1) (x : xs) = drop n xs
```

```
drop :: Int → [a] → [a]
drop 0       xs       = xs
drop n       [ ]      = [ ]
drop (n + 1) (x : xs) = drop n xs
```

```
zip :: [Int] → [Int] → [(Int, Int)]
zip (x : xs) (y : ys) = (x, y) : zip xs ys
zip xs       ys       = [ ]
```

```
zip :: [a] → [b] → [(a, b)]
zip (x : xs) (y : ys) = (x, y) : zip xs ys
zip xs       ys       = [ ]
```

```
fst :: (a, b) → a

head :: [a] → a

take :: Int → [a] → [a]

id :: a → a
```

Sichere Verwendung polymorpher Funktionen

```
zip :: [a] → [b] → [(a, b)]
zip (x : xs) (y : ys)  =  (x, y) : zip xs ys
zip xs       ys        =  [ ]
```

```
> zip "abc" [ True, False, True ]
[('a', True), ('b', False), ('c', True)]

> :t "abc"
"abc" :: [Char]

> :t [ True, False, True ]
[True, False, True] :: [Bool]

> :t [ ('a', True), ('b', False), ('c', True) ]
[('a', True), ('b', False), ('c', True)] :: [(Char, Bool)]
```

Polymorphe Datentypen

- Abstraktion möglich von:

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

- zu:

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

- mit wie folgt getypten Datenkonstruktoren:

```
> :t Leaf
Leaf :: a → Tree a
> :t Node
Node :: Tree a → a → Tree a → Tree a
```

Polymorphe Datentypen

- Mögliche Werte für:

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

sind etwa: Leaf 3 :: Tree Int
Node (Leaf 'a') 'b' (Leaf 'c') :: Tree Char

aber nicht: Node (Leaf 'a') 3 (Leaf 'c')

- Beispielfunktion:

```
height :: Tree a → Int
height (Leaf _)        = 0
height (Node t1 _ t2)  = 1 + max (height t1) (height t2)
```

Polymorphe Typsynonyme und Typisomorphe

- Abstraktion genauso möglich für type und newtype:

```
type PairList a b = [(a, b)]
```

```
newtype InfList a = Cons (a, InfList a)
```

Exkurs: Semantische Konsequenzen parametrischer Polymorphie

- Typen mit Typvariablen implizieren (in einer „puren“ Sprache wie Haskell) nicht-triviale Einschränkungen des Verhaltens der entsprechenden Funktionen.

- Als einfaches Experiment, überlegen Sie sich mal jeweils drei Funktionen folgender Typen:
  - $a \to [a]$
  - $Int \to a \to [a]$
  - $(a, b) \to a$
  - $(a, a) \to a$
  - a
  - $a \to a$
  - $[a] \to a$
  - $[a] \to Int$
  - $[a] \to [a]$

- Jenseits dieser Art „Charakterisierungen“ auch praktischer angelegte Konsequenzen, zum Beispiel, für jede beliebige Funktion $fun :: [a] \to [a]$ gilt:

$$fun\ [\ g\ x \mid x \leftarrow xs\ ] = [\ g\ y \mid y \leftarrow fun\ xs\ ]$$

Deskriptive Programmierung

Ad-hoc Polymorphie

Vordefinierte Typklassen, insbesondere automatisches „deriving“

- Das freizügige Einführen immer neuer Typen mag zunächst unattraktiv erscheinen, da man jeweils auch bestimmte Funktionalität (neu)-implementieren müsste (für Ein- und Ausgabe, für „Rechnen“ auf Aufzählungstypen, …).

- Diese Sorgen erübrigen sich jedoch durch Mechanismen für generische Funktionalität, zum Beispiel:

```
data Color  =  Red | Green | Blue | White | Black  deriving (Enum, Bounded)

allColors  =  [minBound .. maxBound] :: [Color]
```

```
data Expr  =  Lit Int | Add Expr Expr | Mul Expr Expr  deriving (Read, Show, Eq)
```

…

- Funktioniert für data und für newtype. (… während für type nicht sinnvoll – Warum?)

Vordefinierte Typklassen, Instanzdefinitionen von Hand

Am einfachsten erklärt durch Beispiele:

```
data Color  =  Red | Green | Blue | White | Black  deriving (Enum, Bounded)

instance Show Color where
  show Red     =  "rot"
  show Green  =  "gruen"
  …
```

```
newtype Rat  =  Rat (Int, Int)

instance Show Rat where
  show (Rat (n, m))  =  show n ++ " / " ++ show m
```

```
data Expr  =  Lit Int | Add Expr Expr | Mul Expr Expr

instance Show Expr where
  show (Lit n)          =  "Lit "  ++  show n  ++  "; "
  show (Add e1 e2)  =  show e1  ++  show e2  ++  "Add; "
  show (Mul e1 e2)  =  show e1  ++  show e2  ++  "Mul; "
```

Natürlich dürfen auch beliebige andere Funktionen aufgerufen werden, nicht nur die gerade definierte (auf anderem oder dem selben Typ).

Zusammenspiel mit parametrischer Polymorphie

- Wir hatten Typvariablen benutzt, um auszudrücken, dass eine bestimmte Funktionalität etwa nicht vom Typ der Elemente einer Liste abhängt:

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

- Wie ist das nun zum Beispiel mit show?

- Sicher wollen wir nicht etwa schreiben:

```
instance Show [Int] where
  show [ ]      = "[ ]"
  show (i : is) = … show i … show is …

instance Show [Color] where
  show [ ]      = "[ ]"
  show (c : cs) = … show c … show cs …
```

Zusammenspiel mit parametrischer Polymorphie

- Parametrisierung über den Elementtyp, aber mit explizitem Constraint:

```
instance Show a ⇒ Show [a] where
  show [ ]        =  "[ ]"
  show (x : xs)  =  … show x … show xs …
```

- Ein solcher Constraint kann auch Abhängigkeit zu einer <u>anderen</u> Typklasse ausdrücken:

```
instance Show a ⇒ Eq a where
  x == y   =   show x == show y
```

- Und auf ganz natürliche Weise können Constraints auch in Typsignaturen „normaler“ Funktionen auftauchen:

```
elem :: Eq a ⇒ a → [a] → Bool
elem x [ ]        =  False
elem x (y : ys)  =  x == y  ||  elem  x  ys
```

Definition eigener Typklassen?

- Zunächst ein Blick auf die Definition zweier Klassen in der Standardbibliothek:

```
class Eq a where
  (==) :: a → a → Bool
  x == y  =  not (x /= y)
  (/=) :: a → a → Bool
  x /= y  =  not (x == y)

class Eq a ⇒ Ord a where
  (<), (<=), (>), (>=) :: a → a → Bool
  …
```

optionale Default-Implementierungen (für `x == y = not (x /= y)` und `x /= y = not (x == y)`)

- Dabei bedeutet Eq a ⇒ Ord a nicht, dass jeder Eq-Typ auch ein Ord-Typ ist, sondern dass ein Typ nur dann zur Typklasse Ord gehören kann, wenn er bereits zur Typklasse Eq gehört. (Und er eben zusätzlich auch noch Operationen (<), (<=), … unterstützt, für deren Default-Implementierungen freilich die Eq-Methoden benutzt werden können.)

- Definition eigener Typklassen einfach analog (siehe Live-Beispiele).

Deskriptive Programmierung

Higher-Order

Higher-Order: Funktionen als Parameter und Resultate

- In Haskell dürfen Funktionen andere Funktionen „manipulieren“ oder „generieren“:

  - Funktionen dürfen Funktionsargumente sein.
  - Funktionen dürfen Funktionswerte sein.

- Bezeichnung für derartige Funktionen (in Anlehnung an Begrifflichkeit aus der Prädikatenlogik):

  Funktionen höherer Ordnung

- Funktionen, die „nur normale Daten“ verarbeiten und erzeugen, sind Funktionen erster Ordnung.

Curryfizierung (1)

- In Haskell werden mehrstellige Funktionen in der Regel als „mehrstufige“ Funktionale aufgefasst (und dadurch u.U. Parameterklammern gespart):

```
zwischen :: Integer → (Integer → (Integer →  Bool))
zwischen  x  y  z     | x <= y && y <= z  =  True
                      | otherwise         =  False
```

- Die Anwendung dieses Prinzips wird heute Curryfizierung genannt (nach Haskell B. Curry, der diese Technik intensiv untersucht hat; der eigentliche „Erfinder“ ist allerdings der Logiker Schönfinkel).

- Die obige Form der zwischen-Funktion wird die „currifizierte“ Form genannt, die konventionellere Form mit einem Parametertupel heißt „uncurrifiziert“.

(im Engl.: „curried“/„uncurried“)

Curryfizierung (2)

- Neben der Klammereinsparung bringt die currifizierte Notation noch den Vorteil mit sich, dass von jeder Funktion verschiedene Varianten (mit verschiedenen Stelligkeiten) automatisch mit zur Verfügung stehen.

```
zwischen :: Integer → (Integer → (Integer →  Bool))
zwischen  x  y  z     | x <= y && y <= z  =  True
                      | otherwise         =  False
```

```
zwischen 2      :: Integer → (Integer → Bool)
zwischen 2 3    :: Integer → Bool
zwischen 2 3 4  :: Bool
```

- Jede solche partielle Applikation hat selbst alle „Rechte“ einer Funktion, darf also insbesondere selbst appliziert werden, weitergereicht werden, gespeichert werden, …

„Sections“

- Ein normalerweise zwischen seinen Argumenten geschriebener Operator kann durch Umschließung mit Klammern in eine vor die Argumente zu schreibende, currifizierte Funktion umgewandelt werden:

```
> (+) 3 4
7
```

- Eine solche „Section“ kann auch eines der Argumente in den Klammern einschließen:

```
> (/) 3 2
1.5
```

vs.

```
> (3/) 2
1.5
```

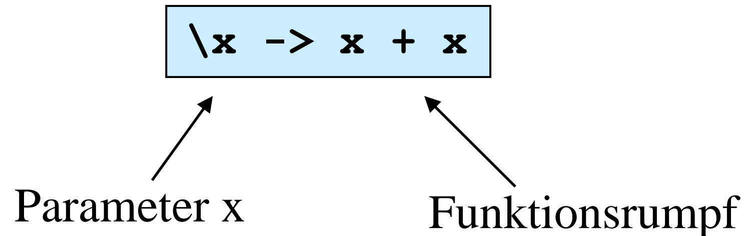
vs.

```
> (/2) 3
1.5
```

- Einige weitere Beispiele: (>3), (1+), (1/), (*2), (++ [42])

Anonyme Funktionen (1)

- Funktionen können anonym erzeugt werden, ohne ihnen einen Namen zu geben, z.B.:



- entspricht der mathematischen Notation von „λ-Abstraktionen“, z.B.:

```
λx.(x + x)
```

- Deren Applikation wird wie normale Funktionsauswertung behandelt, z.B.:

```
> (\x → x + x) 3
6
```

Anonyme Funktionen (2)

- Nützliche Sichtweise im Zusammenhang mit Curryfizierung:

statt:

```
add :: Int → Int → Int
add x y = x + y
```

auch:

```
add :: Int → (Int → Int)
add = \x → \y → x + y
```

- Oder auch:

```
const :: Int → Int → Int
const x _ = x
```

vs.

```
const :: Int → (Int → Int)
const x = \_ → x
```

- übrigens, abkürzende Schreibweise für anonyme Funktionen mehrerer Argumente:

```
Main> (\x -> \y -> 2*x*y) 2 3
12
```

vs.

```
Main> (\x y -> 2*x*y) 2 3
12
```

Higher-Order: etwas künstliche Beispiele

- Funktion als Parameter und Ergebnis:

```
g :: (a → a) → a → a
g f x = f (f x)
```

- Etwas expliziter (mit λ-Abstraktion):

```
g :: (a → a) → (a → a)
g f = \x → f (f x)
```

- Curryfizierung innerhalb der Sprache:

```
curry :: ((a, b) → c) → (a → b → c)
curry f = \x y → f (x, y)
```

- Und umgekehrt:

```
uncurry :: (a → b → c) → ((a, b) → c)
uncurry f = \(x, y) → f x y
```

Oft verwendete Higher-Order Funktionen auf Listen (1)

- Ein sehr nützliches Beispiel einer Funktion, die eine andere Funktion als Parameter akzeptiert (und sie dann auf alle Elemente einer Liste anwendet), ist die map-Funktion:

```
map f  [ ]          =   [ ]
map f  (x : xs)     =   f x : map f xs
```

Funktion als Parameter

- Zwei unterschiedliche Applikationen dieser Funktion:

```
> map square [1, 2, 3]
[1, 4, 9] :: [Integer]
```

```
> map sqrt [2, 3, 4]
[1.41421, 1.73205, 2.0] :: [Double]
```

- Die Funktion map ist polymorph:

```
> :t map
map :: (a → b) → [a] → [b]
```

Oft verwendete Higher-Order Funktionen auf Listen (2)

- Neben map gibt es noch eine Reihe weiterer wichtiger Higher-Order Funktionen für das Arbeiten mit Listen: filter, foldl, foldr, zipWith, scanl, scanr, …

- Die Funktion filter dient zum Extrahieren von Listenelementen, die eine bestimmte Boolesche Bedingung erfüllen:

```
filter  :: (a → Bool) → [a] → [a]
filter p xs  =  [ x | x ← xs, p x ]
```

„Prädikat“

```
> filter even [1, 2, 4, 5, 7, 8]
[2, 4, 8]

> filter even (filter (>3) [1, 2, 4, 5, 7, 8])
[4, 8]
```

Abkürzung für \x → x > 3
(zur Erinnerung: „Section“)

Effektiver Einsatz von Higher-Order Funktionen

- Eher Haskell-un-idiomatisch:

```
fun :: [Int] → Int
fun [ ] = 0
fun (x : xs)  | x < 20      =  5 * x – 3 + fun xs
              | otherwise   =  fun xs
```

- Besser:

```
fun :: [Int] → Int
fun  =  sum  .  map (\x → 5 * x – 3)  .  filter (< 20)
```

- Weitere für diesen Stil nützliche Funktionen: zip, splitAt, takeWhile, repeat, iterate, …

Weitere Beispiele für Nutzen von Higher-Order

- Was bewirkt folgende Funktion (im Kontext von Gloss)?

```
f :: Float → [Float → Picture] → (Float → Picture)
f d fs t = pictures [ translate (i * d) 0 (a t) | (i, a) ← zip [0 ..] fs ]
```

- Und diese?

```
g :: [Float] → [Float → Picture] → (Float → Picture)
g ss fs t = pictures (map (\(s, a) → a (s * t)) (zip ss fs))
```

- Eine Aufgabe in ähnlichem Geiste auf kommendem Übungsblatt.

Weitere Beispiele für Nutzen von Higher-Order

- Erinnern wir uns an:

```
data Expr  =  Lit Int | Add Expr Expr | Mul Expr Expr

eval :: Expr → Int
eval  (Lit n)           =  n
eval  (Add e1 e2)       =  eval e1 + eval e2
eval  (Mul e1 e2)       =  eval e1 * eval e2
```

- Angenommen, wir möchten Subtraktion und Division hinzufügen.

```
...
eval  (Sub e1 e2)       =  eval e1 – eval e2
eval  (Div e1 e2)       =  eval e1 `div` eval e2
```

- Mögliches Problem: Division durch Null, daher …

Weitere Beispiele für Nutzen von Higher-Order

- Um mögliche Division durch Null abzufangen, sollten wir besser so vorgehen:

```
eval :: Expr → Maybe Int
eval  (Lit n)         =  Just n
eval  (Add e1 e2)     =  case eval e1 of
                            Nothing → Nothing
                            Just r1   → case eval e2 of
                                           Nothing → Nothing
                                           Just r2   → Just (r1 + r2)
…
```

- Aber um diese mühsamen case-Kaskaden zu vermeiden, Abstraktion der Essenz in:

```
andThen :: Maybe a → (a → Maybe b) → Maybe b
andThen m f  =  case m of  Nothing → Nothing
                           Just r    → f r
```

- Und dann etwa:

```
eval  (Add e1 e2)  =  eval e1 `andThen` \r1 →
                      eval e2 `andThen` \r2 → Just (r1 + r2)
```

Einschub: idiomatisches Haskell bei „Verzweigungen“

- Generell, statt Bedingungen xs == [ ] oder m == Nothing lieber null xs bzw. isNothing m.

- Außerdem, statt etwa:

```
let y = f x in
  if isJust y
    then something-involving-(fromJust y)-maybe-several-times
    else something-else
```

lieber:

```
case f x of
  Just y'   → the-first-something-from-above-but-with-y'-instead-of-(fromJust y)
  Nothing → something-else-as-above
```

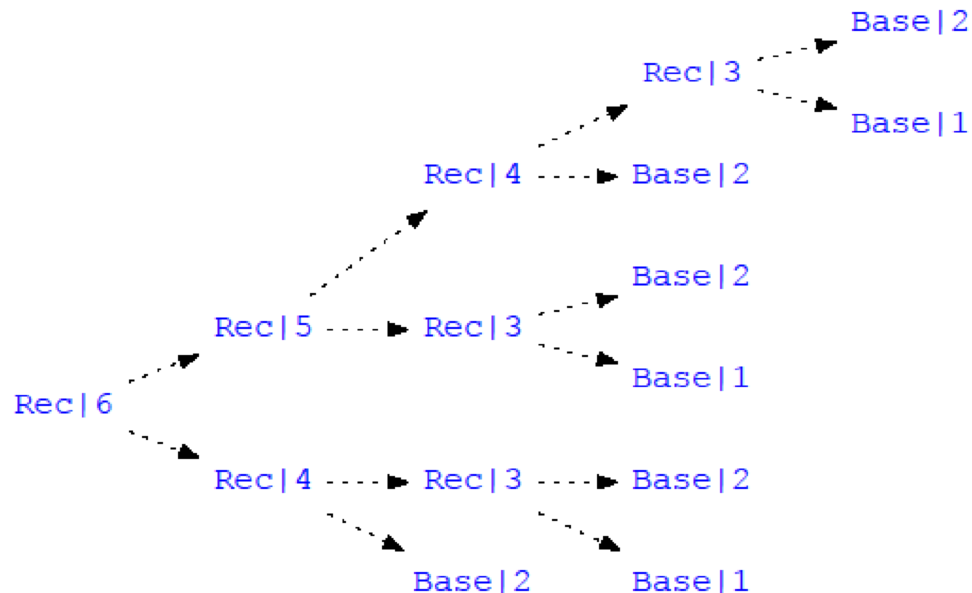
(und ähnlich für analoge Situationen etwa bei Listen).

Higher-Order: ein etwas komplexeres Beispiel, Memoisierung (1)

- Betrachten wir folgendes Programm, sehr ineffizient:

```
fib :: Int → Int
fib n  |  n < 2  =  1
fib n            =  fib (n – 2) + fib (n – 1)
```

- Die Ineffizienz liegt an folgendem „Aufrufgraph“ (für fib 6):



Higher-Order: ein etwas komplexeres Beispiel, Memoisierung (2)

- Betrachten wir folgendes Programm, sehr ineffizient:

```
fib :: Int → Int
fib n  |  n < 2  =  1
fib n            =  fib (n – 2) + fib (n – 1)
```

- Wir können Funktionsresultate „wiederverwendbar“ machen, und zwar auf sehr kanonische Weise, unabhängig von der konkreten fib-Funktion:

```
memo :: (Int → Int) → (Int → Int)
memo f  = g
          where g n    =  table !! n
                table  =  [ f n | n ←  [0 ..] ]
```

```
> let mfib = memo fib
> mfib 30
1346269   -- nach einigen Sekunden
> mfib 30
1346269   -- „sofort“
```

Higher-Order: ein etwas komplexeres Beispiel, Memoisierung (3)

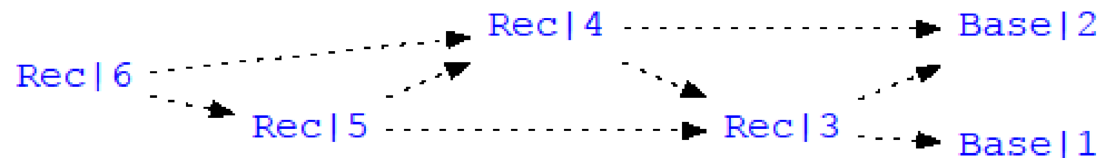
- Noch besser ist es, wenn wir Memoisierung auch innerhalb der Rekursion ausnutzen:

```
mfib = memo fib

fib :: Int → Int
fib n  |  n < 2  =  1
fib n            =  mfib (n – 2) + mfib (n – 1)
```

- Dann nämlich:

```
> fib 30
1346269   -- „sofort“
> fib 31
2178309   -- „noch sofortiger“
```

- „Aufrufgraph“ jetzt:



Strukturelle Rekursion auf Listen als Higher-Order Funktion

```
sum [ ]       = 0
sum (x : xs)  = x + sum xs
```

```
prod [ ]      = 1
prod (x : xs) = x * prod xs
```

- Die Listenfunktionen zum Summieren bzw. Multiplizieren von Listenelementen weisen dasselbe Rekursionsmuster auf, das sich mit Hilfe einer vordefinierten Funktion zum „Falten" von zweistelligen Operatoren in Listen realisieren lässt:

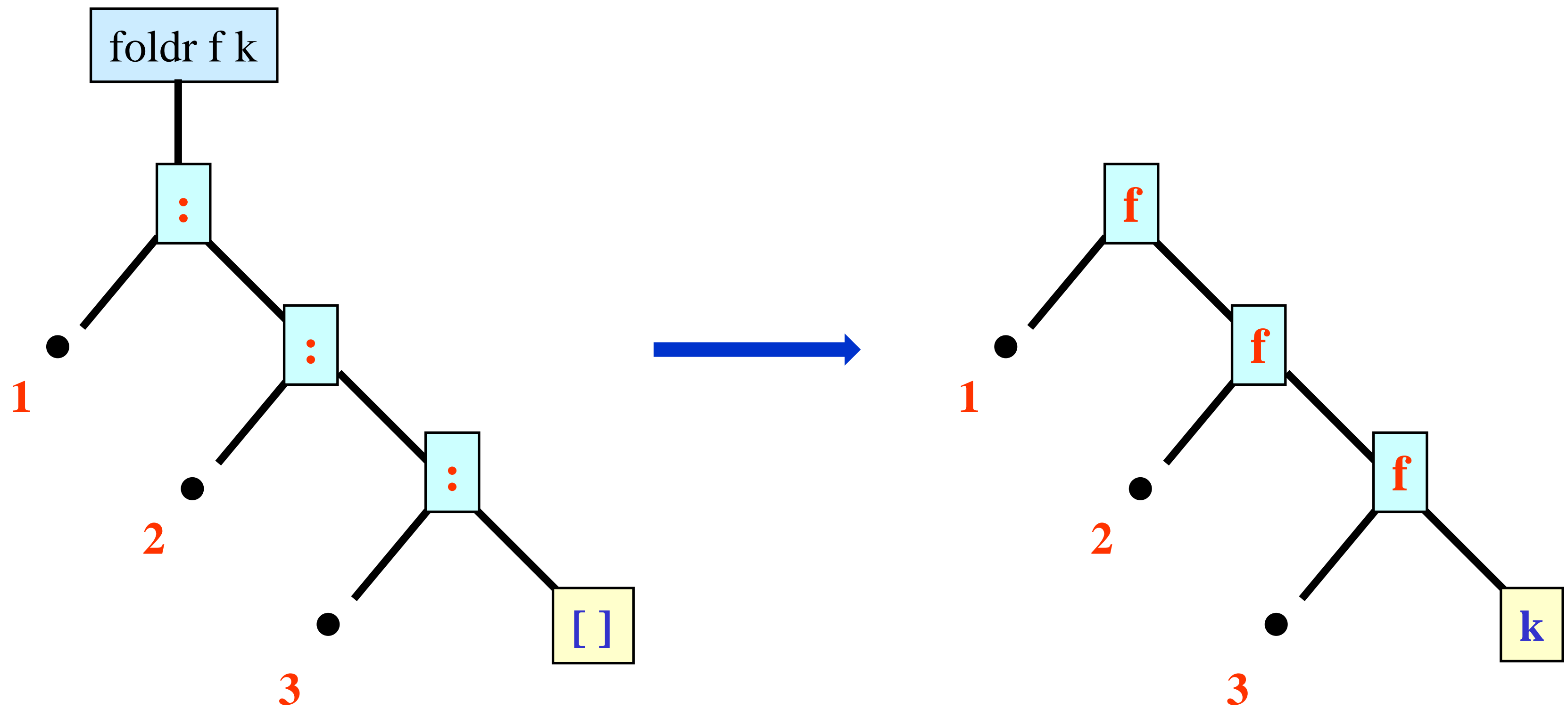
```
foldr :: (a → b → b) → b → [a] → b
foldr f k [ ]      = k
foldr f k (x : xs) = f x (foldr f k xs)
```

engl. „fold": „falten"
(..r steht für „right";
es gibt auch foldl)

- Zum Beispiel Definitionen von sum bzw. prod als Anwendung von foldr:

```
sum, prod :: [Int] → Int
sum  = foldr (+) 0
prod = foldr (*) 1
```

Visualisierung von foldr



Weitere Beispiele für Verwendung von foldr

- Unter Verwendung von foldr sind vordefinierte logische Junktoren implementiert, die auf Listen von Booleschen Werten operieren:

```
and, or :: [Bool] → Bool
and = foldr (&&) True
or = foldr (||) False
```

- „Quantoren“ über Listen sind als Verallgemeinerung dieser Junktoren mittels Komposition realisiert:

```
any, all :: (a → Bool) → [a] → Bool
any p = or . map p
all p = and . map p
```

z.B.: `all (<100) [ x^2 | x ← [1 .. 19] ]`

Allgemeine Strategie zur Verwendung von foldr

- Wann kann eine Funktion mittels foldr ausgedrückt werden?

- Wann immer es möglich ist, sie in folgende Form zu bringen:

```
g [ ]       =  k
g (x : xs)  =  f  x (g xs)
```

für irgendwelche k und f

- Dann:

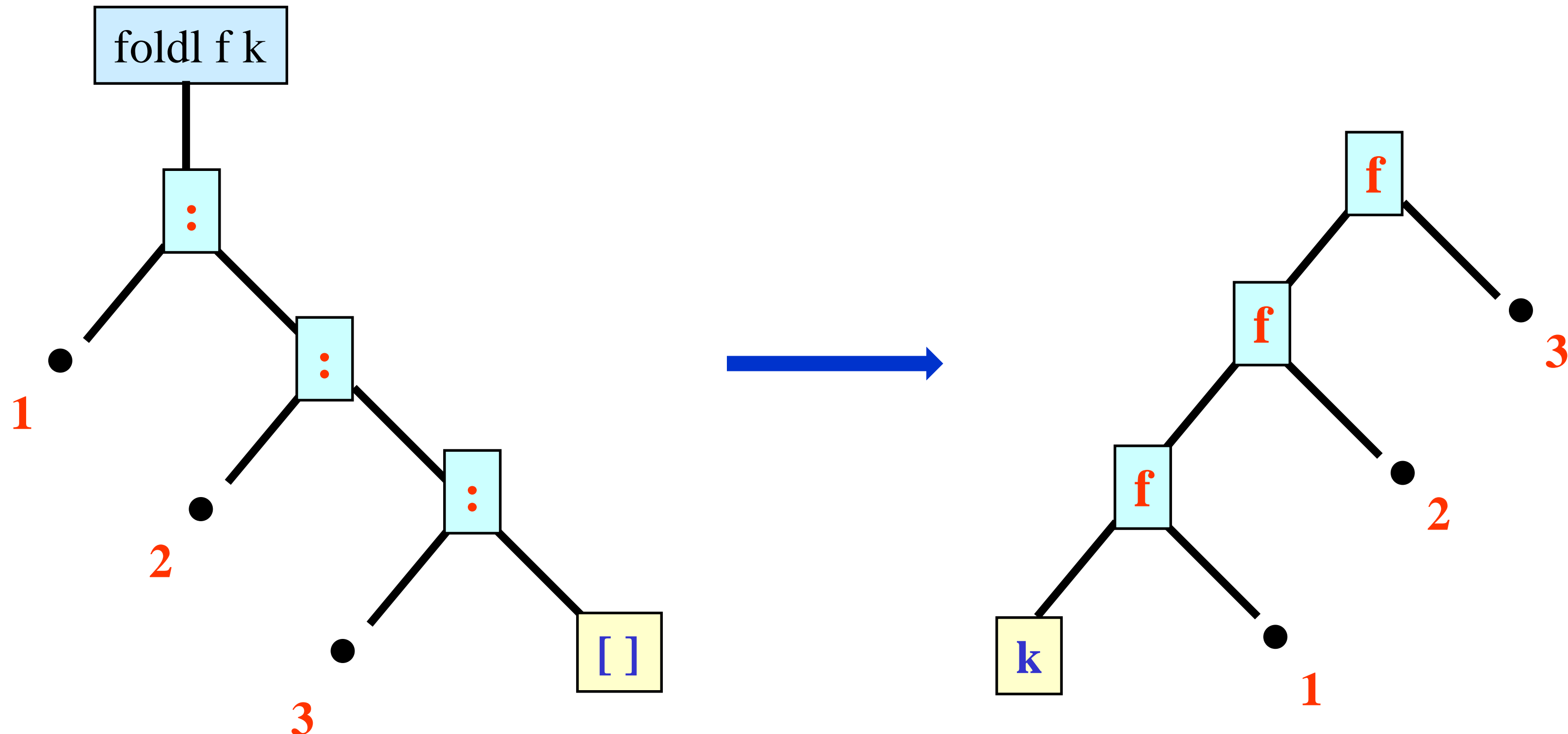
```
g  =  foldr  f  k
```

- Dies liefert eine einfache Charakterisierung strukturell rekursiver Funktionen auf Listen!

Eine linkslastige Variante von foldr

- Neben foldr gibt es:

```
foldl :: (b → a → b) → b → [a] → b
foldl f k [ ]      = k
foldl f k (x : xs) = foldl f (f k x) xs
```



Variationen auf foldl und foldr

- Gibt auch alle Zwischenergebnisse von foldl aus:

```
scanl :: (b → a → b) → b → [a] → [b]
scanl f  k xs = k : case xs of
                      [ ]      → [ ]
                      x : xs' → scanl f  (f  k  x)  xs'
```

- Zum Beispiel:

```
> scanl (+) 0 [1 .. 5]
[0, 1, 3, 6, 10, 15]
```

- In gewissem Sinne dual zu foldr:

```
unfoldr :: (b → Maybe (a, b)) → b → [a]
unfoldr f b = case f b of
                 Nothing    → [ ]
                 Just (a, b') → a : unfoldr f b'
```

Higher-Order Abstraktion auf der Ebene von Typen

- Einige der betrachteten Funktionen (map, foldr, …) lassen sich außer auf Listen auch auf anderen Datentypen sinnvoll realisieren. Dazu Verwendung des Typklassenkonzepts, aber nun higher-order Typvariablen:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

- Instanzen existieren neben Listen zum Beispiel für Maybe und andere vordefinierte Datentypen.

- Und mit neueren GHCs kann man sogar entsprechende Instanzen für viele nutzerdefinierte Datentypen deriven lassen.

Deskriptive Programmierung

Ein- und Ausgabe in Haskell

Deklarative Programmierung als Programmierparadigma

Die deklarative Programmierung ist ein Programmierparadigma, welches auf mathematischer, rechnerunabhängiger Theorie beruht.

Zu den deklarativen Programmiersprachen gehören:

- funktionale Sprachen (u.a. LISP, ML, Miranda, Gofer, Haskell)
- logische Sprachen (u.a. Prolog)
- funktional-logische Sprachen (u.a. Babel, Escher, Curry, Oz)
- Datenflusssprachen (wie Val oder Linda)

(aus Wikipedia, 07.04.08)

- In der Regel erlauben deklarative Sprachen in irgendeiner Form die Einbettung imperativer Programmteile, mehr oder weniger direkt und/oder „diszipliniert“.

- Andere Programmiersprachenkategorien, einigermaßen orthogonal zu dekl./imp.:
  - Objektorientierte oder ereignisorientierte Sprachen
  - Parallelverarbeitende/nebenläufige Sprachen
  - Stark oder schwach, statisch oder dynamisch, oder gar nicht getypte Sprachen

Ein-/Ausgabe in Haskell, ganz einfaches Beispiel

- In „reinen“ Funktionen ist keine Interaktion mit Betriebssystem/Nutzer/… möglich.
- Es gibt jedoch eine spezielle do-Notation, die Interaktion ermöglicht, und aus der man „normale“ Funktionen aufrufen kann.

Einfaches Beispiel:

```
prod :: [Int] → Int
prod [ ]       =  1
prod (x : xs)  =  x * prod xs
```

reine Funktion

```
main = do n ← readLn
          m ← readLn
          print (prod [n .. m])
```

„Hauptprogramm“

```
5
8
1680
```

Eingabe → 5
Eingabe → 8
Ausgabe → 1680

Programmablauf

Prinzipielles zu Ein-/Ausgabe in Haskell: IO-Typen …

- Es gibt einen vordefinierten Typkonstruktor IO, so dass sich für jeden konkreten Typ Int, Bool, [(Int, Tree Bool)] etc. der Typ IO Int, IO Bool, … bilden lässt.

- Die Interpretation eines Typs IO a ist, dass Elemente dieses Typs nicht selbst konkrete Werte sind, sondern „nur“ potentiell beliebig komplexe Sequenzen von Ein- und Ausgabeoperationen sowie von dabei eingelesenen Werten abhängigen Berechnungen, bei denen schließlich ein Wert des Typs a entsteht.

- Ein (eigenständig lauffähiges) Haskell-Programm insgesamt hat immer einen „IO-Typ“, in der Regel einfach main :: IO ().

- Zum Erzeugen von „IO-Werten“ gibt es vordefinierte Primitiven (und man sieht ihnen diesen IO-Charakter an Hand ihres Typs an):

```
getChar :: IO Char
getLine :: IO String
readLn :: Read a ⇒ IO a
```

```
putChar :: Char → IO ()
putStr, putStrLn :: String → IO ()
print :: Show a ⇒ a → IO ()
```

Prinzipielles zu Ein-/Ausgabe in Haskell: … und do-Notation

- Um IO-behaftete Berechnungen zu kombinieren (also basierend auf den IO-Primitiven komplexere Aktionssequenzen zu „bauen“), gibt es die do-Notation.

- Allgemeine Form:

$$\begin{array}{l} \text{do } cmd_1 \\ \quad x_2 \leftarrow cmd_2 \\ \quad x_3 \leftarrow cmd_3 \\ \quad cmd_4 \\ \quad x_5 \leftarrow cmd_5 \\ \quad \ldots \end{array}$$

Der do-Block insgesamt hat den Typ des letzten cmd_n. Zu diesem ist generell kein x_n vorhanden.

wobei die cmd_i jeweils IO-Typen haben und an die x_i (sofern explizit vorhanden) jeweils ein Wert des in cmd_i gekapselten Typs gebunden wird (und ab dieser Stelle im gesamten do-Block verwendet werden kann), nämlich gerade das Ergebnis der Ausführung von cmd_i.

- Oftmals auch noch nützlich (z.B. am Ende eines do-Blocks), vordefinierte Funktion return :: a $\rightarrow$ IO a, die ohne jegliche Ein-/Ausgabe einfach ihr Argument liefert.

Prinzipielles zu Ein-/Ausgabe in Haskell: IO-Typen und do-Notation

- Ein „komplexeres“ Beispiel:

```
dialog = do putStr "Eingabe: "
            s ← getLine
            if s == "end"
                then return ()
                else do let n = read s
                        putStrLn ("Ausgabe: " ++ show (n * n))
                        dialog
```

- Was „nein, nicht, auf keinen Fall“ geht, ist aus einem IO-Wert direkt (abseits der expliziten Sequenzialisierung und Bindung in einem do-Block) den gekapselten Wert zu entnehmen.

- Neben den gesehenen Primitiven für Ein-/Ausgabe per Terminal gibt es Primitiven und Bibliotheken für File-IO, Netzwerkkommunikation, GUIs, …

- Natürlich stehen auch im Kontext von IO-behafteten Berechnungen alle Features und Abstraktionsmittel von Haskell zur Verfügung, also wir definieren Funktionen mit Rekursion, verwenden Datentypen, Polymorphie, Higher-Order, …

Eigene „Kontrollstrukturen“

- Wie betont, stehen auch im Kontext von IO-behafteten Berechnungen alle Abstraktionsmittel von Haskell zur Verfügung, also insbesondere Polymorphie und Definition von Funktionen höherer Ordnung.

- Dies wird gern benutzt für Dinge wie:

```
while :: (a → Bool) → (a → IO a) → (a → IO a)
while p body  =  loop
   where  loop x  =  if  p x  then  do  x' ← body x
                                       loop x'
                                 else  return x
```

- Was ist dann wohl die Ausgabe folgenden Aufrufs?

```
> while (< 10) (\n → do {print n; return (n + 1)}) 0
```

Ein einfaches „interaktives Spiel“

Towers of Hanoi:

- es gibt drei Türme/Plätze: A, B und C
- zu Beginn liegen n Scheiben unterschiedlicher Größe auf A; B und C sind leer
- Ziel ist es, die Scheiben von A nach B zu transportieren
- zu keinem Zeitpunkt darf eine Scheibe auf einer kleineren anderen liegen

Beispiel mit drei Scheiben:

- zu Beginn „Konfiguration“ (A: 1,2,3; B leer; C leer)
- Zug A $\mapsto$ B, führt zu (A: 2,3; B: 1; C leer)
- Zug A $\mapsto$ C, führt zu (A: 3; B: 1; C: 2)
- Zug B $\mapsto$ C, führt zu (A: 3; B leer; C: 1,2)
- Zug A $\mapsto$ B, führt zu (A leer; B: 3; C: 1,2)
- Zug C $\mapsto$ A, führt zu (A: 1; B: 3; C: 2)
- Zug C $\mapsto$ B, führt zu (A: 1; B: 2,3; C leer)
- Zug A $\mapsto$ B, führt zu (A leer; B: 1,2,3; C leer)

Ein einfaches „interaktives Spiel“

Towers of Hanoi, allgemeine Strategie (Divide and Conquer):

- um n Scheiben von A nach B unter Nutzung von C zu bringen:
  - zunächst n – 1 Scheiben von A nach C unter Nutzung von B
  - dann eine Scheibe von A nach B
  - schließlich n – 1 Scheiben von C nach B unter Nutzung von A
- um n – 1 Scheiben von A nach C unter Nutzung von B zu bringen:
  - zunächst n – 2 Scheiben von A nach B unter Nutzung von C
  - dann eine Scheibe von A nach C
  - schließlich n – 2 Scheiben von B nach C unter Nutzung von A
- …

```
data Place = A | B | C  deriving Show

towers 0 i j k = [ ]
towers n i j k = towers (n – 1) i k j ++ [ (i, j) ] ++ towers (n – 1) k j i
```

```
> towers 3 A B C
[ (A, B), (A, C), (B, C), (A, B), (C, A), (C, B), (A, B) ]
```

Kommentiertes Ergebnis des „Live-Programmierens“ (1)

in der Version des Vorjahrs

```
module Main where

data Place = A | B | C  deriving (Show, Read, Eq)

type Move = (Place, Place)
type Conf = ([Int], [Int], [Int])

run :: [Move] → Conf → [Conf]
run [ ]          c = [ c ]
run (m : ms) c = c : run ms (step m c)

step :: Move → Conf → Conf
step (A, B) (a : as, bs, cs) = (as, a : bs, cs)
step (A, C) (a : as, bs, cs) = (as, bs, a : cs)
step (B, A) (as, b : bs, cs) = (b : as, bs, cs)
step (B, C) (as, b : bs, cs) = (as, bs, b : cs)
step (C, A) (as, bs, c : cs) = (c : as, bs, cs)
step (C, B) (as, bs, c : cs) = (as, c : bs, cs)
```

Typsynonyme zur Abkürzung

„Simulierter Lauf“ einer Zugfolge auf einer gegebenen Konfiguration

Kommentiertes Ergebnis des „Live-Programmierens“ (2)

in der Version des Vorjahrs

```
animate n [ ]        =  return ()
animate n (c : cs)  =  do output' c n
                          getLine
                          animate n cs

main  =  do n ← readLn
            animate n (run (towers n A B C) ([1 .. n], [ ], [ ]))

output' (as, bs, cs) n = output n
                          (replicate (n – length as) 0 ++ as,
                           replicate (n – length bs) 0 ++ bs,
                           replicate (n – length cs) 0 ++ cs)

output n ([ ], [ ], [ ])              =  return ()
output n (a : as, b : bs, c : cs)  =  do putStr (disc a n)
                                          putStr (disc b n)
                                          putStrLn (disc c n)
                                          output n (as, bs, cs)
```

jetzt „Hauptschleife“

„hübsche Ausgabe“

Kommentiertes Ergebnis des „Live-Programmierens“ (3)

in der Version des Vorjahrs

```
disc :: Int → Int → String
disc 0 n  =  replicate (2 * n – 1) ' '
disc i  n  =  replicate (n – i) ' ' ++ replicate (2 * i – 1) '*'
              ++ replicate (n – i) ' '

towers 0 i j k = [ ]
towers n i j k = towers (n – 1) i k j ++ [ (i, j) ] ++ towers (n – 1) k j i
```

reine Berechnung einer String-darstellung

unveränderte „Ursprungslogik“

Anmerkungen:

- inkrementelle Entwicklung:
  - zunächst nur run und step; statt animate nur print
  - dann animate, mit print statt output'
  - schließlich output, output', disc
- zwischendurch jeweils Testen von Teilfunktionalität
- nicht überall Funktionstypen „gepflegt“, stattdessen inferieren lassen

Anregungen zum Ausprobieren

- Wir könnten das „interaktive Programm mit Visualisierung“ zu Towers of Hanoi dahingehend ändern, dass der Nutzer die Züge vorgibt.
  - … dabei zunächst annehmen, dass stets nur korrekte Züge eingegeben werden,
  - … oder explizite Checks mit (Fehler-)Meldung an den Nutzer einfügen,
  - … vielleicht sogar die Möglichkeit schaffen, dass der Nutzer zu jedem Zeitpunkt um „Hilfe“ bitten kann, woraufhin das Puzzle vom aktuellen Stand aus wieder automatisch gelöst wird.

Funktionale Programmierung in Haskell: Zwischenbilanz (1)

- Prinzip der FP:
  - Spezifikation = Folge von Funktionsdefinitionen
  - Funktionsdefinition = Folge von definierenden Gleichungen
  - Operationalisierung = stufenweise Reduktion von Ausdrücken auf Werte
- Ausdrücke:
  - Konstanten, Variablen, strukturierte Ausdrücke: Listen, Tupel
  - Applikationen
  - list comprehensions
- Systeme definierender Gleichungen:
  - Kopf, Rumpf (mit div. Beschränkungen)
  - (ggfs.) mehrelementige Parameterlisten
  - Wächter
- syntaktische Besonderheiten von Haskell:
  - von mathematischer Notation abweichende Funktionssyntax
  - lokale Definitionen (let, where)
  - Layoutregel

Funktionale Programmierung in Haskell: Zwischenbilanz (2)

- Reduktion/Auswertung:
  - pattern matching, eindeutige Fallauswahl, Rekursion
  - lazy evaluation
  - besondere Rolle von IO, do-Blöcke
- Listen:
  - Klammerdarstellung vs. Baumdarstellung (:), pattern matching
  - spez. Listenfunktionen (z.B. length, ++, !!)
  - arithmetische Sequenzen, unendliche Listen, list comprehensions
- Typen (starke Typisierung, Typprüfung, Typinferenz):
  - Datentypen
    - Basisdatentypen (Integer etc.)
    - strukturierte Typen (Listen, Tupel)
    - algebraische Datentypdeklarationen, Konstruktoren
  - polymorphe Typen, Typvariablen
  - Funktionstypen
    - Funktionstypdeklarationen, Curryfizierung
  - Typklassen, Deklarationen, Instanzdefinitionen

Funktionale Programmierung in Haskell: Zwischenbilanz (3)

- Funktionen höherer Ordnung:
  - Funktionen als Parameter und/oder als Resultate
  - partielle Applikation, Sections
  - Lambda-Ausdrücke
  - Funktionen höherer Ordnung auf Listen: map, filter, foldr, …

- Verwendung expliziter Rekursionsschemata

Deskriptive Programmierung

Parserkombinatoren

Zur Erinnerung, Anfang der Vorlesung: Verarbeitung arithmetischer Ausdrücke

- Wir wollten gültige arithmetische Ausdrücke beschreiben, zum Beispiel angelehnt an eine formale Grammatik:

```
expr    ::=  term + expr | term
term    ::=  factor * term | factor
factor  ::=  nat | (expr)
```

- … um dann zum Beispiel sowas zu können:

```
data Expr = Lit Int | Add Expr Expr | Mul Expr Expr
```

```
parse :: String → Expr
???
```

```
> parse "2+3*5"
Add (Lit 2) (Mul (Lit 3) (Lit 5))
```

```
eval :: Expr → Int
…
```

```
calc :: String → Int
calc s = eval (parse s)
```

```
> calc "2+3*5"
17
```

Zur Erinnerung, Anfang der Vorlesung: Verarbeitung arithmetischer Ausdrücke

- Wir hatten schon diskutiert, dass man natürlich die Regeln (zu Vorrang etc.) der Grammatik berücksichtigen muss, dann „naiv“ könnte man "2+3*5" ja statt als Add (Lit 2) (Mul (Lit 3) (Lit 5)) auch als Mul (Add (Lit 2) (Lit 3)) (Lit 5) lesen.

- Gezeigt bzw. erstmal nur behauptet hatte ich, dass man die Grammatik selbst als „Programm“ ausdrücken kann:

```
expr   =  ( Add  <$>  term  <*  char '+'  <*>  expr )  |||  term

term   =  ( Mul  <$>  factor  <*  char '*'  <*>  term )  |||  factor

factor =  ( Lit  <$>  nat )  |||  ( char '('  *>  expr  <*  char ')' )
```

und dann erhält:

```
>  parse expr "2+3*5"
Add (Lit 2) (Mul (Lit 3) (Lit 5))
```

- Genau das wollen wir jetzt realisieren.

- Da ähnliche Problemstellungen natürlich in vielen Bereichen auftauchen; wollen wir eine möglichst allgemeine Lösung entwickeln.

- Zum Beispiel eine Bibliothek:

```
type Parser = …

parse :: Parser → String → …

integer, identifier :: Parser

(|||) :: Parser → Parser → Parser

(+++) :: Parser → Parser → Parser
…
```

… wobei wir die notwendigen Kombinatoren für die spezielle syntaktische Form des Beispiels (<$>, <*>, <*, *>) zunächst noch zurückstellen.

Design einer Parser-Bibliothek

- Schon die zweiten „…“ in:

```
type Parser = …

parse :: Parser → String → …
```

zeigen, dass man wohl etwas allgemeiner starten muss.

- Besser:

```
type Parser a = …

parse :: Parser a → String → a

integer :: Parser Int

identifier :: Parser String

(|||) :: Parser a → Parser a → Parser a

(+++) :: Parser a → Parser b → Parser ?
```

Design einer Parser-Bibliothek

- Was wäre nun eine gute Repräsentation oder Implementierung für

```
type Parser a = …
```

?

- Naheliegende Idee, unter Verwendung von Higher-Order:

```
type Parser a = String → a
```

- Aber im Allgemeinen könnte ein (Teil-)Parser nur einen Teil der Eingabe „verbrauchen“, es bliebe also ein Rest-String übrig.

```
type Parser a = String → (a, String)
```

- Und wir sollten darauf vorbereitet sein, dass ein Parse-Versuch auch fehlschlagen kann.

```
type Parser a = String → Maybe (a, String)
```

Design einer Parser-Bibliothek

- Wie ist für

```
type Parser a = String → Maybe (a, String)
```

nun die Funktion

```
parse :: Parser a → String → a
```

zu implementieren?
(Zur Verwendung nachdem man einen Parser mittels der anderen Operationen der Bibliothek entwickelt hat.)

- Relativ natürlich:

```
parse :: Parser a → String → a
parse p inp = case p inp of
                Nothing        → error "invalid input"
                Just (x, "")   → x
                Just (_, rest) → error ("unused input: " ++ rest)
```

Design einer Parser-Bibliothek

- Machen wir uns nun an die Implementierung einiger „Parser-Bausteine“:

```
item :: Parser Char
item = \inp → case inp of
                ""      →  Nothing
                c : cs  →  Just (c, cs)
```

- Dann:

```
> parse item "a"
'a'

> parse item "b"
'b'

> parse item ""
Program error: invalid input

> parse item "ab"
Program error: unused input: b
```

Design einer Parser-Bibliothek

- Es geht auch etwas wählerischer:

```
digit :: Parser Char
digit = \inp → case inp of
                   ""      →  Nothing
                   c : cs  →  if '0' <= c && c <= '9' then Just (c, cs) else Nothing
```

- Dann:

```
> parse digit "a"
Program error: invalid input

> parse digit "5"
'5'

> parse digit ""
Program error: invalid input

> parse digit "ab"
Program error: invalid input
```

Design einer Parser-Bibliothek

- Analog:

```
lower :: Parser Char
lower = …
```

- Zum Ausdrücken von Alternativen:

```
infixr 3 |||

(|||) :: Parser a → Parser a → Parser a
p ||| q = …
```

- Gewünscht:

```
> parse (digit ||| lower) "a"
'a'

> parse (digit ||| lower) "5"
'5'
```

Design einer Parser-Bibliothek

- Implementierung des Auswahl-Operators:

```
(|||) :: Parser a → Parser a → Parser a
p ||| q = \inp → case p inp of
                   Just (x, rest)  →  Just (x, rest)
                   Nothing         →  q inp
```

- Der Operator ist assoziativ (daher macht auch die Deklaration „infixr 3 |||“ Sinn), und es gibt ein neutrales Element:

```
failure :: Parser a
failure = \inp → Nothing
```

```
> parse failure "a"
Program error: invalid input

> parse (failure ||| lower) "a"
'a'
```

Design einer Parser-Bibliothek

- Implementierung eines Operators zur Konkatenation von Parsern:

```
(+++) :: Parser a → Parser b → Parser ?
p +++ q = …
```

- Denkbar wäre `(+++) :: Parser a → Parser b → Parser (a, b)` , aber dann wäre es nicht möglich, Abhängigkeitsbedingungen auszudrücken.

- Stattdessen:

```
(++>) :: Parser a → (a → Parser c) → Parser c

(+++) :: Parser a → Parser b → Parser b
```

- Dann zum Beispiel ausdrückbar:

```
digit  ++>  \d → if d < '5' then digit else lower
```

- Implementierung der beiden Operatoren zur Konkatenation von Parsern:

```
infixr 4 +++, ++>

(+++) :: Parser a → Parser b → Parser b
p +++ q = \inp → case p inp of
                   Nothing        → Nothing
                   Just (_, rest) → q rest

(++>) :: Parser a → (a → Parser b) → Parser b
p ++> f = \inp → case p inp of
                   Nothing        → Nothing
                   Just (x, rest) → f x rest
```

- Tatsächlich, für den Operator, der das Ergebnis des ersten Parsers ignoriert:

```
(+++) :: Parser a → Parser b → Parser b
p +++ q  =  p  ++>  \_ → q
```

- Können wir dennoch eine Konkatenation implementieren, bei der beide Ergebnisse kombiniert werden?

- Zum Beispiel zum Parsen eines Kleinbuchstabens und einer Ziffer, und Rückgabe beider Komponenten:

```
> parse pair "a1"
('a', '1')
```

- Möglich, unter Verwendung einer weiteren Primitive:

```
yield :: a → Parser a
yield x = \inp → Just (x, inp)
```

- Dann nämlich:

```
pair = lower ++> \x → digit ++> \y → yield (x, y)
```

Design einer Parser-Bibliothek

- Das scheint ein allgemein nützliches Kombinationsprinzip zu sein, daher Abstraktion in eine extra Hilfsfunktion:

```
liftP :: (a → b → c) → Parser a → Parser b → Parser c
liftP f p q  =  p ++> \x → q ++> \y → yield (f x y)
```

- Dann:

```
>  parse (liftP (,) lower digit) "a1"
('a', '1')

>  parse (liftP (,) digit lower) "a1"
Program error: invalid input

>  parse (liftP max lower lower) "nm"
'n'
```

Design einer Parser-Bibliothek

- Wenn wir schon mal beim Abstrahieren sind:

```
mapP :: (a → b) → Parser a → Parser b
mapP f p  =  p ++> \x → yield (f x)
```

- Dann etwa:

```
digitAsInt :: Parser Int
digitAsInt  =  mapP (\d → length ['1' .. d]) digit
```

- Sowie:

```
sat :: (Char → Bool) → Parser Char
sat p  =  item ++> \x → if p x then yield x else failure
```

- Dann etwa:

```
digit :: Parser Char
digit = sat isDigit                         -- isDigit :: Char → Bool
```

```
char :: Char → Parser ()
char x  =  sat (== x) +++ yield ()
```

Parser-Bibliothek: Zwischenbilanz

parse :: Parser a → String → a

item, digit, lower :: Parser Char

yield :: a → Parser a

failure :: Parser a

(| | |) :: Parser a → Parser a → Parser a

(++>) :: Parser a → (a → Parser b) → Parser b

(+++) :: Parser a → Parser b → Parser b

liftP :: (a → b → c) → Parser a → Parser b → Parser c

mapP :: (a → b) → Parser a → Parser b

sat :: (Char → Bool) → Parser Char

Design einer Parser-Bibliothek

- Eigentlich wollen wir im Beispiel ja (unter anderem) natürliche Zahlen parsen.
- Wir könnten wie folgt vorgehen:

nat_1 :: Parser Int
nat_1 = digitAsInt

nat_2 = digitAsInt ++> \\$d_1 \rightarrow$ digitAsInt ++> \\$d_2 \rightarrow$ yield (10 * d_1 + d_2)

nat_3 = digitAsInt ++> \\$d_1 \rightarrow$ nat_2 ++> \\$n_2 \rightarrow$ yield (100 * d_1 + n_2)

nat_4 = liftP (\\d_1 $n_3 \rightarrow$ 1000 * d_1 + n_3) digitAsInt nat_3

…

nat :: Parser Int
nat = nat_9 ||| nat_8 ||| nat_7 ||| nat_6 ||| nat_5 ||| nat_4 ||| nat_3 ||| nat_2 ||| nat_1

- Das ist natürlich nicht wirklich befriedigend!

Design einer Parser-Bibliothek

- Es wäre gut, allgemeine Wiederholungen ausdrücken zu können.

- Also:

```
many :: Parser a → Parser [a]
many p  =  (p ++> \x → many p ++> \xs → yield (x : xs))
           ||| yield [ ]

many₁ :: Parser a → Parser [a]
many₁ p  =  p ++> \x → many p ++> \xs → yield (x : xs)
```

- Oder, äquivalent:

```
many :: Parser a → Parser [a]
many p = many₁ p ||| yield [ ]

many₁ :: Parser a → Parser [a]
many₁ p = liftP (:) p (many p)
```

Parsen arithmetischer Ausdrücke

Nun, zum Parsen natürlicher Zahlen:

1. Versuch:

```
nat :: Parser [Int]
nat = many₁ digitAsInt
```

```
> parse nat "123"
[1, 2, 3]
```

2. Versuch, mit Nachbearbeitung:

```
nat :: Parser Int
nat = mapP (foldl (\n d → 10 * n + d) 0) (many₁ digitAsInt)
```

```
> parse nat "123"
123
```

Parsen arithmetischer Ausdrücke

- Ausgangsspezifikation/Grammatik zum Erkennen arithmetischer Ausdrücke als Ganzes:

```
expr    ::=  term + expr | term
term    ::=  factor * term | factor
factor  ::=  nat | (expr)
```

- Nun, Umsetzung:

```
expr :: Parser ()
expr = term +++ char '+' +++ expr  |||  term

term :: Parser ()
term = factor +++ char '*' +++ term  |||  factor

factor :: Parser ()
factor = nat +++ yield ()  |||  char '(' +++ expr +++ char ')'
```

- Test:

```
> parse expr "2+3*5"
()
```

Parsen arithmetischer Ausdrücke

- Wir wollen natürlich auch die Ergebnisse des Parsens sehen, also:

```
expr :: Parser Expr
expr  =  (term  ++>  \t → char  '+'  +++  expr  ++>  \e → yield (Add t e))
          |||  term

term :: Parser Expr
term  =  (factor  ++>  \f → char  '*'  +++  term  ++>  \t → yield (Mul f t))
          |||  factor

factor :: Parser Expr
factor  =  (nat  ++>  \n → yield (Lit n))
            |||  char  '('  +++  expr  ++>  \e → char ')'  +++  yield e
```

- Oder, unter geeigneter Nutzung der eingeführten Higher-Order Funktionen:

```
expr    =  liftP Add term (char '+' +++ expr)  |||  term
term    =  liftP Mul factor (char '*' +++ term)  |||  factor
factor  =  mapP Lit nat  |||  char '(' +++ expr ++> \e → char ')' +++ yield e
```

Parsen arithmetischer Ausdrücke

- Tests:

```
> parse expr "2+3*5"
Add (Lit 2) (Mul (Lit 3) (Lit 5))
```

```
> parse expr "2*3+5"
Add (Mul (Lit 2) (Lit 3)) (Lit 5)
```

- Also mit:

```
calc :: String → Int
calc s = eval (parse expr s)
```

dann:

```
> calc "2+3*5"
17
```

```
> calc"2*3+5"
11
```

Parser-Bibliothek und Parsen arithmetischer Ausdrücke

„Lektion“:

- allgemeine Parser-Bibliothek (ParserCore.hs, Parser.hs):

type Parser a	parse	item	yield	failure	(|||)	(++>)

(+++)	digit	lower	…	sat	liftP	mapP
many	$many_1$	char				

- darauf aufbauend, Parser für spezifische Anwendungen, z.B. (Calc.hs):

```
import ParserCore
import Parser

data Expr

nat            factor            term            expr
```

Ein zusätzliches Interface zur Parser-Bibliothek

Etwas „(gar nicht so) dunkle Magie“ (MParserCore.hs):

```
import qualified ParserCore

newtype Parser a  =  P (ParserCore.Parser a)
unP :: Parser a → ParserCore.Parser a
unP (P p)  =  p

parse :: Parser a → String → a
parse  =  ParserCore.parse . unP

item :: Parser Char
item  =  P ParserCore.item
…

instance Monad Parser where
  return  =  yield
  (>>=)  =  (++>)
  fail _  =  failure
```

Ein zusätzliches Interface zur Parser-Bibliothek

Nun Verwendung von do-Blöcken möglich (nicht erzwungen), zum Beispiel:

```
term :: Parser Expr
term = do f ← factor
          char '*'
          t ← term
          return (Mul f t)
       ||| factor
```

```
factor :: Parser Expr
factor = mapP Lit nat
         ||| do char '('
                e ← expr
                char ')'
                return e
```

statt:

```
term :: Parser Expr
term  =  (factor  ++>  \f → char  '*'  +++  term  ++>  \t → yield (Mul f t))
         |||  factor

factor :: Parser Expr
factor  =  mapP Lit nat
           |||  char  '('  +++  expr  ++>  \e → char ')'  +++  yield e
```

Ein zusätzliches Interface zur Parser-Bibliothek

- Allgemeine Regeln zur Verwendung von do-Notation für unsere Parser-Bibliothek:

$$\begin{array}{l} \text{do } prs_1 \\ \quad x_2 \leftarrow prs_2 \\ \quad x_3 \leftarrow prs_3 \\ \quad prs_4 \\ \quad x_5 \leftarrow prs_5 \\ \quad \ldots \end{array}$$

Der do-Block insgesamt hat den Typ des letzten prs_n. Zu diesem ist generell kein x_n vorhanden.

wobei die prs_i jeweils einen Typ der Form Parser t_i haben und an die x_i (sofern explizit vorhanden) dann jeweils ein Wert des Typs t_i gebunden wird (und ab dieser Stelle im gesamten do-Block verwendet werden kann), nämlich gerade das durch prs_i gelieferte Ergebnis.

- Es handelt sich tatsächlich nur um „syntaktischen Zucker“. Obiges Beispiel würde automatisch umgewandelt in:

$$prs_1 \text{ +++ } (prs_2 \text{ ++> } (\backslash x_2 \rightarrow prs_3 \text{ ++> } (\backslash x_3 \rightarrow prs_4 \text{ +++ } (prs_5 \text{ ++> } (\backslash x_5 \rightarrow \ldots)))))$$

Ein wichtiger (und nützlicher) Spezialfall

- Häufig tritt folgender Spezialfall auf:

$$\begin{aligned}\text{do } & x_1 \leftarrow prs_1 \\ & x_2 \leftarrow prs_2 \\ & \ldots \\ & x_n \leftarrow prs_n \\ & \text{return } (f\ x_1\ x_2 \ldots x_n)\end{aligned}$$

wobei f eine normale Funktion ist und keines der prs_j von irgendeinem x_i abhängt (und wobei durchaus auch einige x_i ganz weggelassen sein können).

- Dann ist die Vergabe von Namen für die Ergebnisse der prs_i eigentlich überflüssig, nur die Reihenfolge ist wesentlich.

- Für diesen Fall kann dann folgende vereinfachte Form verwendet werden:

$$f <\$> prs_1 <*> prs_2 <*> \ldots <*> prs_n$$

Ein wichtiger (und nützlicher) Spezialfall

- Ermöglicht wird dies durch zusätzlich zur Verfügung stehende Kombinatoren:

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

(<$>) :: (a → b) → Parser a → Parser b

(<*>) :: Parser (a → b) → Parser a → Parser b
```

(sowie Varianten <$, <*, *>, siehe Dokumentation von Control.Applicative), welche durch folgende minimale Definition realisiert sind:

```
instance Functor Parser where
  fmap  =  mapP

instance Applicative Parser where
  pure    =  yield
  (<*>)   =  liftP ($)
```

(restliche Definitionen ergeben sich daraus).

Ein wichtiger (und nützlicher) Spezialfall

- Nun erschließt sich (hoffentlich) auch die ursprünglich angegebene Lösung für das Parsen arithmetischer Ausdrücke:

```
expr   = ( Add <$> term <* char '+' <*> expr ) ||| term

term   = ( Mul <$> factor <* char '*' <*> term ) ||| factor

factor = ( Lit <$> nat ) ||| ( char '(' *> expr <* char ')' )
```

- Denn, zum Beispiel, die erste Zeile steht für:

```
expr = ( pure (\t _ e → Add t e) <*> term <*> char '+' <*> expr )
       ||| term
```

und somit für:

```
expr = do t ← term
          _ ← char '+'
          e ← expr
          return (Add t e)
       ||| term
```

- Zur Erinnerung:

```
> parse (liftP (,) lower digit) "a1"
('a', '1')
```

bzw.:

```
> parse ((,) <$> lower <*> digit) "a1"
('a', '1')
```

- Auch noch erwartbar:

```
> parse (liftP (,) (many1 lower) (many1 digit)) "abc123"
("abc", "123")
```

- Aber, problematisch:

```
> parse (liftP (,) (many1 lower) (many1 lower)) "abcdef"
Program error: invalid input
```

Design der Parser-Bibliothek: ein bisher ignoriertes Problem

- Ursache für:

```
> parse (liftP (,) (many₁ lower) (many₁ lower)) "abcdef"
Program error: invalid input
```

muss irgendwo liegen in:

```
liftP :: (a → b → c) → Parser a → Parser b → Parser c
liftP f p q = p ++> \x → q ++> \y → yield (f x y)
```

```
many₁ :: Parser a → Parser [a]
many₁ p = liftP (:) p (many₁ p ||| yield [ ])
```

```
(|||) :: Parser a → Parser a → Parser a
p ||| q = \inp → case p inp of
                   Just (x, rest)  →  Just (x, rest)
                   Nothing         →  q inp
```

Design der Parser-Bibliothek: ein bisher ignoriertes Problem

- Tatsächliche Ursache:

```
many₁ :: Parser a → Parser [a]
many₁ p = liftP (:) p (many₁ p ||| yield [ ])
```

und

```
(|||) :: Parser a → Parser a → Parser a
p ||| q = \inp → case p inp of
                   Just (x, rest)  →  Just (x, rest)
                   Nothing         →  q inp
```

implizieren „greedy matching“.

- Das heißt, das erste $many_1$ lower in

```
> parse (liftP (,) (many₁ lower) (many₁ lower)) "abcdef"
```

„verbraucht“ die gesamte Eingabe "abcdef".

Design der Parser-Bibliothek: ein bisher ignoriertes Problem

- Vielleicht sollten wir also die Reihenfolge in many_1 vertauschen?

```
many₁ :: Parser a → Parser [a]
many₁ p = liftP (:) p (yield [ ] ||| many₁ p)
```

- Nicht wirklich besser:

```
>  parse (liftP (,) (many₁ lower) (many₁ lower)) "abcdef"
Program error: unused input: cdef
```

- Das eigentliche Problem ist die willkürliche Bevorzugung von p in:

```
(|||) :: Parser a → Parser a → Parser a
p ||| q = \inp → case p inp of
                    Just (x, rest)  →  Just (x, rest)
                    Nothing         →  q inp
```

Design der Parser-Bibliothek: eine Lösung für erkanntes Problem

- Um p und q in

```
(|||) :: Parser a → Parser a → Parser a
p ||| q = \inp → case p inp of
                     Just (x, rest)  →  Just (x, rest)
                     Nothing         →  q inp
```

  gleich zu behandeln, Abkehr von

```
type Parser a = String → Maybe (a, String)
```

  nötig.

- Idee:

```
type Parser a = String → [ (a, String) ]
```

- Dann:

```
(|||) :: Parser a → Parser a → Parser a
p ||| q = \inp → p inp ++ q inp
```

Design der Parser-Bibliothek: eine Lösung für erkanntes Problem

- Natürlich auch Anpassungen anderer Funktionen aus ParserCore.hs nötig:

```
type Parser a     parse     item     yield     failure     (| | |)     (++>)
```

- Einige Änderungen sehr leicht:

```
failure :: Parser a
failure = \inp → Nothing
```

```
failure :: Parser a
failure = \inp → [ ]
```

```
yield :: a → Parser a
yield x = \inp → Just (x, inp)
```

```
yield :: a → Parser a
yield x = \inp → [ (x, inp) ]
```

```
item :: Parser Char
item = \inp → case inp of
                  ""       →  Nothing
                  x : xs   →  Just (x, xs)
```

```
item :: Parser Char
item = \inp → case inp of
                  ""       →  [ ]
                  x : xs   →  [ (x, xs) ]
```

- Andere Anpassungen erfordern etwas mehr Einsicht …

Design der Parser-Bibliothek: eine Lösung für erkanntes Problem

```
(++>) :: Parser a → (a → Parser b) → Parser b
p ++> f = \inp → case p inp of
                   Nothing          →  Nothing
                   Just (x, rest)   →  f x rest
```

```
(++>) :: Parser a → (a → Parser b) → Parser b
p ++> f = \inp → concatMap (\(x, rest) → f x rest) (p inp)
```

```
parse :: Parser a → String → a
parse p inp = case p inp of
                Nothing          →  error "invalid input“
                Just (x, "")     →  x
                Just (_, rest)   →  error ("unused input: " ++ rest)
```

```
parse :: Parser a → String → [a]
parse p inp = [ x | (x, rest) ← p inp, rest == "" ]
```

Design der Parser-Bibliothek: eine Lösung für erkanntes Problem

- Andere Funktionen, in Parser.hs:

(+++)	digit	lower	…	sat	liftP	mapP
many	$many_1$	char				

brauchen nicht angepasst zu werden!

- Nach Ersetzung von ParserCore.hs durch LParserCore.hs, jetzt:

```
> parse (liftP (,) (many1 lower) (many1 lower)) "abcdef"
[("abcde","f"), ("abcd","ef"), ("abc","def"), ("ab","cdef"), ("a","bcdef")]
```

bzw. (bei yield [ ] ||| $many_1$ p statt $many_1$ p ||| yield [ ] in $many_1$):

```
> parse (liftP (,) (many1 lower) (many1 lower)) "abcdef"
[("a","bcdef"), ("ab","cdef"), ("abc","def"), ("abcd","ef"), ("abcde","f")]
```

Zusammenfassung/Übersicht Parser-Bibliothek

- Alternativen ParserCore.hs:

  ```
  type Parser a = String → Maybe (a, String)
  parse     …     (| | |)     (++>)
  ```

  oder LParserCore.hs:

  ```
  type Parser a = String → [ (a, String) ]
  parse     …     (| | |)     (++>)
  ```

- MParserCore.hs (importiert ParserCore.hs oder LParserCore.hs), zur Verwendung von do-Notation, <$>, <*>, …; und macht Parser-Typ abstrakt/opak

- Parser.hs:

  ```
  (+++)   digit   lower   …   sat   liftP   mapP
  many    many1   char
  ```

- Verwendung von LParserCore.hs liefert immer eine Obermenge (oder die gleiche Menge) der Parse-Ergebnisse bei Verwendung von ParserCore.hs.

- Bei „deterministischen Grammatiken“, kein Unterschied!

Zusammenfassung/Übersicht Parser-Bibliothek

- Alternativen ParserCore.hs:

  type Parser a = String → Maybe (a, String)
  parse … (| | |) (++>)

  oder LParserCore.hs:

  type Parser a = String → [(a, String)]

- MParser… r Verwendung von do-N… k

- Parser.h…

  (+++)

  many

- Verwend… er die gleiche Menge) …

- Bei „dete…

Dr. Seuss on Parser Monads:

Haiku

To recurse through lists,
Simply work on beginnings
Until it's the end

Tom Murphy

The Poetry of Errors

I think there's a case that I missed,
For GHC seems to insist,
That when I run main,
it is all in vain:
**** Exception: Prelude.head: empty list*

Wouter Swierstra

Haiku

With strongly typed hands,
I recurse guardingly
in comprehensive repose.

Todd Coram

Mehr in: „Special Poetry and Fiction Edition of The Monad.Reader"