A *pretty-printer* converts tree structured data (eg. XML documents, syntax trees) into a nicely formatted string

- should support describing alternative layouts
- choosing an optimal layout given the line width
- be efficient

Literature

D. C. Oppen
"Prettyprinting"

J. Hughes
"The Design of a Pretty-printing Library"

P. Wadler
"A prettier printer"

Intuition (Oppen)

Given the stream of characters

```
var x:integer; y:char; begin x:=1; y:='a'; end
```

Intuition (Oppen)

Given the stream of characters

```
var x:integer; y:char; begin x:=1; y:='a'; end
```

For line width of 40, we might want

```
var x:integer; y:char;
  begin x:=1; y:='a'; end
```

or

```
var x:integer;
    y:char;
begin
  x:=1;
  y:='a';
end
```

Intuition (Oppen)

Given the stream of characters

```
var x:integer; y:char; begin x:=1; y:=′a′; end
```

For line width of 40, we might want

```
var x:integer; y:char;
  begin x:=1; y:=′a′; end
```

or

```
var x:integer;
    y:char;
begin
  x:=1;
  y:=′a′;
end
```

but not

```
var x:integer; y:
char; begin x:=1;
y:=′a′; end
```

Even Simpler

Given the stream of characters

```
f(a, b, c, d) + g(a, b, c, d)
```

For line width of 20, we might want

```
f(a, b, c, d) +
  g(a, b, c, d)
```

or

```
f(a, b, c, d)
  + g(a, b, c, d)
```

but not

```
f(a, b, c, d) + g(a,
                  b,
                  c,
                  d)
```

General idea

- Add grouping characters ⦃ and ⦄ to the stream.
- Break lines at spaces according to the groups.
- Look ahead to check if the group fits on a single line.

```
⦃⦃f(a, b, c, d)⦄ + ⦃g(a, b, c, d)⦄⦄
```

- Complicated description!

Pretty-printing boxes

- Extensions to the ideas of Oppen
- Used in OCaml
- Adds to the stream
    - possible-line-break symbol
    - horizontal box delimiters
    - vertical box delimiters (+indentation)
    - horizontal-vertical box delimiters (+indentation)

Pretty-printing boxes (cont.)

- horizontal box
  - possible-line-break ⟶ spaces
- vertical box
  - possible-line-break ⟶ newline+indentation
- horizontal-vertical box

$$\begin{cases} \text{horizontal box}, & \text{if it fits on a single line} \\ \text{vertical box} & \text{otherwise} \end{cases}$$

Pretty-printing boxes (cont.)

- Pro:
    - optimal – does not miss line breaks to avoid overflow
    - bounded – makes the choice after w characters
    - worst case: $O(nw)$ time, and $O(w)$ space
- Con:

Pretty-printing boxes (cont.)

- Pro:
  - optimal – does not miss line breaks to avoid overflow
  - bounded – makes the choice after w characters
  - worst case: $O(nw)$ time, and $O(w)$ space
- Con:
  - ???

Pretty-printing boxes (cont.)

- Pro:
  - optimal – does not miss line breaks to avoid overflow
  - bounded – makes the choice after w characters
  - worst case: $O(nw)$ time, and $O(w)$ space
- Con:
  - ???
  - imperative implementation
  - not as simple as it might be

Pretty-printing boxes (cont.)

- Pro:
  - optimal – does not miss line breaks to avoid overflow
  - bounded – makes the choice after w characters
  - worst case: $O(nw)$ time, and $O(w)$ space

- Con:
  - ???
  - imperative implementation
  - not as simple as it might be

We need a haskell-y implementation!

Pretty printing ver. 0 — using strings

Example: arithmetic expressions

```
data Expr = Num Int | Add Expr Expr

showExpr (Num x)        =  show x
showExpr (Add e1 e2)    =  showExpr e1   ++
                           "+"           ++
                           showExpr e2
```

Pretty printing ver. 0 — using strings

Example: arithmetic expressions

```
data Expr = Num Int | Add Expr Expr

showExpr (Num x)       =  show x
showExpr (Add e1 e2)   =  showExpr e1  ++
                          "+"          ++
                          showExpr e2
```

NB!

- Very inefficient
- Hard to create different layouts

Pretty printing ver. 1

Abstract interface

```
nil     :: Doc
text    :: String -> Doc
(<>)    :: Doc -> Doc -> Doc
pretty  :: Doc -> String
```

Pretty printing ver. 1

Abstract interface

```
nil      :: Doc
text     :: String -> Doc
(<>)     :: Doc -> Doc -> Doc
pretty   :: Doc -> String
```

Example

```
showExpr = pretty . showE
   where  showE (Num x)        =  text (show x)
          showE (Add e1 e2)    =  showE e1   <>
                                  text "+"   <>
                                  showE e2
```

Pretty printing ver. 1 — implementation

Representation of documents

```
data Doc  =  Nil
          |  Text String
          |  Doc :<> Doc
```

Pretty printing ver. 1 — implementation

Representation of documents

```
data Doc  =  Nil
          |  Text String
          |  Doc :<> Doc
```

Construction of documents

```
nil    =  Nil
text   =  Text
(<>)   =  (:<>)
```

Pretty printing ver. 1 — implementation

Printing documents

```
pretty d = convert [d]

convert []                  =  ""
convert (Nil : ds)          =  convert ds
convert (Text s : ds)       =  s ++ convert ds
convert (d1 :<> d2 : ds)    =  convert (d1:d2:ds)
```

Pretty printing ver. 2 — nesting and layouts

Abstract interface

```
line  ::  Doc
nest  ::  Int -> Doc -> Doc
```

Pretty printing ver. 2 — nesting and layouts

Abstract interface

```
line  ::  Doc
nest  ::  Int -> Doc -> Doc
```

Example

```
data Tree = Node String [Tree]

tree = Node "aa" [  Node "b" [ Node "c" [] ],
                    Node "dd" [],
                    Node "e" [ Node "f" [] ] ]
```

Pretty printing ver. 2 — nesting and layouts

Example

```
ppTree (Node s ts) =  text s <>
                      nest (length s) (ppBracket ts)
  where  ppBracket []     =  nil
         ppBracket ts     =  text "[" <>
                             nest 1 (ppTrees ts) <>
                             text "]"
         ppTrees [t]      =  ppTree t
         ppTrees (t:ts)   =  ppTree t <> text "," <>
                             line <> ppTrees ts
```

```
Main> putStr . pretty . ppTree $ tree
aa[b[c],
   dd,
   e[f]]
```

Pretty printing ver. 2 — nesting and layouts

Example

```
ppTree (Node s ts) = text s <> ppBracket ts
  where  ppBracket []   =  nil
         ppBracket ts   =  text "[" <>
                           nest 2 (line <> ppTrees ts) <>
                           line <> text "]"
```

```
Main> putStr . pretty . ppTree $ tree
aa[
  b[
    c
  ],
  dd,
  e[
    f
  ]
]
```

Pretty printing ver. 2 — implementation

Representation of documents

```
data Doc  =  ...
          |  Line
          |  Nest Int Doc
```

Pretty printing ver. 2 — implementation

Representation of documents

```
data Doc  =  ...
          |  Line
          |  Nest Int Doc
```

Construction of documents

```
line  =  Line
nest  =  Nest
```

Pretty printing ver. 2 — implementation

Printing documents

```
pretty d = convert [(d,0)]

convert []                    =  ""
convert ((Nil,i) : ds)        =  convert ds
convert ((Text s,i) : ds)     =  s ++ convert ds
convert ((d1 :<> d2,i) : ds)  =  convert ((d1,i):(d2,i):ds)
convert ((Line,i) : ds)       =  "\n"  ++ copy i ' '
                                       ++ convert ds
convert ((Nest n d,i) : ds)   =  convert ((d,n+i):ds)
```

Pretty printing ver. 3 — alternative layouts

Abstract interface

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

Pretty printing ver. 3 — alternative layouts

Abstract interface

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

Example

```
ppTree  (Node s ts) =
  group (text s <> nest (length s) (ppBracket ts))
```

```
Main> putStr . pretty 10 . ppTree $ tree
aa[b[c],
   dd,
   e[f]]
Main> putStr . pretty 20 . ppTree $ tree
aa[b[c], dd, e[f]]
```

Pretty printing ver. 3 — implementation

Representation of documents

```
data Doc  =  ...
          |  Doc :<|> Doc
```

Pretty printing ver. 3 — implementation

Representation of documents

```
data Doc  =  ...
          |  Doc :<|> Doc
```

Construction of documents

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

flatten Nil            =  Nil
flatten (Text s)       =  Text s
flatten (d1 :<> d2)    =  flatten d1 :<> flatten d2
flatten Line           =  Text " "
flatten (Nest i d)     =  Nest i (flatten d)
flatten (d1 :<|> d2)   =  flatten d1
```

Pretty printing ver. 3 — implementation

Printing documents

```
pretty w d  =  convert w 0 [(d,0)]

convert w c []                 =  ""
convert w c ((Nil,i) : ds)     =  convert w c ds
convert w c ((Text s,i) : ds)
            =  s ++ convert w (c + length s) ds
convert w c ((d1 :<> d2,i) : ds)
            =  convert w c ((d1,i):(d2,i):ds)
convert w c ((Line,i) : ds)
            =  "\n" ++ copy i ' ' ++ convert w i ds
convert w c ((Nest n d,i) : ds)
            =  convert w c ((d,n+i):ds)
```

Pretty printing ver. 3 — implementation

Printing documents (cont.)

```
convert w c  ((d1 :<|> d2,i) : ds)
          =  better w c  (convert w c ((d1,i):ds))
                         (convert w c ((d2,i):ds))
```

Pretty printing ver. 3 — implementation

Printing documents (cont.)

```
convert w c  ((d1 :<|> d2,i) : ds)
           =  better w c  (convert w c ((d1,i):ds))
                          (convert w c ((d2,i):ds))
```

Choosing the best layout

```
better w c s1 s2  = if fits (w-c) s1 then s1 else s2
fits w xs    = w >= 0 &&
               (null xs  ||  head xs == '\n'
                         ||  fits (w-1) (tail xs))
```

Pretty printing ver. 4 — extensions

Forcing newlines

```
data Doc  =  ...
          |  ForceLine

forceline  =  ForceLine

flatten ForceLine = Text (replicate maxBound ' ')

convert w c ((ForceLine,i) : ds) =
     convert w c ((Line,i) : ds)
```

Conclusion

- Good design
- Reasonably fast for small code snippets
- Still, does not scale very well.
    - Remove groups and sacrifice optimality?