Chapter 8

Pretty-printing

This chapter is devoted to an example of how to build a small library in Haskell. A library is an organised collection of types and functions made available to users for carrying out some task. The task we have chosen to discuss is *pretty-printing*, the idea of taking a piece of text and laying it out over a number of lines in such a way as to make the content easier to view and understand. We will ignore many of the devices for improving the readability of a piece of text, devices such as a change of colour or size of font. Instead we concentrate only on where to put the line breaks and how to indent the contents of a line. The library won't help you to lay out bits of mathematics, but it can help in presenting tree-shaped information, or in displaying lists of words as paragraphs.

8.1 Setting the scene

Let's begin with the problem of displaying conditional expressions. In this book we have used three ways of displaying such expressions:

```
if p then expr1 else expr2

if p then expr1
else expr2

if p
then expr1
else expr2
```

These three layouts, which occupy one, two or three lines, respectively, are considered acceptable, but the following two are not:

```
if p then
expr1 else expr2

if p
then expr1 else expr2
```

The decision as to what is or is not acceptable is down to me, the author. You may disagree with my choices (some do), and a flexible library should provide you with the ability to make your own reasonable choices. In any case, two basic questions have to be answered. Firstly, how can we describe the acceptable alternatives while rejecting the unacceptable ones? Secondly, how do we choose between the acceptable alternatives?

A quick answer to the second question is that the choice depends on the permitted line width. For instance we might choose a layout with the fewest lines, subject to the condition that each line fits within the allotted line width. Much more on this later.

As to the first question, one answer is just to write out all the acceptable alternatives. That's going to involve a lot of writing. A better alternative is to provide the user with a suitable *layout description language*. As a rough and ready guide we might write something like

```
if p <0> then expr1 (<0> + <1>) else expr2 +
if p <1> then expr1 <1> else expr2
```

where `<0>` means a single space, `<1>` means a line break and `+` means 'or'. The expression above yields our three layouts described earlier. However, the danger with providing the user with an unfettered choice of alternatives is that it becomes difficult to make a decision about the best layout without exploring every possible alternative, and that could take a long time.

Another possibility is to allow only restricted choice by forcing the user to describe layouts in terms of certain functions and operations provided by the library. For example, consider the description

```
group (group (if p <1> then expr1) <> <1> else expr2)
```

where `group` augments a set of layouts with one additional layout in which every `<1>` is replaced by `<0>`, thereby flattening the layout to just one line, and `(<>)` means concatenation lifted to sets of alternatives. For example,

```
group (if p <1> then expr1)
  = {if p <0> then expr1, if p <1> then expr1}
```

```
group (if p <1> then expr1) <> <1> else expr2
  = {if p <0> then expr1 <1> else expr2,
     if p <1> then expr1 <1> else expr2}
group (group (if p <1> then expr1) <> <1> else expr2)
  = {if p <0> then expr1 <0> else expr2,
     if p <0> then expr1 <1> else expr2,
     if p <1> then expr1 <1> else expr2}
```

Thus our set of three acceptable layouts is captured by the above description which contains two occurrences of `group`.

There is another aspect to the problem of displaying conditional expressions. What if `expr1` or `expr2` are themselves conditional expressions? Here we might want to allow a layout like

```
if p
then if q
     then expr1
     else expr2
else expr3
```

The point is that we should allow for *indentation* in our description language. Indentation means putting in a suitable number of spaces after each line break. This idea can be captured by providing a function `nest` so that `nest i x` is a layout in which *each* line break in layout `x` is followed by `i` spaces.

8.2 Documents

For the sake of a name let us agree to call a *document* some entity that represents the set of possible layouts of a piece of text. Documents are given as elements of the type `Doc` whose definition is left for later on. On the other hand, a layout is simply a string:

```
type Layout = String
```

We are deliberately being cagey about what a document actually is because we want to consider two representations of `Doc`. For now we concentrate on the operations on documents that our library might provide.

The first operation is a function

```
pretty :: Int -> Doc -> Layout
```

that takes a given line width and a document, and returns the best layout. How to define this function efficiently is really the main concern of the chapter.

The second operation is a function

```
layouts :: Doc -> [Layout]
```

that returns the set of possible layouts as a list. Why should we want such a function when we have `pretty`? Well, it takes a little experimentation to find the definitions that describe the layouts we regard as acceptable. The way to experiment is to formulate an initial definition and then rework it after inspecting all the resulting layouts on a small number of examples. That way we can see whether some layouts should be excluded or others added. So, whatever our final representation of documents turns out to be, we should provide `layouts` as a sensible diagnostic tool for the user.

The remaining operations deal with constructing documents. First up is the operation of concatenating two documents to give a new one:

```
(<>) :: Doc -> Doc -> Doc
```

Document concatenation should surely be an associative operation so we require of any implementation of `(<>)` that

```
(x <> y) <> z) = x <> (y <> z)
```

for all documents `x`, `y` and `z`.

Whenever there is an associative operation there is usually an identity element, so we also provide an empty document

```
nil :: Doc
```

We require `nil <> x = x` and `x <> nil = x` for all documents `x`.

The next operation is a function

```
text :: String -> Doc
```

that takes a string not containing newlines into a document. To provide for documents containing more than one line, we can provide another basic document

```
line :: Doc
```

For example,

```
text "Hello" <> line <> text "World!"
```

is a document with a single layout that consists of two lines. You might think that `line` is unnecessary because we could always allow newline characters in text strings, but to indent a document we would then have to inspect the contents of every `text`. Far better is to have an explicit newline document; that way we know where line breaks are.

Next, the function

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

provides a way of nesting documents: `nest i` indents a document by inserting `i` spaces *after* every newline. Note the emphasis: indentation is not done at the beginning of a document unless it begins with a newline. The reason for this choice is explained below.

Finally, to complete a library of eight operations, we have the function

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

This is the function that produces multiple layouts. The function `group` takes a document and adds an extra layout, one that consists of a single line of text with no line breaks.

We have named eight operations and given informal descriptions of what they are intended to mean, but can we be more precise about their properties and the relationships between them? An even more fundamental question is whether these operations are sufficiently flexible to allow for a reasonable class of layouts.

Let's first concentrate on what equational laws we might want. Finding such laws can boost our confidence that we have in hand an adequate and smoothly integrated box of tools, and that there isn't some crucial gadget we have missed. Such laws can also influence the meanings of operations and guide implementations. We have already asserted that `(<>)` should be associative with identity element `nil`, but what else should we require?

Well, for `text` we want the following properties:

```
text (s ++ t) = text s <> text t
text ""       = nil
```

In mathematical language this asserts that `text` is a *homomorphism* from string concatenation to document concatenation. An impressive (and possibly intimidating) name for something quite simple. Note that the associativity of string concatenation implies the associativity of document concatenation, at least for `text` documents.

For `nest` we require the following equations to hold:

```
nest i (x <> y)   = nest i x <> nest i y
nest i nil        = nil
nest i (text s)   = text s
nest i line       = line <> text (replicate i ' ')
nest i (nest j x) = nest (i+j) x
nest 0 x          = x
nest i (group x)  = group (nest i x)
```

All very reasonable (except possibly for the last), and we could give some of them mathematical names (`nest i` distributes through concatenation, `nest` is a homomorphism from numerical addition to functional composition and `nest i` commutes with `group`). The third law fails if `nest` were to indent from the beginning of a document; and it would also fail if we allowed text strings to contain newline characters. The last law holds because grouping adds a layout with no line breaks, and nesting has no effect on such a layout. See Exercise D for a more precise argument.

Turning to the properties of `layouts`, we require that

```
layouts (x <> y)   = layouts x <++> layouts y
layouts nil        = [""]
layouts (text s)   = [s]
layouts line       = ["\n"]
layouts (nest i x) = map (nestl i) (layouts x)
layouts (group x)  = layouts (flatten x) ++ layouts x
```

The operation `(<++>)` is lifted concatenation:

```
xss <++> yss = [xs ++ ys | xs <- xss, ys <- yss]
```

The function `nestl :: Int -> Layout -> Layout` is defined by

```
nestl i    = concat (map indent i)
indent i c = if c=='\n' then c:replicate i ' ' else [c]
```

Finally, `flatten :: Doc -> Doc` is the function that converts a document into one with a single layout in which each newline and its associated indentation is replaced by a single space. This function is not provided in the public interface of our documents library, though it will be needed internally. It is a missing gadget in the sense that we need it to complete the description of the algebraic laws.

We require that `flatten` should satisfy the following conditions:

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

That makes 24 laws in total (one for `<>`, two each for `nil` and `text`, seven for `nest` and six each for `layouts` and `flatten`). Many of the laws look like constructive Haskell definitions of functions over a data type in which `nil`, `text` and so on are constructors. More on this is in Section 8.6.

The eight operations certainly seem reasonable enough, but do they give us sufficient flexibility to describe the layouts we might want? The proof of the pudding is in the eating, so in a moment we will pause to consider three examples. Before doing so, some implementation of documents, however quick and dirty, will be needed to test the examples.

8.3 A direct implementation

One obvious choice of representation is to identify a document with its list of layouts:

```
type Doc = [Layout]
```

Such a representation is called a *shallow embedding*. With a shallow embedding, the library functions are implemented directly in terms of the values of interest (here, `layouts`). Later on we will abandon this representation in favour of a more structured alternative, but it is the obvious one to try first.

Here are the definitions of the operations above (we will leave `pretty` until later):

```
layouts   = id
x <> y    = x <++> y
nil       = [""]
line      = ["\n"]
text s    = [s]
nest i    = map (nestl i)
group x   = flatten x ++ x
flatten x = [flattenl (head x)]
```

We have already defined `nestl`, and `flattenl` is defined by

```
flattenl :: Layout -> Layout
flattenl [] = []
flattenl (c:cs)
   | c=='\n'   = ' ':flattenl (dropWhile (== ' ') cs)
   | otherwise = c:flattenl cs
```

Do the 24 laws hold for this implementation? Well, let's go through them. Lifted concatentation `<++>` is associative with `[[]]` as identity element, so the first three laws are okay. The two laws of `text` are easy to check, and the six laws of `layouts` are immediate. All but two laws of `nest` are routine. The remaining two, namely

```
nest i . nest j = nest (i+j)
nest i . group  = group . nest i
```

involve a bit of work (see Exercises C and D). That leaves the laws of `flatten`. Three are easy, and one can show

```
flatten . nest i = flatten
flatten . group  = flatten
```

with a bit of work (see Exercises E and F). But the stumbling block is the law

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

This one is false. Take `x = line` and `y = text "  hello"`. Then

```
flatten (x <> y) = ["hello"]
flatten x <> flatten y = ["   hello"]
```

and the two results are different. The reason is that `flatten` removes the effect of nesting, but does not remove spaces after newlines if they are present in an unnested document. On the other hand, `flattenl` removes spaces after every newline in the document.

Rather than try to fix up this deficiency, we can accept the less than perfect implementation and move on. One can show that all layouts of a document flatten to the same string (see the Answer to Exercise E). The shallow embedding also possesses another property that we will exploit in the definition of `pretty`. To see what it is, consider the function `shape` that returns the shape of a layout:

```
shape :: Layout -> [Int]
shape = map length . lines
```

The prelude function `lines` breaks up a string on newline characters, returning a list of strings without newlines. Thus the shape of a layout is the list of lengths of the lines that make up the layout. The crucial property of `layouts` is that the list

of shapes of the layouts of a document is in lexicographically decreasing order. For example, one of the documents described in the following section has 13 possible layouts whose shapes are given by

```
[[94],[50,43],[50,28,19],[50,15,17,19],[10,39,43],
[10,39,28,19],[10,39,15,17,19],[10,28,15,43],
[10,28,15,28,19],[10,28,15,15,17,19],[10,13,19,15,43],
[10,13,19,15,28,19],[10,13,19,15,15,17,19]]
```

This list is in decreasing lexicographic order. The reason the property holds is that `layouts (group x)` puts the flattened layout at the head of the list of layouts of document `x`, and a flattened layout consists of a single line. Exercise G goes into more details.

8.4 Examples

Our first example deals with laying out conditional expressions. For present purposes a conditional expression can be represented as an element of the data type `CExpr`, where

```
data CExpr = Expr String | If String CExpr CExpr
```

Here is a function `cexpr` that specifies the acceptable layouts described earlier:

```
cexpr :: CExpr -> Doc
cexpr (Expr p) = text p
cexpr (If p x y)
       = group (group (text "if " <> text p <>
                       line <> text "then " <>
                       nest 5 (cexpr x)) <>
                line <> text "else " <>
                nest 5 (cexpr y))
```

This definition is similar to our previous version, except for the nesting of the subexpressions.

For example, two of the 13 possible layouts for one particular expression are as follows:

```
if wealthy
then if happy then lucky you else tough
else if in love then content else miserable
```

```
if wealthy
then if happy
     then lucky you
     else tough
else if in love
     then content
     else miserable
```

You can see from the last expression why we have chosen an indentation of five spaces. The 13 possible layouts for this particular conditional expression have the shapes displayed in the previous section.

The second example concerns how to lay out general trees, trees with an arbitrary number of subtrees:

```
data GenTree a = Node a [GenTree a]
```

Here is an example tree, laid out in two different ways:

```
Node 1
  [Node 2
     [Node 7 [],
      Node 8 []],
   Node 3
     [Node 9
        [Node 10 [],
         Node 11 []]],
   Node 4 [],
   Node 5
     [Node 6 []]]

Node 1
  [Node 2 [Node 7 [], Node 8 []],
   Node 3 [Node 9 [Node 10 [], Node 11 []]],
   Node 4 [],
   Node 5 [Node 6 []]]
```

The function `gtree` that produced these trees (coincidentally, also among a total of 13 different ways) was defined as follows:

```
gtree :: Show a => GenTree a -> Doc
gtree (Node x [])
    = text ("Node " ++ show x ++ " []")
```

```
gtree (Node x ts)
    =  text ("Node " ++ show x) <>
       group (nest 2 (line <> bracket ts))
```

The first clause says that a tree with no subtrees is always displayed on a single line; the second clause says that a tree with at least one subtree is displayed either on a single line or has its subtrees each displayed on a new line with an indentation of two units. The function `bracket` is defined by

```
bracket :: Show a => [GenTree a] -> Doc
bracket ts = text "[" <> nest 1 (gtrees ts) <> text "]"

gtrees [t]    = gtree t
gtrees (t:ts) = gtree t <> text "," <> line <> gtrees ts
```

To be honest, it took a little time and experimentation to find the definitions above (for which the function `layouts` proved indispensable), and the result is certainly not the only way to lay out trees.

Finally, here is a way of laying out a piece of text (a string of characters containing spaces and newlines, not a document `text`) as a single paragraph:

```
para :: String -> Doc
para = cvt . map text . words

cvt [] = nil
cvt (x:xs)
    = x <> foldr (<>) nil [group (line <> x) | x <- xs]
```

First, the words of the text are computed using the standard library function `words`, a function we have encountered a number of times before. Then each word is converted into a document using `text`. Finally, each word, apart from the first, is laid out either on the same line or on a new line. If there are $n+1$ words in the text, and so n inter-word spaces, the code above describes 2^n possible layouts. We certainly don't want to examine all these layouts in computing one that will fit within a given line width.

8.5 The best layout

As we said above, the best layout depends on the maximum permitted line width. That's a simple decision, but not the only one. In general a pretty layout of a nested document will consist of a ribbon of text snaking across the page, and it is arguable

that the width of the ribbon should also play a part in determining the best layout. After all, is the best layout on an infinitely wide page one in which everything is placed on one line? However, for simplicity we will ignore this very reasonable refinement and take only the line width as the deciding factor.

There is also another decision to be made. Suppose we choose the best layout, according to some criterion, among those layouts all of whose lines fit within the given line width. That's fine if there is at least one such layout, but what if there isn't? The two options are either to abandon the formatting process with a suitable error message, or else to do the best we can, accepting that the width may be exceeded.

Psychologically and practically the second option seems the better one, so let us explore what it entails. We can start by comparing the first lines, ℓ_1 and ℓ_2, of two layouts. We can decide that line ℓ_1 is better than ℓ_2 if: (i) both lines fit into width w and ℓ_1 is longer than ℓ_2; (ii) ℓ_1 fits w but ℓ_2 doesn't; or (iii) neither fits w and ℓ_1 is shorter than ℓ_2. The decision is a reasonable one because it should be capable of being implemented by a *greedy* strategy: fill up the first line as much as possible without exceeding the line width; and if that is not possible, stop as soon as the width is exceeded.

The comparison test above doesn't determine what should happen if the two lines have the same length. But it is a consequence of the fact that all layouts flatten to the same string that two first lines with the same length will be the *same* line. Consequently, the first line is fixed and the comparison can pass to the second pair of lines. And so on.

The second property about decreasing shapes can be used to simplify the comparison test slightly because if layout *lx* precedes layout *ly* in the list of layouts, then the first line of *lx* is known to be at least as long as the first line of *ly*. And if the two lines are equally long, then the same statement is true of the second lines. And so on.

Given our shallow embedding of documents, here is a simple implementation of the function `pretty` that finds the best layout:

```
pretty :: Int -> Doc -> Layout
pretty w = fst . foldr1 choose . map augment
 where
 augment lx = (lx,shape lx)
 choose alx aly
  = if better (snd alx) (snd aly) then alx else aly
 better [] ks         = True
```

```
better js []          = False
better (j:js) (k:ks) | j == k    = better js ks
                     | otherwise = (j <= w)
```

Each layout is augmented with shape information to guide the choice of layout, which is then determined by a simple search. The test `better` implements the comparison operation described above. Finally, shape information is discarded.

This definition of `pretty` is hopelessly inefficient because every layout is computed and examined. If there are n possible choices of whether to have a line break or not, there are 2^n layouts to be examined and pretty-printing will be very slow indeed. For example,

```
ghci> putStrLn $ pretty 30 $ para pg
This is a fairly short
paragraph with just twenty-two
words. The problem is that
pretty-printing it takes time,
in fact 31.32 seconds.
(31.32 secs, 17650013284 bytes)
```

Ouch! What is worse, pretty-printing a longer paragraph will cause GHCi to crash with an 'out of memory' message. An exponential time and space algorithm is not acceptable.

What is wanted is an algorithm for `pretty` that can decide on which first line to choose without looking ahead more than w characters. The algorithm should also be efficient, taking linear time in the size of the document being pretty-printed. Ideally the running time should be independent of w, but a running time that does depend on w is acceptable if a faster one means a much more complicated program.

8.6 A term representation

The problem with identifying a document with its list of possible layouts is that useful structure is lost. Rather than bring all the alternatives to the top level as a list, we really want to bury them as deep as possible. For example, consider the following two expressions for a document:

```
A<0>B<0>D + A<0>B<1>D + A<1>C<0>E + A<1>C<1>E
A(<0>B(<0>D + <1>D) + <1>C(<0>E + <1>E))
```

As before, `<0>` denotes a single space and `<1>` a single line break. The five letters denote five nonempty texts. Since all four alternatives have to flatten to the same document, we require that `B<0>D = C<0>E`. In the first expression (which is essentially what is given by representing a document by its list of layouts) we have four layouts to compare. In the second expression we can shortcut some of the comparisons. For example, if we know that the common prefix `A` cannot fit in the given width, the first two layouts can be thrown away without further comparisons. Even better, if we choose between alternatives from the innermost to the outermost, we can base the comparison test on just the first lines of layouts. For instance, if we choose the better of `C<0>E` and `C<1>E` first, then that choice is not changed by subsequent choices.

The way to maintain the structure of documents is to represent a document as a tree:

```
data Doc = Nil
         | Line
         | Text String
         | Nest Int Doc
         | Group Doc
         | Doc :<>: Doc
```

Note the use of an infix constructor in the last line. Haskell allows infix operators as constructors, but they have to begin with a colon. They do not have to end with a colon as well, but it seems more attractive if they do. This tree is called an *abstract syntax tree*; each operation of the library is represented by its own constructor. An implementation in terms of abstract syntax trees is known as a *deep embedding*.

We will *not* provide the user with the details of the data type `Doc`, just its name. To explain why not, it is useful to insert a short digression about Haskell data types. In Haskell the effect of a `data` declaration is to introduce a new data type by describing how its values are constructed. Each value is named by an expression built only from the constructors of the data type, in other words a *term*. Moreover, different terms denote different values (provided there are no strictness flags). We can define functions on the data type by pattern matching on the constructors. There is therefore no need to state what the operations on the data type are – we can just define them. Types in which the values are described, but the operations are not, are called *concrete* types.

The situation is exactly the reverse with *abstract* data types. Here the operations are named, but not how the values are constructed, at least not publicly. For example, `Float` is an abstract data type; we are given the names of the primitive arithmetic

and comparison operations, and also a way of displaying floating-point numbers, but it is not stated how such numbers are actually represented. We cannot define functions on these numbers by pattern matching, but only in terms of the given operations. What can and should be stated publicly are intended meanings and the algebraic properties of the operations. However, Haskell provides no means for such descriptions beyond informal comments.

As it stands, `Doc` is a concrete type. But in our understanding of this type, different terms do not denote different values. For instance, we intend each constructor to be a replacement for the corresponding operation. Thus

```
nil      = Nil
line     = Line
text s   = Text s
nest i x = Nest i x
group x  = Group x
x <> y   = x :<>: y
```

We also want to keep the algebraic properties of these operations, so equations such as

```
(x :<>: y) :<>: z = x :<>: (y :<>: z)
Nest i (Nest j x) = Nest (i+j) x
```

should hold. But of course they do not. The solution is to use the module structure to hide the constructors of `Doc` from the user and insist only that the laws are 'observably' true. For instance we require

```
layouts ((x :<>: y) :<>: z) = layouts (x :<>: (y :<>: z))
```

The only way we can observe documents is through `layouts`; from the user's point of view if two documents produce the same layouts, then they are essentially the same document.

Let's get back to programming. Here is one definition of `layouts`. It is just the laws of `layouts` that we saw earlier, but now expressed as a proper Haskell definition:

```
layouts :: Doc -> [Layout]
layouts (x :<>: y) = layouts x <++> layouts y
layouts Nil        = [""]
layouts Line       = ["\n"]
layouts (Text s)   = [s]
layouts (Nest i x) = map (nestl i) (layouts x)
layouts (Group x)  = layouts (flatten x) ++ layouts x
```

The function `flatten` is similarly defined by

```
flatten :: Doc -> Doc
flatten (x :<>: y) = flatten x :<>: flatten y
flatten Nil        = Nil
flatten Line       = Text " "
flatten (Text s)   = Text s
flatten (Nest i x) = flatten x
flatten (Group x)  = flatten x
```

With these definitions, our 24 laws are either true by definition, or are observably true in the sense above.

The definition of `layouts` is simple enough, but it is unnecessarily inefficient. There are two separate reasons why this is so. First, consider the function `egotist` defined by

```
egotist :: Int -> Doc
egotist n | n==0      = nil
          | otherwise = egotist (n-1) <> text "me"
```

The document `egotist n` is a very boring one, and its sole layout consists of a string of `n` repetitions of `me`. By the way, we could have expressed the definition using `Nil`, `(:<>:)` and `Text` but, as we have said, we are not going to make these constructors public. As it stands, the definition of `egotist` could have been made by a user of the library. Anyway, back to the main point, which is that the association of the `(<>)` operations is to the left, and it takes $\Theta(n^2)$ steps to compute its layout(s). The `(++)` operations pile up to the left. The situation is entirely analogous to the fact that `concat` defined in terms of `foldl` is an order of magnitude less efficient than one defined in terms of `foldr`.

The second source of inefficiency concerns nesting. For example, consider the function `egoist` defined by

```
egoist :: Int -> Doc
egoist n | n==0      = nil
         | otherwise = nest 1 (text "me" <> egoist (n-1))
```

There are no line breaks in sight, so `egoist n` describes the same boring document as `egotist n`. But although the concatenation associates to the right, it still takes quadratic time to construct the layout. Each nesting operation is carried out by running through the entire document. Try it and see.

The way to solve the first problem is to delay concatenation, representing a concatenated document by a list of its component documents. The way to solve the second problem is to delay nesting, representing a nested document by a pair consisting of an indentation to be applied only when necessary and the document it is to be applied to. Combining both solutions, we represent a document by a list of indentation-document pairs. Specifically, consider the function `toDoc` defined by

```
toDoc :: [(Int,Doc)] -> Doc
toDoc ids = foldr (:<>:) Nil [Nest i x | (i,x) <- ids]
```

We can now calculate a definition of a function `layr` such that

```
layr = layouts . toDoc
```

and then define a new version of `layouts` based on `layr`. We leave the details as an exercise, but here is the result:

```
layouts x = layr [(0,x)]
layr []                  = [""]
layr ((i,x :<>: y):ids) = layr ((i,x):(i,y):ids)
layr ((i,Nil):ids)       = layr ids
layr ((i,Line):ids)      = ['\n':replicate i ' ' ++ ls
                           | ls <- layr ids]
layr ((i,Text s):ids)    = [s ++ ls | ls <- layr ids]
layr ((i,Nest j x):ids) = layr ((i+j,x):ids)
layr ((i,Group x):ids)  = layr ((i,flatten x):ids) ++
                           layr ((i,x):ids)
```

This definition takes linear time for each layout. Exactly the same template is used for the function `pretty`, which chooses a single best layout:

```
pretty w x = best w [(0,x)]
 where
 best r []                  = ""
 best r ((i,x :<>: y):ids) = best r ((i,x):(i,y):ids)
 best r ((i,Nil):ids)       = best r ids
 best r ((i,Line):ids)      = '\n':replicate i ' ' ++
                              best (w-i) ids
 best r ((i,Text s):ids)    = s ++ best (r-length s) ids
 best r ((i,Nest j x):ids) = best r ((i+j,x):ids)
 best r ((i,Group x):ids)  = better r
                               (best r ((i,flatten x):ids))
                               (best r ((i,x):ids))
```

The first argument of `best` is the remaining space available on the current line. This function is made local to the definition of `pretty` to avoid having to carry around the maximum line width `w` as an additional argument.

That leaves us with the problem of computing `better r lx ly`. Here we can make use of the fact that the first line of `lx` is guaranteed to be at least as long as the first line of `ly`. Thus it suffices to compare the length of the first line of `lx` with `r`. If the former fits within the latter, we choose `lx`; otherwise we choose `ly`. We therefore define

```
better r lx ly = if fits r lx then lx else ly
```

But we don't want to compute the length of the whole of the first line of `lx` since that looks ahead too far. Instead, we take a more miserly approach:

```
fits r _ | r<0 = False
fits r []      = True
fits r (c:cs)  = if c == '\n' then True
                 else fits (r-1) cs
```

For exactly the same reason it is essential that the second and third arguments to `better` are computed lazily, that is, the two layouts are evaluated just enough to determine which is the better one, and no further.

Let's revisit our troublesome paragraph:

```
ghci> putStrLn $ pretty 30 $ para pg
This is a fairly short
paragraph with just twenty-two
words. The problem is that
pretty-printing it takes time,
in fact 31.32 seconds.
(0.00 secs, 1602992 bytes)
```

Much better. Exercise L discusses what we can say about the running time of `pretty`.

The final task is to put our small library together as a module. Here is the main declaration:

```
module Pretty
   (Doc, Layout,
    nil, line, text,
    nest, (<>), group,
    layouts, pretty, layout) where
```

The module name is `Pretty` and the file containing the above declaration and the definitions of the library functions has to be saved in a file called `Pretty.lhs`.

The module exports 11 entities. Firstly, there is the name `Doc` of the abstract type of documents. The constructors of this type are not exported. (By the way, if we did want to export all the constructors we can write `Doc ()` in the export list, and if we wanted just, say, `Nil` and `Text`, we can write `Doc (Nil, Text)`.) Secondly, there is the name `Layout` which is just a synonym for `String`. The next eight constants and functions are the ones we have defined above. The final function `layout` is used for printing a layout:

```
layout :: Layout -> IO ()
layout = putStrLn
```

And that's it. Of course, in a really useful library a number of additional combinators could be provided. For example, we could provide

```
(<+>),(<|>) :: Doc -> Doc -> Doc
x <+> y = x <> text " " <> y
x <|> y = x <> line <> y

spread,stack :: [Doc] -> Doc
spread = foldr (<+>) nil
stack  = foldr (<|>) nil
```

No doubt the reader can think of many others.

8.7 Exercises

Exercise A

A picky user of the library wants just three layouts for a certain document:

```
A B C       A B       A
            C         B C
```

Can the user do it with the given functions?

Exercise B

The layouts of a document are given as a list. But are they all different? Either prove that they are or give a counterexample.

By the way, is it obvious from the laws that each document has a nonempty set of layouts?

Exercise C

The next four exercises refer to the shallow embedding of Section 8.3. Prove, by equational reasoning, that

```
nest i . nest j  = nest (i + j)
```

You will need a subsidiary result about `nestl`, which you don't have to prove.

Exercise D

Continuing on from the previous question, prove that

```
nest i (group x) = group (nest i x)
```

by equational reasoning (at the point level). Again, you will need a subsidiary result.

Exercise E

Continuing on, prove that `flatten . group = flatten`. You will need a subsidiary result.

Exercise F

The final law is `flatten . nest i = flatten`. And, yes, you will need yet another subsidiary result.

Exercise G

We said in the text that the prelude function `lines` breaks up a string on newline characters. In fact, `lines` treats a newline as a terminator character, so both `lines "hello"` and `lines "hello\n"` return the same result. It is arguable that a better definition treats newlines as *separator* characters, so there is always one more line than there are newlines. Define a function `lines` that has this behaviour. We will need the new definition below.

Now, the proof that `map shape` applied to the layouts of a document returns a lexicographically decreasing sequence of list of integers can be structured into the following steps. First, define

```
msl   = map shape . layouts
shape = map length . lines
```

where `lines` refers to the revised version above. We have to prove that `msl` returns a decreasing sequence on every document. To this end, we can define functions `nesty` and `groupy` so that

```
nesty i . msl = msl . nest i
groupy . msl  = msl . group
```

and an operation `<+>` so that

```
msl x <+> msl y = msl (x <> y)
```

(It is this equation that requires the revised definition of `lines`.) The proof is then completed by showing that if `xs` and `ys` are decreasing, then so are `nesty i xs` and `groupy xs` and `xs <+> ys`. All this exercise asks though is that you construct definitions of `nesty`, `groupy` and `<+>`.

Exercise H

Write a function `doc :: Doc -> Doc` that describes how to lay out elements of `Doc` where `Doc` is the abstract syntax tree representation in Section 8.6.

Exercise I

Consider a function `prettybad` that chooses a best layout from the list `layouts` by taking the first layout all of whose lines fit within the given width, and the last layout if this is not possible. Does `prettybad` always compute the same layout as `pretty`? (Hint: think about paragraphs.)

Exercise J

Using the algebraic properties of the constructors of `Doc`, calculate the efficient version of `layouts`.

Exercise K

We have designed `pretty w` to be *optimal*, meaning that it chooses line breaks to avoid overflowing lines if at all possible. We also have that `pretty w` is *bounded*, meaning that it can make the choice about the next line break without looking at more than the next `w` characters of the input. Given that, what do you expect GHCi's response would be to the commands

```
layout $ pretty 5 $ para pg
layout $ pretty 10 $ cexpr ce
```

where

```
pg = "Hello World!" ++ undefined
ce = If "happy" (Expr "great") undefined
```

Exercise L

We cannot relate the cost of `pretty w x` to the size of `x` without saying what the size of a document is. Here is a reasonable measure:

```
size :: Doc -> Int
size Nil          = 1
size Line         = 1
size (Text s)     = 1
size (Nest i x)   = 1 + size x
size (x :<>: y)   = 1 + size x + size y
size (Group x)    = 1 + size x
```

Under this definition both the documents

```
nest 20 (line <> text "!")
nest 40 (line <> text "!")
```

have size two. But it takes twice as long to produce the second layout, so the cost of `pretty` cannot be linear in the document size.

Instead of having `pretty` produce the final layout, a string, we can interpose an additional data type of layouts:

```
data Layout = Empty
            | String String Layout
            | Break Int Layout
```

and define `layout :: Layout -> String` by

```
layout Empty        = ""
layout (String s x) = s ++ layout x
layout (Break i x)  = '\n':replicate i ' ' ++ layout x
```

We have

```
pretty w = layout . prettyl w
```

where the new function `prettyl` produces a `Layout` rather than a string. Define `prettyl`.

A fairer question to ask is whether `prettyl w x` takes linear time in the size of `x`. Does it?

8.8 Answers

Answer to Exercise A

No. There is no way of allowing both `A<0>B<1>C` and `A<1>B<0>C` without also having both of `A<0>B<0>C` and `A<1>B<1>C`. These four are given by the expression

```
group (A <> line <> B) <> group (line <> C)
```

Answer to Exercise B

The layouts of a document are not necessarily all different. For example

```
layouts (group (text "hello")) = ["hello","hello"]
```

Yes, it is obvious that each document has a nonempty set of layouts. Look at the laws of `layouts`. The basic documents have a nonempty list of layouts and this property is preserved by the other operations.

Answer to Exercise C

The calculation is:

```
  nest i . nest j
=   {definition of nest}
  map (nestl i) . map (nestl j)
=   {functor law of map}
  map (nestl i . nestl j)
=   {claim}
  map (nestl (i+j))
=   {definition of nest}
  nest (i+j)
```

The claim is that `nestl i . nestl j = nestl (i+j)`, which follows – after a short calculation – from

```
indent (i+j) = concat . map (indent i) . indent j
```

We omit the proof.

Answer to Exercise D

We reason:

```
  nest i (group x)
=   {definition of group}
  nest i (flatten x ++ x)
=   {since nest i = map (nestl i)}
  nest i (flatten x) ++ nest i x
=   {claim}
  flatten (nest i x) ++ nest i x
=   {definition of group}
  group (nest i x)
```

The claim follows from

```
  nest i . flatten
=   {since there are no newlines in flatten x}
  flatten
=   {since flatten . nest i = flatten (Exercise F)}
  flatten . nest i
```

Answer to Exercise E

We reason:

```
  flatten . group
=   {definition of flatten and group}
  one . flattenl . flattenl . head
=   {claim}
  one . flattenl . head
=   {definition of flatten}
  flatten
```

The claim is that `flattenl` is *idempotent*:

```
flattenl . flattenl = flattenl
```

This follows because `flattenl` returns a layout with no newlines.

By the way, it is the idempotence of `flattenl` that ensures all layouts of a document flatten to the same string. The only function that introduces multiple layouts is `group`, whose definition is

```
group x = flatten x ++ x
```

We have therefore to show that flattening the first element of this list gives the same string as flattening the second element. Thus we need to show

```
flattenl . head . flatten = flattenl . head
```

This follows at once from the definition of `flatten` and the idempotence of the function `flattenl`.

Answer to Exercise F

We reason:

```
  flatten . nest i
=   {definitions}
  one . flattenl . head . map (nestl i)
=   {since head . map f = f . head}
  one . flattenl . nestl i . head
=   {claim}
  one . flattenl . head
=   {definition of flatten}
  flatten
```

The claim is that `flattenl . nestl i = flattenl`.

Answer to Exercise G

We can define

```
lines xs = if null zs then [ys]
           else ys:lines (tail zs)
           where (ys,zs) = break (=='\n') xs
```

The function `groupy` is defined by

```
groupy :: [[Int]] -> [[Int]]
groupy (xs:xss) = [sum xs + length xs - 1]:xs:xss
```

The function `nesty` is defined by

```
nesty :: :: Int -> [[Int]] -> [[Int]]
nesty i = map (add i)
          where add i (x:xs) = x:[i+x | x <- xs]
```

The function `(<+>)` is defined by

```
(<+>) :: [[Int]] -> [[Int]] -> [[Int]]
xss <+> yss = [glue xs ys | xs <- xss, ys <- yss]
 where glue xs ys = init xs ++ [last xs + head ys] ++
                    tail ys
```

Answer to Exercise H

One possibility, which no doubt can be improved on:

```
doc :: Doc -> Doc
doc Nil        = text "Nil"
doc Line       = text "Line"
doc (Text s)   = text ("Text " ++ show s)
doc (Nest i x) = text ("Nest " ++ show i) <>
                 group (nest 2 (line <> paren (doc x)))
doc (x :<>: y) = doc x <> text " :<>:" <>
                 group (line <> nest 3 (doc y))
doc (Group x)  = text "Group " <>
                 group (nest 2 (line <> paren (doc x)))

paren x = text "(" <> nest 1 x <> text ")"
```

Answer to Exercise I

No. Consider a paragraph whose longest word is one character longer than the line width. In this case, `prettybad` will lay out each word on a single line, while `pretty` will still fill lines with groups of words provided they fit. For example:

```
ghci> putStrLn $ pretty 11 $ para pg4
A lost and
lonely
hippopotamus
went into a
bar.
```

Answer to Exercise J

First we show `layouts x = layr [(0,x)]`:

```
  layr [(0,x)]
=   {definition of layr}
  layouts (toDoc [(0,x)])
=   {definition of toDoc}
  layouts (Nest 0 x :<>: Nil)
=   {laws of Doc}
  layouts x
```

It remains to give a recursive definition of `layr`. We will just give two clauses:

```
  toDoc ((i,Nest j x):ids)
=   {definition of toDoc}
  Nest i (Nest j x) :<>: toDoc ids
=   {laws}
  Nest (i+j) x :<>: toDoc ids
=   {definition of toDoc}
  toDoc ((i+j x):ids)
```

Hence `layr ((i,Nest j x):ids) = layr ((i+j x):ids)`. Next:

```
  toDoc ((i,x:<>:y):ids)
=   {definition of toDoc}
  Nest i (x :<>: y) <> toDoc ids
=   {laws}
  Nest i x :<>: Nest i y :<>: toDoc ids
=   {definition of toDoc}
  toDoc ((i,x):(i,y):ids)
```

Hence `layr ((i,x:<>:y):ids) = layr ((i,x):(i,y):ids)`.

Answer to Exercise K

```
ghci> layout $ pretty 5 $ para pg
Hello
World1*** Exception: Prelude.undefined
```

```
ghci> layout $ pretty 10 $ cexpr ce
if happy
then great
else *** Exception: Prelude.undefined
```

Answer to Exercise L

The definition is

```
prettyl :: Int -> Doc -> Layout
prettyl w x = best w [(0,x)]
 where
 best r []                  = Empty
 best r ((i,Nil):ids)       = best r ids
 best r ((i,Line):ids)      = Break i (best (w-i) ids)
 best r ((i,Text s):ids)    = String s (best (r-length s) ids)
 best r ((i,Nest j x):ids) = best r ((i+j,x):ids)
 best r ((i,x :<>: y):ids) = best r ((i,x):(i,y):ids)
 best r ((i,Group x):ids)  = better r
                                 (best r ((i,flatten x):ids))
                                 (best r ((i,x):ids))
```

where `better` is changed to read

```
better r lx ly = if fits r (layout lx) then lx else ly
```

The number of steps required to evaluate `better r` is proportional to `r` and thus at most `w`.

Now, `prettyl` takes linear time if `best` does. The second argument of `best` is a list of indentation-document pairs, and we can define the size of this list by

```
isize ids = sum [size x | (i,x) <- ids]
```

For each of the inner five clauses in the definition of `best`, the size decreases by 1. For instance

```
isize ((i,x :<>: y):ids)
= size (x :<> y) + isize ids
= 1 + size x + size y + isize ids
= 1 + isize ((i,x):(i,y):ids)
```

It follows that if we let $T(s)$ denote the running time of `best r` on an input of size s, then $T(0) = \Theta(1)$ from the first clause of `best`, and $T(s+1) = \Theta(1) + T(s)$ for

each of the five inner clauses, and

$$T(s+1) = \Theta(w) + \mathit{maximum}\ [T(k) + T(s-k) | k \leftarrow [1\,..\,s-1]]$$

for the last clause. And now we can deduce that $T(s) = \Theta(ws)$.

In conclusion, our algorithm for `pretty` is linear, though not independently of w.

8.9 Chapter notes

We referred to pretty-printing as a library, but another name for it is an *embedded domain specific language* (EDSL). It is a language for pretty-printing documents embedded in the host language Haskell. Many people believe that the growing success of Haskell is due to its ability to host a variety of EDSLs without fuss.

The detailed material in this chapter has been based closely on work by Philip Wadler, see 'A prettier printer', Chapter 11 in *The Fun of Programming* in *Cornerstones of Computing Series* (Palgrave MacMillan, 2003). The main difference is that Wadler used an explicit alternation operator in the term representation of `Doc` (though it was hidden from the user) rather than the constructor `Group`. Jeremy Gibbons suggested that the latter was a better fit with the idea of a deep embedding.

An earlier functional pretty-printing library based on a different set of combinators was described by John Hughes, 'The design of a pretty-printer library', in Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, Springer, 1995. Hughes' library was later reworked by Simon Peyton Jones and installed as a Haskell library

```
Text.PrettyPrint.HughesPJ
```

Another pretty-printing library, in an imperative rather than functional style, was constructed 30 years ago by Derek Oppen, 'Pretty-printing'. *ACM Transactions on Programming Languages and Systems* 2(4), 465–483, 1980 and is widely used as the basis of pretty-printing facilities in a number of languages. More recently, efficient pretty-printing algorithms in a functional style have been described by Olaf Chitil, 'Pretty printing with lazy dequeues', *ACM Transactions on Programming Languages and Systems* 27(1),163–184, 2005, and by Olaf Chitil and Doaitse Swierstra, 'Linear, bounded, functional pretty-printing', *Journal of Functional Programming* 19(1), 1–16, 2009. These algorithms are considerably more complicated than the one described in the text.