Pretty Printing

A brief primer on how code auto-formatters work

Pretty printing is an art and a science — the distinctly human endeavor of formatting code to look *pretty*.

I recently learned about how pretty printers work after I implemented one for my early-stage programming language, Poly, to formalize the language's style. In the process of researching pretty printers, I was amazed to find that the technique is over 40 years old and the elegant underlying algorithm back then is still in use in modern, popular tools like Prettier.

In this computational essay, we'll build an HTML formatter (and a bonus syntax highlighter) from the ground up to learn how pretty printing works.

```
<div class="observablehq-root">
  <div
    class="observablehq observablehq--worker observablehq--changed"
    dir="auto" style="margin-bottom: 17px;"
  >
    <div>
      <h1 id="pretty-printing">Pretty Printing</h1>
      <p>
        <em>
          A brief primer on how code auto-formatters work
        </em>
      </p>

      <p>
        Pretty printing is an art and a science — the
        distinctly human endeavor of formatting code to
        look <em>pretty</em> .
      </p>
```

real-time formatting of the source code of this notebook (computed in-browser)

Overview

Why format source code? In addition to making code look nicer, formatters...

- **help catch bugs** — formatted code reveals mistakes like mismatched braces.
- **unify code style** — eliminate debates on appropriate style by enforcing one.
- **improve developer experience** — make working with code editors pleasant. For example, the *"Format on save"* option provided in apps like Visual Studio Code is a satisfying way to prettify error-free code every time you hit save.
- **result in concise diffs** — repositories like GitHub show new changes, or *diffs*, in source code. Some formatting approaches aim to make diffs as small as possible for minor tweaks.

Documents

A *document* is a tree-like structure that can contain text, newlines, nested blocks, and unions of multiple possible layouts. The purpose of the document type is to provide the building blocks to represent how we want source text to be formatted. Here's how we define our document structure:

```
Doc = ▸ Object {nil: Object, text: ƒ(…), line: ƒ(…), nest: ƒ(indent, doc), concat: ƒ(docs), union: ƒ(a, b)}
```

Text

The simplest type of document is a *text document*, which just contains some text. Text documents should not contain newline characters, since line breaks are handled specially.

Line

Line documents represent line breaks and are useful in handling indentation and other behavior. Documents can be flattened to a single line with a flatten method we'll define below; when flattened, line breaks turn into the specified reduced text, typically a space or empty string.

Concatenation

Concatenation documents take a list of documents and join them into one document. For instance, the example below illustrates joining text and line documents to display some multi-line text.

```
Hello
world
```

Nesting

Indentation is commonly used in formatting to represent enclosed data. A *nest document* takes in a number and a sub-document, representing indenting the sub-document passed in a number of spaces. In practice, indenting means each occurrence of a newline is prepended with a number of spaces.

```
[
  arrayElement,
]
```

Union

A *union document* is the secret sauce to pretty printing. The other document types just define a fixed layout, with no mechanism to handle text wrapping. A union document stores two sub-documents that flatten to the same layout, where the first one has a longer first line. When pretty printing, the first document of the union is tried, and if the maximum line width is exceeded, the second document is rendered instead.

Small width:	*Large width:*
`Hi` `there`	`Hi there`

The Pretty Printing Algorithm

So, now we are equipped with a structure to describe how we might want to format documents and are ready to develop our pretty printing algorithm. We'll follow the approach outlined by Philip Wadler in "A Prettier Printer" (2003), which is a modern version of Derek Oppen's 1980 paper "Prettyprinting".

Let's outline our approach and some helper methods. The crux is we want a method that

consumes the document sequentially and outputs formatted text as it goes. The method should only have to look ahead a line at a time, and essentially it should optimistically try to place everything on the same line if possible, only breaking lines if needed.

The three helper methods that we'll define below are as follows:

- *lazy*: wraps a function so that it only has to be computed once and then the value is cached thereafter
- *layout*: returns a structure which stores some formatted text and a lazy function that returns the next layout structure (functions as a lazy linked list)
- *fits*: returns if there's enough room on the current line to output the passed in layout structure

We are setting up routines to lazily evaluate layout possibilities, which allows us to only compute what's needed in a performant manner while retaining the ability to express an exponential pool of possibilities.

```
lazy = ƒ(closure)
```

```
layout = ƒ(…)
```

```
fits = ƒ(remainingWidth, layoutDoc)
```

We're ready to define the *best* method, which finds the ideal layout structure given a *max line width* and document. To make the function work well, it's a recursive method that is actually passed a list of *document pairs,* which each contain a document structure and that document's current level of indentation. The method is also passed the *line width,* to account for how much room is left in the current line.

The method proceeds by examining the first document pair and recursively calling itself as necessary, expanding the list of document pairs to examine based on the first document type. Union documents are picked apart with the *fits* method, expanding to use the first document if there is room and the second if not. When text and line documents are encountered, they output layout structures which we can unravel later to reconstruct the formatted text.

```
best = ƒ(maxLineWidth, lineWidth, documentPairs)
```

Once *best* returns, it will output a layout structure, which is a linked list of text and lines. Let's add a convenience method to *unravel* the layout, which converts this linked list into an array of `{text, color}` objects which can then be rendered to the screen!

```
unravelLayout = ƒ(layoutDoc)
```

```
render = ƒ(unraveledLayout)
```

Finally, the *pretty* method combines all of the above, taking in the desired line width and the document and returning the unraveled best layout.

```
pretty = ƒ(lineWidth, doc)
```

Usage in Practice

Let's see how one could make a text-wrapping formatter function in practice. The definition of *union* above had an example with the text "Hi there" on one line or two depending on avaialble width. That can be expressed as follows:

```
hiThere = ▸ Object {type: "union", a: ƒ(), b: ƒ()}
```

Essentially, this union will display the text "Hi" and "there" separated by a space if there's room and a newline if there's not.

```
Hi there
```

```
Hi
there
```

In practice, there are many use cases where you want text to go on the same line if possible and separate lines otherwise. To help construct formatters, certain helper functions are useful.

The *flatten* method takes a document and returns a version where everything occurs on the same line. It does this by converting line documents to their reduced text (typically space or empty text) and always picking the first option in a union (which will always be the longer line by definition).

The *group* function is a convenience method to automatically construct a union of the

flattened version of a document with itself.

```
flatten = ƒ(doc)
```

```
group = ƒ(doc)
```

Our "Hi there" above example can be simplified with *group*:

```
hiThere2 = ▸ Object {type: "union", a: ƒ(), b: ƒ()}
```

```
Hi there
```

```
Hi
there
```

Formatting HTML code

Let's show a more practical pretty printer example. Generally, to put a pretty printer to use, we need to define methods on top of a parse tree to transform it into a document. As an example, we'll make a pretty printer for HTML, since the browser has built-in methods to parse HTML.

Let's start by defining an HTML structure that can contain elements and text nodes:

```
HTML = ▸ Object {element: ƒ(…), text: ƒ(text)}
```

Below, we define helpers to format HTML code. *showHTML* traverses the HTML structure, calling *showTag* to render HTML tags. *showList* renders a list of items without spaces in-between and relies on *show* to ensure list items are grouped in a way so that HTML tags tend to have lines to themselves. *showAttribute* renders the key/value pair for an HTML element attribute.

```
Color = ▸ Object {tag: "green", property: "#ae39e8", attribute: "blue"}
```

```
showHTML = ƒ(html)
```

```
showTag = ƒ(html, endTag)
```

```
showAttribute = ƒ(attr)
```

```
showList = ƒ(fn, l)
```

```
show = ƒ(l)
```

To prettify at HTML structure, we define *prettyHtml*, which pretty-prints the concatenated list-style output of *showHTML*.

```
prettyHtml = f(lineWidth, element)
```

Now, let's test the method with an example HTML structure.

```
testHtml = ▸ Object {type: "element", tag: "p", attributes: Array(3), children: Array(6)}
```

```
<p color="blue" font="Times" size="10">
  Here is some <em>emphasized</em> text. Here is a
  <a href="https://example.com">link</a> elsewhere.
</p>
```

```
<p
  color="blue"
  font="Times"
  size="10"
>
  Here is
  some
  <em>
    emphasized
  </em>
  text.
  Here is
  a
  <a
    href="https://example.com"
  >
    link
  </a>
  elsewhere.
</p>
```

Hooray, it works! And notice how we got syntax highlighting practically for free by painting the text as we format it.

Now for the fun part: JavaScript has free access to the page's HTML, which means we can write a method to convert an actual HTML element into the HTML structure we defined above.

```
htmlFromDom = f(element)
```

To get the HTML structure for the source code of this Observable notebook, we can call

htmlFromDom on the notebook's HTML element. At the very top of this notebook, I stored the rendered markdown of the intro paragraph as a variable called `doc`. It turns out, `doc.parentElement.parentElement` points to the HTML element that contains all the nodes in this notebook.

```
documentNodes = ▸ Object {type: "element", tag: "div", attributes: Array(1), children: Array(73)}
```

Let's render it! Try dragging the slider to change the print width and reformat the document. You can also scroll through the formatted document to see its full contents.

(Note: because there's so much HTML in this notebook, the formatted text is super slow to display using our `render` method (HTML-based). Instead, I define a `draw` method you can find below which renders to canvas instead and is much more performant.)

Print width: 76

```
<div class="observablehq-root">
  <div
    class="observablehq observablehq--worker observablehq--changed"
    dir="auto" style="margin-bottom: 17px;"
  >
    <div>
      <h1 id="pretty-printing">Pretty Printing</h1>
      <p><em>A brief primer on how code auto-formatters work</em></p>
      <p>
        Pretty printing is an art and a science — the distinctly human
        endeavor of formatting code to look <em>pretty</em> .
      </p>

      <p>
        I recently learned about how pretty printers work after I
        implemented one for my early-stage programming language,
        <a href="poly.dev">Poly</a> , to formalize the language's style. In
        the process of researching pretty printers, I was amazed to find
        that the technique is over 40 years old and the elegant underlying
```

So, there you have it. Hopefully this guide serves as a fun way to explore an interesting algorithm and an instructive guide for anyone trying to make a formatter for their language.

If you want to learn more, I recommend reading the papers that informed this piece:

- "A Prettier Printer" by Philip Wadler (2003)
- "Prettyprinting" by Derek Oppen (1980)

Supporting code for drawing and demoing

```
drawFontHeight = 12
```

```
drawLineHeight = 1.3
```

```
draw = ƒ(…)
```

```
display = ƒ(…)
```

```
displaySmallAndLargeWidth = ƒ(doc)
```

```
viewof demoLineWidth = ▸ EventTarget {value: 40}
```

```
table = ƒ(elem1, w1, elem2, w2)
```

```
update
```

```
import {set, Inputs} from "@observablehq/synchronized-inputs"
```