Today's Topic

- Pretty Printing
  Like parsing a typical demo-application

Pretty Printing

Pretty Printing

...like lexical and syntactical analysis another typical application for demonstrating the elegance of functional programming.

What's it all about?

A *pretty printer* is...

- a tool (often a library of routines) designed for converting a *tree* into plain *text*

Essential goal...

- a minimum number of lines while preserving and reflecting the structure of the tree by indentation

"Good" Pretty-Printer

...distinguished by properly balancing

- Simplicity of usage
- Flexibility of the format
- "Prettiness" of output

Reference

The following presentation is based on...

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

Distinguishing Feature

...of the "Prettier Printer" proposed by Philip Wadler:

- There is only a single way to concatenate documents, which is
  - associative
  - with left-unit and right-unit

Why "prettier" than "pretty"?

Wadler considers his "Prettier Printer" an improvement of the pretty printer library proposed by John Hughes, which is widely recognized as a standard.

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

Hughes' library enjoys the following characteristics:

- Two ways to concatenate documents (horizontal and vertical), one of which
  - vertical: without unit
  - horizontal: with right-unit (but no left-unit)
- ca. 40% more code, ca. 40% slower as Wadler's proposal

A Simple Pretty Printer: Basic App.

Characteristic: For each document there shall be only one possible layout (e.g., no attempt is made to compress structure onto a single line).

The *basic operators* needed are:

```
(<>)   :: Doc -> Doc -> Doc   -- ass. concatenation of docs.
nil    :: Doc                 -- The empty document:
                                 Right and left unit for (<>)
text   :: String -> Doc       -- Conversion function: Converts
                                 a string to a document
line   :: Doc                 -- Line break
nest   :: Int -> Doc -> Doc   -- Adding indentation
layout :: Doc -> String       -- Output: Converts a document
                                 to a string
```

Convention:

- Arguments of `text` are free of *newline* characters

A Simple Implementation

Implement...

- doc as strings (i.e. as data type String)

with...

- (<>) ...concatenation of strings
- nil ...empty string
- text ...identity on strings
- line ...new line
- nest i ...indentation: adding i spaces (after each line break by means of line) ⇝ essential difference to Hughes' pretty printer allowing to drop one concatenation operator
- layout ...identity on strings

Example

...converting trees into documents (here: Strings) which are output as text (here: Strings).

Consider the following type of trees:

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

A concrete value B of type Tree...

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


...and its desired output

A text, where indentation reflects the tree structure...

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

...sibling trees start on a new line, properly indented.

Implementation

The below implementation achieves this...

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

Another possibly wanted output of B

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

...each subtree starts on a new line, properly indented.

An implementation producing the latter output

```
data Tree             = Node String [Tree]

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

showBracket' :: [Tree] -> Doc
showBracket' []       = nil
showBracket' ts       = text "[" <> nest 2 (line <> showTrees' ts)
                          <> line <> text "]"

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


A Normal Form of Documents

Documents can always be reduced to *normal form*

Normal form...

- text alternating with line breaks nested to a given indentation

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

where

- each s_j is a (possibly empty) string
- each i_j is a (possibly zero) natural number

Normal Forms: An Example 1(3)

The document...

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

Normal Forms: An Example 2(3)

...prints as follows:

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


Normal Forms: An Example 3(3)

Here it is its normal form:

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


Why does it work?

...because of the properties (laws) the functions enjoy.

In more detail...

...because of the fact that

- <> is associative with unit `nil` and
- the following laws (see next slide):

Properties of the Functions – Laws 1(2)

We have the following (pairs of) laws (except for the last one):

```
text (s ++ t)   = text s <> text t     (text is a homomorphism from
text ""         = nil                   string concatenation to
                                        document concatenation)

nest (i+j) x    = nest i (nest j x)    (nest is a homomorphism from
nest 0 x        = x                     addition to composition)

nest i (x <> y) = nest i x <> nest i y (nest distributes through
nest i nil      = nil                   document concatenation)

nest i (text s) = text s       (Nesting is absorbed by text;
                                Different to Hughes' pretty printer)
```

Properties of the Functions – Laws 2(2)

Impact

- The above laws are sufficient to ensure that documents can always be transformed into normal form
  - first four laws: applied left to right
  - last three laws: applied right to left

Further Properties – Laws

...relating documents to their layouts:

```
layout (x <> y)      = layout x ++ layout y (layout is a homomorphism
layout nil           = ""                    from document
                                             concatenation to
                                             string concatenation)

layout (text s)      = s                    (layout is the inverse
                                             of text)

layout (nest i line) = '\n' : copy i ' '    (layout of a nested
                                             line is a newline
                                             followed by one space
                                             for each level of
                                             indentation)
```


The Implementation of `Doc`

Intuition

...represent documents as a concatenation of items, where each item is a text or a line break indented to a given amount.

...realized as a sum type (the *algebra of documents*):

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

The constructors relate to the document operators as follows:

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


Example

Using the algebraic type `Doc`, the normal form (considered previously)...

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

...is represented by the following value of this algebraic type `Doc`:

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

Derived Implementations 1(2)

Implementations of the document operators can easily be derived from the above equations:

```
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
```


Derived Implementations 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          = ""
```


Correctness of the derived Implementations

...can be shown for each of them, e.g.:

- Derivation of `(s 'Text' x) <> y = s 'Text' (x <> y)`

```
    (s 'Text' x) <> y
 =   { Definition of Text }
    (text s <> x) <> y
 =   { Associativity of <> }
    text s <> (x <> y)
 =   { Definition of Text }
    s 'Text' (x <> y)
```

The remaining equations can be shown by similar reasoning

Documents with Multiple Layouts: Adding Flexibility

- *Up to now*... documents were equivalent to a string (i.e., they have a fixed single layout)
- *Next*... documents shall be equivalent to a set of strings (i.e., they may have multiple layouts)

where each string corresponds to a layout.

This can be rendered possible by just adding a new function

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

Informally:
Given a document, representing a set of layouts, `group` returns the set with one new element added, which represents the layout in which everything is compressed on one line: Replace each newline (plus indentation) by a single space.

Preferred Layouts

Beauty needs to be specified...

- `pretty` replaces `layout`

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

  which picks the prettiest layout depending on the preferred maximum line width argument

  Remark: `pretty`'s integer-argument specifies the preferred maximum line length of the output (and hence the prettiest layout out of the set of alternatives at hand).

Example

Using the modified `showTree` function based on `group`...

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

...the call of `pretty 30` (once completely specified) will yield the output:

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

This ensures:

- Trees are fit onto one line where possible (i.e., length ≤ 30)
- Insertion of sufficiently many line breaks in order to avoid exceeding the given maximum line length

Implementation of the new Functions

The following supporting functions are required:

```
-- Forming the union of two sets of layouts
(<|>)   :: Doc -> Doc -> Doc

-- Replacement of each line break (and its associated
-- indentation) by a single space
flatten :: Doc -> Doc
```

- *Observation* ...a document always represents a non-empty set of layouts
- *Requirements*
  - ...in `(x <|> y)` all layouts of `x` and `y` enjoy the same flat layout (mandatory invariant of `<|>`)
  - ...each first line in `x` is at least as long as each first line in `y` (second invariant)
- *Note* ...`<|>` and `flatten` are not directly exposed to the user (only via `group` and other supporting functions)

Properties (Laws) of `(<|>)`

...operators on simple documents are extended pointwise through union:

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

Properties (Laws) of `flatten`

...the interaction of `flatten` with other document operators:

```
flatten (x <|> y) = flatten x    -- distribution law

flatten (x <> y)  = flatten x <> flatten y
flatten nil       = nil
flatten (text s)  = text s
flatten line      = text " "     -- the most interesting case:
                                 -- linebreaks are replaced by
                                 -- a single space
flatten (nest i x) = flatten x
```


Implementation of `group`

...by means of `flatten` and (<>), the implementation of `group` can be given:

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

Intuitively: `group` adds the flattened layout to to a set of layouts.

Note: A document always represents a non-empty set of layouts where all layouts in the set flatten to the same layout.

Normal Form

Based on the previous laws each document can be reduced to a *normal form* of the form

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

where each `xi` is in the normal form of simple documents (which was introduced previously).

Selecting a "best" Layout out of a Set of Layouts

...by defining an ordering relation on lines in dependence of the given maximum line length

Out of two lines...

- which do not exceed the maximum length, select the longer one
- of which at least one exceeds the maximum length, select the shorter one

Note: Sometimes we have to pick a layout where some line exceeds the limit (a key difference to the approach of Hughes). However, this is done only, if this is unavoidable.

The Adapted Implementation of Doc

The new implementation of Doc as algebraic type. It is similar to the previous one except for the new construct representing the union of two documents:

```
data Doc = -- As before: The first 3 alternatives
           Nil
         | String 'Text' Doc
         | Int 'Line' Doc
           -- New: We add a construct representing the
              union of two documents
         | Doc 'Union' Doc
```


Relationship of Constructors and Document Operators

The following relationships hold between the constructors and the document operators...

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


Example 1(8)

The document...

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


Example 2(8)

...has the following possible 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
```

Example 3(8)

Task: ...print the above document under the constraint that the maximum line width is 5
⇝ the right-most layout of the previous slide is requested

Initial (performance) considerations:

- Factoring out `"hello"` of all the layouts in `x` and `y`

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

- Defining additionally the interplay of `(<>)` and `nest` with `Union`

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


Example 4(8)

Implementations of `group` and `flatten` can easily be derived:

```
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
```


Example 5(8)

Considerations on correctness (similar reasoning as earlier):

Derivation of `group (i 'Line' x)` (see line two) (preserving the invariant required by `union`)

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


Example 6(8)

Correctness considerations (cont'd):

Derivation of `group (s 'Text' x)` (see line three)

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

Example 7(8)

Selecting the "best" layout:

```
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
```

Remark:

- `best` ...converts a "union"-afflicted document into a "union"-free document
- Argument `w` ...maximum line width
- Argument `k` ...already consumed letters (including indentation) on current line

Example 8(8)

Check, if the first document line stays within the maximum line length `w`...

```
fits w x | w<0      = False                  -- cannot fit
fits w Nil          = True                   -- fits trivially
fits w (s 'Text' x) = fits (w - length s) x -- fits if x fits into
                                             -- the remaining space
                                             -- after placing s
fits w (i 'Line' x) = True                   -- yes, it fits
```

Last but not least, the output routine (layout remains unchanged): Select the best layout and convert it to a string...

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


Enhancing Performance: A More Efficient Variant

Sources of inefficiency:

1. Concatenation of documents might pile up to the left
2. Nesting of documents adds a layer of processing to increment the indentation of the inner document

Problem fix:

- For 1.): Add an explicit representation for concatenation, and generalize each operation to act on a list of concatenated documents
- For 2.:) Add an explicit representation for nesting, and maintain a current indentation that is incremented as nesting operators are processed

Enhancing Performance: A More Efficient Variant (cont'd)

Implementing this fix by means of a new implementation of documents:

```
data DOC = NIL                -- Here is one constructor
         | DOC :<> DOC        -- corresponding to each
         | NEST Int DOC       -- operator that builds a
         | TEXT String        -- document
         | LINE
         | DOC :<|> DOC
```

Remark:

- In distinction to the previous document type we here use capital letters in order to avoid name clashes with the previous definitions

Implementing the Document Operators

Defining the operators to build a document are straightforward:

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


Implementing `group` and `flatten`

As before, we require the following invariants:

- ...in (x :<|> y) all layouts in x and y flatten to the same layout
- ...no first line in x is shorter than any first line in y

Definitions of `group` and `flatten` are then straightforward:

```
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
```


Representation Function

...generating the document from an indentation-afflicted document ("indentation-document pair")

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


Selecting the "best" Layout

Generalizing the function "best" by composing the old function with the representation function to work on lists of indentation-document pairs...

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

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

where the definition is derived from the old one...

```
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))
```

Preparing the XML-Application 1(3)

First some useful convenience 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)

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


Preparing the XML-Application 2(3)

Further supportive functions:

```
-- An often recurring output pattern
bracket l x r            = group (text l <>
                                  nest 2 (line <> x) <>
                                  line <> text r)

-- Abbreviation of the alternative tree layout function
showBracket' ts          = bracket "[" (showTrees' ts) "]"

-- Filling up lines (using words out of the Haskell Standard Lib.)
x <+/> y                 = x <> (text " " :<|> line) <> y
fillwords                = folddoc (<+/>) . map text . words
```


Preparing the XML-Application 3(3)

`fill`, a variant of `fillwords`

$\leadsto$ ...collapses a list of documents to a single document

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


Application 1(2)

Printing XML-documents (simplified 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)]
```

Application 2(2)

Continuation...

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

showTag n a   = text n <> showFill showAtts a

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


XML Example 1

...for a given maximum line length of 30 letters:

```
<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>
```


XML Example 2

...for a given maximum line length of 60 letters:

```
<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>
```


XML Example 3:

...after dropping of `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>
```

...start and close tags are crammed together with other text $\leadsto$ less beautifully than before.

Overview of the Code 1(11)

Source: Philip Wadler. *A Prettier Printer*. In Jeremy Gibbons, Oege de Moor (Eds.), *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
```


Overview of the Code 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
```


Overview of the Code 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
```

Overview of the Code 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)
```

Overview of the Code 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))
```


Overview of the Code 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
```


Overview of the Code 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
```


Overview of the Code 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))
```

Overview of the Code 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)
```


Overview of the Code 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))) ""
```


Overview of the Code 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))
```


Further Readings 1(2)

On an imperative Pretty Printer

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

...and a functional realization of it:

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

Further Readings 2(2)

Overview on the evolution of a Pretty Printer Library and origin of the development of the *Prettier Printers* proposed by Phil Wadler.

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

...a variant implemented in the Glasgow Haskell Compiler

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

Einladung zum epilog

epilog SS 2010

"Die Fakultät für Informatik präsentiert zwei mal pro Jahr die Diplomarbeiten des letzten halben Jahres in einer Posterausstellung und ausgewählten Vorträgen und gibt einen Einblick in das breite Spektrum der Themen und Aufgabenstellungen der Abschlussarbeiten".

ZEIT: Donnerstag, 10. Juni 2010, ab 15:00 Uhr

ORT: TU Wien, Freihaus, Wiedner Hauptsraße 8, 2.OG, FH Hörsaal 6

MEHR INFO: `http://www.informatik.tuwien.ac.at/epilog`

Einladung zum Kolloquiumsvortrag

Die Complang-Gruppe lädt ein zu folgendem Vortrag...

Automatic Verification of Concurrent Programs in Chalice

Prof. Dr. Peter Müller
ETH Zürich, Schweiz

ZEIT: Dienstag, 18. Mai 2010, 15:30 Uhr

ORT: TU Wien, Seminarraum Argentinierstr. 8, Erdgeschoss

MEHR INFO: `http://www.complang.tuwien.ac.at/talks/mueller2010-05-18` (in Kürze)

Alle Interessenten sind herzlich willkommen!

Opportunity!

HaL5 : Haskell in Leipzig, zum Fünften

Leipzig, Mediencampus (`mediencampus-villa-ida.de/`), Germany, Fri, 4 June 2010.

(Registration fee: 20 EUR (including Tutorials, Workshop, and Barbecue-Party)).

More Infos: `http://iba-cg.de/hal5.html`

Next Course Meeting

- Thu, May 13, 2010: No lecture (public holiday)
- Thu, May 20, 2010: 4.15 p.m. to 5.45 p.m., lecture room on the ground floor of the building Argentinierstr. 8