

## Tutorial for Gigamonkeys Markup

Gigamonkeys Markup, or Markup, is a typesetting language from Peter Seibel that was successfully used to document large amounts of Lisp code in his book, *Practical Common Lisp* (<http://www.gigamonkeys.com/book/>).

The package is easy to install given the directions at <http://www.gigamonkeys.com/lisp/markup/>. From this point, I will assume that you have a working installation of Markup.

### Start Markup.

Load Emacs and start slime.

```
; SLIME 2005-12-22
CL-USER> (asdf:oos 'asdf:load-op :com.gigamonkeys.markup)
; compilation noise
CL-USER> , change-package <ret> COM.GIGAMONKEYS.MARKUP <ret>
MARKUP>
```

Now, create a new file in Emacs. `Playground.txt` will be used for this tutorial.

Add the following text to `Playground.txt`:

```
* My first Markup file!

This is paragraph of text that will be typeset by lisp.
Isn't it special?

How is it special?
```

Switch to the SLIME REPL and run:

```
MARKUP> (render :html "Playground.txt")
```

You now have an HTML file named `Playground.html`. Open it, and you will see `Playground1.html`.

You've noticed two things by now:

- Headings are prefixed by asteriks. This makes Markup play nicely with Emacs outline-mode.
- Paragraphs are seperated by newlines. This gives Markup the easy-edit WikiWiki nature.

### Adding more.

You should answer that question:

```
How is it special?

\bullets{
```

```

\item{A unique combination of TeX and Wiki markup.}
\item{Written completely in Lisp, which promises tighter
      integration in the future with Lisp tools such as Climacs.}
\item{Adding code examples is trivial.}
}

```

Switch to the SLIME REPL, render your file again. You should get `Playground2.html`.

## Code Examples

Code examples are done with four-space indentation. This is easy in Emacs. Either press space four times, or type `C-u <space>`. Once the first four are added, `<Tab>` or `C-j` will automatically put you in the right spot. Another option is to add a rectangle of four space rows with `C-x r t C-u <space> <ret>`.

I personally like the rectangle method, it makes me feel like reading the Emacs manual from cover to cover was a constructive use of my time as a child.

Add the following to `Playground.txt`:

```

Here is a code example:

CL-USER> (format t "~A~%" "Aesthetic Hello World!")

```

Render the html, and get `Playground3.html`.

## What else can I do?

Show, not tell:

```

*** Another level of headings!
**** Another!

Block quotes. Notice the two spaces
of indentation. The main difference between
block quotes and code examples is that code
examples are in a monospaced font.

Simple links:
\url{http://www.gigamonkeys.com}

Normal links:
\link{\href{http://www.gigamonkeys.com}\text{Gigamonkeys Consulting}}

Endnotes\note{This bit appears at the end.}

\sidebarhead{A Sidebar Heading!}
\sidebar{
  A larger section of text, offset for visibility.
}

```

Render it again, and get `Playground4.html`.

## Extend it!

Notice that the endnote was referenced with just a little number, not a link?

<http://www.UserScripts.org> has a script to fix this annoyance of Peter's pages, but we can do it

at the rendering level.

Each rendering method has its own lisp file. Find `html.lisp` in the Markup folder (possibly `~/sbcl/site/markup/`).

Find the following at the beginning of the file:

```
(defparameter *tag-translations* `((:example . :pre)
                                   (:sidebarhead . (:div :class "sidebarhead"))
                                   (:sidebar . (:div :class "sidebar"))
                                   (:note . (:div :class "note"))
                                   (:note-ref . :sup)
                                   (:bullets . :ul)
                                   (:item . :li)
                                   (:url . htmlize-url)
                                   (:link . htmlize-link)))

(defun htmlize-url (tag body)
  (declare (ignore tag))
  `(:a :href ,@body ,@body))
```

From these two top level forms, it is obvious that `*tag-translations*` is an alist mapping parsed tag types to either FOO tags or function names. The ones we will modify will be `note` and `note-ref`.

Change the values in the alist:

```
(:note . htmlize-note)
(:note-ref . htmlize-note-ref)
```

Make two functions:

```
(defun htmlize-note (tag body)
  (declare (ignore tag))
  `((:div :class "note") ,@body))

(defun htmlize-note-ref (tag body)
  (declare (ignore tag))
  `(:sup ,@body))
```

I've duplicated the existing functionality so that we can add gradual changes. I've copied the `declare` forms because `tag` contains the tag, and we already know what it is: `note`.

HTML links within the same file involve creating an anchor and a link to it. The anchor must have a unique name, and for this I will use a global counter. To keep the references and the actual notes in sync, I will use two, in fact.

Add this above the functions to create the counters:

```
(defvar *note-counter* 0)
(defvar *note-ref-counter* 0)
```

Now, we can use them to include an anchor in `htmlize-note`:

```
(defun htmlize-note (tag body)
  (declare (ignore tag))
  (incf *note-counter*)
  `((:div :class "note")
    ,@body
    ((:a :id ,*note-counter*))))
```

And to add the appropriate link in `htmlize-note-ref`:

```
(defun htmlize-note-ref (tag body)
  (declare (ignore tag))
  (incf *note-ref-counter*)
  `(:sup ((:a :href ,(format nil "#~A" *note-ref-counter*))
         ,@body)))
```

Try it! You might need to manually reset `*note-counter*` and `*note-ref-counter*` if you've been playing with the code. We haven't included anything to reset the counters on each render, so the two counters might not be in sync.

It is an easy fix. Add the following to the top of `render-as`'s definition:

```
(setf *note-counter* 0)
(setf *note-ref-counter* 0)
```

Try the render on `Playground.txt`, and you should get `Playground5.html`. (My example shows the following tweak.)

As an afterthought, add a link back. The number for the endnote is generated in `make-foo-html`, make it a link back to the body text.

Hint:<sup>1</sup> `Answer:answer.txt`

## Conclusions

Markup is a great library for documenting Common Lisp code with examples. Escaping code is kept to a minimum with the simple indentation method of marking blocks of text as example code.

The library is also clean and easy to modify and extend, as I have hopefully shown.

If I had done this with DocBook, it would've been a pain in the ass. Likewise, TeX, in all of its glory, probably would've been overkill for me throwing together a tutorial on Markup in the wee hours of the night.

Facts:

- DocBook is bad.
- Markup is good.
- Since Markup is good, all Common Lisp documentation tools should generate Markup.
- Profit.<sup>2</sup>

## Any questions? Ask pimaniac in #lisp.

---

1. Change the loop and the `htmlize-href` function.

2. The previous statement was questionable.