# Some thoughts on the design of rustfmt

## Use cases

A formatting tool can be used in different ways and the different use cases can affect the design of the tool. The use cases I'm particularly concerned with are:

- running on a whole repo before check-in
    - in particular, to replace the `make tidy` pass on the Rust distro
- running on code from another project that you are adding to your own
- using for mass changes in code style over a project

Some valid use cases for a formatting tool which I am explicitly not trying to address (although it would be nice, if possible):

- running 'as you type' in an IDE
- running on arbitrary snippets of code
- running on Rust-like code, specifically code which doesn't parse
- use as a pretty printer inside the compiler
- refactoring
- formatting totally unformatted source code

## Scope and vision

I do not subscribe to the notion that a formatting tool should only change whitespace. I believe that we should semantics preserving, but not necessarily syntax preserving, i.e., we can change the AST of a program.

I.e., we might change glob imports to list or single imports, re-order imports, move bounds to where clauses, combine multiple impls into a single impl, etc.

However, we will not change the names of variables or make any changes which *could* change the semantics. To be ever so slightly formal, we might imagine a compilers high level intermediate representation, we should strive to only make changes which do not change the HIR, even if they do change the AST.

I would like to be able to output refactoring scripts for making deeper changes though. (E.g., renaming variables to satisfy our style guidelines).

My long term goal is that all style lints can be moved from the compiler to rustfmt and, as well as warning, can either fix problems or emit refactoring scripts to do so.

### Configurability

I believe reformatting should be configurable to some extent. We should read in options from a configuration file and reformat accordingly. We should supply at least a config file which matches the Rust style guidelines.

There should be multiple modes for running the tool. As well as simply replacing each file, we should be able to show the user a list of the changes we would make, or show a list of violations without corrections (the difference being that there are multiple ways to satisfy a given set of style guidelines, and we should distinguish violations from deviations from our own model).

## Implementation philosophy

Some details of the philosophy behind the implementation.

### Operate on the AST

A reformatting tool can be based on either the AST or a token stream (in Rust this is actually a stream of token trees, but it's not a fundamental difference). There are pros and cons to the two approaches. I have chosen to use the AST approach. The primary reasons are that it allows us to do more sophisticated manipulations, rather than just change whitespace, and it gives us more context when making those changes.

The advantage of the tokens approach is that you can operate on non-parsable code. I don't care too much about that, it would be nice, but I think being able to perform sophisticated transformations is more important. In the future, I hope to (optionally) be able to use type information for informing reformatting too. One specific case of unparsable code is macros. Using tokens is certainly easier here, but I believe it is perfectly solvable with the AST approach. At the limit, we can operate on just tokens in the macro case.

I believe that there is not in fact that much difference between the two approaches. Due to imperfect span information, under the AST approach, we sometimes are reduced to examining tokens or do some re-lexing of our own. Under the tokens approach, you need to implement your own (much simpler) parser. I believe that as the tool gets more sophisticated, you end up doing more at the token-level, or having an increasingly sophisticated parser, until at the limit you have the same tool.

However, I believe starting from the AST gets you more quickly to a usable and useful tool.

### Heuristic rather than algorithmic

Many formatting tools use a very general algorithmic or even algebraic tool for pretty printing. This results in very elegant code, but I believe does not give the best results. I prefer a more ad hoc approach where each expression/item is formatted using custom rules. We hopefully don't end up with too much code due to good old fashioned abstraction and code sharing. This will give a bigger code base, but hopefully a better result.

It also means that there will be some cases we can't format and we have to give up. I think that is OK. Hopefully, they are rare enough that manually fixing them is not painful. Better to have a tool that gives great code in 99% of cases and fails in 1% than a tool which gives 50% great code and 50% ugly code, but never fails.

### Incremental development

I want rustfmt to be useful as soon as possible and to always be useful. I specifically don't want to have to wait for a feature (or worse, the whole tool) to be perfect before it is useful. The main ways this is achieved is to output the source code where we can't yet reformat, be able to turn off new features until they are ready, and the 'do no harm' principle (see next section).

### First, do no harm

Until rustfmt is perfect, there will always be a trade-off between doing more and doing existing things well. I want to err on the side of the latter. Specifically, rustfmt should never take OK code and make it look worse. If we can't make it better, we should leave it as is. That might mean being less aggressive than we like or using configurability.

### Use the source code as guidance

There are often multiple ways to format code and satisfy standards. Where this is the case, we should use the source code as a hint for reformatting. Furthermore, where the code has been formatted in a particular way that satisfies the coding standard, it should not be changed (this is sometimes not possible or not worthwhile due to uniformity being desirable, but it is a useful goal).

### Architecture details

We use the AST from syntex_syntax, an export of rustc's libsyntax. We use syntex_syntax's visit module to walk the AST to find starting points for reformatting. Eventually, we should reformat everything and we shouldn't need the visit module. We keep track of the last formatted position in the code, and when we reformat the next piece of code we make sure to output the span for all the code in between (handled by missed_spans.rs).

We read in formatting configuration from a `rustfmt.toml` file if there is one. The options and their defaults are defined in `config.rs`. A `Config` object is passed throughout the formatting code, and each formatting routine looks there for its configuration.

Our visitor keeps track of the desired current indent due to blocks ( `block_indent`). Each `visit_*` method reformats code according to this indent, `config.comment_width()` and `config.max_width()`. Most reformatting that is done in the `visit_*` methods is a bit hacky and is meant to be temporary until it can be done properly.

There are a bunch of methods called `rewrite_*`. They do the bulk of the reformatting. These take the AST node to be reformatted (this may not literally be an AST node from syntex_syntax: there might be multiple parameters describing a logical node), the current indent, and the current width budget. They return a `String` (or sometimes an `Option<String>`) which formats the code in the box given by the indent and width budget. If the method fails, it returns `None` and the calling method then has to fallback in some way to give the callee more space.

So, in summary, to format a node, we calculate the width budget and then walk down the tree from the node. At a leaf, we generate an actual string and then unwind, combining these strings as we go back up the tree.

For example, consider a method definition:

```
    fn foo(a: A, b: B) {
        ...
    }
```

We start at indent 4, the rewrite function for the whole function knows it must write `fn foo(` before the arguments and `) {` after them, assuming the max width is 100, it thus asks the rewrite argument list function to rewrite with an indent of 11 and in a width of 86. Assuming that is possible (obviously in this case), it returns a string for the arguments and it can make a string for the function header. If the arguments couldn't be fitted in that space, we might try to fallback to a hanging indent, so we try again with indent 8 and width 89.