

Praktische Informatik 3: Einführung in die Funktionale Programmierung
Vorlesung vom 26.01.2011: Kombinatoren

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Rev. 1347

1 [39]

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben
 - ▶ Effizient Funktional Programmieren
 - ▶ Fallstudie: Kombinatoren
 - ▶ Eine Einführung in Scala
 - ▶ Rückblick & Ausblick

2 [39]

Kombinatoren im engeren Sinne

Definition (Kombinator)

Ein **Kombinator** ist ein punktfrei definierte Funktion höherer Ordnung.

- ▶ Herkunft: Kombinatorlogik (Schönfinkel, 1924)

$$\begin{aligned}K x y &\triangleright x \\S x y z &\triangleright x z (y z) \\I x &\triangleright x\end{aligned}$$

S, K, I sind Kombinatoren

- ▶ Fun fact #1: kann alle berechenbaren Funktionen ausdrücken
- ▶ Fun fact #2: S und K sind genug: $I = S K K$

3 [39]

Kombinatoren als Entwurfsmuster

- ▶ Kombination von Basisoperationen zu komplexen Operationen
- ▶ Kombinatoren als Muster zur Problemlösung:
 - ▶ Einfache Basisoperationen
 - ▶ Wenige Kombinationsoperationen
 - ▶ Alle anderen Operationen abgeleitet
- ▶ Kompositionalität:
 - ▶ Gesamtproblem läßt sich zerlegen
 - ▶ Gesamtlösung durch Zusammensetzen der Einzellösungen

4 [39]

Beispiele

- ▶ Parserkombinatoren
- ▶ Grafikkombinatoren mit der HGL
- ▶ Grafikkombinatoren mit tinySVG

5 [39]

Beispiel #1: Parser

- ▶ Parser bilden Eingabe auf Parsierungen ab
 - ▶ Mehrere Parsierungen möglich
 - ▶ Backtracking möglich
- ▶ Kombinatoransatz:
 - ▶ Basisparser erkennen Terminalsymbole
 - ▶ Parserkombinatoren zur Konstruktion:
 - ▶ Sequenzierung (erst A , dann B)
 - ▶ Alternierung (entweder A oder B)
 - ▶ Abgeleitete Kombinatoren (z.B. Listen A^* , nicht-leere Listen A^+)

6 [39]

Modellierung in Haskell

Welcher Typ für Parser?

```
type Parse a b = [a] -> [(b, [a])]
```

- ▶ Parametrisiert über Eingabetyp (Token) a und Ergebnis b
- ▶ Parser übersetzt Token in abstrakte Syntax
- ▶ Muss Rest der Eingabe modellieren
- ▶ Muss mehrdeutige Ergebnisse modellieren
- ▶ Beispiel: $"a+b*c" \rightsquigarrow [$
(Var "a", "+b*c"),
(Plus (Var "a") (Var "b"), "*c"),
(Plus (Var "a") (Times (Var "b") (Var "c")), "")]

7 [39]

Basisparser

- ▶ Erkennt nichts:

```
none :: Parse a b  
none = const []
```

- ▶ Erkennt alles:

```
succeed :: b -> Parse a b  
succeed b inp = [(b, inp)]
```

- ▶ Erkennt einzelne Token:

```
spot :: (a -> Bool) -> Parse a a  
spot p [] = []  
spot p (x:xs) = if p x then [(x, xs)] else []
```

```
token :: Eq a => a -> Parse a a  
token t = spot (\c -> t == c)
```

- ▶ Warum nicht none, succeed durch spot? Typ!

8 [39]

Basiskombinatoren: alt, >*>

▶ Alternierung:

- ▶ Erste Alternative wird bevorzugt

```
infixl 3 'alt'
alt :: Parse a b → Parse a b → Parse a b
alt p1 p2 i = p1 i ++ p2 i
```

▶ Sequenzierung:

- ▶ Rest des ersten Parsers als Eingabe für den zweiten

```
infixl 5 >*>
(>*>) :: Parse a b → Parse a c → Parse a (b, c)
(>*>) p1 p2 i =
  concatMap (\(b, r) →
    map (\(c, s) → ((b, c), s)) (p2 r)) (p1 i)
```

9 [39]

Basiskombinatoren: use

▶ Rückgabe weiterverarbeiten:

```
infix 4 'use', 'use2'
use :: Parse a b → (b → c) → Parse a c
use p f i = map (\(o, r) → (f o, r)) (p i)

use2 :: Parse a (b, c) → (b → c → d) → Parse a d
use2 p f = use p (uncurry f)
```

▶ Damit z.B. Sequenzierung rechts/links:

```
infixl 5 >, >*>
(>) :: Parse a b → Parse a c → Parse a c
(>*) :: Parse a b → Parse a c → Parse a b
p1 > p2 = p1 >*> p2 'use' snd
p1 >*> p2 = p1 >*> p2 'use' fst
```

10 [39]

Abgeleitete Kombinatoren

▶ Listen:

$A^+ ::= AA^* | \varepsilon$

```
list :: Parse a b → Parse a [b]
list p = p >*> list p 'use2' (:)
      'alt' succeed []
```

▶ Nicht-leere Listen: $A^+ ::= AA^*$

```
some :: Parse a b → Parse a [b]
some p = p >*> list p 'use2' (:)
      'alt' succeed []
```

- ▶ NB. Präzedenzen: >*> (5) vor use (4) vor alt (3)

11 [39]

Verkapselung

▶ Hauptfunktion:

- ▶ Eingabe muß vollständig parsiert werden
- ▶ Auf Mehrdeutigkeit prüfen

```
parse :: Parse a b → [a] → Either String b
parse p i =
  case filter (null . snd) $ p i of
    [] → Left "Input does not parse"
    [(e, _)] → Right e
    _ → Left "Input is ambiguous"
```

▶ Schnittstelle:

- ▶ Nach außen nur Typ Parse sichtbar, plus Operationen darauf

12 [39]

Grammatik für Arithmetische Ausdrücke

```
Expr ::= Term + Term | Term
Term  ::= Factor * Factor | Factor
Factor ::= Variable | (Expr)
Variable ::= Char+
Char ::= a | ... | z | A | ... | Z
```

13 [39]

Abstrakte Syntax für Arithmetische Ausdrücke

▶ Zur Grammatik **abstrakte Syntax**

```
data Expr = Plus Expr Expr
          | Times Expr Expr
          | Var String
```

- ▶ Hier Unterscheidung Term, Factor, Number unnötig.

14 [39]

Parsierung Arithmetischer Ausdrücke

- ▶ Token: Char
- ▶ Parsierung von Factor

```
pFactor :: Parse Char Expr
pFactor = some (spot isAlpha) 'use' Var
      'alt' token '(' *> pExpr >*> token ')'
```

▶ Parsierung von Term

```
pTerm :: Parse Char Expr
pTerm =
  pFactor >*> token '*' >*> pFactor 'use2' Times
  'alt' pFactor
```

▶ Parsierung von Expr

```
pExpr :: Parse Char Expr
pExpr = pTerm >*> token '+' >*> pTerm 'use2' Plus
      'alt' pTerm
```

15 [39]

Die Hauptfunktion

- ▶ Lexing: Leerzeichen aus der Eingabe entfernen

```
parseExpr :: String → Expr
parseExpr i =
  case parse pExpr (filter (not.isSpace) i) of
    Right e → e
    Left err → error err
```

16 [39]

Ein kleiner Fehler

- ▶ **Mangel:** $a+b+c$ führt zu Syntaxfehler — Fehler in der Grammatik

- ▶ Behebung: **Änderung** der Grammatik

```
Expr ::= Term + Expr | Term
Term  ::= Factor * Term | Factor
Factor ::= Variable | (Expr)
Variable ::= Char+
Char ::= a | ... | z | A | ... | Z
```

- ▶ Abstrakte Syntax bleibt

17 [39]

Änderung des Parsers

- ▶ Entsprechende Änderung des Parsers in pTerm

```
pTerm :: Parse Char Expr
pTerm =
  pFactor >* token '*' >*> pTerm 'use2' Times
  'alt' pFactor
```

- ▶ ... und in pExpr:

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pExpr 'use2' Plus
  'alt' pTerm
```

- ▶ pFactor und Hauptfunktion bleiben.

18 [39]

Zusammenfassung Parserkombinatoren

- ▶ **Systematische Konstruktion** des Parsers aus der Grammatik.
- ▶ **Kompositional:**
 - ▶ Lokale Änderung der Grammatik führt zu lokaler Änderung im Parser
 - ▶ Vgl. Parsergeneratoren (yacc/bison, antlr, happy)
- ▶ Struktur von Parse zur Benutzung irrelevant
 - ▶ Vorsicht bei **Mehrdeutigkeiten** in der Grammatik (Performance-Fälle)
 - ▶ Einfache Implementierung (wie oben) skaliert nicht
 - ▶ Effiziente Implementation mit **gleicher Schnittstelle** auch für **große** Eingaben geeignet.

19 [39]

Beispiel #2: Die Haskell Graphics Library HGL

- ▶ **Kompakte Grafikbücherei** für einfache Grafiken und Animationen.
- ▶ **Gleiche Schnittstelle** zu X Windows (X11) und Microsoft Windows.
- ▶ Bietet:
 - ▶ Fenster
 - ▶ verschiedene Zeichenfunktionen
 - ▶ Unterstützung für Animation

20 [39]

Übersicht HGL

- ▶ **Grafiken**

```
type Graphic
```

- ▶ **Atomare Grafiken:**

- ▶ Ellipsen, Linien, Polygone, ...

- ▶ **Modifikation mit Attributen:**

- ▶ Pinsel, Stifte und Textfarben
- ▶ Farben

- ▶ **Kombination** von Grafiken

- ▶ Überlagerung

21 [39]

Basisdatentypen

- ▶ Winkel (Grad, nicht Bogenmaß)

```
type Angle = Double
```

- ▶ Dimensionen (Pixel)

```
type Dimension = Int
```

- ▶ Punkte (Ursprung: links oben)

```
type Point = (Dimension, Dimension)
```

22 [39]

Atomare Grafiken (1)

- ▶ **Ellipse** (gefüllt) innerhalb des gegebenen Rechtecks

```
ellipse :: Point -> Point -> Graphic
```

- ▶ **Ellipse** (gefüllt) innerhalb des Parallelograms:

```
shearEllipse :: Point -> Point -> Point -> Graphic
```

- ▶ **Bogenabschnitt** einer Ellipse (math. positiven Drehsinn):

```
arc :: Point -> Point -> Angle -> Angle -> Graphic
```

23 [39]

Atomare Grafiken (2)

- ▶ **Strecke, Streckenzug:**

```
line :: Point -> Point -> Graphic
polyline :: [Point] -> Graphic
```

- ▶ **Polygon** (gefüllt)

```
polygon :: [Point] -> Graphic
```

- ▶ **Text:**

```
text :: Point -> String -> Graphic
```

- ▶ **Leere Grafik:**

```
emptyGraphic :: Graphic
```

24 [39]

Modifikation von Grafiken

- ▶ Andere **Fonts**, **Farben**, Hintergrundfarben, ...:

```
withFont      :: Font    → Graphic → Graphic
withTextColor :: RGB     → Graphic → Graphic
withBkColor   :: RGB     → Graphic → Graphic
withBkMode    :: BkMode  → Graphic → Graphic
withPen       :: Pen     → Graphic → Graphic
withBrush     :: Brush   → Graphic → Graphic
withRGB       :: RGB     → Graphic → Graphic
withTextAlign :: Alignment → Graphic → Graphic
```

25 [39]

Farben

- ▶ Nützliche Abkürzung: benannte Farben

```
data Color = Black | Blue | Green | Cyan | Red
           | Magenta | Yellow | White
  deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)
```

- ▶ Benannte Farben sind einfach `colorTable :: Array Color RGB`

- ▶ Dazu Modifikator:

```
withColor :: Color → Graphic → Graphic
withColor c = withRGB (colorTable ! c)
```

26 [39]

Kombination von Grafiken

- ▶ Überlagerung (erste über zweiter):

```
overGraphic :: Graphic → Graphic → Graphic
```

- ▶ Verallgemeinerung:

```
overGraphics :: [Graphic] → Graphic
overGraphics = foldr overGraphic emptyGraphic
```

27 [39]

Fenster

- ▶ **Elementare** Funktionen:

```
getGraphic :: Window → IO Graphic
setGraphic :: Window → Graphic → IO ()
```

- ▶ **Abgeleitete** Funktionen:

- ▶ In Fenster zeichnen:

```
drawInWindow :: Window → Graphic → IO ()
drawInWindow w g = do
  old ← getGraphic w
  setGraphic w (g `overGraphic` old)
```

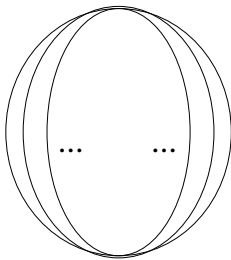
- ▶ Grafik löschen

```
clearWindow :: Window → IO ()
clearWindow w = setGraphic w emptyGraphic
```

28 [39]

Ein einfaches Beispiel: der Ball

- ▶ **Ziel**: einen gestreiften Ball zeichnen
- ▶ **Algorithmus**: als Folge von konzentrischen Ellipsen
 - ▶ Start mit Eckpunkten (x_1, y_1) und (x_2, y_2) .
 - ▶ Verringerung von x um Δ_x , y bleibt gleich.
 - ▶ Dabei Farbe verändern.



29 [39]

Ein einfaches Beispiel: Der Ball

- ▶ Liste aller Farben `cols`
- ▶ Listen der x -Position (y -Position ist konstant), $\Delta_x = 25$
- ▶ Hilfsfunktion `drawEllipse`

```
drawBall :: Point → Point → Graphic
drawBall (x1, y1) (x2, y2) =
  let cols = cycle [Red, Green, Blue]
      midx = (x2 - x1) `div` 2
      xls = [x1, x1 + 25 .. midx]
      xrs = [x2, x2 - 25 .. midx]
      drawEllipse c xl xr = withColor c $
                            ellipse (xl, y1) (xr, y2)
      gs = zipWith3 drawEllipse cols xls xrs
  in overGraphics (reverse gs)
```

30 [39]

Ein einfaches Beispiel: Der Ball

- ▶ Hauptprogramm (**Zeigen**)

```
main :: IO ()
main = runGraphics $ do
  w ← openWindow "Balls!" (500, 500)
  drawInWindow w $ drawBall (25, 25) (485, 485)
  getKey w
  closeWindow w
```

31 [39]

Animation

Alles dreht sich, alles bewegt sich...

- ▶ Animation: über der Zeit veränderliche Grafik
- ▶ Unterstützung von Animationen in HGL:
 - ▶ Timer ermöglichen getaktete Darstellung
 - ▶ Gepufferte Darstellung ermöglicht flickerfreie Darstellung
- ▶ Öffnen eines Fensters mit Animationsunterstützung:
 - ▶ Initiale Position, Grafikzwischenpuffer, Timer-Takt in Millisekunden

```
openWindowEx :: Title → Maybe Point → Size →
              RedrawMode → Maybe Time → IO Window
data RedrawMode
  = Unbuffered | DoubleBuffered
```

32 [39]

Der springende Ball

- ▶ Ball hat Position und Geschwindigkeit:

```
data Ball = Ball { p :: Point,
                  v :: Point }
```

- ▶ Ball zeichnen: Roter Kreis an Position \vec{p}

```
drawBall :: Ball → Graphic
drawBall (Ball {p= p}) =
  withColor Red (circle p 15)
```

- ▶ Kreis zeichnen:

```
circle :: Point → Int → Graphic
circle (px, py) r = ellipse (px- r, py- r) (px+ r, py+ r)
```

33 [39]

Bewegung des Balles

- ▶ Geschwindigkeit \vec{v} zu Position \vec{p} addieren
- ▶ In X-Richtung: modulo Fenstergröße 500
- ▶ In Y-Richtung: wenn Fensterrand 500 erreicht, Geschwindigkeit invertieren
- ▶ Geschwindigkeit in Y-Richtung nimmt immer um 1 ab

```
move (Ball {p= (px, py), v= (vx, vy)}) =
  Ball {p= (px', py'), v= (vx, vy')} where
  px' = (px+ vx) `mod` 500
  py0 = py+ vy
  py' = if py0 > 500 then 500-(py0-500) else py0
  vy' = (if py0 > 500 then -vy else vy)+ 1
```

34 [39]

Der springende Ball

- ▶ Hauptschleife: Ball zeichnen, auf Tick warten, Folgeposition

```
loop w b =
  do setGraphic w (drawBall b)
     getWindowTick w
     loop w (move b)
```

- ▶ Hauptprogram: Fenster öffnen, Starten der Hauptschleife

```
main = runGraphics $
  do w ← openWindowEx "Bounce!" Nothing (500, 500)
     DoubleBuffered (Just 30)
     loop w (Ball {p=(0, 10), v= (5, 0)})
```

35 [39]

Zusammenfassung HGL

- ▶ Abstrakte und portable **Grafikprogrammierung**
- ▶ **Verkapselung** von **systemnaher** Schnittstelle durch Kombinatoren
- ▶ Kombinatoransatz: Kombination **elementarer** Grafiken zu komplexen Grafikprogrammen
- ▶ Rudimentäre Unterstützung von **Animation** durch Timer und Puffer
- ▶ Kombinatoransatz hier:

```
type Time = Int
type Animation = Int → Graphic
```

36 [39]

Beispiel #3: tinySVG

- ▶ **Scalable Vector Graphics (SVG)**

- ▶ XML-Standard für Vektorgrafiken
- ▶ Unterstützt Vektorgrafiken (Pfade), Rastergrafiken und Text
- ▶ Ein Kreis: `<circle x="20" y="30" r="50" />`

- ▶ Übersetzung in Kombinatorbücherei in Haskell (**tinySVG**):

- ▶ Elementare Operationen:

```
circle :: Point → Double → Graphics
line   :: Point → Point → Graphics
```

- ▶ Kombinatoren zum Transformieren:

```
group :: [Graphics] → Graphics
rotate :: Double → Graphics → Graphics
scale  :: Double → Graphics → Graphics
```

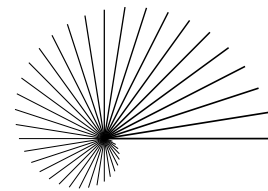
- ▶ Ausgabe:

```
toXML :: Double → Double → Graphics → String
```

37 [39]

tinySVG in Aktion

```
snailShell :: Int → Graphics
snailShell n =
  let rs = [fromInt i / fromInt n | i ← [1..n]]
      theLine = line (pt' 0 0) (pt' 100 0)
      lines = [rotate (r*360) $ scale r theLine
               | r ← rs]
  in group lines
```



38 [39]

Zusammenfassung

- ▶ **Kombinatoransatz:**

- ▶ Einfache Basisoperationen
- ▶ Wenige Kombinationsoperatoren
- ▶ Ideal in funktionalen Sprachen, generell nützlich

- ▶ **Parserkombinatoren:**

- ▶ Von Grund auf in Haskell
- ▶ Kombinatoren abstrahieren über Implementation

- ▶ **Grafik mit HGL:**

- ▶ Verkapselung von **Systemschnittstellen**
- ▶ Kombinatoren abstrahieren **Systemschnittstellen**

- ▶ **Grafik mit tinySVG:**

- ▶ Kombinatoren abstrahieren über **XML-Syntax**

39 [39]