The tagpdf package, v0.1

Ulrike Fischer*

July 5, 2018

This package is not meant for normal document production.
You need a current expl3 version to use it.
This package is incomplete, experimental and quite probably contains bugs.
You need some knowledge about TEX, pdf and perhaps even lua to use it.

Issues, comments, suggestions should be added as issues to the github tracker:

https://github.com/u-fischer/tagpdf

Contents

*fischer@troubleshooting-tex.de

1. Introduction

Since many year the creation of accessible pdf-files with LaTeX which conform to the PDF/UA standard has been on the agenda of TeX-meetings. Many people agree that this is important and Ross Moore has done quite some work on it. There is also a TUG-mailing list and a webpage [2] dedicated to this theme.

But in my opinion missing are means to *experiment* with tagging and accessibility. Means to try out, how difficult it is to tag some structures, means to try out, how much tagging is really needed (standards and validators don't need to be right . . .), means to test what else is needed so that a pdf works e.g. with a screen reader. Without such experiments it is imho quite difficult to get a feeling about what has to be done, which kernel changes are needed, how packages should be adapted.

This package tries to close this gap by offering *core* commands to tag a pdf.[1]

My hope is that the knowledge gained by the use of this package will at the end allow to decide if and how code to do tagging should be part of the LaTeX kernel.

The package does not patch commands from other packages. It is also not an aim of the package to develop such patches. While at the end changes to various commands in many classes and packages will be needed to get tagged pdf files – and the examples accompaigning the package try (or will try) to show various strategies – these changes should in my opinion be done by the class, package and document writers themselves using a sensible API provided by the kernel and not by some external package that adds patches everywhere and would need constant maintenance – one only need to look at packages like tex4ht or bidi or hyperref to see how difficult and sometimes fragile this is.

So this package deliberately concentrates on the basics – and this already quite a lot, there are much more details involved as I expected when I started.

I'm sure that it has bugs. Bugs reports, suggestions and comments can be added to the issue tracker on github. `https://github.com/u-fischer/tagpdf`

[1] In case you don't know what this means: there will be some explanations later on.

1.1. Tagging and accessibility

While the package is named `tagpdf` the goal is actually *accessible* pdf-files. Tagging is *one* requirement for accessibility but there are others. I will mention some later on in this documentation, and – if sensible – I will also try to add code, keys or tips for them.

So the name of the package is a bit wrong. As excuse I can only say that it is shorter and easier to pronounce.

1.2. Engines and modes

The package works currently with pdflatex and lualatex.

The package has two modes: the *generic mode* which should work in theory with every engine and the *lua mode* which works only with lualatex.

I implemented the generic mode first. Mostly because my tex skills are much better than my lua skills and I wanted to get the tex side right before starting to fight with attributes and node traversing.

While the generic mode is not bad and I spent quite some time to get it working I nevertheless think that the lua mode is the future and the only one that will be usable for larger documents. pdf is a page orientated format and so the ability of luatex to manipulate pages and nodes after the TEX-processing is really useful here. Also with luatex characters are normally already given as unicode. The main problem with luatex is how to insert "fake spaces" between words.

1.3. References

My main reference was the free reference for pdf 1.7. [1]. This document is from 2006.

In the meantime pdf 2.0. has been released. I know that it contains also for accessibility relevant changes. But the specification is not available for free, also currently imho neither pdftex nor luatex actually target the creation of pdf 2.0. So I'm ignoring this for the moment.

1.4. Validation

pdf's created with the commands of this package must be validated:

- One must check that the pdf is *syntactically* correct. It is rather easy to create broken pdf: e.g. if a chunk is opened on one page but closed on the next page.
- One must check how good the requirements of the PDF/UA standard are followed *formally*.
- One must check how good the accessibility is *practically*.

Syntax validation and formal standard validation can be done with preflight of the (non-free) adobe acrobat. It can also be done also with the free PDF Accessibility Checker (PAC 3) [4]. There is also the validator veraPDF [3]. But I didn't try it yet and have no idea if it is useful here.

Practical validation is naturally the more complicated part. It needs screen reader, users which actually knows how to handle them, can test documents and can report where a pdf has real accessibility problems.

Preflight woes

Sadly validators can not be always trusted. As an example for an reason that I don't understand the adobe preflight don't like the list structure `L`. It is also possible that validators contradict: that the one says everything is okay, while the other complains.

1.5. Examples wanted!

To make the package usable examples are needed: example that demonstrates how various structures can be tagged and which patches are needed, examples for the test suite, examples that demonstrates problems.

> Feedback, contribuations and corrections are welcome!

All examples should use the tagpdfsetup key `uncompress` described in the next section so that uncompressed pdf are created and the internal objects and structures can be inspected and – hopefully soon – be compared by the l3build checks.

2. Setup

Activation needed!

When the package is loaded it will – apart from loading more packages and defining a lot of things – not do anything. You will have to activate it with `\tagpdfsetup`, see below.

2.1. Modes and package options

The package has two different modes: The **generic mode** works (in theory, currently only with pdftex and luatex) probably with all engines, the **lua mode** only with luatex. The differences between both modes will be described later. The mode can be set with package options:

`luamode`

This is the default mode. It will use the generic mode if the document is processed with pdflatex and the lua mode with lualatex.

`genericmode`

This will force the generic mode for all engines.

2.2. Setup and activation

The following command setups the general behaviour of the package. The command should be normally used only in the preamble (for a few keys it could also make sense to change them in the document).

`\tagpdfsetup{`⟨*key-val-list*⟩`}`

The key-val list understands the following keys:

`activate-mc` Boolean, initially false. Activates the code related to marked content.

`activate-struct` Boolean, initially false. Activates the code related to structures. Should be used only if `activate-mc` has been used too.

`activate-tree` Boolean, initially false. Activates the code related to trees. Should be used only if the two other keys has been used too.

`activate-all` Boolean, initially false. Activates everything, that normally the sensible thing to do.

`add-new-tag` See section 4 for a description.

`check-tags` Boolean, initially true. Activates some safety checks.

`compresslevel` Value is an integer between 0 and 9. It sets both the pdfcompresslevel and the pdfobjcompresslevel.

`log` Choice key, possible values `none`, `v`, `vv`, `vvv`, `all`. Setups the log level. Changing the value affects currently mostly the luamode: "higher" values gives more messages in the log. The current levels and messages have been setup in a quite ad-hoc manner and will need improvement.

`tabsorder` Choice key, possible values are `row`, `column`, `structure`, `none`. This decides if a `/Tabs` value is written to the dictionary of the page objects. Not really needed for tagging itself, but one of the things you probably need for accessibility checks. So I added it. Currently the tabsorder is the same for all pages. Perhaps this should be changed ....

luamode `tagunmarked` Boolean, initially true. When this boolean is true, the lua code will try to mark everything that has not been marked yet as an artifact. The benefit is that one doesn't have to mark up every deco rule oneself. The danger is that it perhaps marks things that shouldn't be marked – it hasn't been tested yet with complicated documents containing annotations etc.

`uncompress` Equivalent to using `compresslevel=0`.

3. Tagging

pdf is a page orientated graphic format. It simply puts ink and glyphs at various coordinates on a page. A simple stream of a page can look like this[2]:

```
stream
   BT
   /F27 14.3462 Tf                    %select font
   89.291 746.742 Td                  %move point
   [(1)-574(Intro)-32(duction)]TJ     %print text
   /F24 10.9091 Tf                    %select font
    0 -24.35 Td                       %move point
    [(Let's)-331(start)]TJ            %print text
    205.635 -605.688 Td               %move point
    [(1)]TJ                           %print text
   ET
endstream
```

From this stream one can extract the characters and their placement on the page but not their semantic meaning (the first line is actually a section heading, the last the page number). And while in the example the order is correct there is actually no garanty that the stream contains the text in the order it should be read.

Tagging means to enrich the pdf with information about the *semantic* meaning and the *reading order*. (Tagging can do more, one can also store all sorts of layout information like font properties and indentation with tags. But as I already wrote this package concentrates on the part of tagging that is needed to improve accessibility.)

3.1. Three tasks

To tag a pdf three tasks must be carried out:

mc-task

1. **The mark-content-task**: The document must add "labels" to the page stream which allows to identify and reference the various chunks of text and other content. This is the most difficult part of tagging – both for the document writer but also for the package code. At first there can be quite many chunks as every one is a leaf node of the structure and so often a rather small unit. At second the chunks must be defined page-wise – and this is not easy when you don't know where the page breaks are. At last some text is created automatically, e.g. the toc, references, citations, list numbers etc and it is not always easy to mark them correctly.

struct-task

2. **The structure-task**: The document must declare the structure. This means marking the start and end of semantically connected portions of the document (correctly nested as a tree). This too means some work for the document writer, but less than for the mc-task: at first quite often the mc-task and the structure-task can be combined, e.g. when you mark up a list number or a tabular cell or a section header; at second one doesn't have to worry about page breaks so quite often one can patch standard environments to declare the structure. On the other side a

[2] The appendix contains some remarks about the syntax of a pdf file

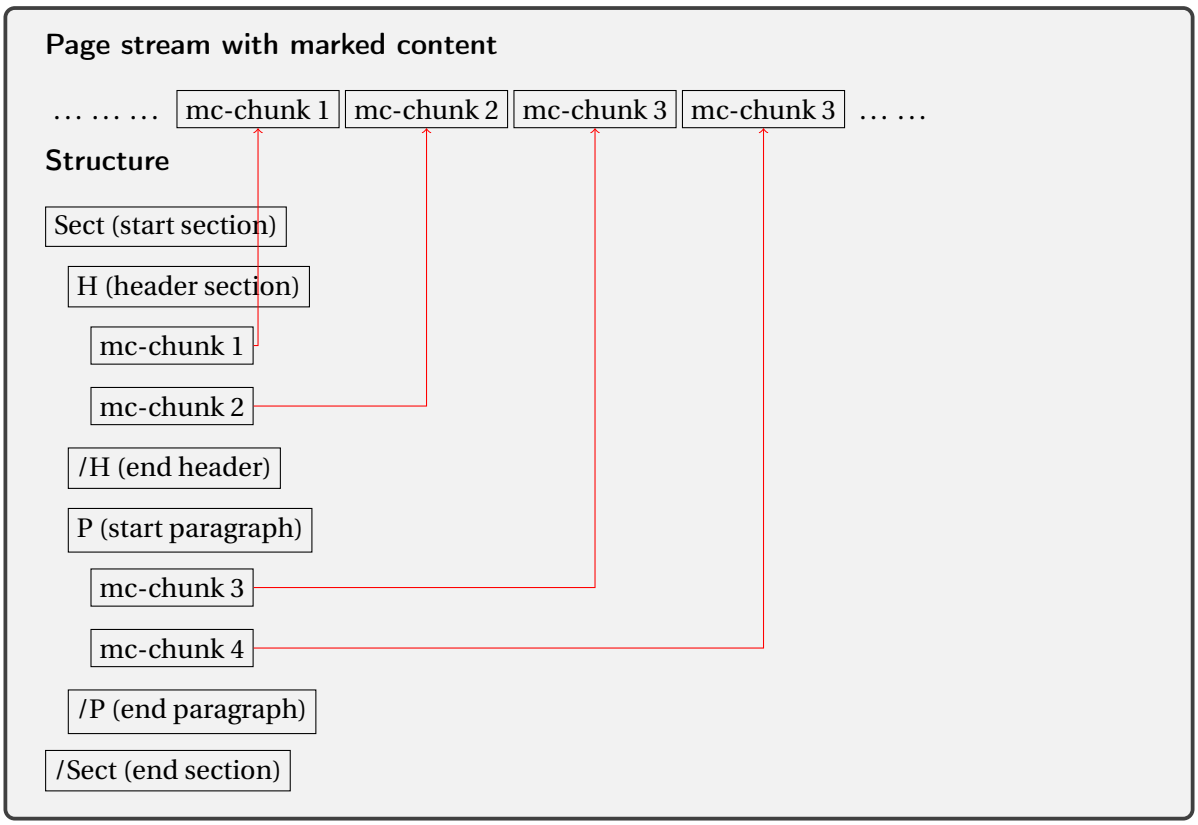


Figure 1: Schematical description of the relation between marked content in the page stream and the structure

number of structures end in LaTeX only implicitly – e.g. an item ends at the next item, so getting the pdf structure right still means that additional mark up must be added.

tree-task 3. **The tree management**: At last the structure must be written into the pdf. For every structure an object of type `StructElem` must be created and flushed with keys for the parents and the kids. A parenttree must be created to get a reference from the mc-chunks to the parent structure. A rolemap must be written. And a number of dictionary entries. All this is hopefully done automatically and correctly by the package ....

3.2. Task 1: Marking the chunks: the mark-content-step

To be able to refer to parts of the text in the structure, the text in the page stream must get "labels". In the pdf reference they are called "marked content". The three main variants needed here are:

Artifacts They are marked with of a pair of keywords, `BMC` and `EMC` which surrounds the text. `BMC` has a single prefix argument, the fix tag name `/Artifact`. Artifacts should be used for irrelevant text and page content that should be ignored in the structure. Sadly it is often not possible to leave such text simply unmarked – the accessibility tests in Acrobat and other validators complain.

```
/Artifact BMC
 text to be marked
/EMC
```

Artifacts with a type They are marked with of a pair of keywords, BDC and EMC which surrounds the text. BDC has two arguments: again the tag name /Artifact and a following dictionary which allows to specify the suppressed info. Text in header and footer can e.g. be declared as pagination like this:

```
/Artifact <</Type /Pagination>> BDC
 text to be marked
/EMC
```

Content Content is marked also with of a pair of keywords, BDC and EMC. The first argument of BDC is a tag name which describes the structural type of the text.[3] Examples are /P (paragraph), /H2 (header), /TD (table cell). The reference mentions a number of standard types but it is possible to add more or to use different names.

In the second argument of BDC – in the property dictionary – more data can be stored. *Required* is an /MCID-key which takes an integer as a value:

```
/H <</MCID 3>> BDC
 text to be marked
/EMC
```

This integer is used to identify the chunk when building the structure tree. The chunks are numbered by page starting with 0. As the numbers are also used as an index in an array they shouldn't be "holes" in the numbering system[4].

It is possible to add more entries to the property dictionary, e.g. a title, alternative text or a local language setting.

The needed markers can be added with low level code e.g. like this (in pdftex syntax):

```
\pdfliteral page {/H <</MCID 3>> BDC}%
 text to be marked
\pdfliteral page {EMC}%
```

This sounds easy. But there are quite a number of traps.

1. Pdf is a page oriented format. And this means that the start BDC/BMC and the corresponding end EMC must be on the same page. So marking e.g. a section title like in the following example won't always work as the literal before the section could end on the previous page:

   ```
   \pdfliteral page {/H <</MCID 3>> BDC} %problem: possible pagebreak here
    \section{mysection}
   \pdfliteral page {EMC}%
   ```

[3] There is quite some redundancy in the specification here. The structural type is also set in the structure tree. One wonders if it isn't enough to use always /SPAN here.

[4] It is perhaps possible to handle a numbering scheme not starting by 0 and having holes, but it will enlarge the pdf as one would need dummy objects.

Using the literals *inside* the section argument is better, but then one has to take care that they don't wander into the header and the toc.

2. Literals are "whatsits" nodes and can change spacing, page and line breaking. The literal *behind* the section in the previous example could e.g. lead to a lonely section title at the end of the page.
3. The `/MCID` numbers must be unique on a page. So you can't use the literal in a saved box that you reuse in various places. This is e. g. a problem with `longtable` as it saves the table header and footer in a box.
4. The `/MCID`-chunks are leaf nodes in the structure tree, so they shouldn't be nested.
5. Often text in a document is created automatically or moved around: entries in the table of contents, index, bibliography and more. To mark these text chunks correctly one has to analyze the code creating such content to find suitable places to inject the literals.
6. The literals are inserted directly and not at shipout. This means that due to the asynchronous page breaking of TEX the MCID-number can be wrong even if the counter is reset at every page (this package uses in generic mode a label-ref-system to get around this problem. This sadly means that three compilations are needed until everything has settled down).
7. There exist environments that process their content more than once – examples are `align` and `tabularx`. So one has to check for doublettes and holes in the counting system.
8. Pdf is a page oriented format. And this means that the start and the end marker must be on the same page ... *so what to do with normal paragraphs that split over pages??*. This question will be handled in subsection 3.2.5.

3.2.1. Generic mode versus lua mode in the mc-task

While in generic mode the commands insert the literals directly and so have all the problems described above the lua mode works quite differently: The tagging commands don't insert literals but set some *attributes* which are attached to all the following nodes. When the page is shipped out some lua code is called which wanders through the shipout box and injects the literals at the places where the attributes changes.

This means that quite a number of problems mentioned above are not relevant for the lua mode:

1. Pagebreaks between start and end of the marker are *not* a problem. So you can mark a complete paragraph. If a pagebreak occur directly after an start marker or before an end marker this can lead to empty chunks in the pdf and so bloat up pdf a bit, but this is imho not really a problem (compared to the size increase by the rest of the tagging).
2. The commands don't insert literals directly and so affect line and page breaking much less.
3. The numbering of the MCID are done at shipout, so no label/ref system is needed.
4. The code can do some marking automatically. Currently everything that has not been marked up by the document is marked as artifact. This can probably be extended and improved.

3.2.2. Commands to mark content and chunks

Generic mode only

It is vital that the end command is executed on the same page as the begin command. So think carefully how to place them. For strategies how to handle paragraphs that split over pages see subsection 3.2.5.

```
\tagmcbegin{⟨key-val-list⟩}
\uftag_mc_begin:n{⟨key-val-list⟩}
```

These commands insert the begin of the marked content code in the pdf. They don't start a paragraph. The user command additionally issues an `\ignorespaces` to suppress spaces after itself. Such markers should not be nested. The command will warn you if this happens.

The key-val list understands the following keys:

`tag` This is required, unless you use the `artifact` key. The value of the key is normally one of the standard type listed in section 4. It is possible to setup new tags, see the same section.

`artifact` This will setup the marked content as an artifact. The key should be used for content that should be ignored. The key can take one of the values `pagination`, `layout`, `page`, `background` and `notype` (this is the default). Text in the header and footer should be marked with `artifact=pagination`.

It is not quite clear if rules and other decorative graphical objects needs to be marked up as artifacts. Acrobat seems not to mind if not, but PAC 3 complained.

The validators complain if some text is not marked up, but it is not quite clear if this is a serious problem.

lua mode only

The lua mode will mark up everything unmarked as `artifact=notype`. You can suppress this behaviour by setting the tagpdfsetup key `tagunmarked` to false. See section 2.2.

`stash` Normally marked content will be stored in the "current" structure. This may not be what you want. As an example you may perhaps want to put a marginnote behind or before the paragraph it is in the tex-code. With this boolean key the content is marked but not stored in the kid-key of the current structure.

`label` This key sets a label by which you can call the marked content later in another structure (if it has been stashed with the previous key). Internally the label name will start with `tagpdf-`.

`raw` This key allows you to add more entries to the properties dictionary. The value must be correct, low-level pdf. E.g. `raw=/Alt (Hello)` will insert an alternative Text. (I will probably add keys for `/Alt` and `/Actualtext` later, but I haven't made up my mind regarding the encoding yes).

```
\tagmcend
\uftag_mc_end:
```

These commands insert the end code of the marked content. The user command also issues at first an \unskip. Both commands check if there has been a begin marker and issue a warning if not.

```
\tagmcuse{⟨labelname⟩}
\uftag_mc_use:n{⟨labelname⟩}
```

These commands allow you to record a marked content that you stashed away into the current structure. Be aware that a marked content can be used only once – the command will warn you if you try to use it a second time.

```
\tagmcifinTF{⟨truecode⟩}{⟨false⟩}
\_uftag_mc_if_in:TF{⟨truecode⟩}{⟨false⟩}
```

These commands check if a marked content is currently open and allows you to e.g. add the end marker if yes.

3.2.3. Tips

- Mark commands inside floats should work fine (but need perhaps some compilation rounds in generic mode).
- In case you want to use it inside a \savebox (or some command that saves the text internally in a box): If the box is used directly, there is probably no problem. If the use is later, stash the marked content and add the needed \tagmcuse directly before oder after the box when you use it.
- Don't use a saved box with markers twice.
- If boxes are unboxed you will have to analyze the pdf to check if everything is ok.
- If you use complicated structures and commands (breakable boxes like the one from tcolorbox, multicol, many footnotes) you will have to check the pdf.

3.2.4. Math

Math is a problem. I have seen an example where *every single symbol* has been marked up with tags from MathML along with an `/ActualText` entry and an entry with alternate text which describes how to read the symbol. The pdf then looked like this

```
/mn <</MCID 6 /ActualText<FEFF0034>/Alt( : open bracket: four )>>BDC
...
/mn <</MCID 7 /ActualText<FEFF0033>/Alt( third s )>>BDC
...
/mo <</MCID 8 /ActualText<FEFF2062>/Alt( times )>>BDC
```

If this is really the way to go one would need some script to add the mark-up as doing it manually is too much work and would make the source unreadable – at least with pdflatex and the generic mode. In lua mode is it probably possible to hook into the `mlist_to_hlist` callback and add marker automatically.

But I'm not sure that this is the best way to do math. It looks rather odd that a document should have to tell a screen reader in such detail how to read an equation. It would be much more efficient, sensible and flexible if a complete representation of the equation in mathML could be stored in the pdf and the task how to read this aloud delegated to the screen reader. More investigations are needed here.

3.2.5. Split paragraphs

Generic mode only

A problem are paragraphs with page breaks. As already mentioned the end marker `EMC` must be added on the same page as the begin marker. But it is in pdflatex *very* difficult to inject something at the page break automatically. One can manipulate the shipout box to some extend in the output routine, but this is not easy and it gets even more difficult if inserts like footnotes and floats are involved: the end of the paragraph is then somewhere in the middle of the box.

So with pdflatex in generic mode one currently has to do the splitting manually.

The example `mc-manual-para-split` demonstrates how this can be done. The general idea is to use `\vadjust` in the right place:

```
\tagmcbegin{tag=P}
...
fringilla, ligula wisi commodo felis, ut adipiscing felis dui in
enim. Suspendisse malesuada ultrices ante.% page break
\vadjust{\tagmcend\pagebreak\tagmcbegin{tag=P}}
Pellentesque scelerisque
...
sit amet, lacus.\tagmcend
```

3.3. Task 2: Marking the structure

The structure is represented in the pdf with a number of objects of type `StructElem` which build a tree: each of this objects points back to its parent and normally has a number of kid elements, which are either again structure elements or – as leafs of the tree – the marked contents chunks marked up with the `tagmc`-commands. The root of the tree is the `StructTreeRoot`.

3.3.1. Structure types

The tree should reflect the *semantic* meaning of the text. That means that the text should be marked as section, list, table head, table cell and so on. A number of standard structure types is predefined, see section 4 but it is allowed to create more. If you want to use types of your own you must declare them. E.g. this declares two new types `TAB` and FIG and base them on `P`:

```
\tagpdfsetup{
  add-new-tag = TAB/P,
  add-new-tag = FIG/P,
           }
```

3.3.2. Sectioning

The sectioning units can be structured in two ways: a flat, html-like and a more xml-like version. The flat version creates a structure like this:

```
<H1>section header</H1>
<P> text</P>
<H2>subsection header</H2>
...
```

So here the headers are marked according their level with `H1`, `H2`, etc.

In the xml-like tree the complete text of a sectioning unit is surrounded with the `Sect` tag, and all headers with the tag `H`. Here the nesting defines the level of a sectioning header.

```
<Sect>
 <H>section header</H>
 <P> text</p>
 <Sect>
   <H>subsection header</H>
   ...
 </Sect>
</Sect>
```

The flat version is more LaTeX-like and it is rather straightforward to patch `\chapter`, `\section` and so on to insert the appropriates `H...` start and end markers. The xml-like tree is more difficult to automate. If such a tree is wanted I would recommend to use – like the context format – explizit commands to start and end a sectioning unit.

3.3.3. Commands to define the structure

The following commands can be used to define the tree structure:

```
\tagstructbegin{key-val-list}
\uftag_struct_begin:n{key-val-list}
```

These commands start a new structure.

The key-val list understands the following keys:

`tag` This is required. The value of the key is normally one of the standard type listed in section **??**. It is possible to setup new tags/types, see section **??**.

`stash` Normally a new structure inserts itself as a kid into the currently active structure. This key prohibits this. The structure is nevertheless from now on "the current active structure" and parent for following marked content and structures.

`label` This key sets a label by which you can use the structure later in another structure. Internally the label name will start with `tagpdfstruct-`.

`title`,`alttext`,`actualtext` These keys allow to set the dictionary entries `/Title`, `/Alt` and `/Actualtext`. But I haven't yet decided which is the suitable format for the values, so currently you must ensure yourself that the values lead to valid pdf content.

```
\tagstructend
\uftag_struct_end:
```

This ends a structure.

```
\tagstructuse{⟨label⟩}
\uftag_struct_use:n{⟨label⟩}
```

These commands insert a structure previously stashed away as kid into the currently active structure. A structure should be used only once, if the structure already has a parent you will get a warning.

3.3.4. Root structure

A document should have at least one structure which contains the whole document. A suitable tag is `Document` or `Article`. I'm considering to automatically inserting it.

3.4. Task 3: tree Management

When all the document content has been correctly marked and the data for the trees has been collected they must be flushed to the pdf. This is done automatically (if the package has been activated) with the following command in `\AfterEndDocument`:

```
\uftag_finish_structure:
```

This will hopefully write all the needed objects and values to the pdf. (Beside the already mentioned `StructTreeRoot` and `StructElem` objects, additionally a so-called `ParentTree` is needed which records the parents of all the marked contents bits, a `Rolemap` and a few more values and dictionaries).

I'm not quite sure if this shouldn't be a really internal command.

3.5. A fully marked up document body

The following shows the marking need for a section, a sentence and a list with two items. It is obvious that one wouldn't want to do like this for real documents. If tagging should be usable, the commands must be hidden as much as possible inside suitable LaTeX commands and enviroments.

```
\begin{document}

\tagstructbegin{tag=Document}

 \tagstructbegin{tag=Sect}
  \tagstructbegin{tag=H}
   \tagmcbegin{tag=H} %avoid page break!
    \section{Section}
   \tagmcend
  \tagstructend
  \tagstructbegin{tag=P}
   \tagmcbegin{tag=P,raw=/Alt (x)}
    a paragraph\par x
   \tagmcend
  \tagstructend

 \tagstructbegin{tag=L} %List
  \tagstructbegin{tag=LI}
   \tagstructbegin{tag=Lbl}
    \tagmcbegin{tag=Lbl}
    1.
    \tagmcend
   \tagstructend
   \tagstructbegin{tag=LBody}
    \tagmcbegin{tag=P}
```

```
    List item body
    \tagmcend
   \tagstructend %lbody
  \tagstructend %Li

  \tagstructbegin{tag=LI}
   \tagstructbegin{tag=Lbl}
    \tagmcbegin{tag=Lbl}
    2.
    \tagmcend
   \tagstructend
   \tagstructbegin{tag=LBody}
    \tagmcbegin{tag=P}
    another List item body
    \tagmcend
   \tagstructend %lbody
  \tagstructend %Li
  \tagstructend %L

  \tagstructend %Sect
\tagstructend %Document
\tagfinish
\end{document}
```

4. Standard type and new tags

The pdf reference mentions a number of standard types: Document, Part, Art, Sect, Div, BlockQuote, Caption, TOC, TOCI, Index, NonStruct, H, H1, H2, H3, H4, H5, H6, P, L, LI, Lbl, Lbody, Table, TR, TH, TD, THead, TBody, TFoot, Span, Quote, Note, Reference, BibEntry, Code, Link, Annot, Figure, Formula, Form, Artifact

Their meaning can be looked up in the pdf-reference[5].

New tags can be defined in the setup command with the key `add-new-tag`. It takes a value consisting of two names separated by a slash. The first is the new name, the second a known (e.g. a standard) tag it should be mapped too. Example:

```
\tagpdfsetup{add-new-type = section/H1}
```

5. Accessibility is not only tagging

A tagged pdf is needed for accessibility but this is not enough. As already mentioned there are more requirements:

[5] https://wwwimages2.adobe.com/content/dam/acom/en/devnet/pdf/pdf_reference_archive/pdf_reference_1-7.pdf

- The language must be declared by adding a `/Lang xx-XX` to the pdf catalog or – if the language changes for a part of the text to the structure or the marked content – this can be rather easily done with existing packages.
- All characters must have an unicode representation or a suitable alternative text. With lualatex and open type (unicode) fonts this is normally not a problem. With pdflatex it could need

  ```
  \input{glyphtounicode}
  \pdfgentounicode=1
  ```

  and perhaps some`\pdfglyphtounicode` commands.
- Hard and soft hyphen must be distinct.
- Spaces between words should be space glyphs and not only a horizontal movement.
- Various small infos must be present in the catalog dictionary, info dictionary and the page dictionaries.

If suitable I will add code for this tasks to this packages. But some of them can also be done already with existing packages like hyperref, hyperxmp, pdfx.

6. To-do

- Add commands and keys to enable/disable the checks.
- Check/extend the code for language tags.
- Think about math.
- Think about Links/Annotations
- Keys for alternative and actualtext. How to define the input encoding? Like in Accsupp?
- Check twocolumn documents
- Examples
- Write more Tests
- Write more Tests
- “Fake spaces”
- Unicode
- Hyphenation char
- Think about included (tagged) pdf. Can one handle them?
- Improve the documentation
- Tag as proof of concept the documentation
- Document the code better

- Create dtx
- Find someone to check and improve the lua code
- Move more things to lua in the luamode
- Find someone to check and improve the rest of the code
- bidi?

References

[1] Adobe Systems Incorporated. *PDF Reference, sixth edition*. 2006. URL: https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf.

[2] TeX User Group. *PDF accessibility and PDF standards*. URL: https://tug.org/twg/accessibility/.

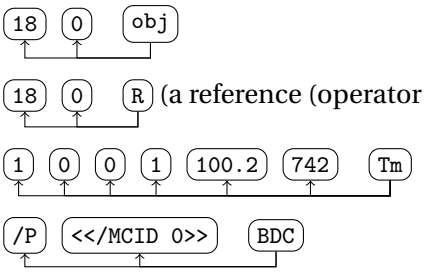
[3] veraPDF consortium. *veraPDF*. URL: http://verapdf.org/.

[4] "Zugang für alle" Schweizerische Stiftung zur behindertengerechten Technologienutzung. *PDF Accessibility Checker (PAC 3)*. URL: http://www.access-for-all.ch/ch/pdf-werkstatt/pdf-accessibility-checker-pac.html (visited on 07/05/2018).

A. Some remarks about the pdf syntax

This is not meant as a full reference only as a background to make the examples and remarks easier to understand.

postfix notation pdf uses in various places postfix notation. This means that the operator is behind its arguments:



Names pdf knows a sort of variable called a "name". Names start with a slash and may include any regular characters, but not delimiter or white-space characters. Uppercase and lowercase letters are considered distinct: `/A` and `/a` are different names. `/.notdef` and `/Adobe#20Green` are valid names.

Quite a number of the options of `tagpdf` actually define such a name which is later added to the pdf. I recommend *strongly* not to use spaces and exotic chars in such names. While it is possible to escape such names it is rather a pain when moving them through the various lists and commands and quite probably I forgot some place where it is needed.

Strings There are two types of strings: *Literal strings* are enclosed in round parentheses. They normally contain a mix of ascii chars and octal numbers:

`(gr\374\377ehello[]\050\051).`

Hexadezimal strings are enclosed in angle brackets. They allow for a representation of all characters the whole unicode ranges. This is the default output of lualatex.

`<003B00600243013D0032>`.

Arrays Arrays are enclosed by square brackets. They can contain all sort of objects including more arrays. As an example here an array which contains five objects: a number, an object reference, a string, a dictionary and another array. Be aware that despite the spaces `15 0 R` is *one* element of the array.

`[0 15 0 R (hello) <</Type /X>> [1 2 3]]`

`0` `15 0 R` `(hello)` `<</Type /X>>` `[1 2 3]`

Dictionaries Dictionaries are enclosed by double angle brackets. They contain key-value pairs. The key is always a name. The value can be all sort of objects including more dictionaries. It doesn't matter in which order the keys are given.

Dictionaries can be written all in one line:
`<</Type/Page/Contents 3 0 R/Resources 1 0 R/Parent 5 0 R>>` but at least for examples a layout with line breaks and indentation is more readable:

```
<<
  /Type      /Page
  /Contents  3 0 R
  /Resources 1 0 R
  /MediaBox  [0 0 595.276 841.89]
  /Parent    5 0 R
>>
```

(indirect) objects These are enclosed by the keywords `obj` (which has two numbers as prefix arguments) and `endobj`. The first argument is the object number, the second a generation number – if a pdf is edited objects with a larger generation number can be added. As with pdflatex/lualatex the pdf is always new we can safely assume that the number is always 0. Objects can be referenced in other places with the `R` operator. The content of an object can be all sort of things.

streams A stream is a sequence of bytes. It can be long and is used for the real content of pdf: text, fonts, content of graphics. A stream starts with a dictionary which at least sets the `/Length` name to the length of the stream followed by the stream content enclosed by the keywords `stream` and `endstream`

Here an example of a stream, an object definition and reference. In the object 2 (a page object) the `/Contents` key references the object 3 and this then contains the text of the page in a stream. `Tf`, `Tm` and `TJ` are (postfix) operators, the first chooses the font with the name `/F15` at the size 10.9, the second displaces the reference point on the page and the third inserts the text.

```
% a page object (shortened)
2 0 obj
  <<
    /Type/Page
    /Contents 3 0 R
    /Resources 1 0 R
    ...
  >>
endobj

%the /Contents object (/Length value is wrong)
3 0 obj
 <</Length 153 >>
 stream
  BT
   /F15 10.9 Tf 1 0 0 1 100.2 746.742 Tm [(hello)]TJ
  ET
 endstream
endobj
```

In such a stream the BT–ET pair encloses texts while drawing and graphics are outside of such pairs.

Number tree This is a more complex data structure that is meant to index objects by numbers. In the core is an array with number-value pairs. A simple version of number tree which has the keys 0 and 3 is

```
6 0 obj
<<
  /Nums [
          0 [ 20 0 R 22 0 R]
          3   21 0 R
        ]
>>
endobj
```

This maps 0 to an array and 2 to the object reference `21 0 R`. Number trees can be split over various nodes – root, intermediate and leaf nodes. We will need such a tree for the *parent tree*.