## Heutiges Thema

**Pretty Printing**
...neben lexikalischer und syntaktischer Analyse eine weitere typische Demonstrationsanwendung funktionalen Programmierstils.



## Worum geht es?

Ein *pretty printer* ist...

- ein Werkzeug (oft in Form einer Bibliothek von Routinen) zur Ausgabe baumartiger Datenstrukturen als Text

Wesentliche Ziele...

- minimale Anzahl von Zeilen bei Wahrung und Veranschaulichung der Baumstruktur, etwa durch geeignete Einrückungen



## "Gute" Pretty-Printer

...zeichnen sich dadurch aus, in ausgewogener Weise folgende Eigenschaften zu vereinen

- Einfachheit der Benutzung
- Flexibilität des Formats
- "Optimalität" des Ergebnisses



## Literaturhinweis

Die folgende Darstellung orientiert sich an...

- Phil Wadler. *A Prettier Printer*. In Jeremy Gibbons, Oege de Moor (Hrsg.), *The Fun of Programming*. Palgrave MacMillan, 2003.



## Besonderes Kennzeichen

...des von Wadler entwickelten "Prettier Printer":

- Genau eine Möglichkeit zur Konkatenation von Dokumenten, die
  - assoziativ
  - mit Links- und Rechtseins

  ist.



## Warum "hübscher" statt "hübsch"?

Wadlers "Prettier Printer" versteht sich als Verbesserung der von John Hughes in

- *The design of a pretty-printer library*. In Johan Jeuring, Erik Meijers (Hrsg.), *Advanced Functional Programming*, LNCS 925, Springer, 1995.

vorgestellten Bibliothek mit folgenden Charakteristika:

- Zwei Varianten zur Dokumentkonkatenation, davon
  - eine Variante ohne Einselement
  - eine Variante (nur) mit Rechtseins
- ca. 40% mehr Code, ca. 40% langsamer als Wadlers Vorschlag



## Pretty Printer – Das Basissystem

*Kennzeichen*: Für jedes Dokument gibt es nur ein mögliches Layout.

Die dazu erforderlichen *Basisfunktionen* sind:

```
(<>)   :: Doc -> Doc -> Doc   -- ass. Konkatenation
nil    :: Doc                 -- Rechts- und Linkseins von (<>)
text   :: String -> Doc       -- Konversionsfunktion
line   :: Doc                 -- Zeilenvorschub
nest   :: Int -> Doc -> Doc   -- Einrueckung
layout :: Doc -> String       -- Ausgabe
```

*Vereinbarung*:

- Argumente für `text` frei von Zeichen für Zeilenumbruch



## Eine einfache Implementierung

Implementiere...

- `doc` als Zeichenreihen (d.h. als `String`)

Damit wird...

- `(<>)` ...Zeichenreihenkonkatenation
- `nil` ...leere Zeichenreihe
- `text` ...Identität auf Zeichenreihen
- `line` ...neue Zeile
- `nest i` ...$i$ Leerzeichen Einrückung (nach jedem Zeilenvorschub mittels `line`)
- `layout` ...Identität auf Zeichenreihen

## Beispiel

...zur Umwandlung von Bäumen in Dokumente (hier: `Strings`) und deren Ausgabe als Text (hier: `Strings`).

Betrachte folgenden Baumtyp:

```
data Tree = Node String [Tree]
```

Ein konkreter Baum `B`...

```
Node "aaa" [Node "bbbbb" [Node "cc" [], Node "dd" []],
            Node "eee" [],
            Node "ffff" [Node "gg" [],
                         Node "hhh" [],
                         Node "ii" []
                        ]
           ]
```



## Gewünschte Ausgabe

...von `B`:

```
aaa[bbbbb[ccc,
          dd],
    eee,
    ffff[gg,
         hhh,
         ii]]
```



## Realisierung

```
data Tree           = Node String [Tree]

showTree :: Tree -> Doc
showTree (Node s ts) = text s <> nest (length s) (showBracket ts)

showBracket :: [Tree] -> Doc
showBracket []      = nil
showBracket ts      = text "[" <> nest 1 (showTrees ts)
                                      <> text "]"
showTree :: [Tree] -> Doc
showTrees [t]       = showTree t
showTrees (t:ts)    = showTree t <> text "," <> line
                                  <> showTrees ts
```



## Alternativ gewünschte Ausgabe von `B`

```
aaa[
  bbbbb[
    ccc,
    dd
  ],
  eee,
  ffff[
    gg,
    hhh,
    ii
  ]
]
```



## Realisierung der alternativen Ausgabe

```
data Tree            = Node String [Tree]

showTree' :: Tree -> Doc
showTree' (Node s ts) = text s <> showBracket' ts

showBracket' :: [Tree] -> Doc
showBracket' []      = nil
showBracket' ts      = bracket "[" (showTrees' ts) "]"

showTree' :: [Tree] -> Doc
showTrees' [t]       = showTree t
showTrees' (t:ts)    = showTree t <> text "," <> line
                                   <> showTrees ts
```



## Normalform von Dokumenten

Normalform...

- alternierende Abfolge von Text und Zeilenumbrüchen mit vorgegebener Einrückung

```
text s0 <> nest i1 line <> text s1 <> ...
                             <> nest ik line <> text sk
```

Es gilt:

- Dokumente können stets auf *Normalform* reduziert werden



## Beispiel zur Normalform 1(3)

Das Dokument...

```
text "bbbbb" <> text "[" <>
nest 2 (
     line <> text "ccc" <> text "," <>
     line <> text "dd"
) <>
line <> text "]"
```



## Beispiel zur Normalform 2(3)

...hat die Normalform:

```
text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"
```

## Beispiel zur Normalform 3(3)

…und wird somit wie folgt ausgegeben:

```
bbbbb[
  ccc,
  dd
]
```



## Eigenschaften d. Funktionen – Gesetze

Es gilt:

```
text (s ++ t)    = text s <> text t      (text Homomorphismus von
text ""          = nil                    Zeichenreihen- auf
                                          Dokumentkonkatenation)

nest (i+j) x     = nest i (nest j x)     (nest Homomorphismus von
nest 0 x         = x                      Addition auf Komposition)

nest i (x <> y) = nest i x <> nest i y (nest distributiert ueber
nest i nil       = nil                    Dokumentkonkatenation)

nest i (text s) = text s         (Text absorbiert Einrueckungen)
```

*Bedeutung*

- Hinreichend dafür, Dokumente stets in Normalform überführen zu können (dafür: erste vier Gesetze: Anwendung von links nach rechts; letzte drei Gesetze: Anwendung von rechts nach links)

## Weitere Eigenschaften – Gesetze

…über den Zusammenhang von Dokumenten und ihren Druckdarstellungen

```
layout (x <> y)      = layout x ++ layout y (layout Homomorphismus
layout nil           = ""                     von Dokument- auf
                                              Zeichenreihenkonka-
                                              tenation)

layout (text s)      = s                    (layout invers zu text)

layout (nest i line) = '\n' : copy i ' '
```



## Die Implementierung von Doc

…als Summentyp:

```
data Doc              = Nil
                      | String 'Text' Doc
                      | Int 'Line' Doc
```

…und der Zusammenhang zu den Dokumentoperatoren:

```
Nil        = nil
s 'Text' x = text s <> x
i 'Line' x = nest i line <> x
```



## Beispiel

Die (weiter oben schon betrachtete) Normalform…

```
text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"
```

…hat die Darstellung:

```
"bbbbb[" 'Text' (
2 'Line' ("ccc," 'Text' (
2 'Line' ("dd," 'Text' (
0 'Line' ("]," 'Text' Nil)))))
```



## Abgeleitete Implementierungen 1(2)

…der Dokumentoperationen:

```
nil                = Nil
text s             = s 'Text' Nil
line               = 0 'Line' Nil

(s 'Text' x) <> y  = s 'Text' (x <> y)
(i 'Line' x) <> y  = i 'Line' (x <> y)
Nil <> y           = y
```



## Abgeleitete Implementierungen 2(2)

```
nest i (s 'Text' x) = s 'Text' nest i x
nest i (j 'Line' x) = (i+j) 'Line' nest i x
nest i Nil = Nil

layout (s 'Text' x) = s ++ layout x
layout (i 'Line' x) = '\n' : copy i ' ' ++ layout x
layout Nil          = ""
```



## Zur Korrektheit

…der abgeleiteten Implementierungen:

- Herleitung von `(s 'Text' x) <> y = s 'Text' (x <> y)`

```
  (s 'Text' x) <> y
=   { Definition Text }
  (text s <> x) <> y
=   { Assoziativitaet von <> }
  text s <> (x <> y)
=   { Definition Text }
  s 'Text' (x <> y)
```

- Übrige Gleichungen: Herleitung analog

## Dokumente mit mehreren Layouts

- *Bislang*... Dokumente äquivalent zu einer Zeichenreihe
- *Jetzt*... Dokumente äquivalent zu einer Menge von Zeichenreihen

wobei eine Zeichenreihe jeweils einem Layout entspricht.

Zentral:

```
group :: Doc -> Doc
```

*Informell*:
...retourniert ein zusätzliches Element, das in einer Zeile ausgegeben wird



## Bevorzugte Layouts

- `layout` wird durch `pretty` ersetzt

  ```
  pretty :: Int -> Doc -> String
  ```

- `pretty`'s Integer-Argument spezifiziert die bevorzugte maximale Zeilenlänge der Ausgabe



## Anwendungsbeispiel

Mit...

```
showTree (Node s ts) = group (text s
                    <> nest (length s) (showBracket ts))
```

...führt der Aufruf von `pretty 30` zur Ausgabe:

```
aaa[bbbbb[ccc, dd],
    eee,
    ffff[gg, hhh, ii]]
```

Dabei ist gewährleistet

- Ausgabe in einer Zeile wo möglich (d.h. Länge $\leq 30$)
- Einfügen ausreichend vieler Zeilenumbrüche, um die vorgegebene maximale Zeilenlänge nirgends zu überschreiten



## Implementierung der neuen Funktionen

Folgende Hilfsfunktionen werden zunächst eingeführt:

```
-- Vereinigung zweier Mengen von Layouts
(<|>)   :: Doc -> Doc -> Doc
-- Ersetzen jedes Zeilenvorschubs (inclusive
-- anschliessendem Einrueckens) durch einen Zwischenraum
flatten :: Doc -> Doc
```

- *Beobachtung* ...Dokumente repräsentieren stets eine nicht-leere Menge von Layouts
- *Forderungen*
  - ...in `(x <|> y)` besitzen alle layouts von `x` und `y` dasselbe flache Layout
  - ...jede erste Zeile in `x` ist mindestens so lang wie jede erste Zeile in `y`



## Eigenschaften von `(<|>)`

```
(x <|> y) <> z   = (x <> z) <|> (y <> z)
x <> (y <|> z)   = (x <> y) <|> (x <> z)
nest i (x <|> y) = nest i x <|> nest i y
```



## Eigenschaften von `flatten`

```
flatten (x <|> y) = flatten x

flatten (x <> y)  = flatten x <> flatten y
flatten nil       = nil
flatten (text s)  = text s
flatten line      = text " "
flatten (nest i x) = flatten x
```



## Implementierung von `group`

...mittels `flatten` und `(<>)`

```
group x = flatten x <|> x
```



## Normalform

Mit vorigen Festlegungen lässt sich jedes Dokument zu folgender *Normalform* reduzieren

```
x1 <|> ... <|> xn
```

wobei jedes `xi` in der (früher eingeführten) Normalform einfacher Dokumente ist.

## Auswahl eines "besten" Layouts

...durch Festlegung einer Ordnungsrelation über Zeilen in Abhängigkeit der vorgegebenen maximalen Zeilenlänge

Von zwei Zeilen...

- die die maximale Länge nicht überschreiten, ist die längere vorzuziehen
- von denen mindestens eine die maximale Länge überschreitet, ist die kürzere vorzuziehen



## Die adaptierte Implementierung von `Doc`

```
data Doc = -- Die ersten 3 Alternativen wie bisher
           Nil
         | String 'Text' Doc
         | Int 'Line' Doc
           -- Neu kommt hinzu
         | Doc 'Union' Doc
```



## Zusammenhang der Konstruktoren und Dokumentoperatoren

```
Nil          = nil
s 'Text' x   = text s <> x
i 'Line' x   = nest i line <> x
x 'Union' y  = x <|> y
```



## Beispiel 1(8)

Das Dokument...

```
group(
    group(
        group(
            group( text "hello" <> line <> text "a")
         <> line <> text "b")
     <> line <> text "c")
 <> line <> text "d")
```



## Beispiel 2(8)

...besitzt die Darstellungen (layouts)

```
hello a b c d    hello a b c    hello a b    hello a    hello
                 d              c            b          a
                                d            c          b
                                             d          c
                                                        d
```



## Beispiel 3(8)

*Aufgabe*: ...gib obiges Dokument aus, wobei die maximale Zeilenlänge 5 betrage

⇝...die rechte Darstellung der vorigen Folie ist gefordert

*Vorüberlegungen*:

- ...Ausklammern von `"hello"` für `x` und `y` geeignet

```
"hello" 'Text' ((" " 'Text' x) 'Union' (0 'Line' y))
```

- ...Festlegen des Zusammenspiels von `(<>)` und `nest` mit `Union`

```
(x 'Union' y) <> z   = (x <> z) 'Union' (y <> z)
nest k (x 'Union' y) = nest k x 'Union' nest k y
```



## Beispiel 4(8)

Implementierungen von `group` und `flatten`

```
group Nil             = Nil
group (i 'Line' x)    = (" " 'Text' flatten x) 'Union'
                                               (i 'Line' x)
group (s 'Text' x)    = s 'Text' group x
group (x 'Union' y)   = group x 'Union' y

flatten Nil           = Nil
flatten (i 'Line' x)  = " " 'Text' flatten x
flatten (s 'Text' x)  = s 'Text' flatten x
flatten (x 'Union' y) = flatten x
```



## Beispiel 5(8)

Überlegungen zur Korrektheit...

Herleitung von `group (i 'Line' x)` (vgl. Zeile zwei)

```
  group (i 'Line' x)
=   { Definition Line }
  group (nest i line <> x)
=   { Definition group}
  flatten (nest i line <> x) <|> (nest i line s <> x)
=   { Definition flatten }
  (text " " <> flatten x) <|> (nest i line <> x)
=   { Definition Text, Union, Line }
  (" " 'Text' flatten x) 'Union' (i 'Line' x)
```

## Beispiel 6(8)

Überlegungen zur Korrektheit...

Herleitung von `group (s 'Text' x)` (vgl. Zeile drei)

```
  group (s 'Text' x)
=   { Definition Text }
  group (text s <> x)
=   { Definition group}
  flatten (text s <> x) <|> (text s <> x)
=   { Definition flatten }
  (text s <> flatten x) <|> (text s <> x)
=   { <> distributiert ueber <|> }
  text s <> (flatten x <|> x)
=   { Definition group }
  text s <> group x
=   { Definition Text }
  s 'Text' group x
```



## Beispiel 7(8)

Auswahl des "besten" Layouts...

```
best w k Nil          = Nil
best w k (i 'Line' x)  = i 'Line' best w i x
best w k (s 'Text' x)  = s 'Text' best w (k + length s) x
best w k (x 'Union' y) = better w k (best w k x) (best w k y)

better w k x y         = if fits (w-k) x then x else y
```

*Bemerkung*:

- `best` ...überführt ein "union"-behaftetes Dokument in ein "union"-freies Dokument
- Argument `w` ...maximale Zeilenlänge
- Argument `k` ...bereits verbrauchte Zeichen (einschließlich Einrückung) auf aktueller Zeile



## Beispiel 8(8)

Überprüfung, ob die erste Dokumentzeile innerhalb der maximalen Zeilenlänge bleibt...

```
fits w x | w<0      = False
fits w Nil          = True
fits w (s 'Text' x) = fits (w - length s) x
fits w (i 'Line' x) = True
```

Zu guter letzt, die Ausgaberoutine...

```
pretty w x          = layout (best w 0 x)
```



## Eine effizientere Variante

...mittels einer neuen Implementierung von Dokumenten

```
data DOC = NIL
         | DOC :<> DOC
         | NEST Int DOC
         | TEXT String
         | LINE
         | DOC :<|> DOC
```

*Bemerkung*:

- Zur Unterscheidung vom vorherigen Dokumententyp werden hier Großbuchstaben verwandt



## Implementierung der Dokumentenoperatoren

```
nil      = NIL
x <> y   = x :<> y
nest i x = NEST i x
text s   = TEXT s
line     = LINE
```



## Implementierungen von `group` und `flatten`

Wie bisher, wird gefordert:

- ...in `(x :<|> y)` besitzen alle layouts von `x` und `y` dasselbe flache Layout
- ...jede erste Zeile in `x` ist mindestens so lang wie jede erste Zeile in `y`

```
group x            = flatten x :<|> x

flatten NIL        = NIL
flatten (x :<> y)  = flatten x:<> flatten y
flatten (NEST i x) = NEST i (flatten x)
flatten (TEXT s)   = TEXT s
flatten LINE       = TEXT " "
flatten (x :<|> y) = flatten x
```



## Repräsentationsfunktion

...erstellen des Dokuments aus einem einrückungsbehafteten Dokuments

```
rep z = fold (<>) nil [nest i x | (i,x) <- z ]
```



## Auswahl des "besten" Layouts

Verallgemeinerung der Funktion "best"...

```
be w k z = best w k (rep z)     (Hypothese)

best w k x                = be w k [(0,x)]
```

wobei...

```
be w k []                 = Nil
be w k ((i,NIL):z)        = be w k z
be w k ((i,x :<> y) : z)  = be w k ((i,x) : (i,y) : z)
be w k ((i,NEST j x) : z) = be w k ((i+j),x) : z)
be w k ((i,TEXT s) : z)   = s 'Text' be w (k+length s) z
be w k ((i,LINE) : z)     = i 'Line' be w i z
be w k ((i.x :<|> y) : z) = better w k (be w k ((i.x) : z))
                                       (be w k (i,y) : z))
```

## In Vorbereitung weiterer Anwendungen 1(3)

…zunächst einige nützliche Hilfsfunktionen

```
x <+> y                  = x <> text " " <> y
x </> y                  = x <> line <> y

folddoc f []             = nil
folddoc f [x]            = x
folddoc f (x:xs)         = f x (folddoc f xs)

spread                   = folddoc (<+>)
stack                    = folddoc (</>)
```



## In Vorbereitung weiterer Anwendungen 2(3)

…weitere Hilfsfunktionen

```
-- Haeufig wiederkehrendes Ausgabemuster
bracket l x r             = group (text l <>
                                   nest 2 (line <> x) <>
                                   line <> text r)

-- Abkuerzung der alternativen Baumlayoutfunktion
showBracket' ts           = bracket "[" (showTrees' ts) "]"

-- Auffuellen von Zeilen (mit words aus Haskells Standardbib.)
x <+/> y                  = x <> (text " " :<|> line) <> y
fillwords                 = folddoc (<+/>) . map text . words
```



## In Vorbereitung weiterer Anwendungen 3(3)

`fill`, eine Variante von `fillwords`
⇝ …kollabiert eine Dokumentenliste zu einem Dokument

```
fill []         = nil
fill [x]        = x
fill (x:y:zs)   = (flatten x <+> fill (flatten y : zs)) :<|>
                    (x </> fill (y : zs))
```



## Anwendung 1(2)

Ausgabe von XML-Dokumenten (vereinfachte Syntax)…

```
data XML              = Elt String [Att] [XML]
                      | Txt String

data Att              = Att String String

showXML x             = folddoc (<>) (showXMLs x)

showXMLs (Elt n a []) = [text "<" <> showTag n a <> text "/>"
showXMLs (Elt n a c)  = [text "<" <> showTag n a <> text ">" <>
                         showFill showXMLs c <>
                         text "</" <> text n <> text ">"]
showXMLs (Txt s)      = map text (words s)

showAtts (Att n v)    = [text n <> text "=" <> text (quoted v)]
```



## Anwendung 2(2)

Fortsetzung…

```
quoted s      = "\"" ++ s ++ "\""

showTag n a   = text n <> showFill showAtts a

showFill f [] = nil
showFill f xs = bracket "" (fill (concat (map f xs))) ""
```



## Ausgabebeispiel 1

…für vorgegebene maximale Zeilenlänge von 30 Zeichen

```
<p
  color="red" font="Times"
  size="10"
>
  Here is some
  <em> emphasized </em> text.
  Here is a
  <a
    href="http://www.eg.com/"
  > link </a>
  elsewhere.
</p>
```



## Ausgabebeispiel 2

…für vorgegebene maximale Zeilenlänge von 60 Zeichen

```
<p color="red" font="Times" size="10" >
  Here is some <em> emphasized </em> text. Here is a
  <a href="http://www.eg.com/" > link </a> elsewhere.
</p>
```



## Ausgabebeispiel 3

…nach Weglassen von `flatten` in `fill`

```
<p color="red" font="Times" size="10" >
  Here is some <em>
    emphasized
  </em> text. Here is a <a
    href="http://www.eg.com/"
  > link </a> elsewhere.
</p>
```

## Der Code im Zusammenhang 1(11)

*Quelle*: Phil Wadler. *A Prettier Printer*. In Jeremy Gibbons, Oege de Moor (Hrsg.), *The Fun of Programming*. Palgrave MacMillan, 2003.

```
-- The pretty printer
infixr 5:<|>
infixr 6:<>
infixr 6 <>

data DOC                = NIL
                        | DOC :<> DOC
                        | NEST Int DOC
                        | TEXT String
                        | LINE
                        | DOC :<|> DOC

data Doc                = Nil
                        | String 'Text' Doc
                        | Int 'Line' Doc
```



## Der Code im Zusammenhang 2(11)

```
nil                     = NIL
x <> y                  = x :<> y
nest i x                = NEST i x
text s                  = TEXT s
line                    = LINE

group x                 = flatten x :<|> x

flatten NIL             = NIL
flatten (x :<> y)       = flatten x:<> flatten y
flatten (NEST i x)      = NEST i (flatten x)
flatten (TEXT s)        = TEXT s
flatten LINE            = TEXT " "
flatten (x :<|> y)      = flatten x
```



## Der Code im Zusammenhang 3(11)

```
layout Nil              = ""
layout (s 'Text' x)     = s ++ layout x
layout (i 'Line' x)     = '\n': copy i ' ' ++ layout x

copy i x                = [x | _ <- [1..i]]

best w k x              = be w k [(0,x)]

be w k []               = Nil
be w k ((i,NIL):z)      = be w k z
be w k ((i,x :<> y) : z)  = be w k ((i,x) : (i,y) : z)
be w k ((i,NEST j x) : z) = be w k ((i+j),x) : z)
be w k ((i,TEXT s) : z)   = s 'Text' be w (k+length s) z
be w k ((i,LINE) : z)     = i 'Line' be w i z
be w k ((i.x :<|> y) : z) = better w k (be w k ((i.x) : z))
                                       (be w k (i,y) : z))

better w k x y          = if fits (w-k) x then x else y
```

## Der Code im Zusammenhang 4(11)

```
fits w x | w<0          = False
fits w Nil              = True
fits w (s 'Text' x)     = fits (w - length s) x
fits w (i 'Line' x)     = True

pretty w x              = layout (best w 0 x)

-- Utility functions
x <+> y                 = x <> text " " <> y
x </> y                 = x <> line <> y

folddoc f []            = nil
folddoc f [x]           = x
folddoc f (x:xs)        = f x (folddoc f xs)
```



## Der Code im Zusammenhang 5(11)

```
spread          = folddoc (<+>)
stack           = folddoc (</>)

bracket l x r   = group (text l <>
                         nest 2 (line <> x) <>
                         line <> text r)
x <+/> y        = x <> (text " " :<|> line) <> y

fillwords       = folddoc (<+/>) . map text . words

fill []         = nil
fill [x]        = x
fill (x:y:zs)   = (flatten x <+> fill (flatten y : zs))
                  :<|> (x </> fill (y : zs)
```



## Der Code im Zusammenhang 6(11)

```
-- Tree example
data Tree           = Node String [Tree]

showTree (Node s ts) = group (text s <>
                         nest (length s) (showBracket ts))

showBracket []      = nil
showBracket ts      = text "[" <> nest 1 (showTrees ts)
                                             <> text "]"

showTrees [t]       = showTree t
showTrees (t:ts)    = showTree t <> text "," <> line
                                        <> showTrees ts
```



## Der Code im Zusammenhang 7(11)

```
showTree' (Node s ts) = text s <> showBracket' ts

showBracket' []       = nil
showBracket' ts       = bracket "[" (showTrees' ts) "]"

showTrees' [t]        = showTree t
showTrees' (t:ts)     = showTree t <> text "," <> line
                                               <> showTrees ts
```



## Der Code im Zusammenhang 8(11)

```
tree                = Node "aaa"[ Node "bbbb"[ Node "ccc"[],
                                               Node "dd"[]
                                  ],
                                  Node "eee"[],
                                  Node "ffff"[ Node "gg"[],
                                               Node "hhh"[],
                                               Node "ii"[]
                                  ]
                    ]

testtree w          = putStr(pretty w (showTree tree))
testtree' w         = putStr(pretty w (showTree' tree))
```

## Der Code im Zusammenhang 9(11)

```
-- XML Example

data XML              = Elt String [Att] [XML]
                      | Txt String

data Att              = Att String String

showXML x             = folddoc (<>) (showXMLs x)

showXMLs (Elt n a []) = [text "<" <> showTag n a <> text "/>"
showXMLs (Elt n a c)  = [text "<" <> showTag n a <> text ">" <>
                         showFill showXMLs c <>
                         text "</" <> text n <> text ">"]
showXMLs (Txt s)      = map text (words s)
```



## Der Code im Zusammenhang 10(11)

```
showAtts (Att n v) = [text n <> text "=" <> text (quoted v)]

quoted s           = "\"" ++ s ++ "\""

showTag n a        = text n <> showFill showAtts a

showFill f []      = nil
showFill f xs      = bracket "" (fill (concat (map f xs))) ""
```



## Der Code im Zusammenhang 11(11)

```
xml      = Elt "p"[Att "color" "red",
                   Att "font" "Times",
                   Att "size" "10"
                  ] [ Txt "Here is some",
                      Elt "em" [] [ Txt "emphasized"],
                      Txt "text.",
                      Txt "Here is a",
                      Elt "a" [ Att "href" "http://www.eg.com/"]
                              [ Txt "link" ],
                      Txt "elsewhere."
                  ]

testXML w = putStr (pretty w (showXML xml))
```



## Weiterführende und vertiefende Literatur 1(2)

Imperativer Pretty Printer

- Derek Oppen. *Pretty-printing*. ACM Transactions on Programming Languages and Systems, 2(4):465-483, 1980.

...und in seiner funktionalen Realisierung

- Olaf Chitil. *Pretty printing with lazy dequeues*. In ACM SIGPLAN Haskell Workshop, 183-201, Florenz, Italien, 2001. Universiteit Utrecht UU-CS-2001-23.



## Weiterführende und vertiefende Literatur 2(2)

Überblick über die Evolution einer Pretty Printer Bibliothek und Ausgangspunkt der Entwicklung des von Ph. Wadler beschriebenen *Prettier Printers*.

- John Hughes. *The design of a pretty-printer library*. In Johan Jeuring, Erik Meijers (Hrsg.), *Advanced Functional Programming*, LNCS 925, Springer, 1995.

...eine im Glasgow Haskell Compiler implementierte Variante

- Simon Peyton Jones. *Haskell pretty-printer library*. `http://www.haskell.org/libraries/#prettyprinting`, 1997.



## Zum Ausklang: Haskell und seine Entwicklung(sgeschichte)

16 Jahre Haskell: Ein Rückblick zum 15. Geburtstag

von

**Simon Peyton-Jones**

*Wearing the Hair Shirt: A Retrospective on Haskell*

`http://research.microsoft.com/users/simonpj/papers/haskell-retrospective/`