GROPDF-ZIG DEV DIARY

MOTIVATION

In the past, I have written a little library, I call `mato` - short for Markdown Transformer framework[1]. In that library, I make heavy use of `groff` - GNU roff - which does the job of transforming output, my library generates, into PDF files. It does that fairly quickly, say around half a second for the average document.

But this step is by far the most expensive step in a whole chain of transformation steps: first `mato` reads the input file, and creates a abstract syntaxt tree from it. That tree is being processed several times and then rendered into `groff` input format. After that `groff` takes over. It is itself built in a similar way as a compiler, it has a front-end doing the parsing and a backend, doing the ouput generation. These backends are called *devices* in `groff` parlance.

One such device, I use heavily is the PDF device, implemented by `gropdf.pl`. As the name suggests, it is written in Perl[2] and thus rather expensive with regards to CPU consumption.

Because, I wanted to build something in `zig`, I decided, this would be a good toy project, to practise my skills.

FIRST ITERATION - GETTING SOMETHING

My first goal is to see some out put in macos' preview app. I have a sample PDF file copied over from a book called "PDF Explained" by O'Reilly. That book served as a starting

[1] I now notice for the first time, that the `o` character is out of place. It is a historical left-over, because initially, I called it `matote`, Markdown to TeX converter, but at some point dropped the TeX and switch it to groff, because it is much simpler and much quicker.
[2] Some of you might know this scripting language still!

point, but I soon learned, that the Adobe "PDF Reference 1.7" (see here).

Implementation

My first shot at implementing this with `zig` was basically creating a set of `structs` - probably as I would have done in `java` or `c++`: `PdfObject` which has a `PdfType` and a `PdfDict`, both type aliases for an `enum` in the first case and what zig thinks of for a string `[]const u8`.

Now, this is interesting: I chose to store the object type *in* the object itself, something, I never would have done in `java` or `c++`. I did this, because I had to integrate these `PdfObjects` into each other more concrete object:

```
const PdfPage = struct {
  pdfObject: PdfObject,
  ...
}
```

That is because zig has no notion of inheritance and or struct embedding.

Challenges

The first problem my current implementation has, is that the output stream is chosen in the pdf implementation. So, the first challgenge becomes, factoring that out. That was easily done, by letting main choose the writer and change the interface of `PdfDocument.print` to accept a that `writer: anytype`. This is what `zig` thinks of as templates.

The next challgenge might be the object model and hierarchy, or putting more text into the implementation. I've looked at a minimal grout output:

```
$ groff -Tascii -Z samples/simple/minimal.groff
x T ascii
x res 240 24 40
```

```
x init
p1
x font 1 R
f1
s10
V80
H0
md
DFd
tThis
wh24
tfile
wh24
tcontains
...
```

So, it should be fairly easy to write a reading loop interpreting this: Text, or words are being introduced by `t` and interword spaces with `wh`, in the first go I could ignore all the rest and I would have a very simple pdf device!