MOTIVATION

Currently, `mato` parses markdown formatted files with a parser into an abstract syntax tree (AST). For example, the following little file:

```
This is a sentence^(And this is a footnote.)
that is quite long.
```

is parsed into an AST, like the following:

```
Document(DEFAULT,
  Cat(
    Cat(
      Cat(
        Literal("This is a sentence"),
        Footnote(
          Literal("And this is a footnote."))),
      Literal(" that is quite long.")),
    LineBreak))
```

But, not all information that is necessary to render a `groff` file is present in this AST. What is missing is outlined in the following paragraphs.

Preamble is missing from AST

Every `groff` file that is generated by `mato` has a *preamble*, i.e., a default section of `groff` macros, that defines a certain default style. The first few lines of the default preamble look like the following:

```
.PRINTSTYLE TYPESET
.PAGE 5.8i 9i 1i 1i .75i .75i
.PAPER A5
.FAMILY Minion
.PT_SIZE 11
.AUTOLEAD 2
```

This preamble is basically a character string, that is being read in during startup of `matopdf`[1] and spat out as is during rendering at the beginning of the document.

As it is not part of the AST, as shown above, it has to be communicated to the renderer in some other way. That other way is a **context** object. And that context object is then being consulted in the renderer.

We can see the code location, where the storing of the preamble happens in the constructor of an AST processor called `MetaDataExtractor`:

```
pub fn new(preamble: &str) -> Box<dyn Process> {
    let mut map = HashMap::new();
    let cp = preamble.to_string();
    if !cp.is_empty() {
        map.insert("preamble".to_string(),
          preamble.to_string());
    }
    Box::new(MetaDataExtractor {
      ctx: map, doc_type: "".to_owned() })
}
```

Here, we see, that a new `HashMap` is being created and the preamble is stored in there under a key called `preamble`.

An interesting aside, that already is hinting at the problem, is the AST line of the constructor, where a new object, or

[1] Or even during compile time.

struct of type `MetaDataExtractor` is created and the context object is being put in.

And we can find the location where said preamble is looked up again in the renderer:

```
if self.ctx.contains_key("preamble") {
    let value = self.ctx.get("preamble").unwrap();
    result = format!("{}\\n{}\\n", result, value);
    self.ctx.remove("preamble");
}
```

PROBLEM STATEMENT AND ANALYSIS

Problem statement

I claim that *context objects are a bad thing*.

But, why, you probably ask. Aren't they used almost everywhere in software development? Aren't they a vital part of many software architectures?

I see currently the following problems:

- *Loss of type safety* - You put strings into the context object and you look-up strings again. Nobody guards you against dumb errors, misspelling the key name. You have to look out for yourself and that convolutes your code, as you must guard against a key not being present, or no value being stored under a present key.
- *Loss of locality* - You put something into the context object at one place in your code and it can be very hard to find the places where the key is looked up again. The same argument applies vice versa.
- *Loss of conciseness* - Your code get's ugly, as you begin to look-up stuff in context objects. The context object tends to creep into every function parameter list.
- *Loss of expressiveness* - A function parameter list with a context object does not communicate which information is really needed by the function.

As all of these problems aggregate, I think it is favorable to strive towards other solutions. Solutions that are *type safe*, preserve *locality* and *conciseness* and keeps your function signatures expressive.

APPROACH

In my current understanding, it might be a good idea to store as much information in the AST as possible. That way it is at every moment obvious, what information is being communicated between different phases of processing. Furthermore, the rendering is much more obvious, as `rusts match` expressions help here.

Document Type as a first example

In the AST above, we see that `Document` has a member with a value of `DEFAULT`.

```
Document(DEFAULT,
...
```

Not obvious in that snapshot of an AST is that this is a member called `doc_type`:

```
pub enum Exp {
    Document(DocType, Box<Exp>),
...
```

Formerly, the document type was also stored in the context object. At some point in the past, I decided to investigate the possibilites of reducing the amount of information stored in the context object.

This member is the result of this endeavours.