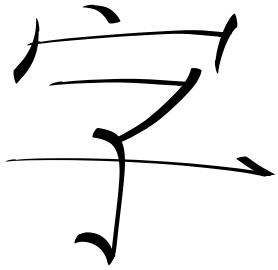

Glyph

Rapid Document Authoring Framework

v0.1.0

by *Fabio Cevasco*

April 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), avoiding mutual calls.
- Include text files in other text files.
- Include the value of any configuration setting (like 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/0.1.0/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)
> - BlueCloth *or* RDiscount *or* Maruku *or* Kramdown (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 the document.author setting 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 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`
> - 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 main purposes 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[#link_id|This is the link text]`.

A macro can often have one or more aliases. For example, `=>` is an alias for the link macro, so the following macro calls are equivalent:

```
=>[#test|Test Section]
link[#test|Test Section]
```

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 which 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 its alias `.`) with the `[=` and `=]` delimiters. By itself, the escape macro doesn't do anything: it just evaluates to its contents, but the special delimiters act as an escape 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 be more or less 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 delimiters 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 macro parameters.
`\.`		An escaped dot evaluates to nothing. Useful to separate macro identifiers from other characters: `_\.=>[#link|This link is emphasized using Textile]_`

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 commonly used in Book Design. Each one of them is a simple wrapper for a `<div>` tag with a `class` attribute 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 used to generate the Table of Contents automatically, and it must be used with no contents (i.e.: `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 call 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 and any of its subdirectories

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

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

> **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 as 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 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 link[#test].

...

This is link[#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/link
```

Basically, if you use the link macro and the anchor macro, 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 link macro, the anchor's title is used as such.

Besides using the anchor macro 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 link 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]` → Fri Apr 09 17:28:17 +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

Same as for links, you can also include images and figures using Textile or Markdown. If you want additional features, you can use the img macro and the fig macro, 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 could 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, CSS3 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).

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 v7.0, 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) inside 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 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 `interpret` 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 and 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

This chapter lists the most common error messages that can be returned when running a Glyph command. It does not aim to be an exhaustive list, especially if you extended Glyph by creating your own macros.

4.1 Generic Errors

Error Message	Description
Document contains syntax errors	This error is returned if the document was not parsed because of one or more syntax error. **At present, no indication on the exact location of the error(s) is provided**, so the only way to determine what went wrong is to try compiling a single file at a time by commenting out files included within `document.glyph` as necessary, and examine more closely the source of the files that do not compile.
Invalid alias: macro '*macro-name*' already exists	The alias name supplied to the `macro_alias` method has already been used for another macro or alias.
Undefined macro '*macro-name*'	The document contains a macro that does not exist, i.e. it is not a standard or used-defined Glyph macro or alias.
An error occurred when generating *file-name*.pdf	Returned if Prince could not generate the PDF file or if Prince is not installed. Normally, Prince provides additional details on the specific error(s).
Glyph cannot generate PDF. Please specify a valid tools.pdf_generator setting	Returned if the tools.pdf_generator setting has not be set to a valid PDF renderer. Currently, the only supported value for this setting is `prince`.
The current directory is not a valid Glyph project	Returned if a glyph command was executed outside a valid glyph project directory.
Invalid snippet file	The `snippet.yml` file contains invalid data. Most likely, it does not evaluate to a Ruby `Hash`.
Directory '*directory-name*' is not empty	Returned when executing the init command in a directory that is not empty.
File '*file-name*' already exists	Returned if the name of an existing file was specified as a parameter for the add command.

4.2 Command Errors

Error Message	Description
Please specify a file name	No file name was specified for the add command.
Output target not specified	Returned if no target was specified for the compile command *and* if the document.output setting is not set.
Unknown output target '*target-name*'	An unsupported output target was specified for the compile command. Only the following output targets are supported: - `html` - `pdf`
Too few/too many arguments	Returned if the config command was used with no arguments or more than two arguments respectively.
Unknown setting '*setting-name*'	The name of an unknown setting was specified for the config command.

4.3 Macro Errors

The following errors are displayed in the form:

macro-path message

Where:

- *macro-path* is the full path to the macro that returned the error, within the document syntax tree, e.g. `document/body/bodymatter/chapter/section/header/&` if the error occurrent in a snippet within the header of a section in the `bodymatter` part of the document.
- *message* is the error message.

Error Message	Description
Mutual inclusion	This error is returned if a catch-22 situation occurs with macro inclusion, for example if the body of a snippet includes a reference to the same snippet.
Snippet '*snippet-id*' does not exist	Returned by the snippet macro if an invalid snippet was supplied.
File '*file-name*' not found	Returned by the include macro if an invalid file was supplied.
Filter macro '*macro-name*' not found	Returned by the include macro macro if the filters.by_file_extension setting is set to `true` but the file extension of the included file is not recognized as a filter macro.
RedCloth gem not installed. Please run: gem insall RedCloth	Returned by the textile macro if the RedCloth gem is not installed.
No MarkDown converter installed. Please run: gem insall bluecloth	Returned by the markdown macro if no valid Markdown converter gem is installed. Glyph checks for: BlueCloth, Maruku, Kramdown and RDiscount.
Image/Figure not found	Retured by the img macro or the fig macro respectively, if the specified image file could not be found within the `images/` folder.
Bookmark '*bookmark-name*' already exists	Returned by the anchor macro or by the header macro if the anchor ID supplied as parameter has already been used in the document.
Bookmark '*bookmark-name*' already exists	Returned by the link macro if the anchor ID supplied as parameter has not been used in the document.
Stylesheet '*file-name*' not found	Returned by the style macro if the `.css` or `.sass` file supplied as parameter was not found in the `styles/` directory.
Haml is not installed. Please run: gem install haml	Returned by the style macro if a `.sass` file was passed as parameter but the Haml gem is not installed.

Appendix A – Command Reference

Glyph's command-line interface has been built using the gli (Git-like interface) gem. Therefore, Glyph commands are all written like this:

`glyph` *global-options* command *options parameters*

Where:

- *global-options* and *options* are in the form: `-n` *value* or `--name=`*value*, e.g. `-f pdf` or `--format=pdf`
- *parameters* are separated by whitespaces, and can be wrapped in quotes.

A.1 Global Options

A.1.1 `-d, --debug`

If specified, the command is executed in debug mode and additional diagnostic information is printed on the screen.

A.2 `add`

Creates a new text file in the `text/` folder.

Example: `glyph add introduction.textile`

A.2.1 Parameters

Parameter	Description
file-name	The name (or relative path) of the new file to be created.

A.3 `compile`

Compiles a Glyph document into an output file. If no options are specified, the `document.glyph` file is used as source to produce a standalone HTML file.

Example: `glyph compile -f pdf`

A.3.1 Options

Option	Description
-s (--source)	The source file to compile. **Default Value:** `document.glyph`
-f (--format)	The format of the output file. **Default Value:** `html` **Possible Values:** `html, pdf`

A.4 `config`

Gets or sets a configuration setting in the project or global configuration file (for more information on configuration files, see Project Configuration).

Examples:

```
glyph config document.filename
glyph config -g document.author "Fabio Cevasco"
```

A.4.1 Options

Option	Description
-g (--global)	If specified, the global configuration file is processed instead of the project file. **Default Value:** `false`

A.4.2 Parameters

Parameter	Description
setting	The name of a valid configuration setting.
value	The new value of the configuration setting.

A.5 `help`

Prints information about all Glyph commands or about one specific command.

Examples:
```
glyph help
glyph help compile
```

A.5.1 Parameters

Parameter	Description
command	A valid Glyph command.

A.6 `init`

Creates a new Glyph project in the current directory (if empty).

Example: `glyph init`

A.7 `todo`

Prints all the todo items saved using the todo macro.

Example: `glyph todo`

Appendix B – Macro Reference

B.1 Common Macros

B.1.1 `comment`

Evaluates to nothing. Used to add comments in a Glyph document that will not be displayed in output files.

Aliases: `--`
Example: `--[This is a comment. It will not be displayed in the output]`

B.1.2 `config`

Evaluates to the configuration setting referenced by its value.

Aliases: `$`
Example: `$[document.author]`

B.1.3 `escape`

Evaluates to its value. Commonly used with the escaping delimiters `[=` and `=]`.

Aliases: `.`
Example: `.[=Macros are escaped here =>[#test].=]`

B.1.4 `include`

Evaluates to the contents of a text file stored in the `text/` directory referenced by its value. If the filters.by_file_extension setting is `true`, filters the contents of the file using the filter macro corresponding to the file extension.

Aliases: `@`
Example: `@[introduction.textile]`

B.1.5 `ruby`

Evaluates its value as Ruby code (using `Kernel#instance_eval`).

Aliases: `%`

Examples:
```
%[Time.now]
%[Glyph::VERSION]
```

B.1.6 `snippet`

Evaluates to the snippet referenced by its value.

Aliases: `&`
Example: `&[glang]`

B.1.7 `todo`

Saves the value as a TODO item, which can be printed using the todo command.

Example: `todo[Remember to do this.]`

B.2 Filter Macros

B.2.1 `markdown`

Uses a markdown converter (BlueCloth, RDiscount, Maruku or Kramdown) to transform the value into HTML if the filters.target setting is set to `html`.

If the filters.by_file_extension setting is `true`, this macro is called automatically on included files with a `.markdown` or a `.md` extension.

Example: `markdown[This is *emphasized* text.]`

B.2.2 `textile`

Uses the RedCloth gem to transform the value into HTML or LaTeX, depending on the value of the filters.target setting.

If the filters.by_file_extension setting is `true`, this macro is called automatically on included files with a `.textile` extension.

Example: `textile[This is a *strong emphasis*.]`

B.3 Block Macros

B.3.1 box

Creates a titled box (`<div>` tag).

Example:

```
box[Why boxes?|
Boxes can be used to make a section of text stand out from the rest of the document.
]
```

B.3.2 code

Used to render a block of code within `<pre>` and `<code>` tags. For inline code, see the codeph macro.

Example:

```
code[
  def hello
    puts "Hello World"
  end
]
```

B.3.3 fig

Includes an image in the document, with an optional caption.

Examples:

```
fig[diagram.png]
fig[graph.png|Monthly pageviews]
```

B.3.4 `note`

Creates a note `div` containing the value.

Aliases: `important, caution, tip`
Example: `note[This is a note.]`

B.3.5 `img`

Includes an image in the document, optionally scaled according to the specified width and height. The image must be stored within the `images/` directory of the current project.

Examples:
```
img[icon.png]
img[holidays/landscape.jpg|70%]
img[logo.svg|50%|50%]
```

B.3.6 `pubdate`

Evaluates to a date string (in the format: *current-month current-year*; or *%B %Y*), within a `<div>` tag.

Example: `pubdate[]`

B.3.7 `subtitle`

Renders the subtitle of the document (based on the document.subtitle setting) within a `<h2>` tag.

Example: `subtitle[]`

B.3.8 `table`

Evaluates to an HTML table. Used in conjunction with the `tr`, `td` and `th` macros.

Example:

```
table[
  tr[
    th[Name]
    th[Value]
  ]
  tr[
    td[A]
    td[1]
  ]
  tr[
    td[B]
    td[2]
  ]
]
```

B.3.9 `td`

See `table`.

B.3.10 `title`

Renders the title of the document (based on the document.title setting) within a `<h1>` tag.

Example: `title[]`

B.3.11 `th`

See `table`.

B.3.12 `tr`

See `table`.

B.4 Inline Macros

B.4.1 anchor

Creates a named anchor (or bookmark).

Aliases: `bookmark, #`
Example: `#[test|Test Bookmark]`

B.4.2 codeph

Wraps the value in a `<code>` tag.

Example: `codeph[Kernel.instance_eval]`

B.4.3 fmi

Creates a *For More Information* link (for an example usage, see the link macro).

Example: `fmi[creating links|#links]`

B.4.4 link

Creates an hyperlink (for more information on creating links, see Links and Bookmarks).

Aliases: `=>`

Examples:

```
=>[#introduction]
=>[#troub|Troubleshooting]
=>[http://www.h3rald.com|H3RALD.com]
```

B.5 Structure Macros

B.5.1 body

Creates a `<body>` tag.

B.5.2 `div`

Creates a `<div>` tag.

Aliases: `acknowledgement`, `addendum`, `afterword`, `appendix`, `bibliography`, `book`, `chapter`, `colophon`, `dedication`, `epilogue`, `foreword`, `glossary`, `imprint`, `index`, `inspiration`, `introduction`, `lof`, `lot`, `part`, `postscript`, `preface`, `prologue`, `promotion`, `references`, `section`, `section`, `volume`

B.5.3 `document`

The root macro used in every Glyph document. It creates an `<html>` tag.

B.5.4 `head`

Creates a `<head>` tag, pre-populated with `title` and author/copyright `<meta>` tags.

B.5.5 `header`

Creates an `h2`, `h3`, `h4`, etc. header (for more information on using headers, see Sections and Headers).

Examples:

```
header[Introduction]
header[Getting Started|gs]
```

B.5.6 `section`

See `div`.

B.5.7 `style`

Embeds the content of a CSS or Sass file within a `<style>` tag (for more information on stylesheets, see Adding Stylesheets).

Example: `style[default.css]`

B.5.8 `toc`

Generates a *Table of Contents* based on how sections and headers are nested in the current document.

Example: `toc[]`

Appendix C – Configuration Reference

C.1 `document.*`

The following configuration settings are related to the current Glyph document. Therefore, you should update them right after creating a project.

C.1.1 `document.author`

The author of the document.

Default (YAML):

```
""
```

C.1.2 `document.filename`

The name of the output file.

Default (YAML):

```
""
```

C.1.3 `document.output`

The format of the output file. It can be set to any value stored in the document.output_targets setting.

Default (YAML):

```
html
```

C.1.4 `document.output_targets`

An `Array` containing all the possible output formats. This setting should not be changed by the user.

Default (YAML):

```
- :html
- :pdf
```

C.1.5 `document.source`

The main source file to compile. It can be also be overridden by calling the compile command with the `-s` option.

Default (YAML):

```
document.glyph
```

C.1.6 `document.subtitle`

The subtitle of the document, displayed using the subtitle macro.

Default (YAML):

```
""
```

C.1.7 `document.title`

The title of the document, displayed using the title macro.

Default (YAML):

```
""
```

C.2 `filters.*`

C.2.1 `filters.by_file_extension`

If set to `true`, a filter macro is applied to included files, based on their extensions (for more information on including files, see Including Files and Snippets).

Default (YAML):

```
true
```

C.2.2 `filters.markdown.converter`

The name of the markdown converter to use with the markdown macro. It can be set to one of the following values:

- BlueCloth
- RDiscount
- Maruku
- Kramdown

If not set, Glyph tests for the presence of each gem in the same order, until one is found.

Default (YAML):

```
bluecloth
```

C.2.3 `filters.redcloth.restrictions`

An `Array` containing restrictions applied to RedCloth, used by the textile macro (see RedCloth Documentation for more information).

Default (YAML):

```
[]
```

C.2.4 `filters.target`

The output target for filters. It can be set to `html` (for RedCloth and MarkDown) or `latex` (RedCloth-only).

Default (YAML):

```
html
```

C.3 `structure.*`

The following configuration settings are used internally by Glyph and should not be changed by the user.

C.3.1 `structure.backmatter`

The section types used in the document backmatter.

Default (YAML):

```
- :epilogue
- :afterword
- :postscript
- :appendix
- :addendum
- :glossary
- :colophon
- :bibliography
- :promotion
- :references
- :index
- :lot
- :lof
```

C.3.2 `structure.bodymatter`

The section types used in the document bodymatter.

Default (YAML):

```
- :volume
- :book
- :part
- :chapter
```

C.3.3 `structure.frontmatter`

The section types used in the document frontmatter.

Default (YAML):

```
- :preface
- :imprint
- :dedication
- :inspiration
- :foreword
- :introduction
- :acknowledgement
- :prologue
```

C.3.4 `structure.hidden`

The section types that will not be shown in the Table of Contents.

Default (YAML):

```
- :imprint
- :dedication
- :inspiration
- :postscript
- :colophon
- :promotion
```

C.3.5 `structure.special`

The section types that will be considered *special* and whose children will not be included in the Table of Contents.

Default (YAML):

```
- :preface
- :foreword
- :introduction
- :acknowledgement
- :prologue
- :epilogue
- :addendum
- :glossary
- :bibliography
- :references
- :index
- :lot
- :lof
```

C.4 `tools.*`

C.4.1 `tools.pdf_generator`

The external program used to generate PDF files. It can only be set to `prince`.

Default (YAML):

```
prince
```