HyML Documentation

Release 1a

Marko Manninen

Aug 03, 2017

Contents

HyML (acronym for Hy Markup Language) is a set of macros to generate XML, XHTML, and HTML code in Hy.

HyML MiNiMaL macro is departed from the more extensive document and validation oriented "full" version of HyML. HyML MiNiMaL is meant to be used as a minimal codebase to generate XML (Extensible Markup Language).

Contents:

CHAPTER 1

HyML MiNiMaL

Minimal markup language generator in Hy

HyML (acronym for Hy Markup Language) is a set of macros to generate XML, XHTML, and HTML code in Hy.

HyML MiNiMaL (`ml`) macro is departed from the more extensive document and validation oriented "full" version of HyML.

HyML MiNiMaL is meant to be used as a minimal codebase to generate XML (Extensible Markup Language) with the next principal features:

1. closely resembling syntax with XML
2. ability to embed Hy / Python program code within markup
3. processing lists and external templates
4. using custom variables and functions

You can use HyML MiNiMaL:

- for static XML / XHTML / HTML content and file generation
- to render html code for the Jupyter Notebook, Sphinx docs, etc.
- to attach it to a server for dynamic html output generation (sHyte 0.2)
- for practice and study DSL and macro (Lisp) programming
- challenge your imagination

To compare with the full HyML XML / HTML macros, MiNiMaL means that there is no tag name validation and no tag and attribute minimize techniques utilized. If you need them, you should see full HyML documentation.

Note: HyML refers to XML, althought "Markup Language" -term itself is more generic term.

Ready, steady, go!

Project is hosted at: https://github.com/markomanninen/hyml

Install

For easy install, use pip Python repository installer:

```
$ pip install hyml
```

This will install only the necessary source files for HyML, no example templates nor Jupyter Notebook files.

There are no other dependencies except Hy language upon Python of cource. If Hy does not exist on your computer, it will be installed (or updated to the version 0.12.1 or greater) at the same time.

At the moment, the latest version of HyML is 0.2.6 (pre-release).

Import

Then import MiNiMaL macros:

```
(require (hyml.minimal (*)))
```

This will load the next macros for usage:

- main markup macro: `ml`
- `defvar` and `deffun` macros for custom variable and function setter
- `include` macro for using templates
- `list-comp*` list comprehension helper macro

Optionally `ml>` render macro will be loaded, if code is executed on Jupyter Notebook / IPython environment with the `display.HTML` function available.

If you intend to use xml code `indent` function, you should also import it:

```
(import (hyml.minimal (indent)))
```

With indent function you can make printed XML output prettier and more readable.

Run

Finally, run the simple example:

```
(ml (tag :attr "value" (sub "Content")))
```

That should output:

```
<tag attr="value"><sub>Content</sub></tag>
```

Tests

To run basic tests, you can use Jupyter Notebook document for now.

Jupyter Notebook

If you want to play with the provided HyML Notebook document, you should download the whole HyML repository (or clone it with `$ git clone https://github.com/markomanninen/hyml.git`) to your computer. It contains all necessary templates to get everything running as presented in the HyML MiNiMaL Notebook document.

HyML MiNiMaL codebase

Because codebase for HyML MiNiMaL implementation is roughly 60 lines only (without comments), it is provided here with structural comments and linebreaks for the inspection. More detailed comments are available in the minimal.hy source file.

```
; eval and compile variables, constants and functions for ml, defvar, deffun, and include macros
(eval-and-compile

  ; global registry for variables and functions
  (setv variables-and-functions {})

  ; internal constants
  (def **keyword** "keyword") (def **unquote** "unquote")
  (def **splice** "unquote_splice") (def **unquote-splice** (, **unquote** **splice**))
  (def **quote** "quote") (def **quasi** "quasiquote")
  (def **quasi-quote** (, **quote** **quasi**))

  ; detach keywords and content from code expression
  (defn get-content-attributes [code]
    (setv content [] attributes [] kwd None)
    (for [item code]
         (do (if (iterable? item)
                 (if (= (first item) **unquote**) (setv item (eval (second item) variables-and-functions))
                     (in (first item) **quasi-quote**) (setv item (name (eval item)))))
             (if-not (keyword? item)
               (if (none? kwd)
                   (.append content (parse-mnml item))
                   (.append attributes (, kwd (parse-mnml item)))))
             (if (and (keyword? kwd) (keyword? item))
                 (.append attributes (, kwd (name kwd))))
             (if (keyword? item) (setv kwd item) (setv kwd None))))
    (if (keyword? kwd)
        (.append attributes (, kwd (name kwd))))
    (, content attributes))

  ; recursively parse expression
  (defn parse-mnml [code]
    (if (coll? code)
        (do (setv tag (catch-tag (first code)))
            (if (in tag **unquote-splice**)
                (if (= tag **unquote**)
                    (str (eval (second code) variables-and-functions))
                    (.join "" (map parse-mnml (eval (second code) variables-and-functions))))
                (do (setv (, content attributes) (get-content-attributes (drop 1 code)))
```

```
                    (+ (tag-start tag attributes (empty? content))
                       (if (empty? content) ""
                           (+ (.join "" (map str content)) (+ "</" tag ">")))))))
        (if (none? code) "" (str code))))

  ; detach tag from expression
  (defn catch-tag [code]
    (if (and (iterable? code) (= (first code) **unquote**))
        (eval (second code))
        (try (name (eval code))
             (except (e Exception) (str code)))))

  ; concat attributes
  (defn tag-attributes [attr]
    (if (empty? attr) ""
        (+ " " (.join " " (list-comp
          (% "%s=\"%s\"" (, (name kwd) (name value))) [[kwd value] attr])))))

  ; create start tag
  (defn tag-start [tag-name attr short]
    (+ "<" tag-name (tag-attributes attr) (if short "/>" ">"))))

; global variable registry handler
(defmacro defvar [&rest args]
  (setv l (len args) i 0)
  (while (< i l) (do
    (assoc variables-and-functions (get args i) (get args (inc i)))
    (setv i (+ 2 i)))))

; global function registry handler
(defmacro deffun [name func]
  (assoc variables-and-functions name (eval func)))

; include functionality for template engine
(defmacro include [template]
  `(do (import [hy.importer [tokenize]])
       (with [f (open ~template)]
         (tokenize (+ "~@`(" (f.read) ")")))))

; main MiNiMaL macro to be used. passes code to parse-mnml
(defmacro ml [&rest code]
  (.join "" (map parse-mnml code)))

; macro -macro to be used inside templates
(defmacro macro [name params &rest body]
  `(do
    (defmacro ~name ~params
      `(quote ~~@body)) None))
```

Features

Basic syntax

MiNiMaL macro syntax is simple. It practically follows the rules of Hy syntax.

MiNiMaL macro expression is made of four components:

1. tag name
2. tag attribute-value pair(s)
3. tag text content
4. sub expression(s)

Syntax of the expression consists of:

- parentheses to define hierarchical (nested) structure of the document
- all opened parentheses `(` must have closing parentheses pair `)`
- the first item of the expression is the tag name
- next items in the expression are either:
  - tag attribute-value pairs
  - tag content wrapped with double quotes
  - sub tag expression
  - nothing at all
- between keywords, keyword values, and content there must a whitespace separator OR expression components must be wrapped with double quotes when suitable
- whitespace is not needed when a new expression starts or ends (opening and closing parentheses)

There is no limit on nested levels. There is no limit on how many attribute-value pairs you want to use. Also it doesn't matter in what order you define tag content and keywords, althougt it might be easier to read for others, if keywords are introduced first and then the content. However, all keywords are rendered in the same order they have been presented in the markup. Also content and sub nodes (expressions) are rendered similarly in the given order.

Main differences to XML syntax are:

- instead of < and > wrappers, parentheses `(` and `)` are used
- there can't be a separate end tag
- given expression does not need to have a single root node
- see other possible differences comparing HyML to wiki/XML

Special chars

In addition to basic syntax there are three other symbols for advanced code generation. They are:

- quasiquote `` ` ``
- unquote `~`
- unquote splice `~@`

These all are symbols used in Hy macro notation, so they should be self explanatory. But to make everything clear, in the MiNiMaL macro they may look they work other way around.

Unquote (`~`) and unquote-splice (`~@`) gets you back to the Hy code evaluation mode. And quasiquote (`` ` ``) sets you back to MiNiMaL macro mode. This is natural when you think that MiNiMaL macro is a quoted code in the first place. So if you want to evaluate Hy code inside it, you need to do it inside unquote.

Simple example

The simple example utilizing above features and all four components is:

```
(tag :attr "value" (sub "Content"))
```

`tag` is the first element of the expression, so it regarded as a tag name. `:attr "value"` is the keyword-value (attribute-value) -pair. `(sub` starts a new expression. So there is no other content (or keywords) in the `tag`. Sub node instead has content `"Content"` given.

Output would be:

```
<tag attr="value"><sub>Content</sub></tag>
```

Process components with unquote syntax (~)

Any component (tag name, tag attribute / value, and tag content) can be generated instead of hardcoded to the expression.

Tag name

You can generate a tag name with Hy code by using ~ symbol:

```
(ml (~(+ "t" "a" "g"))) ; output: <tag/>
```

This is useful, if tag names collide with Hy internal symbols and datatypes. For example, the symbol `J` is reserved for complex number type. Instead of writing: `(ml (J))` which produces `<1j/>`, you should use: `(ml (~"J"))` or `(ml ("J"))`.

Attribute name and value

You can generate an attribute name or a value with Hy by using ~ symbol. Generated attribute name must be a keyword type however:

```
(ml (tag ~(keyword (.join "" ['a 't 't 'r])) "value")) ; output: <tag attr="value"/>
```

And same for value:

```
(ml (tag :attr ~(+ 'v 'a 'l 'u 'e))) ; output: <tag attr="value"/>
```

Content

You can generate content with Hy by using ~ symbol:

```
(ml (tag ~(.upper "content"))) ; output: <tag>CONTENT</tag>
```

Using custom variables and functions

You can define custom variables and functions for the MiNiMaL macro. Variables and functions are stored on the common registry and available on the macro expansion. You can access predefined symbols when quoting ~ the expression.

```
; define variables with defvar macro
(defvar firstname "Dennis"
        lastname "McDonald")

; define functions with deffun macro
(deffun wholename (fn [x y] (+ y ", " x)))

; use variables and functions with unquote / unquote splice
(ml (tag ~(wholename firstname lastname)))
```

[output]

```
<tag>McDonald, Dennis</tag>
```

Process lists with unquote splice syntax (~@)

Unquote-splice is a special symbol to be used with the list and the template processing. It is perhaps the most powerful feature in the MiNiMaL macro.

Generate list of items

You can use list comprehension function to generate a list of xml elements. Hy code, sub expressions, and variables / functions work inside unquote spliced expression. You need to quote a line, if it contains a sub MiNiMaL expression.

```
; generate 5 sub tags and use enumerated numeric value as a content
(ml (tag ~@(list-comp `(sub ~(str item)) [item (range 5)])))
```

[output]

```
<tag><sub>0</sub><sub>1</sub><sub>2</sub><sub>3</sub><sub>4</sub></tag>
```

Using templates

One useful extension of the HyML MiNiMaL is that you can define code on external templates and `include` them for generation.

Let us first show the template content existing in the external file:

```
(with [f (open "note.hy")] (print (f.read)))
```

```
(note :src "https://www.w3schools.com/xml/note.xml"
  (to ~to)
  (from ~from)
  (heading ~heading)
  (body ~body))
```

Then we will define variables to be used inside the template:

```
(defvar to "Tove"
        from "Jani"
        heading "Reminder"
        body "Don't forget me this weekend!")
```

And finally use `include` macro to render the template:

```
(ml ~@(include "note.hy"))
```

[output]

```
<note src="https://www.w3schools.com/xml/note.xml">
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Danger: In HyML, but with templates especially, you must realize that inside macro there is a full access to all Hy and Python modules including file systems access and so on. This might raise up some security concerns that you should be aware of.

Templates extra

On HyML package there is also the `render-template` function and the `extend-template` macro available via `HyML.template` module.

`HyML.template` is especially useful when embedding `HyML MiNiMaL` to webserver, Flask for example. Here just the basic use case is shown, more examples you can find from sHyte 0.2 HyML Edition codebase.

In practice, `render-template` function is a shortcut to call `parse-mnml` with parameters and `include` in sequence. The first argument in `render-template` is the template name, and the rest of the arguments are dictionaries to be used on the template. So this is also an alternative way of using (bypassing the usage of) `defvar` and `deffun` macros.

```
; extend-template macro
(require [hyml.template [*]])
; render-template function
(import [hyml.template [*]])
; prepare template variables and functions
(setv template-variables-and-functions
     {"var" "Variable 1" "func" (fn[]"Function 1")})
; render template
(render-template "render.hyml" template-variables-and-functions)
```

[output]

```
<root><var>Variable 1</var><func>Function 1</func></root>
```

On template engines it is a common practice to extend sub template with main template. Say we have a layout template, that is used as a wapper for many other templates. We can refactor layout xml code to another file and keep the changing sub content on other files.

In `HyML MiNiMaL`, `extend-template` macro is used for that.

Lets show an example again. First we have the content of the `layout.hyml` template file:

```
(with [f (open "templates/layout.hyml")] (print (f.read)))
```

```
(html
  (head (title ~title))
  (body ~body))
```

And the content of the `extend.hyml` sub template file:

```
(with [f (open "templates/extend.hyml")] (print (f.read)))
```

```
~(extend-template "layout.hyml"
    {"body" `(p "Page content")})
```

We have decided to set the `title` as a "global" variable but define the `body` on the sub template. The render process goes like this:

```
(setv locvar {"title" "Page title"})
(render-template "extend.hyml" locvar)
```

[output]

```
<html><head><title>Page title</title></head><body><p>Page content</p></body></html>
```

Note that extension name `.hyml` was used here even though it doesn't really matter what file extension is used.

At first it may look overly complicated and verbose to handle templates this way. Major advantage is found when processing multiple nested templates. Difference to simply including template files `~@(include "templates/template.hy")` is that on `include` you pass variables (could be evaluated `HyML` code) to template, but on `extend-template` you pass unevaluated `HyML` code to another template. This will add one more dynamic level on using `HyML MiNiMaL` for XML content generation.

Template directory

Default template directory is set to `"template/"`. You can change directory by changing `template-dir` variable in the `HyML.template` module:

```
(import hyml.template)
(def hyml.template.template-dir "templates-extra/")
```

Macro -macro

One more related feature to templates is a `macro` -macro that can be used inside template files to factorize code for local purposes. If our template file would look like this:

```
~(macro custom [attr]
  `(p :class ~attr))

~(custom ~class)
```

Then rendering it, would yield:

```
(render-template "macro.hyml" {"class" "main"}) ; output: <p class="main"/>
```

Directly calling the `parse-mnml` function

You are not forced to use `ml` macro to generate XML. You can pass quoted code directly to `parse-mnml` function. This can actually be a good idea, for example if you want to generate tags based on a dictionary. First lets see the simple example:

```
(parse-mnml '(tag)) ; output: <tag/>
```

Then let us make it a bit more complicated:

```
; define contacts dictionary
(defvar contacts [
    {:firstname "Eric"
     :lastname "Johnson"
     :telephone "+1-202-555-0170"}
    {:firstname "Mary"
     :lastname "Johnson"
     :telephone "+1-202-555-0185"}])
(ml
  ; root contacts node
  (contacs
    ~@(do
      ; import parse-mnml function at the highest level of unquoted code
      (import (hyml.minimal (parse-mnml)))
      ; contact node
      (list-comp `(contact
        ; last contact detail node
        ~@(list-comp (parse-mnml `(~tag ~val))
            [[tag val] (.items contact)]))
      [contact contacts]))))
```

[output]

```
<contacs>
  <contact>
    <firstname>Eric</firstname>
    <lastname>Johnson</lastname>
    <telephone>+1-202-555-0170</telephone>
  </contact>
  <contact>
    <firstname>Mary</firstname>
    <lastname>Johnson</lastname>
    <telephone>+1-202-555-0185</telephone>
  </contact>
</contacs>
```

With `parse-mnml` function it is also possible to pass an optional dictionary to be used for custom variables and functions on evaluation process. This is NOT possible with `ml` macro.

```
(parse-mnml '(tag :attr ~val) {"val" "val"}) ; output: <tag attr="val"/>
```

With `template` macro you can actually see a very similar behaviour. In cases where variables can be hard coded, you might want to use this option:

```
(template {"val" "val"} `(tag :attr ~val)) ; output: <tag attr="val"/>
```

It doesn't really matter in which order you pass expression and dictionary to the `template` macro. It is also ok to leave dictionary out if expression does not contain any variables. For `template` macro, expression needs to be quasiquoted, if it contains `HyML` code.

Wrapping up everything

So all features of the MiNiMaL macro has now been introduced. Let us wrap everything and create XHTML document that occupies the most of the feature set. Additional comments will be given between the code lines.

```
; define variables
(defvar topic "How do you make XHTML 1.0 Transitional document with HyML?"
        tags ['html 'xhtml 'hyml]
        postedBy "Hege Refsnes"
        contactEmail "hege.refsnes@example.com")

; define function
(deffun valid (fn []
  (ml (p (a :href "http://validator.w3.org/check?uri=referer"
          (img :src "http://www.w3.org/Icons/valid-xhtml10"
               :alt "Valid XHTML 1.0 Transitional"
               :height "31" :width "88"))))))

; let just arficially create a body for the post
; and save it to the external template file
(with [f (open "body.hy" "w")]
  (f.write "(div :class \"body\"
    \"I've been wondering if it is possible to create XHTML 1.0 Transitional
      document by using a brand new HyML?\")"))

; start up the MiNiMaL macro
(ml
  ; xml document declaration
  "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
  "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\"
  \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">"
  ; create html tag with xml namespace and language attributes
  (html :xmlns "http://www.w3.org/1999/xhtml" :lang "en"
    (head
      ; title of the page
      (title "Conforming XHTML 1.0 Transitional Template")
      (meta :http-equiv "Content-Type" :content "text/html; charset=utf-8"))
    (body
      ; wrap everything inside the post div
      (div :class "post"
        ; first is the header of the post
        (div :class "header" ~topic)
        ; then body of the post from external template file
        ~@(include "body.hy")
        ; then the tags in spans
        (div :class "tags"
          ~@(list-comp `(span ~tag) [tag tags]))
        ; finally the footer
        (div :id "footer"
          (p "Posted by: " ~postedBy)
          (p "Email: "
            (a :href ~(+ "mailto:" contactEmail) ~contactEmail) ".")))
       ; proceed valid stamp by a defined function
       ~(valid))))
```

[output]

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
  PUBLIC '-//W3C//DTD XHTML 1.0 Transitional//EN'
  'http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd'>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Conforming XHTML 1.0 Transitional Template</title>
    <meta content="text/html; charset=utf-8" http-equiv="Content-Type"/>
  </head>
  <body>
    <div class="post">
      <div class="header">How do you make XHTML 1.0 Transitional document with HyML?</
↪div>
      <div class="body">I've been wondering if it is possible to create XHTML 1.0␣
↪Transitional
      document by using a brand new HyML?</div>
      <div class="tags">
        <span>html</span>
        <span>xhtml</span>
        <span>hyml</span>
      </div>
      <div id="footer">
        <p>Posted by: Hege Refsnes</p>
        <p>
          Email:
          <a href="mailto:hege.refsnes@example.com">hege.refsnes@example.com</a>
          .
        </p>
      </div>
    </div>
    <p>
      <a href="http://validator.w3.org/check?uri=referer">
        <img alt="Valid XHTML 1.0 Transitional" height="31" src="http://www.w3.org/
↪Icons/valid-xhtml10" width="88"/>
      </a>
    </p>
  </body>
</html>
```

Unintended features

These are not deliberately implemented features, but a consequence of the HyML MiNiMaL implementation and how Hy works.

Nested MiNiMaL macros

It is possible to call MiNiMaL macro again inside unquoted code:

```
(ml (tag ~(+ "Generator inside: " (ml (sub "content")))))
```

[output]

```
<tag>Generator inside: <sub>content</sub></tag>
```

Unrecognized symbols

Unrecognized symbols (that is they are not specified as literals with double quotas and have no whitespace) are regarded as string literals, unless they are unquoted and they are not colliding with internal Hy symbols.

```
(ml (tag :alfred J. Kwak))
```

[output]

```
<tag alfred="J.">Kwak</tag>
```

Quote and quasiquote

Tag names, attribute values, and tag content can be also single pre-quoted strings. It doesn't matter because in the final process of evaluating the component, a string representation of the symbol is retrieved.

```
[(ml ('tag)) (ml (`tag)) (ml (tag)) (ml ("tag"))]
```

[output]

```
['<tag/>', '<tag/>', '<tag/>', '<tag/>']
```

With keywords, however, single pre-queted strings will get parsed as a content.

```
[(ml (tag ':attr)) (ml (tag `:attr))]
```

[output]

```
['<tag>attr</tag>', '<tag>attr</tag>']
```

Keyword specialties

Also, if keyword marker is followed by a string literal, keyword will be empty, thus not a correctly wormed keyword value pair.

```
(ml (tag :"attr")) ; output: <tag ="attr"/>
```

So only working version of keyword notation is `:{symbol}` or unquoted `~(keyword {expression})`.

Note: Keywords without value are interpreted as a keyword having the same value as the keyword name (called boolean attributes in HTML).

```
[(ml (tag :disabled)) (ml (tag ~(keyword "disabled"))) (ml (tag :disabled "disabled
↪"))]
```

[output]

```
['<tag disabled="disabled"/>', '<tag disabled="disabled"/>', '<tag disabled="disabled
↪"/>']
```

If you wish to define multiple boolean attributes together with content, you can collect them at the end of the expression.

Note: In XML boolean attributes cannot be minimized similar to HTML. Attributes always needs to have a value pair.

```
(ml (tag "Content" :disabled :enabled))
```

[output]

```
<tag disabled="disabled" enabled="enabled">Content</tag>
```

One more thing with keywords is that if the same keyword value pair is given multiple times, it will show up in the mark up in the same order, as multiple. Depending on the markup parser, the last attribute might be valuated OR parser might give an error, because by XML Standard attibute names should be unique and not repeated under the same element.

```
(ml (tag :attr :attr "attr2")) ; output: <tag attr="attr" attr="attr2"/>
```

CHAPTER 2

HyML - Markup Language generator for Hy

HyML (acronym for Hy Markup Language) is a set of macros to generate XML, XHTML, and HTML code in Hy.

Introduction

Main features

1. resembling syntax with XML
2. ability to evaluate Hy program code on macro expansion
3. processing lists and templates
4. custom variables (and functions)
5. tag name validation and attribute with html4 and html5 macros
6. start and end tag omission for html4 and html5
7. custom div tag, class and id attribute handlers for (x)html

You can use HyML for:

- static xml/xhtml/html content and file generation
- generating html code for Jupyter Notebook for example
- attached it to the server for dynamic html output generation
- practice and study
- challenge your imagination for any creative use

If you want to skip the nostalgic background rationale part, you can jump straight to the installation and the documentation part.

Motivation

My primary intention is simple and mundane. Study. Study. Study.

First of all, I wanted to study more Lisp language. Seven years ago, I tried Scheme and CommonLisp for form generation and validation purposes. Then Clojure for website session handler. Now, in 2017, I found another nice Lisp dialect which was seemlessly interoperating with Python, the language I've already used for an another decade on many spare time research projects.

This implementation, Pythonic Lisp, is called with a concise two character name, Hy. Well chosen name makes it possible to create many "Hylarious" module names and acronyms when prefixed, infixed, and affixed with other words. Playful compounds can be created such as Hymn, Hy5, Hyway, Shyte, HyLogic, Hyffix, Hypothesis (actually already a Python test library), Archymedes (could have been), and now: **HyML**.

For other Lisp interpreters written in Python should be mentioned. One is from iconic Peter Norvig: Lispy2. Other is McCarthy's original Lisp by Fogus: lithp.py. Yet one more implementation of Paul Graham's Root Lisp by Kjetil Valle.

But none of these interact with Python AST so that both Lisp and Python modules can be called from each sides, which is why I think Hy is an exceptionally interesting implementation.

Previous similar work

As a web developer, most of my time, I'm dealing with different kinds of scripting and markup languages. Code generation and following specifications is the foremost concern. Lisp itself is famous for code generation and domain language oriented macro behaviours. I thought it would be nice to make a generator that creates html code, simplifies creation of it and produces standard well defined code. It turned out that I was not so unique on that endeavour after all:

> "There are plenty of Lisp Markup Languages out there - every Lisp programmer seems to write at least one during his career..."‘
>
> —cl-who

And to be honest, I've made it several times with other languages.

Python

Since Hy is a rather new language wrapper, there was no dedicated generator available (natively written) for it. Or at least I didn't find them. Maybe this is also, because one could easily use Python libraries. Any Python library can be imported to Hy with a simple *import* clause. And vice versa, any Hy module can be imported to Python with the ordinary *(import)* command.

I had made tag generator module for Python four years ago, namely *tagpy*, which is now called Remarkuple3. It is a general purpose class with automatic tag object creation on the fly. It follows strictly XML specifications. I should show some core parts of it.

First the tag class:

```
class TAG(object):
    def __init__(self, *args, **kw):
        """ Construct object, args are content and keywords are attributes """
        for arg in args:
            self.__dict__['content'].append(arg)
        for key, val in kw.items():
            self.__dict__['attributes'][key.lower()] = val
    def __getattr__(self, key):
        """
        Get attribute by key by dot notation: tag.attr. This is a short and nice way,␣
↪but
```

```
        drawback is that Python has some reserved words, that can't be used this way.␣
↪Method
        is also not-case-sensitive, because key is transformed to lower letters.␣
↪Returning
        None if attribute is not found.
        """
        return self.__dict__['attributes'].get(key.lower(), None)
    def __str__(self):
        """
        Represent tag in string format. This is also a nice and short way to output␣
↪the actual
        xml content. Concat content retrieves all nested tags recursively.
        """
        if self.__dict__['content']:
            return '<%s%s>%s</%s>' % (self.__class__.__name__, strattr(self.__dict__[
↪'attributes']),
                                      concat(*self.__dict__['content']), self.__class_
↪_.__name__)
        else:
            return '<%s%s/>' % (self.__class__.__name__, strattr(self.__dict__[
↪'attributes']))
```

Then helper initialization:

```
# create helper class to automaticly create tags based on helper class attribute /␣
↪method overloading
class htmlHelper(object):
    def create(self, tag):
        return type(tag, (TAG,), {})()
    def __getattr__(self, tag):
        return type(tag.lower(), (TAG,), {})

# init helper for inclusion on the module
helper = htmlHelper()
```

And finally usage example:

```
# load xml helper
from remarkuple import helper as h
# create anchor tag
a = h.a()
# create attribute for anchor
a.href = "#"
# add bolded tag text to anchor
a += h.b("Link")
print(a) # <a href="#"><b>Link</b></a>
```

PHP

I also made a PHP version of the HTML generator even earlier in 2007. That program factored classes for each html4 specified tag, and the rest was quite similar to Python version. Here is some parts of the code for comparison, first the generation of the tag classes:

```
$evalstr = '';
// Factorize elements to classes
foreach ($elements as $abbreviation => $element) {
    $abbreviation = strtoupper($abbreviation);
    $arg0 = strtolower($abbreviation);
```

```
    $arg1 = $element['name'];
    $arg2 = $element['omitted'] ? 'true' : 'false';
    $arg3 = $element['nocontent'] ? 'true' : 'false';
    $arg4 = $element['strict'] ? 'true' : 'false';

    $evalstr .= <<<EOF
class HE_$abbreviation extends HtmlElement
{
    function HE_$abbreviation(\$Attributes = null, \$Content = null, \$Index = null) {
        parent::Mm_HtmlElement('$arg0', '$arg1', $arg2, $arg3, $arg4);
        if (isset(\$Attributes) && is_array(\$Attributes)) \$this->attributes->
↪container(\$Attributes);
        if (isset(\$Content)) \$this->add_content(\$Content, \$Index);
    }
}
EOF;
    }
    eval($evalstr);
}
```

Then usage of the HtmlElement class:

```
include 'HtmlElement.php';
$a = new HE_A(array('href' => '#'));
$a->addContent(new HE_B("Link"));
echo $a->render(); // <a href="#"><b>Link</b></a>
```

Doesn't this feel distantly quite Lispy? I mean generating and modifying code is same what macros do. Here it is done with PHP, and can be done with any language. But the thing is that *eval* in other languages is regarded as *evil* but for Lisp users it is a "principia primaria".

Javascript

Both Python and PHP versions are object oriented approaches to xml/html generation. Which is quite good after all. You can collect xml elements inside each other, manipulate them anyway you want before rendering output. One could similarly use world-famous jQuery javascript library, which has become a standard for DOM manipulation:

```
var a = $('<a/>');
a.attr('href', "#");
a.html($('<b>Link</b>');
// there is a small catch here, a -element must be inner element of other
// tag to be possible to be rendered as a whole
var d = $('<div/>').html(a);
console.log(d.html()); //<a href="#"><b>Link</b></a>
```

jQuery will construct tag objects (DOM elements) which you can access by jQuery methods that are too manifold to mention here.

Template engines

Then there are plenty of domain specific html template languages for each and every programming language. Haml for Ruby. Jinja, Mako, and Genchi for Python. Twig, Smarty, and Mustache for PHP.

Common to all is that they separate user interface logic from business and database logic to follow model-view-controller architecture.

Actually by using output buffering control one can easily create a template engine with PHP, that, by the way, is a template language itself already. For example this file.php content:

```
<a href="<?=$href?>"><b><?=$link?></b></a>
```

With this code:

```
<?php
function render($file, $data) {
    $content = file_get_contents($file);
    ob_start() && extract($data);
    eval('?>'.$content);
    $content = ob_get_clean();
    ob_flush();
    return $content;
}
render('file.php', array('href'=>"#", 'link'=>"Link"));
?>
```

Would render:

```
<a href="#"><b>Link</b></a>
```

But now it is time to get back to Python, Lisp, and Hy. While Hy didn't have html generators until now, there are many Lisp implementations as previously told. You can find out some from cliki.net. You may also want to compare different implementations and their final DSL syntax to HyML from @com-informatimago.

Python xml/html generators and processors are available from Pypi. Some do more or less same than HyML, some are just loosely related to HyML.

Benefits and Implementation

One thing in the object oriented method is that code itself doesn't resemble much like xhtml and html. So you are kind of approaching one domain language syntax from other syntax. In some cases it looks like ugly, in many small projects and cases it gives overhead in the amoun of code you need to write to output XML.

In Hy (and List generally), language syntax already resembles structured and nested markup langauge. Basic components of the language are tag notation with <, >, and / characters, tag names, tag attributes, and tag content. This behaves exactly with Lisp notation where the first element inside parentheses is normally a function, but now gets interpreted as a tag name. Keywords are usually indicated with a pair notation (:key "value"). And content is wrapped with double quotation characters. Only difference is that when indicator of nested content in XML is done "outside" of the start tag element, for example:

```
<tag>content</tag>
```

In Hy, the content is inside the expression:

```
(tag "Content")
```

This makes parenthesized notation less verbose, so it tends to save some space. Drawback is of cource the fact that in a large code block there will be a lot of ending parentheses,a s you will find later. This will make the famous LISP acronym expanded to "(L)ots of (I)rritating (S)uperfluous (P)arentheses". But don't let it scare you, like it did me at first. After all, it is like with playing guitars; more different types you play, less it matters what you get on your hands. Soon you find you can't get it enought!

Lisp is also known as "code is data, data is code" -paradigm. This is perfectly visible on the HyML implementation I'm going give some sights now.

Three aspects

Data, was it just data as data or code, in the information technology it has always to do with three different aspects, namely:

1. processing lists (did I mention this somewhere earlier?!)
2. hierarchic structures
3. data types

In HyML the third part is pretty simple. On the output everything is just a plain text. There are no datatypes. In HyML data types has a negligible meaning. You should only give attention keywords that starts with colon (:) punctuation mark and literals that start with " and ends to the counterpart ".

Hierachical structure is defined by nested parentheses. Simple as that.

Processing list can be thought as a core Hy / Lisp language syntax utility, but there is also a specific syntactic feature called unquote-splice, that can delegate a rendered list of elements to the parent element in HyML.

Catch tag if you can

We are talking about internal implementation of the HyML module now, especially the macros.hy file.

Let us take a moment to think of this expression in HyML:

```
(tag :attr "value" (sub "Content"))
```

One of the core parts of the HyML implementation is where to catch a tag name. Because the first element after opening parentheses in Hy is normally referring to a function, in HyML we need to change that functionality so that it refers to a tag name. Thus we need to catch tag name with the following code:

```
(defn catch-tag [code]
  (try
    ; code can be a symbol or a sub program
    ; thats why try to evaluate it. internal symbols like "input"
    ; for example are handled here too. just about anything can be
    ; a tag name
    (name (eval code))
    ; because evaluation most probably fails when code contains
    ; a symbol name that has not been specified on the global namespace,
    ; thats why return quoted code which should work every time.
    ; tag will be tag and evaluation of the code can go on without failing
    ; in the catch-tag part
    (except (e Exception) (eval 'code))))
```

Then the rest of the HyML expression gets interpreted. It can contain basicly just key-value pairs or content. Content can be a string or yet another similar HyML expression. *get-content-attributes* in macros.hy will find out all keyword pairs first and then rest of the expression in regarded as content, which is a string or a nested HyML expression.

Semantic sugar

Then some tag names are specially handled like: *unquote*, *unquote_splice*, , *!__*, *<?xml*, *!DOCTYPE*, and in *html4/5* mode tag names starting with . or # (*dispatch_reader_macro*).

For example ~ (unquote) symbol is used to switch the following expression from macro mode to Hy program mode. Other are mroe closely discussed in the documentation.

Finally when tags are created some rules from specs.hy https://github.com/markomanninen/hyml/blob/master/hyml/specs.hy are used to create either long or short tags and to minimize attributes.

This is basicly it. Without html4/5 functionality code base would be maybe one third of the current code base. Tag validation and minimizing did add a lot of extra code to the module. Being a plain xml generator it would have been comparative to Remarkuple code base.

Templating feature requires using globals variable dictionary as a registry for variables. Macro to expand and evaluate templates is pretty simple:

```
(defmacro include [template]
  `(do
    ; tokenize is needed to parse external file
    (import [hy.importer [tokenize]])
    (with [f (open ~template)]
      ; funky ~@` part is needed as a prefix to the template code
      ; so that code on template wont get directly expanded but only
      ; after everything had been collected by the macro for final evaluation
      (tokenize (+ "~@`(" (f.read) ")")))))
```

One more catch is to use variables from globals dictionary when evaluating code on parser:

```
(.join "" (map ~name (eval (second code) variables)))
```

This makes it possible to use custom variables at the moment in HyML module and maybe custom functions on templates later in future.

Now, with these simple language semantic modifications to Hy, I have managed to do a new programable markup language, HyML, that produces XML / XHTML, and HTML code as an output.

Future work

There is a nice feature set on arclanguage html generator, that still could optimize the size of the codebase of HyML: http://arclanguage.github.io/ref/html.html

Downside of this is that implementation like that adds more functionas to call and maintain, while HyML at this point is a pretty minimal implementation for its purposes.

Quick start

Project is hosted in GitHub: https://github.com/markomanninen/hyml/

Installation

HyML can be installed effortlessly with pip:

$ pip install hyml

HyML requires of cource Python and Hy on a computer. Hy will be automaticly installed, or updated at least to version 0.12.1, if it wasn't already.

Environment check

You should check that your environment meets the same requirements than mine. My environment for the sake of clarity:

```
(import hy sys)
(print "Hy version: " hy.__version__)
(print "Python" sys.version)
```

```
Hy version:  0.12.1
Python 3.5.2 |Anaconda custom (64-bit)| (default, Jul  5 2016, 11:41:13) [MSC v.1900
↪64 bit (AMD64)]
```

So this module has been run on Hy 0.12.1 and Python 3.5.2 installed by Anaconda package in Windows. If any problems occurs, you should report them to: https://github.com/markomanninen/hyml/issues

Import main macros

After installation you can import ML macros with the next code snippet in Hy REPL or Jupyter Notebook with calysto_hy kernel:

```
(require [hyml.macros [*]])
(import (hyml.macros (*)))
```

Let us just try that everything works with a small test:

```
#(tag :attr "val" (sub "Content"))
```

That should output:

```
<tag attr="val"><sub>Content</sub></tag>
```

So is this it, the code generation at its best? With 35 characters of code we made 40 characters xml string. Not to mention some 500 lines of code on a module to make it work! Give me one more change and let me convince you with the next all-in-one example.

Documentation

This is the core documentation part of the HyML.

All-in-one example

First, I'd like to show an example that presents the most of the features included in the HyML module. Then I will go through all the features case by case.

```
(defvar rows [[1 2 3] [1 2 3]])
(print (indent (xhtml5
  "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
  "<!DOCTYPE html>"
  (html :lang "en" :xmlns "http://www.w3.org/1999/xhtml"
    (head (title "Page title"))
    (body
      "<!-- body starts here -->"
      (.main-container
        (h1.main.header
          ~(.capitalize "page header"))
        (ul#main "List"
          ~@(list-comp* [[idx num] (enumerate (range 3))]
            `(li :class ~(if (even? idx) "even" "odd") ~num)))
        (table
          (thead
            (tr (th "Col 1") (th "Col 2") (th "Col 3")))
```

```
          (tbody
            ~@(include "rows.hy")))))))))
```

This will output:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Page title</title>
    </head>
    <body>
        <!-- body starts here -->
        <div class="main_container">
            <h1 class="main header">Page header</h1>
            <ul id="main">
                List
                <li class="even">0</li>
                <li class="odd">1</li>
                <li class="even">2</li>
            </ul>
            <table>
                <thead>
                    <tr>
                        <th>Col 1</th>
                        <th>Col 2</th>
                        <th>Col 3</th>
                    </tr>
                </thead>
                <tbody>
                    <tr>
                        <td>1</td>
                        <td>2</td>
                        <td>3</td>
                    </tr>
                </tbody>
            </table>
        </div>
    </body>
</html>
```

XML, HTML4, HTML5, XHTML, and XHTML5

At the moment HyML module contains `xml`, `html4`, `html5`, `xhtml`, and `xhtml5` macros (called as `ML` macros in short) to generate the (M)arkup (L)anguage code. `xml` is a generic generator which allows using any tag names and attributes. `html4` and `xhtml` macros allows to use only html4 specified tag names. Same applies to `html5` and `xhtml5`. Complete chart of the allowed elements are listed at the end of the document.

Tags can be created with or without attributes, as well as with or without content. For example:

```
(println
  (xml (node))
  (xml (node :attribute "")) ; force to use empty attribute
  (xml (node :attribute "value"))
  (xml (node :attribute "value" "")) ; force to use empty content
  (xml (node :attribute "value" "Content")))
```

Output:

```
<node/>
<node attribute=""/>
<node attribute="value"/>
<node attribute="value"></node>
<node attribute="value">Content</node>
```

However in `html4` and `html5` there are certain tags that cannot have endings so they will be rendered in correct form by the parser. "Forbidden" labeled tags are listed at the end of the document. One of them is for example the meta tag:

```
(html4 (meta :name "keywords" :content "HTML,CSS,XML,JavaScript"))
```

Output:

```
<meta name=keywords content=HTML,CSS,XML,JavaScript>
```

To see and compare the difference in xhtml, let macro print the same:

```
(xhtml (meta :name "keywords" :content "HTML,CSS,XML,JavaScript"))
```

Output:

```
<meta name="keywords" content="HTML,CSS,XML,JavaScript"/>
```

Shorthand macro

`#` (Square Ml) can be used as a shorthand reader macro for generating xml/html/xhtml code:

```
#(html
    (head (title "Page title"))
    (body (div "Page content" :class "container")))
```

Output:

```
<html><head><title>Page title</title></head><body><div class="container">Page content
↪</div></body></html>
```

`#` actually utilizes `xml` macro so same result can be achieved with the next, maybe more convenient and recommended notation:

```
(xml
  (html
    (head (title "Page title"))
    (body (div "Page content" :class "container"))))
```

Output:

```
<html><head><title>Page title</title></head><body><div class="container">Page content
↪</div></body></html>
```

It is not possible to define other `ML` macro to be used with the `#` shorthand reader macro. You could however define your own shorthands following next quidelines:

> (defsharp {unicode-char} [code] (parse-{parser} code))

`{unicode-char}` can be any unicode char you want. `{parser}` must be one of the following available parsers: xml, xhtml, xhtml5, html4, or html5.

With # shorthand you have to provide a single root node for generating code. Directry using ML macros makes it possible to generate multiple instances of code, and might be more informative notation style anyway:

```
(xml (p "Sentence 1") (p "Sentence 2") (p "Sentence 3"))
```

Output:

```
<p>Sentence 1</p><p>Sentence 2</p><p>Sentence 3</p>
```

Let us then render the code, not just printing it. This can be done via html5> macro imported earlier from helpers:

```
(html4> (p "Content is " (b king) !))
```

Output:

Renderers are available for all ML macros: xml>, xhtml>, xhtml5>, html4>, and html5>.

Validation and minimizing

If validation of the html tag names is a concern, then one should use html4, html5, xhtml, and xhtml5 macro family. In the example below if we try to use time element in html4, which is specifically available in html5 only, we will get an HyMLError exception:

```
;(try
; (html4 (time))
; (catch [e [HyMLError]]))
;hytml.macros.HyMLError: Tag 'time' not meeting html4 specs
```

Other features in html4 and html5 macros are attribute and tag minimizing. Under the certain rules start and end tags can be removed from the output. Also boolean attributes can be shortened. In html4 and html5 macros minimizing is a default feature that can't be bypassed. If you do not want to minimize code, you must use xhtml or xhtml5 macro. Contrary in xhtml and xhtml5 macros attribute and tag minimizing is NOT available. Instead all tags are strictly closed and attributes in key="value" format.

HTML4

```
; valid html4 document
(html4 (title) (table (tr (td "Cell 1") (td "Cell 2") (td "Cell 3"))))
```

Output:

```
<title/><table><tr><td>Cell 1<td>Cell 2<td>Cell 3</table>
```

XHTML

```
; in xhtml tags and attributes will be output in complete format
(xhtml (title) (table (tr (td "Cell 1") (td "Cell 2") (td "Cell 3"))))
```

Output:

```
<title/><table><tr><td>Cell 1</td><td>Cell 2</td><td>Cell 3</td></tr></table>
```

Note that above xhtml code is still not a valid xhtml document even tags and attributes are perfectly output. ML macros do no validate structure of the document just tag names. For validation one should use official validator service and follow the html specifications to create a valid document. ML macros can be used to guide on that process but more importantly it is meant to automatize the generation of the xml code while adding programming capabilities on it.

`xml` on the other hand doesn't give a dime of the used tag names. They can be anything, even processed names. Same applies to keywords, values, and contents. You should use more strict `xhtml`, `xhtml5`, `html4`, and `html5` macros to make sure that tag names are corresponding to HTML4 or HTML5 specifications.

```
; see how boolean attribute minimizing works
(html4 (input :disabled "disabled"))
```

Output:

```
<input disabled>
```

Unquoting code

In all `ML` macros you can pass any code in it. See for example:

```
(xml (p "Sum: " (b (apply sum [[1 2 3 4]]))))
```

Output:

```
<p>Sum: <b><apply>sum<[1, 2, 3, 4]/></apply></b></p>
```

But you see, the result was not possibly what you expected. `ML` macros will interpret the first item of the *expression* as a name of the tag. Thus *apply* becomes a tag name. Until the next *expression* everything else is interpreted either as a content or a keyword.

However using ~ (unquote) symbol, `ML` macro behaviour can be stopped for a moment:

```
(xml (p "Sum: " (b ~(apply sum [[1 2 3 4]])) !))
```

Output:

```
<p>Sum: <b>10</b>!</p>
```

So the following expression after ~ will be evaluated and then result is returned back to the original parser. And the rest of the code will be interpreted via macro. In this case it was just an exclamation mark.

> Note that it is not mandatory to wrap strings with `""` if given input doesn't contain any spaces. You could also single quote simple non-spaced letter sequences. So `!` is same as `"!"` in this case.

Quoting and executing normal Hy code inside html gives almost unlimited possibility to use HyML as a templating engine. Of cource there is also a risk to evaluate code that breaks the code execution. Plus uncontrolled template engine code may be a security consern.

Unquote splice

In addition to unquote, one can handle lists and iterators with `~@` (unquote-splice) symbol. This is particularly useful when a list of html elements needs to be passed to the parent element. Take for example this table head generation snippet:

```
(xhtml
 (table (thead
   (tr ~@(list-comp
          `(th :class (if (even? ~i) "even" "odd") ~label " " ~i)
          [[i label] (enumerate (* ["col"] 3))])))))
```

Output:

```
<table><thead><tr><th class="even">col 0</th><th class="odd">col 1</th><th class="even
↪">col 2</th></tr></thead></table>
```

List comprehensions notation might seem a little bit strange for some people. It takes a processing part (expression) as the first argument, and the actual list to be processed as the second argument. On a nested code this will move lists to be processed in first hand to the end of the notation. For example:

```
(xml>
  ~@(list-comp `(ul (b "List")
      ~@(list-comp `(li item " " ~li)
          [li uls]))
    [uls [[1 2] [1 2]]]))
```

Output:

But there is another slighly modified macro to use in similar manner:

list-comp*

Let's do again above example but this time with a dedicated `list-comp*` macro. Now the lists to be processed is passed as the first argument to the `list-comp*` macro and the expression for processing list items is the second argument. Yet the second argument itself contains a new list processing loop until final list item is to be processed. This is perhaps easier to follow for some people:

```
(xhtml
  ~@(list-comp* [uls [[1 2] [1 2]]]
    `(ul (b "List")
      ~@(list-comp* [li uls]
        `(li item " " ~li)))))
```

Output:

```
<ul><b>List</b><li>item 1</li><li>item 2</li></ul><ul><b>List</b><li>item 1</li><li>
↪item 2</li></ul>
```

Of cource it is just a matter of the taste which one you like. `list-comp*` with `unquote-splice` symbol (`~@`) reminds us that it is possible to create any similar custom macros for the HyML processor. `~@` can be thought as a macro caller, not just unquoting and executing Hy code in a normal lisp mode.

Here is a more complex table generation example from the remarkuple Python module docs. One should notice how variables (`col`, `row`, and `cell`) are referenced by quoting them:

```
(html4>
  (table#data
    (caption "Data table")
    (colgroup
      (col :style "background-color:red")
      (col :style "background-color: green")
      (col :style "background-color: blue"))
    (thead
      (tr
        ~@(list-comp* [col ["Column 1" "Column 2" "Column 3"]]
          `(th ~col))))
    (tbody#tbody1
     ~@(list-comp* [row (range 1 3)]
```

```
      `(tr
        ~@(list-comp* [cell (range 3)]
          `(td  ~row "." ~cell)))))
   (tbody#tbody2
    ~@(list-comp* [row (range 1 3)]
      `(tr
        ~@(list-comp* [cell (range 3)]
          `(td  ~row "." ~cell)))))
   (tfoot
     (tr
       (td :colspan "3" "Footer")))))
```

Output:

Address book table from CSV file

We should of course be able to use external source for the html. Let's try with a short csv file:

```
(xhtml>
 (table.data
   (caption "Contacts")
   ~@(list-comp*
     [[idx row] (enumerate (.split (.read (open "data.csv" "r")) "\n"))]
     (if (pos? idx)
         `(tbody
           ~@(list-comp* [item (.split row ",")]
             `(td ~item)))
         `(thead
           ~@(list-comp* [item (.split row ",")]
             `(th ~item)))))))
```

Output:

Templates

It is possible to load code from an external file too. This feature has not been deeply implemented yet, but you get the feeling by the next example. Firt I'm just going to show external template file content:

```
(with [f (open "template.hy")] (print (f.read)))
```

Output:

```
(html :lang ~lang
  (head (title ~title))
  (body
    (p ~body)))
```

Then I use `include` macro to read and process the content:

```
(defvar lang "en"
        title "Page title"
        body "Content")

(xhtml ~@(include "template.hy"))
```

Output:

```
<html lang="en"><head><title>Page title</title></head><body><p>Content</p></body></
↪html>
```

All globally defined variables are available on `ML` macros likewise:

```
(xhtml ~lang ", " ~title ", " ~body)
```

Output:

```
en, Page title, Content
```

The MIT License

CHAPTER 3

HyML - HTML4 / HTML5 specifications

`xml` does not care about the markup specifications other than general tag and attribute notation. It is totally dummy about the naming conventions of the tags or their relation to each other or global structure of the markup document. It is all on the responsibility of the user to make it correct.

`html4` and `html5` as well as `xhtml4` and `xhtml5` macros will render tags as specified below. These macros will minimize code when possible. Using undefined tag will raise an error. Attributes are not validated however. One should use official validator for a proper validation.

This is also the last example of using `ML` macros. xhtml> will delegate xhtml content to IPython.display method so that content is rendered on Jupyter Notebook cell rather than just output as a raw xml string.

Columns in the table are:

- Tag name
- Tag code when using empty content
- Has the optional end tag?
- Has a forbidden content and the end tag?
- Can omit the short tag? For example: `<col>`
- Is the tag in HTML4 specifications?
- Is the tag in HTML5 specifications?

```
(xhtml>
  (table.data
    (caption "HTML Element Specifications")
    (thead
      (tr
        ~@(list-comp* [col ["Tag name" "Code" "Optional" "Forbidden" "Omit" "HTML4"
↪"HTML5"]]
          `(th ~col))))
    (tbody
      ~@(do (import html)
            (import (hyml.macros (specs optional? parse-html4 parse-html5)))
```

```
        (list-comp* [[id row] (.items specs)]
         `(tr
           (td ~(.upper (get row :name)))
           (td ~(html.escape
                 (if (get row :html4)
                     (parse-html4 `(~(get row :name) ""))
                     (parse-html5 `(~(get row :name) "")))))
           (td ~(if (optional? (get row :name)) "✓" ""))
           (td ~(if (get row :forbidden) "✓" ""))
           (td ~(if (get row :omit) "✓" ""))
           (td ~(if (get row :html4) "✓" ""))
           (td ~(if (get row :html5) "✓" "")))))))))
```

Tag name	Code	Optional	Forbidden	Omit	HTML4	HTML5
A	<a>				✓	✓
ABBR	<abbr></abbr>				✓	✓
ACRONYM	<acronym></acronym>				✓	
ADDRESS	<address></address>				✓	✓
APPLET	<applet></applet>				✓	
AREA	<area>		✓	✓	✓	✓
ARTICLE	<article></article>					✓
ASIDE	<aside></aside>					✓
AUDIO	<audio></audio>					✓
B					✓	✓
BASE	<base>		✓	✓	✓	✓
BASEFONT	<basefont>		✓		✓	
BDI	<bdi></bdi>					✓
BDO	<bdo></bdo>				✓	✓
BIG	<big></big>				✓	
BLOCKQUOTE	<blockquote></blockquote>				✓	✓
BODY	<body>	✓			✓	✓
BR	 		✓	✓	✓	✓
BUTTON	<button></button>				✓	✓
CANVAS	<canvas></canvas>					✓
CAPTION	<caption>	✓			✓	✓
CENTER					✓	
CITE	<cite></cite>				✓	✓
CODE	<code></code>				✓	✓
COL	<col>		✓	✓	✓	✓
COLGROUP	<colgroup>	✓			✓	✓
DATALIST	<datalist></datalist>					✓
DD	<dd></dd>				✓	✓
DEL	<del></del>				✓	✓
DETAILS	<details></details>					✓
DFN	<dfn></dfn>				✓	✓
DIALOG	<dialog></dialog>					✓
DIR	<dir></dir>				✓	
DIV					✓	✓
DL	<dl></dl>				✓	✓
DT	<dt></dt>				✓	✓
EM					✓	✓
EMBED	<embed></embed>					✓

Continued on next page

Table 3.1 – continued from previous page

Tag name	Code	Optional	Forbidden	Omit	HTML4	HTML5
FIELDSET	<fieldset></fieldset>				✓	✓
FIGCAPTION	<figcaption></figcaption>					✓
FIGURE	<figure></figure>					✓
FONT					✓	
FOOTER	<footer></footer>					✓
FORM	<form></form>				✓	✓
FRAME	<frame>		✓		✓	
FRAMESET	<frameset></frameset>				✓	
H1					✓	✓
H2					✓	✓
H3					✓	✓
H4					✓	✓
H5					✓	✓
H6					✓	✓
HEAD	<head>	✓			✓	✓
HEADER	<header></header>					✓
HR	<hr>		✓	✓	✓	✓
HTML	<html>	✓			✓	✓
I					✓	✓
IFRAME	<iframe></iframe>				✓	✓
IMG	<img>		✓	✓	✓	✓
INPUT	<input>		✓	✓	✓	✓
INS	<ins></ins>				✓	✓
ISINDEX	<isindex>		✓		✓	✓
KBD	<kbd></kbd>				✓	✓
KEYGEN	<keygen></keygen>			✓		✓
LABEL	<label></label>				✓	✓
LEGEND	<legend></legend>				✓	✓
LI					✓	✓
LINK	<link>		✓	✓	✓	✓
MAIN	<main></main>					✓
MAP	<map></map>				✓	✓
MARK	<mark></mark>					✓
MENU	<menu></menu>				✓	✓
MENUITEM	<menuitem></menuitem>					✓
META	<meta>		✓	✓	✓	✓
METER	<meter></meter>					✓
NAV	<nav></nav>					✓
NOFRAMES	<noframes></noframes>				✓	
NOSCRIPT	<noscript></noscript>				✓	✓
OBJECT	<object></object>				✓	✓
OL					✓	✓
OPTGROUP	<optgroup></optgroup>				✓	✓
OPTION	<option></option>				✓	✓
OUTPUT	<output></output>					✓
P					✓	✓
PARAM	<param>		✓	✓	✓	✓
PICTURE	<picture></picture>					✓
PRE	<pre></pre>				✓	✓

Continued on next page

Table 3.1 – continued from previous page

Tag name	Code	Optional	Forbidden	Omit	HTML4	HTML5
PROGRESS	<progress></progress>					✓
Q	<q></q>				✓	✓
RP	<rp></rp>					✓
RT	<rt></rt>					✓
RUBY	<ruby></ruby>					✓
S	<s></s>				✓	✓
SAMP	<samp></samp>				✓	✓
SCRIPT	<script></script>				✓	✓
SECTION	<section></section>					✓
SELECT	<select></select>				✓	✓
SMALL					✓	✓
SOURCE	<source>		✓	✓		✓
SPAN					✓	✓
STRIKE	<strike></strike>				✓	
STRONG					✓	✓
STYLE	<style></style>				✓	✓
SUB					✓	✓
SUMMARY	<summary></summary>					✓
SUP					✓	✓
TABLE	<table></table>				✓	✓
TBODY	<tbody></tbody>				✓	✓
TD	<td></td>				✓	✓
TEXTAREA	<textarea></textarea>				✓	✓
TFOOT	<tfoot></tfoot>				✓	✓
TH	<th></th>				✓	✓
THEAD	<thead></thead>				✓	✓
TIME	<time></time>					✓
TITLE	<title></title>				✓	✓
TR	<tr></tr>				✓	✓
TRACK	<track>		✓	✓		✓
TT	<tt></tt>				✓	
U	<u></u>				✓	✓
UL					✓	✓
VAR	<var></var>				✓	✓
VIDEO	<video></video>					✓
WBR	<wbr>		✓	✓		✓

CHAPTER 4

HyML tests

Test main features

Assert tests for all main features of HyML library. There should be no output after running these tests. If there is, then *Houston, we have a problem!*

```
;;;;;;;;;
; basic ;
;;;;;;;;;

; empty things
(assert (= (ml) ""))
(assert (= (ml"") ""))
(assert (= (ml "") ""))
(assert (= (ml ("")) "</>"))
; tag names
(assert (= (ml (tag)) "<tag/>"))
(assert (= (ml (TAG)) "<TAG/>"))
(assert (= (ml (~(.upper "tag"))) "<TAG/>"))
(assert (= (ml (tag "")) "<tag></tag>"))
; content cases
(assert (= (ml (tag "content")) "<tag>content</tag>"))
(assert (= (ml (tag "CONTENT")) "<tag>CONTENT</tag>"))
(assert (= (ml (tag ~(.upper "content"))) "<tag>CONTENT</tag>"))
; attribute names and values
(assert (= (ml (tag :attr "val")) "<tag attr=\"val\"/>"))
(assert (= (ml (tag ~(keyword "attr") "val")) "<tag attr=\"val\"/>"))
(assert (= (ml (tag :attr "val" "")) "<tag attr=\"val\"></tag>"))
(assert (= (ml (tag :attr "val" "content")) "<tag attr=\"val\">content</tag>"))
(assert (= (ml (tag :ATTR "val")) "<tag ATTR=\"val\"/>"))
(assert (= (ml (tag ~(keyword (.upper "attr")) "val")) "<tag ATTR=\"val\"/>"))
(assert (= (ml (tag :attr "VAL")) "<tag attr=\"VAL\"/>"))
(assert (= (ml (tag :attr ~(.upper "val"))) "<tag attr=\"VAL\"/>"))
; nested tags
(assert (= (ml (tag (sub))) "<tag><sub/></tag>"))
```

```
; unquote splice
(assert (= (ml (tag ~@(list-comp `(sub ~(str item)) [item [1 2 3]])))
           "<tag><sub>1</sub><sub>2</sub><sub>3</sub></tag>"))
; variables
(defvar x "variable")
(assert (= (ml (tag ~x)) "<tag>variable</tag>"))
; functions
(deffun f (fn [x] x))
(assert (= (ml (tag ~(f "function"))) "<tag>function</tag>"))
; templates
(with [f (open "test/templates/test.hy" "w")]
  (f.write "(tag)"))
(assert (= (ml ~@(include "test/templates/test.hy")) "<tag/>"))
; set up custom template directory
(import hyml.template)
(def hyml.template.template-dir "test/templates/")
; render template function
(import [hyml.template [*]])
(with [f (open "test/templates/test2.hy" "w")]
  (f.write "(tag ~tag)"))
(assert (= (render-template "test2.hy" {"tag" "content"}) "<tag>content</tag>"))
; extend template
(require [hyml.template [*]])
(with [f (open "test/templates/test3.1.hy" "w")]
  (f.write "(tags :attr ~val ~tag)"))
(with [f (open "test/templates/test3.2.hy" "w")]
  (f.write "~(extend-template \"test3.1.hy\" {\"tag\" `(tag)})"))
(assert (= (render-template "test3.2.hy" {"val" "val"}) "<tags attr=\"val\"><tag/></
↪tags>"))
; macro -macro
(with [f (open "test/templates/test4.hy" "w")]
  (f.write "~(macro custom [content] `(tag ~content))~(custom ~content)"))
(assert (= (render-template "test4.hy" {"content" "content"}) "<tag>content</tag>"))
; revert path
(def hyml.template.template-dir "templates/")
; template macro
(assert (= (template {"val" "val"} `(tag :attr ~val)) (template `(tag :attr ~val) {
↪"val" "val"})))

;;;;;;;;;;;;
; special ;
;;;;;;;;;;;;

; tag names
(assert (= (ml (J)) "<1j/>"))
(assert (= (ml (~"J")) "<J/>"))
(assert (= [(ml ('tag)) (ml (`tag)) (ml (tag)) (ml ("tag"))] (* ["<tag/>"] 4)))
; attribute values
(assert (= [(ml (tag :attr 'val)) (ml (tag :attr `val)) (ml (tag :attr val)) (ml (tag␣
↪:attr "val"))]
           (* ["<tag attr=\"val\"/>"] 4)))
; content
(assert (= [(ml (tag 'val)) (ml (tag `val)) (ml (tag val)) (ml (tag "val"))]
           (* ["<tag>val</tag>"] 4)))
; keyword processing
(assert (= [(ml (tag ':attr)) (ml (tag `:attr))] ["<tag>attr</tag>" "<tag>attr</tag>
↪"]))
;(assert (= (ml (tag :"attr")) "<tag =\"attr\"/>"))
```

```
; boolean attributes
(assert (= [(ml (tag :attr "attr")) (ml (tag :attr)) (ml (tag ~(keyword "attr")))]
           ["<tag attr=\"attr\"/>" "<tag attr=\"attr\"/>" "<tag attr=\"attr\"/>"]))
(assert (= (ml (tag :attr1 :attr2)) "<tag attr1=\"attr1\" attr2=\"attr2\"/>"))
(assert (= (ml (tag Content :attr1 :attr2)) "<tag attr1=\"attr1\" attr2=\"attr2\">
↪Content</tag>"))
(assert (= (ml (tag :attr1 :attr2 Content)) "<tag attr1=\"attr1\" attr2=\"Content\"/>
↪"))
; no space between attribute name and value as a string literal
(assert (= (ml (tag :attr"val")) "<tag attr=\"val\"/>"))
; no space between tag, keywords, keyword value, and content string literals
(assert (= (ml (tag"content":attr"val")) "<tag attr=\"val\">content</tag>"))

;;;;;;;;;
; weird ;
;;;;;;;;;

; quote should not be unquoted or surpressed
(assert (= (ml (quote :quote "quote" "quote")) "<quote quote=\"quote\">quote</quote>
↪"))
; tag name, keyword name, value and content can be same
(assert (= (ml (tag :tag "tag" "tag")) "<tag tag=\"tag\">tag</tag>"))
; multiple same attribute names stays in the markup in the reserved order
(assert (= (ml (tag :attr "attr1" :attr "attr2")) "<tag attr=\"attr1\" attr=\"attr2\"/
↪>"))
; parse-mnlm with variable and function dictionary
(assert (= (parse-mnml '(tag :attr ~var ~(func))
                       {"var" "val" "func" (fn[]"Content")}) "<tag attr=\"val\">
↪Content</tag>"))
```