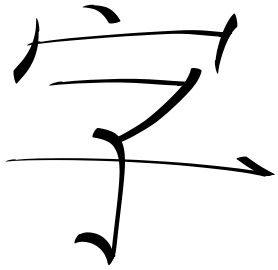

Glyph

Rapid Document Authoring Framework

v0.1.0 (draft)

by *Fabio Cevasco*

March 2010

Table of Contents

Introduction

Glyph is a *Rapid Document Authoring Framework*.

Think of it like a sort of Ruby on Rails but for creating text documents instead of web sites. With Glyph, you can manage your documents tidily in *projects* that can be used to generate deliverables in different formats such as HTML or PDF (through Prince).

Main Features

Glyph uses a simple macro system to perform a wide variety of advanced tasks:

- Generate block-level HTML tags not commonly managed by lightweight markups, like `head`, `body`, `div` and `table`.
- Create and validate internal and external links.
- Include and validate images and figures.
- Automatically determine header levels based on the document structure.
- Automatically generate a Table of Contents based on the document structure.
- Store common snippets of text in a single YAML file and use them anywhere in your document, as many times as you need.
- Store configuration settings in a YAML file and use them anywhere in your document, as many times as you need.
- Evaluate Ruby code within your document.
- Call macros from other macros (including snippets), carefully avoiding mutual calls.
- Include text files in other text files.
- Include the contents of configuration settings (author, title) in the document.
- Filter input explicitly or implicitly, based on file extensions when including files.
- Manage comments and todo items.

Installation

`gem install glyph` — simple, as always.

Essential Glyph commands

Glyph is 100% command line. Its interface resambles Git's for its simplicity and power (thanks to the Gli gem). Here are some example commands:

- `glyph init` — to initialize a new Glyph project in the current (empty) directory.
- `glyph add introduction.textile` — to create a new file called *introduction.textile*.
- `glyph compile` — to compile the current document into a single HTML file.
- `glyph compile -f pdf` — to compile the current document into HTML and then transform it into PDF using Prince.

Glyph macros in a nutshell

Format your documents using Textile or Markdown, and use Glyph Macros to do everything else:

Glyph Source:

```
section[header[Something about Glyph]
You can use Glyph macros in conjunction
 with _Textile_ or _Markdown_ to
produce HTML files effortlessly.
  section[header[What about PDFs?|pdf]
Once you have a single, well-formatted HTML
file, converting it to PDF is
extremely easy with a 3rd-party
renderer like =&gt;[http://www.princexml.com|Prince].
  ]
]
```

HTML Output:

```
<div class="section">
  <h2 id="h_10">Something about Glyph</h2>
  <p>You can use Glyph macros in conjunction with
     <em>Textile</em> or <em>Markdown</em> to
     produce HTML files effortlessly.</p>
  <div class="section">
   <h3 id="pdf">What about PDFs?</h3>
   <p>Once you have a single, well-formatted HTML
      file, converting it to PDF is
      extremely easy with a 3rd-party renderer
      like <a href="http://www.princexml.com">Prince</a>.</p>
  </div>
</div>
```

Resources

- Home Page: http://www.h3rald.com/glyph/
- Repository: http://www.github.com/h3rald/glyph/
- Bug Tracking: http://www.github.com/h3rald/glyph/issues
- Book (PDF): http://github.com/h3rald/glyph/raw/master/book/output/pdf/glyph.pdf
- Reference Documentation: http://yardoc.org/docs/h3rald-glyph/
- User Group: http://groups.google.com/group/glyph-framework

Chapter I – Getting Started

1.1 Creating your first Glyph Project

To install Glyph, simply run `gem install glyph`, like with any other Ruby gem. Then, create a new directory and initialize a new Glyph project, like so:

`mkdir` *test_document*

`cd` *test_document*

```
glyph init
```

That's it. You just created a new Glyph project in the `test_document` directory.

> **Glyph's dependencies**
>
> Glyph requires the following gems:
> - extlib
> - gli
> - treetop
> - rake
>
> Additionally, some Glyph macros may require additional gems, such as:
> - RedCloth (*textile* macro)
> - Maruku *or* Kramdown *or* BlueCloth (*markdown* macro)
> - Haml (if you want to load .sass files with the *style* macro)

Every Glyph project is comprised of the following directories:

- `images/` — used to store the image files used in your document.
- `lib/` — used to store your custom Glyph macros and Rake tasks.
- `output/` — used to store your generated output files.
- `styles/` — used to store your stylesheets.
- `text/*` — used to store your source text files.

Additionally, the following files are also created at top level:

- `config.yml` — containing your Project Configuration.
- `document.glyph` — containing your Document Structure
- `snippets.yml` — containing your text snippets.

1.2 Document Structure

Every Glyph project contains a `document.glyph` file that is typically used to define the document structure. The default `document.glyph`
generated automatically when creating a new project is the following:

```
document[
  head[style[default.css]]
  body[
    titlepage[
      title[]
      author[]
      pubdate[]
    ]
    frontmatter[
      toc[]
      preface[header[Preface]
        @[preface.textile]
      ]
    ]
    bodymatter[
      chapter[header[Chapter #1]
        @[chapter_1.textile]
      ]
      chapter[header[Chapter #2]
        @[chapter_2.textile]
      ]
    ]
    backmatter[
      appendix[header[Appendix A]
        @[appendix_a.textile]
      ]
    ]
  ]
]
```

Even without knowing anything about Glyph Language, you can easily figure out that this file defines a document with a Table of Contents, a Preface and some Chapters. `frontmatter[]`, `preface[]`, `chapter[]`, etc. are all Glyph *macros* used to define — in this case — some structural elements. In practice, this means that if you plan to generate an HTML document, they'll be converted into `<div>` tags.

Be aware that other macros, on the other hand, are used to do something completely different, e.g.:

- `toc[]` generates the document's Table of Contents
- `@[]` or its alias `include[]` is used to copy the contents of another file stored anywhere in the `/text` directory.

Let's now analyze this `document.glyph` more in detail.

- The `document[]` macro wraps every other macro. This is necessary to create the initial `<html>` tag.

- Similarly, `head[]` and `body[]` are used to generate the respective HTML tags. Actually, `head[]` already sets some metadata for you, by default (author and copyright).
- Within `head[]`, the `style[]` macro is used to load the `default.css` stylesheet, which is included by default the `/styles` directory of every Glyph project.
- Immediately after the `body[]` macro, the `titlepage[]` macro is used to define (guess…) the first page of your document. `title[]`, `author[]` and `pubdate[]` insert the title of the document, its author and the publication date (retrieved from the project's configuration settings).
- Then, the `frontmatter[]`, `bodymatter[]` and `backmatter[]` macros are used to further divide the portions of your document according to the rules of book design. They are not mandatory, but they can be used, for example, to number your appendixes with letters instead of numbers and similar.
- `preface[]`, `chapter[]`, `appendix[]` are just a way to wrap content in `<div>` tags, from an HTML point of view, but they are also necessary to nest the content of your document and generate the Table of Contents automatically, together with the `header[]` macro.

1.3 Project Configuration

Glyph stores configuration settings in the following YAML files:

1. Your *Project Configuration* is stored in the `config.yml` file, included in each Glyph Project.
2. Your *Global Configuration* is stored in a `.glyphrc` file in your $HOME (or %HOMEPATH% on Windows) directory (not created by default).
3. The *System Configuration* is stored in the source directory of Glyph itself.

When compiling, Glyph loads all these configuration files and merges them according to the following rules:

- A setting configured in the *Project Configuration* overrides the same setting in both Global and System configuration.
- A setting configured in the *Global Configuration* overrides the same setting in the *System Configuration*

Typically, you should use the *Project Configuration* for all project-specific settings and the *Global Configuration* for settings affecting all your projects (for example, you may want to set 'document.author' in the Global Configuration instead of setting it in the Project Configuration of all your Glyph projects). The *System Configuration* is best left untouched.

Instead of editing your configuration settings directly, you can use the `glyph config` command, as follows:

`glyph config` *setting [value]*

If no *value* is specified, glyph just prints the value of the configuration setting, so typing `glyph config document.author` right after creating a project (assuming you didn't set this in the Global Configuration) will print nothing, because this setting is blank by default.

To change the value of a configuration setting, specify a value right after the setting, like this:

```
glyph config document.author "John Smith"
```

In this way, the document author will be set to *John Smith* for the current project. To save this setting globally, add a `-g` option, like this:

```
glyph config -g document.author "John Smith"
```

Regarding configuration values and data types…

Glyph attempts to "guess" the data type of a configuration values by evaluation (`Kernel#instance_eval`) if the value:
- is wrapped in quotes (`"` or `'`) → String
- starts with a colon (`:`) → Symbol
- is wrapped in square brackets (`[` and `]`) → Array
- is wrapped in curly brackets (`\{` and `\}`) → Hash
- is *true* or *false* → Boolean
- If the value is *nil* → NilClass

Note that this guessing is far from being foolproof: If you type something like *{:test, 2}*, for example, you'll get an error.

There are plenty of configuration settings that can be modified, but most of them are best if left alone (and in the System Configuration file). For a complete reference, see Configuration Reference. Normally, you may just want to change the following ones:

Setting	Description
document.author	The author of the document
document.title	The title of the document
document.filename	The document file name

Chapter II – Authoring Documents

2.1 Text Editing

One of the aims of Glyph is streamlining text editing. Glyph accomplishes this through its own macro language that can be used in conjunction with Textile or Markdown.

2.1.1 Introducing Glyph Macros

By now you probably figured out what a macro looks like: it's an identifier of some kind that wraps a value or parameters within square brackets. More specifically:

- The macro identifier can contain *any* character except for: `[`, `]`, `\`, `|` or spaces.
- The delimiters can be either `[` and `]` or `[=` and `=]` (for more information on differences between delimiters, see Escaping and Quoting).
- The value can be anything, even other macros. If a macro supports more than one parameter, they must be separated with `|`. For example, the `link` (`=>`) macro can take an optional second parameter for the link text: `=>[#link_id|This is the link text]`.

2.1.2 Escaping and Quoting

Glyph doesn't require any special control characters like LaTeX, and its macro syntax is very straightforward and liberal. This however comes with a price: because square brackets are used as delimiters, you must escape any square bracket in your text with a backslash. That's not *too* bad if you think about it, unless you're writing programming code: in that case, escaping every single square bracket can be painful.

If a portion of your text contains an excessive amount of square brackets, you may consider using the `escape` macro (or better, its alias `.`) with `[=` and `=]` as delimiters. By itself, the escape macro doesn't do anything: it just evaluates to its contents, but the special delimiters act as a quote for any square bracket within them. As a consequence, any macro within `[=` and `=]` will *not* be evaluated.

You can use the quoting delimiters with *any* macro identifier. Obviously, using them as delimiters for things like `section` macros may not be a good idea, but they should really be mandatory with the `code` macro, like this:

```
code[=
section[header[A section]

This is a section.

  section[header[A nested section]
This is another section.
  ]
]
=]
```

Note Although quoting delimiters allow you to use square brackets without escaping them, you must still escape them if you want to escape quoting delimiter themselves.

Besides square brackets, there are other characters that must or can be escaped with backslashes, as shown in the following table

Escape Sequence	Evaluates to…	Notes
`\[`	`[`	Square brackets must be escaped unless used as macro delimiters or within a quoting macro.
`\]`	`]`	Square brackets must be escaped unless used as macro delimiters or within a quoting macro.
`\\`	`\`	Backslashes do not have to be escaped by default, but an escaped backslash will evaluate to itself.
`\=`	`=`	Equal signs do not have to be escaped by default, but an escaped equal sign will evaluate to iself.
`|`	`|`	Pipes must be escaped (even within quoting macros) unless they are used to separate two or more macro parameters.
`\.`		An escaped dot evaluates to nothing. Useful to separate macro identifiers from other characters: `_=>[#link|This is an emphasized link]_`

2.1.3 Sections and Headers

Glyph documents are normally organized as a hierarchical tree of nested chapters, appendixes, sections, etc. To define a section, use the `section` macro, like so:

```
section[
  header[Section #1]

Write the section contents here...

  section[
    header[Section #2]

This section is nested into the previous one.

  ] --[End of Section #2]
] --[End of Section #1]
```

This example defines two nested sections, each with its own header. The header is *mandatory*: it will be displayed at the start of the section and in the Table of Contents.

Note an important difference from HTML: there is no explicit level for the headers, as it will be determined at runtime when the document is compiled, based on how sections are nested. The previous code snippet (taken as it is), for example, will be transformed into the following HTML code:

```
<div class="section">
  <h2>Section #1</h2>
  <p>Write the section contents here...</p>
  <div class="section">
    <h3>Section #2</h3>
    <p>This section is nested in the previous one</p>
  </div>
</div>
```

By default, in Glyph the first header level is *2*, so the two headers are rendered as `h2` and `h3`, respectively (`--[...]` macros are *comments*, therefore they are not included in the final output).

There are *a lot* of macros that can be used in the same way as `section`, one for each element of Book Design. Each one of them is a simple wrapper for a `<div>` tag with a class set to its name.

The following table lists the identifiers of all section-like macros, divided according to the part of the book they should be placed in:

Frontmatter	`imprint` †, `dedication` †, `inspiration` †, `foreword` ‡, `introduction` ‡, `acknowledgement` ‡, `prologue` ‡, `toc` *
Bodymatter	`volume`, `book`, `part`, `chapter`
Backmatter	`epilogue` ‡, `afterword` ‡, `postscript` †, `appendix`, `addendum` ‡, `glossary` **‡, `colophon` †, `bibliography` **‡, `promotion` †, `references` **‡, `index` **‡, `lot` **‡, `lof` **‡

*: The `toc` macro is to generate the Table of Contents automatically, and it must be used with no contents (`toc[]`).

**: This macro is likely to be extended in future versions to generate/aggregate content automatically.

†: This section is not listed in the Table of Contents.

‡: Any subsection of this section is not listed in the Table of Contents.

> **Note** `frontmatter`, `bodymatter` and `backmatter` are also valid (and mandatory!) macro identifiers, typically already included in the default `document.glyph` file of every project.

2.1.4 Including Files and Snippets

If you're authoring a user manual, a long article or a book, writing everything inside a single file (`document.glyph`) may not be optimal. For this reason, Glyph provides an `include` macro (aliased by `@`) that can be used to include the contents of any file within the `text/` directory:

```
@[introduction.textile]
```

The macro above loads the contents of the `introduction.textile` file, that can be stored *anywhere* within the `text/` directory.

> **Note** Unlike with image and figures that must be included with their *relative* path to the `images/` folder, you must not specify a relative path when including text files. This is due to the fact that images are copied *as they are* in the `output/<format>/images/` directory and they have to be linked from the output file.
>
> A possible downside of this behavior is that file names must be unique within the entire `text/` directory (or any of its subdirectories)

When including a text file, by default an input filter macro is applied to its contents based on the file extension used:

- `.textile` → `textile`
- `.markdown` or `.md` → `markdown`

> **Tip** You can override this behavior by setting the `filters.by_file_extensions` configuration setting to `false`, like this:
>
> ```
> glyph config filters.by_file_extensions false
> ```

While including the context of an entire file is definitely a useful feature for content reuse, sometimes it can be an overkill. What if, for example, you just want to reuse a short procedure or even a sentence? In this case, you may want to consider using a *snippet* instead.

Snippets are text strings saved in YAML format in the `snippets.yml` file. They can be included anywhere in your document using the `snippet` macro (or its alias `&`).

Example

Consider the following `snippets.yml` file:

```
---
:glang: Glyph Language
:macros: Glyph Macros
:sq_esc: |-
  Square brackets must be escaped
  unless used as macro delimiters or within a quoting macro.
:markups: Textile or Markdown
:test: |-
  This is a
  Test snippet
```

You can use `&[markups]` anywhere in your document instead of having to type "Textile or Markdown" every time. Additionally, later on you can change the value of the `markups` snippets only in the `snippets.yml` file to change it everywhere else in the document.

Tip Snippets (or any other macro) can be nested within other snippets. Glyph takes care of checking if you nested snippets or macros mutually and warns you if necessary.

2.1.5 Links and Bookmarks

Lightweight markups let you create internal and external links in a very easy way, and you can still do so in Glyph. However, if you do so:

- There is no built-in way to check if they are valid
- There is no built-in way to determine the title of a link automatically

If you care about link validation and you want to save some keystrokes, then you should use the following markup-agnostic Glyph Macros:

- `link` (aliased to `=>`) — to create internal and external links.
- `anchor` (aliased to `#`) — to create named anchors (bookmarks) within your document.

Example

The following Glyph code:

```
This is a link to =&gt;[#test].

...

This is =&gt;[#wrong].

This is a #[test|test anchor].
```

Is translated into the following HTML code:

```
<p>This is a link to <a href="#test">test anchor</a>.</p>
<p>...</p>
<p>This is <a href="#wrong">#wrong</a>.</p>
<p>This is a <a id="test">test anchor</a>.</p>
```

Additionally, the following warning message is displayed when compiling:

```
warning: Bookmark 'wrong' does not exist
 -> source: @: aurhoting.textile
 -> path: document/body/bodymatter/chapter/@/textile/section/section/box/=>
```

Basically, if you use the `=>` and `#` macros, Glyph makes sure that:

- All links point to valid anchors within the document (regardless if the anchors are before or after the link, in snippets or included files).
- There are no duplicate anchors within the documents.
- If no title is specified as second parameter for the `=>` macro, the anchor's title is used as such.

Besides using the `#` macro, you can also create an anchor for a header by passing an extra parameter to the `header` macro, like this: `header[Header Title|my_anchor]`.

Note At present, link validation and automatic title retrieval only works with internal links (i.e. the check occurs if the first parameter of the `=>` macro starts with a `#`). In the future, the macro could be extended to support external links as well.

2.1.6 Evaluating Ruby code and Configuration Settings

Glyph Language is not a full-blown programming language, as it does not provide control flow or variables, for example.
However, it is possible to evaluate simple ruby code snippets using the `ruby` macro (aliased to `%`), like this:

- `%[2 + 2]` → 4
- `%[Time.now]` → Sun Mar 28 20:33:01 +0200 2010
- `%[Glyph::VERSION]` → 0.1.0

The scope for the code evaluation is the Kernel module, (with all inclusions required by Glyph itself).

Although it is possible to retrieve Glyph configuration settings in this way (e.g. `%[cfg('document.author')]`), the `config` macro (aliased to `$`) makes things slightly simpler (e.g. `$[document.author]`).

2.1.7 Images and Figures

In a similar way to links, if you want you can include images and figures using Textile or Markdown. If you want additional features, you can use the `img` and `fig` macros, as shown in the following example:

Example

The following Glyph code:

```
img[glyph.svg|20%|20%]

fig[example.png|An example figure.]
```

Is translated into the following HTML code:

```
<img src="images/glyph.svg" width="20%" height="20%" alt="-" />

<div class="figure">
  <img src="images/example.png" alt="-"/>
  <div class="caption">An example figure.</div>
</div>
```

Note In future releases, figures will be numbered automatically and included in a *List of Figures* section.

2.2 Compiling your project

By default, a Glyph project can be *compiled* into an HTML document. Additionally, Glyph can also be used to produce PDF documents through Prince, and in future releases more formats are likely to be supported.

2.2.1 Adding Stylesheets

Currently, Glyph does not provide any native way to format text and pages. The reason is that there's absolutely no need for that: CSS does the job just fine. In particular, CSS 3 offers specific attributes and elements that can be used specifically for paginated documents. That's no replacement for LaTeX by any means, but it is enough if you're not looking for advanced typographical features.

You can embed CSS files using the `style` macro, like this:

```
style[default.css]
```

In this case, the `style` macro looks for a `default.css` file in the `/styles` folder of your Glyph project and embeds it within a `<style>` tag. If you supply a file with a `.sass` extension, it will interpret it as a Sass file and convert it to CSS automatically (if the *Haml* gem is installed).

2.2.2 HTML output

To compile a Glyph project to an HTML document, use the `glyph compile` command within your Glyph project folder. Glyph parses the `document.glyph` file (and all included files and snippets); if no errors are found, Glyph creates an HTML document in the `/output/html` folder. The name of the HTML file can be set in the configuration (`document.filename` setting).

If you don't want to compile the whole project, you can specify a different source, like this:

```
glyph compile -s myfile.textile
```

2.2.3 PDF Output

To generate a PDF document, you must specify `pdf` as format, like this:

```
glyph compile -f pdf
```

The command above will attempt to compile the project into an HTML document and then call Prince to generate a PDF document from it. In order for this to work, you must download and install Prince. It's not open source, but the free version is fully functional, and it just adds a small logo on the first page.

> **Note** Glyph v0.1.0 has been successfully tested with Prince v6.0, and the PDF version of this very book was generated with it.

Chapter III – Extending Glyph

Glyph was created wih extensibility in mind. You can freely extend Glyph Language by creating or overriding macros, to do whatever you like. Macro definitions are written in pure Ruby code and placed in `.rb` files within the `/lib/macros` folder of your project.

3.1 Anatomy of a Macro

This is the source code of a fairly simple macro used to format a note :

```
macro :note do
  %{<div class="#{@name}"><span class="note-title">#{@name.to_s.capitalize}</span>#{@value}

    </div>}
end
```

The `macro` method takes a single Symbol or String parameter, corresponding to the name of the macro. In this case, the entire block (or *body* of the macro) is a String corresponding to what we want the macro to evaluate to: a `<div>` tag containing a note.

The body of the macro is evaluated in the context of the Glyph::Macro class, therefore its instance variables (like `@name` or `@value`) can be used directly.

Why using `@name` instead of just "note"?

For the `note` macro, it absolutely makes no difference. However, by using `@name` it is possible to re-use the same code for the `tip`, `important` and `caution` macros as well, which are in fact only aliases of the `note` macro:

```
macro_alias :important => :note
macro_alias :tip => :note
macro_alias :caution => :note
```

The following table lists all the instance variables that can be used inside macros:

Variable	Description
`@node`	A Node containing information about the macro, within the document syntax tree. Useful for accessing parent and child macros, and the current document. Normally, macro nodes contain the following keys: - `:name`, the name of the macro - `:value`, the value (i.e. the contents, within the delimiters) of the macro - `:source`, a String identifying the source of the macro (a file, a snippet, etc.) - `:document`, the parsed document tree Note that the first three keys can also be accessed via instance variables.
`@name`	The name of the macro
`@value`	The full contents (including parameters and nested macros) within the macro delimiters.
`@source`	A String identifying the source of the macro (a file, a snippet, etc.)
`@params`	The parameters passed to the macro. In other words, the value of the macro split by pipes (`|`).

3.2 Bookmarks and Headers

The Glyph::Macro class also includes a few methods to check and store bookmarks and headers. Consider for example the following source code for the `anchor` macro:

```
macro :anchor do
  ident, title = @params
  macro_error "Bookmark '#{ident}' already exists" if bookmark? ident
  bookmark :id => ident, :title => title
  %{<a id="#{ident}">#{title}</a>}
end
```

The `bookmark?` method can be used to check the existance of a particular ID within the whole document, while the `bookmark` method is used to store bookmark IDs and titles. In a similar way, you can use `header?` and `header` methods to check the existance of headers within the documents or store new ones.

3.3 Using Placeholders

Sometimes you may need to access some data that will not be available until the entire document has been fully parsed and analyzed. For example, in order to be able to validate internal links, it is necessary to know in advance if the bookmark ID referenced in the link exists or not, either before (that's easy) or even *after* the location of the link.

Here's the source code of the `link` macro:

```
macro :link do
  href, title = @params
  if href.match /^#/ then
    anchor = href.gsub(/^#/, '').to_sym
    bmk = bookmark? anchor
    if bmk then
      title ||= bmk[:title]
    else
      plac = placeholder do |document|
        macro_error "Bookmark '#{anchor}' does not exist" unless document.bookmarks[anchor]
        document.bookmarks[anchor][:title]
      end
      title ||= plac
    end
  end
  title ||= href
  %{<a href="#{href}">#{title}</a>}
end
```

If there's already a bookmark stored in the current document, then it is possible to retrieve its title and use it as link text. Otherwise, it is necessary to wait until the entire document has been fully processed and then check if the bookmark exists. To do so, use the `placeholder` method. When called, this method returns an unique placeholder, which is then substituted with the value of the block, right before the document is finalized.

Within the `placeholder` block, the `document` parameter is, by all means, the fully analyzed document.

3.4 Interpreting Glyph Code

What if you need to evaluate some Glyph code *within* a macro? Say for example you want to transform a parameter in a link, and you want to make sure that link gets validated exactly like the others, in this case, you can use the `interpret` method, as follows:

```
macro :fmi do
  topic, href = @params
  link = placeholder do |document|
    interpret "link[#{href}]"
  end
  %{<span class="fmi">for more information on #{topic}, see #{link}</span>}
end
```

When the `interpreter` method is called, the following happens:

1. A new Glyph document is created from the String passed to the method.

2. The bookmarks, headers and placeholders are passed from the main document to the new one. Because they are stored in Arrays or Hashes, they are passed by reference, so for example any new bookmark stored in the new document will also become available in the main document.
3. Any macro included in the String is evaluated, and the resulting text is returned by the method. Note that this new document does not get finalized: in other words, placeholders will be left as they are, and they'll eventually be replaced when *the main document* is finalized.

3.5 Further Reading

For more examples on how to create more complex macros, have a look at the source code of the existing ones.

To gain a deeper understanding on how macros are executed, have a look at the following Glyph classes:

- Glyph::Macro
- Glyph::Interpreter
- Glyph::Document
- Node

Chapter IV – Troubleshooting

4.1 Generic Errors

Error	Description
Undefined macro ‘*macro_name*’	…

4.2 Document Errors

Error	Description
Document contains syntax errors	…
Document has not been analyzed	…

4.3 Macro Errors

Error	Description
Mutual inclusion	…

Appendix A – Command Reference

Appendix B – Macro Reference

Appendix C – Configuration Reference