William Durand

On pretty printers

23 July 2021 — Freiburg, Germany

Pretty printers are tools used to format textual content according to a set of stylistic conventions. Prettier, black, rustfmt are great examples of such tools, which we call “code formatters” because they are applied to source code. Users can usually specify the maximum line length and the type of indentation (spaces or tabs) among other things but those two are responsible for endless debates in our industry 🤷.

After a “I wonder how that works” moment, I researched different ways to implement a code formatter and stumbled upon “A prettier printer” authored by Philipp Wadler. This paper introduces an *Intermediate Representation* (IR) used to format content that can then be printed with multiple possible layouts. This is the foundation of some popular code formatters like Prettier (see also: Prettier’s intermediate representation).

How does Wadler’s algorithm work?

A code formatter takes some code as input, parse it and translate the result into a more beautiful version of the code. The parsing step usually generates an Abstract Syntax Tree (AST), from which we can derive an intermediate representation as presented in Wadler’s paper.

Most implementations based on this paper use two possible layouts: *flat* and *broken*, controlled by the user-defined line length (a.k.a. “print width”). The code formatter should try to append as much content as possible on the same line (*flat* layout) and, when that isn’t possible, the *broken* layout, which describes how to break the content on multiple lines, should be preferred.

Example: pretty-printing JavaScript

In order to better understand how things work, let’s take an example with the JavaScript function definition below, taken from my very own implementation of Wadler’s algorithm in JS, which we are goint to format with that same implementation:

```
const renderDocument = (doc, fits  = DEFAULT_FITS, indentPrefix  = DEF
```

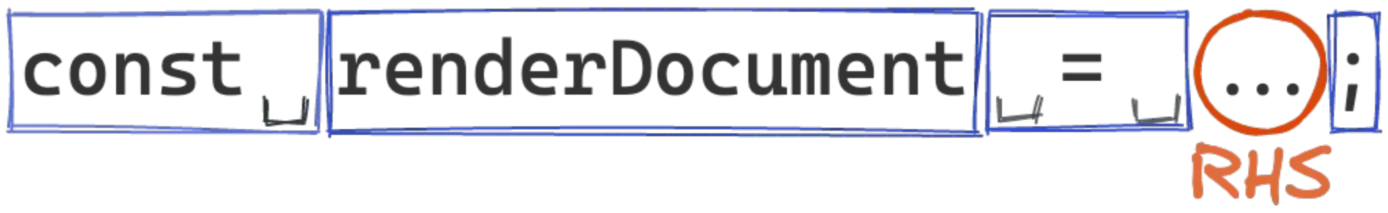
In order to format this code, we first need to parse it. Using the JavaScript parser espree, we get the following AST (some properties have been removed for brevity):

```
Node {
  type: 'Program',
  body: [
    Node {
      type: 'VariableDeclaration',
      kind: 'const',
      declarations: [
        Node {
          type: 'VariableDeclarator',
          id: Node {
            type: 'Identifier ',
            name: 'renderDocument'
          },
          init: Node {
            type: 'ArrowFunctionExpression',
            params: [
              Node {
                type: 'Identifier ',
                name: 'doc'
              },
              Node {
                type: 'AssignmentPattern',
                left: Node {
                  type: 'Identifier ',
                  name: 'fits '
                },
                right: Node {
                  type: 'Identifier ',
                  name: 'DEFAULT_FITS'
                }
              },
              Node {
                type: 'AssignmentPattern',
                left: Node {
                  type: 'Identifier ',
                  name: 'indentPrefix '
                },
                right: Node {
                  type: 'Identifier ',
                  name: 'DEFAULT_INDENT_PREFIX'
                }
              }
            ],
            body: Node {
              type: 'BlockStatement',
              body: []
            }
          }
        }
      ]
    }
  ]
}
```

Let’s walk through the IR generation now. Skipping the `Program` node (which represents the entire source code), we want to start pretty-printing a variable declaration (`VariableDeclaration` and `VariableDeclarator` nodes).

A simple IR [#](#)

We are only interested in the first part of our input for now, which is essentially the JavaScript code shown below where the Right-Hand Side (RHS) of the declaration has been replaced with `...`. We’ll get back to that part later.



In the meantime, this is convenient because it makes the intermediate representation simple: it is a list of strings. A list implies concatenation of all the items and a string is rendered as is.

```
// The JavaScript array below is our IR!
['const ', 'renderDocument', ' = ', '...', ';']
```

```
// This is the result of our IR once rendered:
const renderDocument = ...;
```

Note the semi-colon at the end, which isn't part of the input. When walking the JavaScript AST, we produce IR specifically for JavaScript. This is why the translation logic can append a semi-colon to `VariableDeclaration`. While the IR "building blocks" are generic, the parsing and translation layers are language-specific. Here is an example of translation logic for XML.

Describing different layouts

The RHS is an `ArrowFunctionExpression` and we want the pretty output to be rendered differently depending on the print width value because such functions might have long parameter names, default values, etc. The idea is to print all the arguments of the function on the same line if the total width is less than the print width, otherwise we want to print each argument on its own line. The IR should describe these two options, without having to specify actual values.

The `group` function (Prettier calls it a "command") is used to describe content that should be kept on a single line (like concatenation above) but, if it does not fit, should be broken at specific locations. These locations are defined with different *lines*:

- `HARDLINE`: specify a line break that is always included in the output, even if the expression does not fit on a single line
- `LINE`: specify a line break. If an expression fits on one line, the line break will be replaced with a space
- `SOFTLINE`: specify a line break. The difference from `LINE` is that if the expression fits on a single line, it will be replaced with nothing (`NIL`)

We want to break after each function argument so `LINE` looks like a good fit. Let's create a group and add LINE between each parameter (name, default value if any, and comma).

```
[
  // This is what we had before.
  "const ", "renderDocument", " = ",

  // This is NEW!
  {
    type: "group",
    contents: [
      "(",
      "doc", ",",
      LINE,
      "fits ", " = ", "DEFAULT_FITS", ",",
      LINE,
      "indentPrefix ", " = ", "DEFAULT_INDENT_PREFIX",
      ")", " => ", "{", "}",
    ]
  },

  // This is what we had before.
  ";"
]
```

This would produce the following pretty outputs:

```
//------------------------------------------------------------------
const renderDocument = (doc,
fits  = DEFAULT_FITS,
indentPrefix  = DEFAULT_INDENT_PREFIX) => {};
```

```
//------------------------------------------------------------------
const renderDocument = (doc, fits  = DEFAULT_FITS, indentPrefix  = DEF
```

Ugh! The *broken* layout version isn't pretty. We should probably break *after* the opening parenthesis using a `SOFTLINE`, align all the arguments with one level of indentation and probably break before the closing parenthesis as well.

We can use a new function named `nest()` that will change the current indentation level. This abstraction allows to later render the same input with 2 spaces, 4 spaces or even tabs. Indentation is only updated after a hard line, which is why we don't have to specify that it only applies to the *broken* layout.

Here is the final IR of our function:

```
[
  "const ", "renderDocument", " = ",
  {
    type: "group",
    contents: [
      "(",

      // This is NEW!
      {
        type: "nest",
        indent: 1,
        contents: [
          SOFTLINE,
          "doc", ",",
          LINE,
          "fits ", " = ", "DEFAULT_FITS", ",",
          LINE,
          "indentPrefix ", " = ", "DEFAULT_INDENT_PREFIX",
        ],
      },
      // This is NEW!
      SOFTLINE,

      ")", " => ", "{", "}",
    ],
  },
  ";",
]
```

The renderer

The logic to render the IR to pretty content is generic because the intermediate representation contains all the needed information. When the renderer finds a `group` (see here), it will measure the length of the output when its content is appended on the same line. If this length is lower than the print width, that's what gets rendered, otherwise the renderer switches to the *broken* layout.

`LINE` and `SOFTLINE` are implemented with a `flatChoice` function under the hood, which is defined as follows:

```
{
  type: 'flat-choice',
  whenBroken: '<used when layout is broken>',
  whenFlat: '<used when layout is flat>'
}
```

```
// This is how `LINE` is defined in my implementation.
const LINE = flatChoice(HARDLINE, " ");
```

There are other functions but `group` and `flatChoice` are the ones used to describe multiple layouts and therefore very important.

The IR presented in this article gives us some rendering configuration options "for free":

1. the desired print width: used to determine when to use the *broken* layout (which is used by a `fits()` function under the hood)
2. the "type" of indentation: used when the indentation level changes. With `nest()`, the IR tells the renderer to indent or dedent the content that follows by *N* levels. The renderer is free to use any number of spaces or tabs for a single indentation level
3. the control character to use to indicate the end of a line: while we probably always want `\n`, it could be any other character

Results

Rendering the full intermediate representation shown previously would give different outputs depending on the print width value:

```
//-------------------------------------------------------------------
const renderDocument = (
  doc,
  fits  = DEFAULT_FITS,
  indentPrefix  = DEFAULT_INDENT_PREFIX
) => {};
```

```
//-------------------------------------------------------------------
const renderDocument = (doc, fits  = DEFAULT_FITS, indentPrefix  = DEF
```

🎉🎉🎉

It is more complicated than that.

Is it *that* simple? **No.**

I have been lying to you! Did you see an example with a comment? Or with multiple logical blocks? This is where things start to be complicated.

Empty lines

Let's consider the example thereafter, which has two "logical blocks". First, we have two variables defined and grouped together and then a `console.log` call:

```
const a = 1;
const b = 2;

console.log(a + b);
```

Most users will naturally want to keep the blank line between the two blocks and no one will like tools that produce the following output:

```
const a = 1;
const b = 2;
console.log(a + b);
```

In order to keep logical blocks separated, we need extra logic to determine whether, after each statement, there is one or more blank lines. If that's the case, we can introduce a `HARDLINE` in the IR, which will also collapse multiple blank lines into a single one (here is how I implemented it for JS).

Here is a quote from the Prettier docs:

> *It turns out that empty lines are very hard to automatically generate. The approach that Prettier takes is to preserve empty lines the way they were in the original source code. There are two additional rules:*
>
> - *Prettier collapses multiple blank lines into a single blank line.*
> - *Empty lines at the start and end of blocks (and whole files) are removed. (Files always end with a single newline, though.)*

Depending on the programming language or parser (more on that in the next section), this might not be too difficult. As for comments, this is a completely different story!

Comments

First, most parsers aren't lossless. ASTs are by definition, well... *abstract*. Comments are usually not included in ASTs and when they are present, they aren't attached to nodes. This is a problem because where should comments be placed?

We can try to reattach comments to AST nodes if the parser provides locations or we could use a lossless parser like rowan. Prettier provides a framework to handle comments and it is a lot better than what I implemented to reattach comments.

Here is another quote from the Prettier docs:

> *When it comes to the content of comments, Prettier can't do much really. [...]*
>
> *Then there's the question of where to put the comments. Turns out this is*

> *a really difficult problem. Prettier tries its best to keep your comments roughly where they were, but it's no easy task because comments can be placed almost anywhere.*

Idiomatic patterns

The third constraint that we have when building a code formatter is to fine-tune the pretty output to respect some popular patterns. So far, we've seen how to render the same content depending on the print width but we could go further and have specific layouts depending on the code itself.

Let's take an example with curried arrow functions (see this Prettier patch):

```
// This is the input, which looks quite good already.
const currying =
  (argument1) =>
  (argument2) =>
  (argument3) =>
  (argument4) =>
  (argument5) =>
  (argument6) =>
    foo;
```

With some trivial modifications to my implementation (required because the JS formatter in my demo project does not support *all* JS syntaxes), this is how it would be pretty-printed:

```
const currying = (argument1) => {
  return (argument2) => {
    return (argument3) => {
      return (argument4) => {
        return (argument5) => {
          return (argument6) => {
            return foo;
          };
        };
      };
    };
  };
};
```

Prettier used to print the input above like this:

```
const currying = (argument1) => (argument2) => (argument3) => (argum
  argument5
) => (argument6) => foo;
```

In both cases, this isn't ideal. We really want to keep the input as output because it is more readable in this case. While Wadler's IR and renderer will happily pretty-print any content, fine tuning the translation layer is the complicated part.

Conclusion

I know a lot more about code formatters than before. It is still quite limited to one popular algorithm but I dug enough to understand some of the challenges when it comes to pretty-printing content. This is a hard problem.

While implementing Wadler's algorithm isn't *that* complicated (and we can find many implementations), correctly handling comments and empty lines are crucial and there isn't a popular algorithm for that. Prettier is a great framework to build solid code formatters and it isn't limited to JavaScript: there are plugins for many languages.

It sounds like I felt in love with Prettier or something. No, but knowing how it really works and which problems it actually solves is super interesting and mind-blowing to me.

ℹ️ Feel free to fork and edit this post if you find a typo, thank you so much! This post is licensed under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) license.

Credits

Photo used on social media by Bank Phrom

Recent articles

- Moziversary #7 05 May 2025
- Firefox AI & WebExtensions 24 Mar 2025
- Senior Staff. 01 Mar 2025
- Moziversary #6 01 May 2024
- Introducing xpidump 08 Apr 2024
- Moziversary #5 01 May 2023
- GitHub Container Registry, Proxy and Synology 18 Mar 2023
- Containers and micro virtual machines 11 Jul 2022
- Deep dive into containers 21 Jun 2022
- Developing Firefox in Firefox with Gitpod 03 May 2022

Comments

You can interact on Mastodon or send me an email if you prefer.

GitHub // Mastodon // will@drnd.me 0xa509bcf1c1274f3b
Publications // Talks // Atom feeds: [All] [PHP] [Mozilla]