Eris Legal Markdown: Tutorials

Getting ELM Installed

Option A:

Install the executable from github's releases (https://github.com/eris-ltd/legalmarkdown/releases) somewhere in your path.

Option B:

Build from source. You must have go installed. Legalmarkdown has been tested against go 1.2 and go 1.3:

```
go get -u github.com/eris-ltd/legalmarkdown
cd $GOPATH/src/github.com/eris-ltd/legalmarkdown
go install
```

General Usage

After the library has finished its installation on your system then you can go to your command line and type

```
legalmarkdown parse --template [template_filename] --output [output_filename]
```

Legal Markdown will parse the file and write to the stated output. If you need to pipe from another command or into another command both of the `--template` (or `-t`) and `--output` (or `-o`) flags can be set to `-` which will read from stdin and write to stdout respectively. This is the command you will use if you want to output any text based document rather than a structured document. Again, even though it says markdown it should work for most text based systems.

If you have been working on a template or document and would like the library to build the YAML Front-Matter (see below) automatically for you, then simply type

```
legalmarkdown assemble --template [template_filename] --output [output_filename]
```

All these commands are available from within Go as well if you would prefer to call them programmatically.

```
import (
  "github.com/eris-ltd/legalmarkdown/lmd"
)

lmd.LegalToMarkdown(contents_file, parameters_file, output_file)

// OR

lmd.RawMarkdownToPDF(full_contents_as_string, full_params_as_string)
```

YAML Front-Matter

YAML (http://www.yaml.org/spec/1.2/spec.html) is easy thing to create. At the top of your file (it **MUST** be at the top of the file) you simply put in three hyphens like so: `---` on a single line. Then on the next line you simply put in the `field` followed by a `:` (colon) followed by the `value`. For each line you put the `[field]: [value]` until you have filled everything in that you need. After you have put in all your YAML front-matter then you simply put in a single line with three more hyphens `---` to signal to the library that it is the end of the fields. So YAML would typically look like this:

```
---
party1_address:    "Muj Axmed Jimcaale Road, Hargeisa, Republic of Somaliland"
party1_full:       "Watershed Legal Services, Ltd."
party1_reg:        "The Republic of Somaliland"
party1_rep:        "Mohomoud Abdirahman Nuur"
party1_short:      "(\"Watershed\")"
party1_type:       "private company limited by shares"
---
```

Be Careful with YAML

YAML can be quite testy, so if you use any symbols or parentheses or square brackets, etc. just put the entire field inside of double quotes (“). Also, if you need double quotes within the value then you “escape” them by putting a backslash before the double quotes as shown above. If you need to use a backslash (for instance if you are using latex), then you would escape the backslash by putting two backslashes. If you use the automatic YAML population feature of the library, it will handle this escaping for you.

Features of Eris’ Legal Markdown

Each of the functions below has been fully implemented in Eris' Legal Markdown. Which means that all of these functions are available for utilization directly from the Decerver (https://decerver.io)!

Mixins Function

Mixins are simple markers that can be used throughout the text to identify certain things (Court) or (Company) or (Client) to identify a few. The example above is the YAML Front Matter for mixins. This allows for the creation and utilization of templates that can be reused by simply updating the YAML front-matter and leaving the main text of the template largely untouched.

Mixins are structured in the form of **double curly** brackets. So, for a `` mixin within the body of your document, the YAML front-matter would look like this:

```
court: "Regional Court of Hargeisa"
```

If you do not want a mixin turned on for a particular document just add the mixin in the YAML Frontmatter and then leave it blank, the library will take it out of the text along with any extraneous spaces. If you have a mixin within the body of your document, but not within the YAML front matter, then the library will leave the mixin as is – which is unlikely the result you want to achieve.

Optional Clauses Function

When building templates for contracts, you often build optional clauses or clauses that are mutually exclusive to one another. This functionality is supported by ELM. Here is how to build an optional clause.

In the body of your document you put the entire clause in square-brackets (as you likely normally would) and at the beginning of the square bracket you put a mixin title for the optional clause.

In the YAML Front-Matter you simply add `true` or `false` as the YAML field to turn that entire clause on or off. **Note**, if you do not add the mixin to your header, ELM is just going to leave it as is, which is very unlikely to be what you want to see in your output file.

You are able to nest optional clauses inside of other optional clauses. However, if you do so, make sure that you include all of the sub-provisions of the master provision in the YAML matter, else the library will think that you closed your square brackets earlier than you would like it to.

Another thing to note, if you include nested provisions, you can turn off an inside provision and leave an outside provision on, but if you turn off an outside provision the entire portion will not be produced, even if you turned an inner portion on. Usually, as long as you keep

this rule in mind you can draft around it, and it is generally the case that that will be the result that you will want anyway.

So, this is how the body of the text would look.

```
[{{my_optional_clause}} Both parties agree that upon material breach of this agreement b
y either party they will both commit suicide in homage to Kurt Cobain.]
```

Not sure why you would ever write such a clause, but that is why the functionality exists! Then the YAML Front Matter would look like this:

```
my_optional_clause: true
```

or

```
my_optional_clause: false
```

Structured Headers Function

When creating many legal documents, but especially laws and contracts, we find ourselves constantly repeating structured headers. This gets utterly maddening when working collaboratively with various word processors because each word processor has its own styles and limitations for working with ordered lists and each user of even the same word processor has different defaults. In order to address this problem, we have built functionality into ELM that gets addresses this problem.

Here is how structured headers work in the library.

Wherever you wish to start a block of structured headers just put in ``` Three backticks (~ without the shift on US keyboards) at the beginning of the line. Then start the block of structured headers on the next line. When you are done with the block just put the same three backticks at the beginning of the line and continue your document. If the structured headers run to the end of the document, you do not need to close the backticks if you do not want to.

At the beginning of the line you simply type the level in which the provision resides by the number of lowercase l's (for level) followed by a period and then a space. So a top level provision (perhaps a `Chapter`, or `Article` depending on your document) will begin with `l. The provision ...` A second level provision (a Section or whatnot) will begin with `ll. Both parties agree ...` A third level provision will begin with `lll. Yaddy Yadda ...` And so on. These will reside in the body of the text.

When the library parses the document it will automatically increment and reset each level in the tree that you set up based on the criteria you establish. In the YAML front-matter you will describe the output functionality you need to see by adding the levels by: `level-1` and

then the `:` followed by what the format you would like it to be in. Currently these are the options for the reference portion of the structured header:

1. `level-1: 1.` will format that level of the list as 1. 2. 3. etc.; This is the default functionality;
2. `level-1: (1)` will provide for the same numbering only within parenteticals rather than followed by a period;
3. `level-1: A.` will format with capital letters followed by a period (e.g, A., B., C., etc.);
4. `level-1: (A)` will format the same as the above only with the capital letters in a parentetical;
5. `level-1: a.` will format with lowercase letters followed by a period;
6. `level-1: (a)` will format with lowercase letters within a parentethical;
7. `level-1: I.` will format with capital Roman numerals followed by a period;
8. `level-1: (I)` will format with capital Roman numerals within a parententical;
9. `level-1: i.` will format with lowercase Roman numerals followed by a period;
10. `level-1: (i)` will format with lowercase Roman numerals within a parententical..

Obviously you will replace `level-1` with `level-2`, for each relevant level of the tree. If you do not notate what levels will be used, ELM will simply leave the block as is, which is unlikely the desired behavior.

In addition to the reference portion of the structured header, you can add in whatever text you wish. For example, if you want the top level to be articles with a number and then a period, the next level down to be sections with a number in parentheses, and the next level down to be a letter in parentheses then this is what the YAML front matter would look like.

```
---
level-1: Article 1.
level-2: Section (1)
level-3: (a)
---
```

You can start on any number or letter you wish. So if you want the first article to be `Article 100.` instead of `Article 1.` there is no problem with that.

Note. Be careful if you want to start with letters that also match with Roman Numerals `I, V, X, L, C, D, M` (whether upper or lower case) as the library parses Roman's before letters. If you want a sequence similar to (a), (b), … but you put in (c) as the starting point the library will default to the lowercase version of the Roman Numeral C (100).

No Reset Function

Sometimes in legal documents (particularly in laws) you want to build multiple structured header levels, but you do not want to reset the headers when you are going up the tree. For example, in some laws you will have Chapters, Parts, Sections, … and you will want to track Chapters, Parts and Sections but when you go up to Parts you will not want to reset the Sections.

This functionality is built in with a `no-reset` function. You simply add a `no-reset` field to your YAML header and note the headers that you do not want to reset by their l., ll. notation. Separate those levels you do not want reset with commas. Example YAML line:

```
no-reset: l., ll., lll.
```

This will not reset level-1, level-2, or level-3 when it is parsing the document and those levels will be numbered sequentially through the entire document rather than reseting when going up the tree, levels not in this reset list, (e.g., llll. and lllll.) will be reset when going up a level in the tree. Obviously the level 1 headers will never reset because levels only reset when going a level higher and there is no level 0.

No Indent Function

By default, ELM will indent each level of the tree greater than level-1 by two spaces per level. So level-4 headers are indented six spaces, level-2 headers are indented two spaces, etc. This is the default because in many post-processors the indentation amount sets the level of the output.

However, you may want to keep some of the header levels tight to the margins. This functionality is built into ELM with a `no-indent` function. You simply add a `no-indent` field to your YAML header and not the headers you do not want to indent by their l., ll. notation. Separate those levels you do not want to reset with commas as with the `no-reset` function. Any levels *below* the last level in the `no-indent` function will be indented two spaces for each level.

Titles and Text or Provisions

Sometimes you want to have a title on one line and then some text on the next line all referencing the same provision. This is simple to achieve in a `lmd` document. You type your header level with an l., ll. notation and the text of the title after. On the next line, you just start the 'text' portion (meaning not the title) of the provision. Legal Markdown will figure out that you are separating text from title and parse it accordingly.

Working with Cross Reference Provisions.

Often we need the ability to cross reference between provisions where the text of Section 16 refers back to Section 12. When you're working with templates you may turn on or off provisions after reviewing a draft with a client and so you may not know if the reference point will be Section 12 or 14 in the final document. Also when you're working in a `lmd` file you do not see what the Section reference is within the document (that's the whole point).

Cross references only work within structured headers blocks. They do not work outside of that block. To use cross references, you simply place a reference key (which you can make up and remember, it can contain letters, numbers, or symbols) within pipes "|" (shift + the key above the enter key on US keyboards). First "stake" the cross reference to the provision which you want to reference to. Stakes should go directly after the ll., and before the text of the provision.

Then other provisions within the structured headers block can refer to it (either before or after the reference point within the document). Referencing provisions can utilize the cross references wherever is appropriate for the text. Note, cross references will use the *entire* replacement field so if you have leading text, pre, or preval utilized these will be brought into the cross reference within the text of the referencing provision.

For example, if the YAML front matter looked like this:

```
---
level-1: "# Article 1."
level-2: "Section 1."
level-3: (a)
no-indent: l., ll.
---
```

and the body of the text looked like this:

```
...
ll. |123| This provision will need to be referenced later.
ll. Provision
lll. As stated in |123|, whatever you need to say.
...
```

the output would look like this:

```
Section 7. This provision will need to be referenced later.

Section 8. Provision

    (a) As stated in Section 7, whatever you need to say.
```

Working with Partials

When work with templates it is nice to be a bit more DRY (don't repeat yourself). In order to help with this, ELM has a built in partials feature.

Let's say you put your standard interpretation, notice, severance, boilerplate typically at the end of the contract just before the signature block. Let's also assume that you have multiple contract templates and they all mostly use the same boilerplate final provisions.

If you were lawyering like coders think you would abstract these provisions into their own file within your contracts templates folder. Then you would have all of your templates reference back to that partial. Later, if there is some change in the law you would just go into the partial, make the necessary change in order to adapt your template based on the change, and then all of your templates which refer to that partial will be automatically updated. A bit more simple then updating each and every one of your templates, eh?

Another way to use partials is to build a new contract based on a template by simply opening a new file, importing the template, making the YAML front matter, filling in the front matter, and running the ELM parse. All you had to do was type one line, fill in some fields, and run two commands and you have a full contract directly from your template library.

Partials are simple. They use the `@import [filename]` syntax on a line by itself. So if your final provisions are kept in a file in the same folder called final_provisions.lmd you would put `@import final_provisions.lmd` on its own line (either within a structured headers block or outside of it) and the library will import the contents of the partial before parsing the document.

If your partial was located in another directory you may reference it either by using a relative directory or an absolute directory. To use a relative directory just type into your document as you would on the command line `@import gitlaw/contracts/commercial/partials/final_provisions.lmd` or wherever your partial is. To use an absolute directory just type into your document as you would on the command line `@import /home/compleatang/work/gitlaw/contracts/commercial/partials/final_provisions.lmd` or wherever your partial is.

Note. If you use relative directories, you will need to cd into the directory where the base file is before calling ELM from the command line. If that is a hassle, then you can use absolute paths and call from wherever you like.

Alternative Header Syntax

It can be a pain to count whether you are on level 5 or level 6 for a very complex document with multiple levels. To address this situation, ELM has an alternative header syntax besides the l., ll., lll. syntax. The alternative syntax uses l1., l2., l3., l4., … for level-1, level-2, etc. To use this syntax throughout your document you simply type in l1., l2., etc. into the body of your document, then you will also use that syntax for the no-reset and no-indent functions. Lastly, make a `level-style` field in your YAML front matter and put `l1.` as its value. Then the library will understand to utilize that syntax. By default this syntax is turned off as for the majority of documents it is actually fine to use l., ll., lll.

Date

When you are building documents sometime you simply want to put `date: @today` . Try it! At this point it formats dates according to standard formating outside of the US. But if you want to change that, then simply change the value of the field to `@today_us` . You do not need to have the name of the field be `date` ; indeed, it can be any field name, the import part is that the value of the field is `@today` or `@today_us` .

Signature Block

Want to have ELM build your signature block for you? Just type `@signature(party1:party2)` on a line.

Example

The syntax should be straight-forward. If you learn by seeing rather than by reading, take a look at the Watershed `lmd` repos (https://github.com/watershedlegal) where many ELM contract templates reside for some examples.

Let us say you wanted to output a document that looks like this:

```
Article 1. Provision for Article 1.

  Section 1. Provision for Section 1.1.

      1. Provision for 1.1.1.

      2. Provision for 1.1.2.

  Section 2. Provision for Section 1.2.

      1. Provision for 1.2.1.

      2. Provision for 1.2.2.
```

You can easily do that by doing the following steps.

Step 1: Type the body

```
l. Provision for Article 1.
ll. Provision for Section 1.1.
lll. Provision for 1.1.1.
lll. Provision for 1.1.2.
ll. Provision for Section 1.2.
lll. Provision for 1.2.1.
lll. Provision for 1.2.2.
```

Step 2(a): Fill out the YAML Front-Matter with the Base References you want

```
---
level-1: 1.
level-2: 1.
level-3: 1.
---
```

Step 2(b): (Optional) Add any additional text before the Base Reference

To output the document above, we need to be able to add the words Article and Section before the Base Reference. This is trivial. You just add those words into the YAML front matter before the base reference.

Note. The library looks at the last block of characters in the level-X YAML field to understand what type of Base Reference you need for your document. So you cannot add additional text *after* the Base Reference.

To achieve the above output, we should update our YAML front matter to look like this:

```
---
level-1: Article 1.
level-2: Section 1.
level-3: 1.
---
```

Step 2(c): (Optional) Add Precursors to Headers

Sometimes you want a document which looks like this:

```
Article 1. Provision for Article 1.

  Section 1.1. Provision for Section 1.1.

      1.1.1. Provision for 1.1.1.

      1.1.2. Provision for 1.1.2.

  Section 1.2. Provision for Section 1.2.

      1.2.1. Provision for 1.2.1.

      1.2.2. Provision for 1.2.2.
```

To output the document above, we need to be able to call the reference of the level above (also the level above that in the case of the above example). Legal Markdown allows for this

with the precursor function.

To reference the level above in your reference, you will modify the YAML front matter. In the YAML front matter you add any word or other marker *before* the precursor trigger. If you want to reference the preceding level (like 1.1.1 in the example above) then simply put in `pre` where that is appropriate.

To achieve the above output, we should update our YAML front matter to look like this:

```
---
level-1: "Article 1."
level-2: "Section pre 1."
level-3: "pre 1."
---
```

Step 2(d): (Optional) A Different Type of Precursor to Headers

Sometimes, particularly in laws, the structure is something akin to Chapter 1 and then Section 101, Section 102, … Chapter 9, Section 901, Section 902, etc. You can easily adopt this structure to your document by using the `preval` feature within the YAML front matter. If you combined this structure by also using markdown headers, the YAML front matter would look something like this:

```
---
level-1: "# Chapter 1."
level-2: "## Section preval 1."
level-3: "pre (a)"
---
```

This would output (using the same text from the body of the document typed in step 1) as:

```
# Chapter 1. Provision for Article 1.

  ## Section 101. Provision for Section 1.1.

    101(a) Provision for 1.1.1.

    101(b) Provision for 1.1.2.

  ## Section 102. Provision for Section 1.2.

    102(a) Provision for 1.2.1.

    103(b) Provision for 1.2.2.
...
  ## Section 110. Provision for Section 1.10.

    110(a) Provision for 1.10.1.

    110(b) Provision for 1.10.2.
```

Step 3: Run Legal-Markdown and Primary Processor

If you do not use the `pdf` outputting feature of ELM then run the output from the script through a primary processor.