lowdown — simple markdown translator

Kristaps Dzonsons

lowdown — simple markdown translator

lowdown is a Markdown translator producing HTML5 and *roff* documents in the **ms** and **man** formats. It's just clean, secure, open source C code with no dependencies. Its canonical documentation is lowdown(1) for the formatter, lowdown(5) for the syntax, and the library interface at lowdown(3).

lowdown started as a fork of hoedown to add sandboxing (pledge(2), capsicum(4), or sandbox_init(3)) and *roff* output to securely generate PDFs on OpenBSD with just mandoc(1).

It can also be used to show the difference between two Markdown documents with *lowdown-diff*, documented as the lowdown diffing engine. This uses a tree-based difference algorithm to show semantically-valid Markdown differences. The implementation is inlined from libdiffi.

Want an example? For starters: this page, index.md. The Markdown input is rendered an HTML5 fragment using *lowdown*, then further using sblg. You can also see it as index.pdf, generated from groff(1) from **ms** output. Another example is the GitHub README.md rendered as README.html or README.pdf.

To get *lowdown*, just download, verify, unpack, run `./configure`, then run `doas make install` (or use `sudo`). *lowdown* is a BSD.lv project. Homebrew users can use BSD.lv's tap.

For an argument against using Markdown at all, read Ingo's comments on Markdown.

Output

Of course, *lowdown* supports the usual HTML output. Specifically, it produces HTML5 in XML mode. You can use *lowdown* to create either a snippet or standalone HTML5 document.

It also supports outputting to the **ms** macros, originally implemented for the *roff* typesetting package of Version 7 AT&T UNIX. This way, you can have elegant PDF and PS output by using any modern *troff* system such as groff(1).

Furthermore, it supports the **man** macros, also from Version 7 AT&T UNIX. Beyond the usual *troff* systems, this is also supported by mandoc.

You may be tempted to write manpages in Markdown, but please don't: use mdoc(7), instead — it's built for that purpose! The **man** output is for technical documentation only (section 7).

Both the **ms** and **man** output modes disallow images and equations. The former by definition (although **ms** might have a future with some elbow grease), the latter due to (not insurmountable) complexity of converting LaTeX to eqn(7).

You can control output features by using the **-D** (disable feature) and **-E** (enable feature) flags documented in lowdown.1.html.

Input

Beyond the basic Markdown syntax support, *lowdown* supports the following Markdown features and extensions:

- autolinking
- fenced code
- tables
- superscripts
- footnotes
- disabled inline HTML
- “smartypants”
- metadata
- commonmark (**in progress**)

You can control which parser features are used by using the **-d** (disable feature) and **-e** (enable feature) flags documented in lowdown.1.html.

Examples

I usually use *lowdown* when writing sblg articles when I’m too lazy to write in proper HTML5. (For those not in the know, sblg is a simple tool for knitting together blog articles into a blog feed.) This basically means wrapping the output of *lowdown* in the elements indicating a blog article. I do this in my Makefiles:

```
.md.xml:
     ( echo "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>" ; \
       echo "<article data-sblg-article=\"1\">" ; \
       echo "<header>" ; \
       echo "<h1>" ; \
       lowdown -X title $< ; \
       echo "</h1>" ; \
       echo "<aside>" ; \
       lowdown -X htmlaside $< ; \
       echo "</aside>" ; \
       echo "</header>" ; \
       lowdown $< ; \
       echo "</article>" ; ) >$@
```

If you just want a straight-up HTML5 file, use standalone mode:

```
lowdown -s -o README.html README.md
```

This can use the document’s meta-data to populate the title, CSS file, and so on.

The troff output modes work well to make PS or PDF files, although they will omit equations and only use local PS/EPS images in **-Tms** mode. The extra groff arguments in the following invocation are for UTF-8 processing (**-k** and **-Kutf8**), tables (**-t**), and clickable links and a table of contents (**-mspdf**).

If outputting PDF, use the pdfroff script instead of **-Tpdf** output. This allows image generation to work properly. If not, a blank square will be output in places of your images.

```
lowdown -s -Tms README.md | \
  groff -k -Kutf8 -t -mspdf > README.ps
lowdown -s -Tms README.md | \
  pdfroff -i -k -Kutf8 -t -mspdf > README.pdf
```

On OpenBSD or other BSD systems, you can run *lowdown* within the base system to produce PDF or PS files via mandoc:

```
lowdown -s -Tman README.md | mandoc -Tpdf > README.pdf
```

Read lowdown(1) for details on running the system.

Library

lowdown is also available as a library, lowdown(3). This effectively wraps around everything invoked by lowdown(1), so it's basically the same but... a library.

Testing

The canonical Markdown test, such as found in the original hoedown sources, will not currently work with *lowdown* because of the mandatory "smartypants" and other extensions.

I've extensively run AFL against the compiled sources with no failures — definitely a credit to the hoedown authors (and those from who they forked their own sources). I'll also regularly run the system through valgrind, also without issue.

Hacking

Want to hack on *lowdown*? Of course you do.

First, start in library.c. (The main.c file is just a caller to the library interface.) Both the renderer (which renders the parsed document contents in the output format) and the document (which generates the parse AST) are initialised.

The parse is started in document.c. It is preceded by meta-data parsing, if applicable, which occurs before document parsing but after the BOM. The document is parsed into an AST (abstract syntax tree) that describes the document as a tree of nodes, each node corresponding an input token. Once the entire tree has been generated, the AST is passed into the front-end renderers, which construct output depth-first.

There are three renderers supported: html.c for HTML5 output, nroff.c for **-ms** and **-man** output, and a debugging renderer tree.c.

A note on "real text".

The only time that input is passed directly into the output renderer is when then `normal_text` callback is invoked, blockcode or codespan, raw HTML, or hyperlink components. In both renderers, you can see how the input is properly escaped by passing into escape.c.

After being fully parsed into an output buffer, the output buffer is passed into a "smartypants" rendering, one for each renderer type.

Example

For example, consider the following:

```
## Hello **world**
```

First, the outer block (the subsection) would begin parsing. The parser would then step into the subcomponent: the header contents. It would then render the subcomponents in order: first the regular text

“Hello”, then a bold section. The bold section would be its own subcomponent with its own regular text child, “world”.

When run through the **-Ttree** output, it would generate:

```
LOWDOWN_ROOT
  LOWDOWN_DOC_HEADER
  LOWDOWN_HEADER
    LOWDOWN_NORMAL_TEXT
      data: 6 Bytes: Hello
    LOWDOWN_DOUBLE_EMPHASIS
      LOWDOWN_NORMAL_TEXT
        data: 5 Bytes: world
  LOWDOWN_DOC_FOOTER
```

This tree would then be passed into a front-end, such as the HTML5 front-end with **-Thtml**. The nodes would be appended into a buffer, which would then be passed back into the subsection parser. It would paste the buffer into `<h2>` blocks (in HTML5) or a `.SH` block (troff outputs).

Finally, the subsection block would be fitted into whatever context it was invoked within.

Known Issues (or, How You Can Help)

There are some known issues, mostly in PDF (**-Tms** and **-Tman**) output.

- There needs to be logic to handle when a link is the first or last component of a font change. For example, *[foo](...)* will put the font markers on different lines.
- Footnotes in **-Tms** with groff extensions should use pdfmark to link to and from the definition.
- In all modes, the “smartypants” formatting should be embedded in document output — not in a separate step as implemented in the original sources.