字

Glyph

Rapid Document Authoring Framework

v0.3.0
by *Fabio Cevasco*
June 2010

Table of Contents

Introduction

Glyph is a *Rapid Document Authoring Framework*.

With Glyph, you can manage your documents tidily in *projects* and generate deliverables in different formats such as HTML or PDF (through Prince).

Main Features

Glyph comes with its very own 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.
- Include content only if certain conditions are satisfied.
- Define macros, snippets and configuration settings directly within your document.
- Highlight source code.
- Call macros from other macros (including snippets), avoiding mutual calls.
- Include text files within 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).
- Manage draft comments and todo items.
- Provide a simple, less-verbose syntax to write XML code.

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 --auto` — to keep recompiling the current document every time a file is changed.
- `glyph compile -f pdf` — to compile the current document into HTML and then transform it into PDF using Prince.
- `glyph compile readme.glyph` — to compile a *readme.glyph* located in the current directory into a single HTML file.
- `glyph outline -l 2` — Display the document outline, up to second-level headers.

Glyph macros in a nutshell

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

Glyph Source:

```
section[
  @title[Something about Glyph]
  txt[
You can use Glyph macros in conjunction
with _Textile_ or _Markdown_ to
produce HTML files effortlessly.
  ]
  p[
Alternatively, you can just use em[Glyph itself]
to generate HTML tags.
  ]
  section[
    @title[What about PDFs?]
    @id[pdf]
Once you have a single, well-formatted HTML
file, converting it to PDF is
extremely easy with a 3rd-party
renderer like =>[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>
   <p>
     Alternatively, you can just use <em>Glyph itself</em>
     to generate HTML tags.
   </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
- Development Wiki http://wiki.github.com/h3rald/glyph
- RubyGem Download http://www.rubygems.org/gems/glyph
- Book (PDF): http://github.com/h3rald/glyph/raw/0.3.0/book/output/pdf/glyph.pdf
- Reference Documentation: http://yardoc.org/docs/h3rald-glyph/
- User Group: http://groups.google.com/group/glyph-framework

License

Acknowledgement

Glyph was designed and developed by Fabio Cevasco (h3rald).

Special thanks to the following individuals who contributed to Glyph by reporting and fixing issues and/or proposing and implementing new features:

- Jamie Atkinson (Jabbslad)
- Sebastian Staudt (koraktor)

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
- 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)
- CodeRay *or* UltraViolet (highlight macro)
- directory_watcher (to use auto-regeneration with the compile command)

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 the structure of your document.
- `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:

```
book[
  @frontmatter[
    toc[]
    preface[
      @title[Preface]
      todo[Write the preface]
      include[preface]
    ]
  ]
  @bodymatter[
    chapter[
      @title[Chapter 1]
      todo[Write chapter 1]
      include[chapter_1]
    ]
    chapter[
      @title[Chapter 2]
      todo[Write chapter 2]
      include[chapter_2]
    ]
  ]
  @backmatter[
    appendix[
      @title[Appendix A]
      todo[Write appendix A]
      include[appendix_a]
    ]
  ]
]
```

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 some Chapters and an Appendix.

As you can see, Glyph wraps portions of text within square brackets preceded by an identifier. These identifiers are used for *macros* and *attributes*. The only syntactic difference between macros and attributes is that attributes are preceded by a `@`.

For now, think about a macro as something that performs a certain action and — generally — produces some text output or manipulation of the text inside it. In this way, it becomes easy to understand that the `chapter` macro creates a chapter and the include macro includes an external file, for example.

Attributes “belong” to the macro they’re in, so in this case the book macro has the following attributes:

- `@frontmatter`
- `@bodymatter`
- `@backmatter`

More specifically, in this `document.glyph` file:

- The book macro wraps every other macro and is used to create the document header and default title page.
- Then, the `@frontmatter`, `@bodymatter`, and `@backmatter` attributes are used to 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 through `@title` attributes.

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 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"
```

Tip It is also possible to change configuration settings inside your document, using the config: macro.

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 value 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. For everyday use, you may just want to change the settings defined in the document.* namespace.

Chapter II – Authoring Documents

2.1 .glyph files

The `text` folder of any Glyph folder contains all the text source files used to produce a document. Although there are no restrictions on the extension of the files in this folder, you may want to use `.glyph`, especially if Vim is your favorite text editor.
The reason is simple: a Glyph syntax file is available on vim.org. Although not essential, syntax highlighting does help when editing Glyph files.

2.2 Introducing Glyph Macros

The most important concept to grasp about Glyph is the concept of *macro*.

A Glyph macro is, in a nutshell, 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 also have *attributes*, which look exactly like macros but their identifier starts with a `@`.

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]`

2.3 Macro attributes

Although a macro can take any number of parameters, they are often no more than two or three, for readibility reasons: parameters have no name, but their position within a macro is significant.

If you have something like this:

```
custom_image[test.png|50%|50%|Test Image]
```

it may still be easy enough to understand what each parameter is used for, but:

- you can easily forget that the third parameter is the image width
- if you don't want to resize the image, you still have to pass *empty parameters* to the macro, like this: `custom_image[test2.png|||Test Image]`

To avoid these situations, some macros which would normally take three or four parameters take optional attributes instead, so you can write:

```
image[test.png
  @width[50%]
  @alt[Test Image]
  @height[50%]
]
```

More verbose, of course, but definitely more readable. In this way, if you won't want to scale an image, you can safely omit the `@width` and `@height` attributes.

Note Like parameters, attributes can contain other macros, too.

2.4 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 codeblock macro or the highlight macro, especially when it contains square brackets or even Glyph code, like this:

```
codeblock[=
  section[
    @title[A section]
    @id[test]
This is a section.
    section[
    @title[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]_`

2.5 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[
  @title[Section #1]
Write the section contents here...
  section[
    @title[Section #2]
This section is nested into the previous one.
  ] --[End of Section #2]
] --[End of Section #1]
```

This example defines two nested sections. If the `@title` attribute is specified like in this case, it will be converted to a proper HTML header and it will appear in the table of contents (see the toc macro).

Note an important difference from HTML: there is no need for an 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 takes no parameters.

******: 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 macro identifiers, but they are exposed as attributes for the book macro and the article macro, so if you're using either of these two macros as your root macro for your document, there's no need to use them explicitly.

2.6 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:

- you can't check if they are valid

- you can't infer the link title automatically

If you care about link validation and you want to save some keystrokes, then you should use:

- the link macro (aliased to =>) — to create internal and external links.
- the anchor macro (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: @: authoring.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 name is used as such.

Besides using the anchor macro, you can also create an anchor for a header by passing an `@id` attribute the the section macro, like this:

```
section[
  @title[My Section]
  @id[my_section]
...
]
```

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 (http) links as well.

2.7 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 image macro and the figure macro, as shown in the following example:

Example

The following Glyph code:

```
image[glyph.svg
  @with[20%]
  @height[20%]
]
figure[example.png|An example figure.
  @alt[Example Figure]
]
```

Is translated into the following HTML code:

```
<img src="images/glyph.svg" width="20%" height="20%" />
<div class="figure">
  <img src="images/example.png" alt="Example Figure"/>
  <div class="caption">An example figure.</div>
</div>
```

Any attribute passed to the image macro or the figure macro is automatically passed to the underlying `<img>` tag.

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

2.8 Source Code

If you're a programmer, chances are that you're going to include some source code in your articles and books. Glyph offers two ways to format code blocks effortlessly: the codeblock macro, which simply wraps text into `<pre>` and `<code>` tags, or the highlight macro. The last one requires either Coderay or Ultraviolet, but it provides syntax highlighting for the most common programming languages.

Cosider the following piece of ruby code:

```
def find_child(&block)
  children.each do |c|
    c.descend do |node, level|
      return node if block.call(node)
    end
  end
  nil
end
```

It can be wrapped in a highlight macro, like so:

```
highlight[=ruby|
  def find_child(&block)
    children.each do |c|
      c.descend do |node, level|
        return node if block.call(node)
      end
    end
    nil
  end
=]
```

...to produce the following, using the ultraviolet highlighter:

```
def find_child(&block)
  children.each do |c|
    c.descend do |node, level|
      return node if block.call(node)
    end
  end
  nil
end
```

Some Remarks

- Highlighters require some configuration. For more information on relevant configuration settings, see the filters.* configuration settings.
- If you're using the highlight macro together within the textile macro, you must wrap the macro call within `<notextile>` tags.
- You must always escape pipes (`|`) with the code or the highlight macro.

2.9 Other HTML Elements

So far we examined how to create `<div>` tags for sections, links, images... but what about lists, tables or paragraphs? How is it possible to create them using Glyphs? You have two possibilities (besides using raw HTML code, that is):

- use a lightweight markup supported by Glyph (currently Textile or Markdown)
- rely Glyph's *XML fallback* feature

2.9.1 Textile or Markdown

Textile or Markdown are very easy and intuitive to use, and they can produce HTML markup with almost no effort. Using them with Glyph is as simple as using the textile macro (aliased to `txt`) and the markdown macro (aliased to `md`).

Example

The following Glyph code:

```
textile[
This is a paragraph with some _emphasized_ text.

This is another paragraph with some -deleted- text.
* This is
* a bulletted
* list
]
```

produces the following HTML code:

```
<p>This is a paragraph with some <em>emphasized</em> text.</p>
<p>This is a paragraph with some <del>deleted</del> text.</p>
<ul>
  <li>This is</li>
  <li>a bulletted</li>
  <li>list</li>
</ul>
```

Important Be careful when using block-level HTML with Textile and Markdown: sometimes it may be necessary to add extra empty lines or escape tags.

2.9.2 XML Fallback

Sure Textile and Markdown are great, but sometimes you may want to just use HTML, without the extra verbosity, of course. Take tables for example: Textile offers an easy way to create them, but things may get dirty when you need to have multiple paragraphs or lists within cells.

Very early versions of Glyph used to offered some simple `table`, `tr`, `tr`, `td` macros just for that. Of course the problem was that thy didn't offer any way to customize the markup by adding, for example, CSS classes.

Instead, by default, Glyph can convert any unrecognized macro to the corresponding XML element and macro attributes to XML attributes.

Example

The following Glyph code:

```
table[@class[features]
  tr[
                th[ID]
                th[Priority]
                th[Description]
  ]
  tr[
    td[27]
    td[span[@style[color:red;font-weight:bold;] HIGH]]
    td[HTML output]
  ]
  tr[
    td[42]
    td[span[@style[color:green;font-weight:bols;] LOW]]
    td[
      p[Support for less-used tags:]
      ul[
        li[cite]
        li[sup]
        li[...]
      ]
    ]
  ]
]
```

Is translated into the following HTML code:

```
<table class="features">
  <tr>
    <th>ID</th>
    <th>Priority</th>
    <th>Description</th>
  </tr>
  <tr>
    <td>27</td>
    <td><span style="color:red;font-weight:bold;">HIGH</span></td>
    <td>HTML output</td>
  </tr>
  <tr>
    <td>42</td>
    <td><span style="color:green;font-weight:bold;">LOW</span></td>
    <td>
      <p>Support for less-used tags:</p>
      <ul>
        <li>cite</li>
        <li>sup</li>
        <li>...</li>
      </ul>
    </td>
  </tr>
</table>
```

Basically, if the language.options.xml_fallback setting is set to `true`, any macro unknown to Glyph with at most one parameter will be converted to an XML tag with the same name and any attribute will be converted to the corresponding XML attribute.

> **Important** While macro names and attributes are validated so that an error is returned if they contain illegal character, no check is performed against any particular XML schema.

Additionally, it is possible to force macro-to-XML conversion by prepending an equal sign to any macro, so for example `=snippet[test]` will be converted into `<snippet>test</snippet>`.

2.9.3 Blacklisted XML tags

By default, the following tags are blacklisted and will be ignored:

- applet
- base
- basefont
- embed
- frame
- frameset
- iframe
- isindex
- meta
- noframes
- noscript
- object
- param
- title

> **Tip** You can change this list by modifying the language.options.xml_blacklist setting.

2.10 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* among the default Glyph stylesheets, 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).

2.10.1 Default Stylesheets

Glyph provides the following default stylesheets, that can be referenced directly using the style macro:

File name	Notes
`default.css`	The stylesheet used for this book.
`pagination.css`	A CSS3-compliant stylesheet used for pagination, suitable for PDF generation using Prince.
`coderay.css`	The default Coderay stylesheet, used for syntax highlighting.
`ultraviolet/*`	This folder contains the following Ultraviolet stylesheets, used for syntax highlighting: `active4d.css`, `all_hallows_eve.css`, `amy.css`, `blackboard.css`, `brilliance_black.css`, `brilliance_dull.css`, `cobalt.css`, `dawn.css`, `eiffel.css`, `espresso_libre.css`, `idle.css`, `iplastic.css`, `lazy.css`, `mac_classic.css`, `magicwb_amiga.css`, `pastels_on_dark.css`, `slush_poppies.css`, `spacecadet.css`, `sunburst.css`, `twilight.css`, `zenburnesque.css`

2.11 Including Files and Snippets

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

```
include[general/introduction.textile]
```

The macro call above loads the contents of the `introduction.textile` file, within the `text/general` directory.

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

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

You can override this behavior by setting the `filters.by_file_extensions` configuration setting to `false`. If no extension is specified, `.glyph` is assumed.

Tip The include macro can also be used to include (and evaluate) ruby files (with a `.rb` extension). In this case, the ruby file must be placed within the `lib/` directory of the current project.

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 or a single word? 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 `&`).

Tip Besides storing snippets in the `snippets.yml` file, you can also define them right in your document, using the snippet: macro.

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` snippet 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.

2.12 Evaluating Ruby code and Configuration Settings

Glyph Language is not a full-blown programming language and it is currently not Turing-complete (it does not provide loops). However, it is possible to evaluate simple ruby code snippets using the `ruby` macro (aliased to `%`), like this:

- `%[2 + 2]` ⟶ 4
- `%[Time.now]` ⟶ Sun Jun 13 14:33:00 +0200 2010
- `%[Glyph::VERSION]` ⟶ 0.3.0

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]`).

2.13 Conditional Macros

Sometimes you may want text to be included in a document only if certain conditions are satisfied. For example, you may want to display a disclaimer section only if the document is a draft (see the document.draft setting), or use a particular stylesheet only if when you generate a PDF document.

To do so, you can use the condition macro (aliased by `?`), and a set of additional macros that can be used as conditional operators i.e.:

- eq macro
- not macro
- and macro
- or macro
- match macro

Consider the following code:

```
?[$[document.draft]|
This is a first draft of the Glyph Book]
?[not[$[document.draft]]|
This is the official version of the Glyph Book]
```

In this case, if `document.draft` is set to `true`, “This is a first draft of the Glyph Book” will be displayed; if not, “This is the official version of the Glyph Book” will be displayed instead.

The condition macro takes two parameters:

- the first one is the condition to evaluate
- the second one is the text to include in the document only if the condition is satisfied.

Note that *both* parameters can contain macros, of course, so you can write things like:

```
?[and[
    eq[$[document.output]|pdf]
    |
    eq[$[tools.pdf_generator]|prince]
    ]
  |
  style[pagination.css]]
```

In this case, the `pagination.css` stylesheet is included only when you're generating a PDF document using Prince XML.

2.13.1 Results of conditional expressions

The condition macro in Glyph works in a similar way as conditionals in programming languages: if the conditional expression (supplied as first parameter) is satisfied then the second parameter is executed or displayed. But when is a conditional expression satisfied? Glyph is a simple mini-language to perform text manipulation, and has no types, it can only understand text, therefore:

- A conditional expression is satisfied if it evaluates to a non-empty string except “false”.
- A conditional expression is not satisfied if it evaluates to an empty string or the string “false”.

Chapter III – Generating Output Files

3.1 Compiling a 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 output targets will be supported.

3.1.1 HTML output

To compile a Glyph project to an HTML document, use the 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).

3.1.2 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 v0.3.0 has been successfully tested with Prince v7.0, and the PDF version of this very book was generated with it.

3.1.3 Auto Regeneration

You can also call the compile command with a `--auto` switch. If you do so, your project will be recompiled automatically every time any source file is changed.

Note Auto regeneration requires the directory_watcher gem to be installed.

3.2 Compiling single Glyph files

Glyph's primary goal is to author complex documents like books or manuals. In order to do so, a Glyph project is required to keep everything organized and automated via a set of predefined conventions, exactly like Ruby on Rails or other similar frameworks do.

If you want to write a one-page article or a short draft, however, creating and managing Glyph projects can be an unnecessary burden. Luckily, you don't have to: you can use Glyph to compile single files containing Glyph code, by adding one parameter (or two if you want to specify a custom destination file) to the compile command, like this:

```
glyph compile source.glyph destination.htm
```

This command will process a file called `source.glyph` and produce an HTML file called `destination.htm`.

3.2.1 Limitations

This sort of “lite” mode comes with a few minor limitations:

- Snippets can only be defined inside the source file, using the snippet: macro.
- Project configuration settings can only be defined inside the source file, using the config: macro.
- Custom macros can only be defined inside the source file, using the macro: macro.
- Images must be referenced with their absolute path, or a path relative to the current directory, and will not be copied anywhere when the output file is generated.
- Stylesheets must be referenced with their absolute path, or a path relative to the current directory, or the name of an existing Glyph system stylesheet.
- The files included through the include macro must be referenced with their absolute path, or a path relative to the current directory.

3.3 Using Glyph programmatically

Besides using Glyph from the command line, you can also use it straight from your code. Glyph’s public API is simple and can be used to:

- Retrieve and update configuration settings (using `Glyph[]` and `Glyph[]=`)
- Filter text to HTML (using `Glyph#filter`)
- Compile Glyph source files into HTML or PDF files (using `Glyph#compile`)

That’s pretty much it. Of course, both the `filter` and `compile` method cause Glyph to run in *lite* mode, so the same limitations apply.

Tip
For an example on how to use Glyph programmatically (specifically in conjunction with the nanoc static site generator), see h3rald.com source code, in particular:

- lib/glyph-data.rb — updating configuration settings.
- lib/glyph-data.rb — using the `Glyph#filter` method.
- Rules — using the `Glyph#compile` method to generate PDF files.

3.3.1 Modes

It is possible to specify some flags (or "modes") to make Glyph behave slightly different than normal, as shown in the following table (by default, none of these is used).

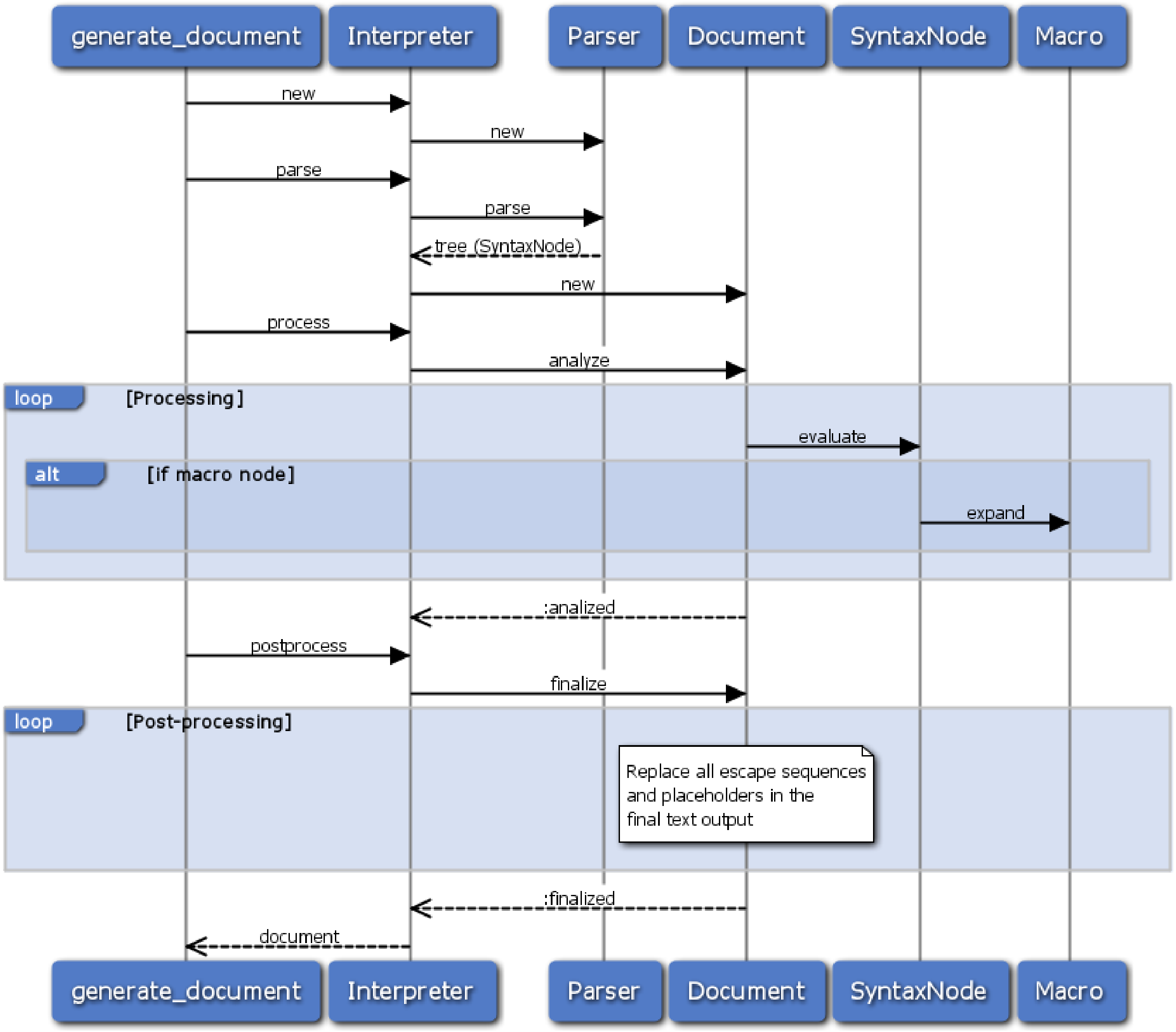
Name	Writer Method	Reader Method	Description
Test Mode	`Glyph.test_mode=`	`Glyph.test?`	Used internally by the `rake spec` task to run Glyph's specs.
Library Mode	`Glyph.library_mode=`	`Glyph.library?`	If enabled, the compile command command will raise exceptions instead of printing errors on the screen. Enabled by the `Glyph.compile` command.
Debug Mode	`Glyph.debug_mode=`	`Glyph.debug?`	If enabled, additional diagnostic information (such as backtraces or macro values) will be displayed. Enabled by specifying the debug switch when running a Glyph command.
Lite Mode	`Glyph.lite_mode=`	`Glyph.lite?`	Used to compile single files. Enabled by: ▪ The `Glyph.compile` and `Glyph.filter` methods. ▪ The compile command, if at least one parameter is supplied.
Safe Mode	`Glyph.safe_mode=`	`Glyph.safe?`	If enabled, the following macros cannot be used and will return an error: ▪ ruby macro ▪ macro: macro ▪ include macro ▪ rewrite: macro ▪ config: macro

Chapter IV – Extending Glyph

4.1 A quick look at Glyph's internals

If you plan on extending Glyph, knowing how it works inside helps. It is not mandatory by any means, but it definitely helps, especially when creating complex macros.

What happens behind the scenes when you call `glyph compile`? Glyph's code is parsed, analyzed and then translated into text, and here's how:



A sequence diagram for document generation

From the diagram, it is possible to divide the document generation process into three phases:

- The *Parsing Phase* starts when a chunk of Glyph code is passed (by the `generate:document` Rake task, for example) to a `Glyph::Interpreter`. The interpreter initializes a `Glyph::Parser` that parses the code and returns an *Abstract Syntax Tree* (AST) of `Glyph::SyntaxNode` objects.
- The *Analysis Phase* (Processing) starts when the interpreter method calls the `analyze` method, instantiating a new `Glyph::Document`. The `Glyph::Document` object evaluates the AST expanding all macro nodesth (that's when macros are executed) and generates string.
- The *Finalization Phase* (Post-Processing) starts when the interpreter calls the `finalyze` method, causing the `Glyph::Document` object to perform a series of finalizations on the string obtained after analysis, i.e. it replaces escape sequences and placeholders.

4.1.1 Example: A short note

As an example, consider the following Glyph code:

```
fmi[something|#test]
...
section[
  @title[Test Section]
  @id[test]
...
]
```

This simple snippet uses the fmi macro to link to a section later on in the document. When parsed, the produced AST is the following:

```
{:name=>:"--"}
  {:name=>:fmi, :escape=>false}
    {:name=>:"0"}
      {:value=>"something"}
    {:name=>:"1"}
      {:value=>"#test"}
  {:value=>"\n"}
  {:value=>"\[", :escaped=>true}
  {:value=>"..."}
  {:value=>"\]", :escaped=>true}
  {:value=>"\n"}
  {:name=>:section, :escape=>false}
    {:name=>:"0"}
      {:value=>"\n\t"}
      {:value=>"\n\t"}
      {:value=>"\n"}
      {:value=>"\[", :escaped=>true}
      {:value=>"..."}
      {:value=>"\]", :escaped=>true}
      {:value=>"\n"}
    {:name=>:title, :escape=>false}
      {:value=>"Test Section"}
    {:name=>:id, :escape=>false}
      {:value=>"test"}
```

This output is produced by calling the `inspect` method on the AST. Each `Glyph::SyntaxNode` object in the tree is basically an ordinary Glyph Hash with a parent and 0 or more chidren, so the code snippets above shows how the syntax nodes are nested.

The AST contains information about macro, parameter and attribute names, and escaping, and raw text values (the nodes without a `:name` key), but nothing more.

When the AST is analyzed, the resulting textual output is the following:

```
<span class="fmi">for more information on something, see ‡‡‡‡‡PLACEHOLDER ¤ 1‡‡‡‡‡
</span>
\[...\]
<div class="section">
<h2 id="test">Test Section</h2>
\[...\]

</div>
```

This looks almost perfect, except that:

- There's a nasty placeholder instead of a link: this is due to the fact that when the link is processed, there is no `#text` anchor in the document, but there may be one afterwards (and there will be).
- There are some escaped brackets.

Finally, when the document is finalized, placeholders and escape sequences are removed and the final result is the following:

```
<span class="fmi">for more information on something, see <a href="#test">Test Section</a></spar
[...]
<div class="section">
<h2 id="test">Test Section</h2>
[...]

</div>
```

4.2 Defining Macros

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.

Alternative Ways to Define Macros

You can also define macros:

- inside your document, using the macro: macro.
- Using the include macro specifying the path to an `.rb` file containing macro definitions stored in the `lib/` directory (useful especially when compiling single Glyph files).

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.

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

Variable	Description
`@node`	A `Glyph::MacroNode` containing information about the macro. Useful for accessing parent and child macros, and the current `Glyph::Document`. Normally, instances of the `MacroNode` class contain the following keys: ▪ `:name`, the name of the macro. ▪ `:source`, a `String` identifying the source of the macro (a file, a snippet, etc.) ▪ `:value`, the value of the macro (populated after the document has been parsed and analyzed). ▪ `:escape`, whether the macro is a quoting macro or not. ▪ `:document`, the instance of `Document` the macro is contained in (populated after the document has been parsed and analyzed). Note that the first two keys can also be accessed via instance variables.
`@name`	The name of the macro.
`@source`	A `String` identifying the source of the macro (a file, a snippet, etc.).

4.3 Parameters and Attributes

Perhaps the most common things to do in a macro definition is accessing parameters and attributes. When doing so, it is important to consider whether we want to retrieve the *raw value* of and attribute or parameter or its *expanded value*. The difference between the two will become clearer in the following sections and also in the Interpreting Glyph Code section.

4.3.1 Accessing Expanded Values

Normally, you just want to get the value of an attribute or parameter and use it in the macro. This means, in other words, its *expanded* value, i.e. the value resulting from the expansion of the macros (if any) within the attribute or parameter.

To access expanded values, use the following methods:

- `parameter` (or `param`): Returns the expanded value of the parameter specified by number. Other parameters are not expanded.
- `value`: Returns the expanded value of the first parameter (i.e. like `parameter(0)`).
- `attribute` (or `attr`): Returns the expanded value of the attribute specified by name. Other attributes are not expanded.
- `parameters` (or `params`): Returns an array of expanded parameters.
- `attributes` (or `attrs`): Returns a hash of expanded attributes.

4.3.2 Accessing Raw Values

While accessing expanded values is simple and immediate, in some cases it may not produce the desired results. Consider the following macro definition:

```
macro :nest_section do
  interpret %{section[
    @title[A]
    section[
      @title[B]
      #{value}
    ]
  ]}
end
```

And suppose to use it as follows:

```
nest_section[
  section[
    @title[Inner Section]
    ...
  ]
]
```

It produces the following HTML code:

```
<div class="section">
  <h2 id="h_2">A</h2>
  <div class="section">
    <h3 id="h_3">B</h3>
    <div class="section">
      <h2 id="h_1">Inner Section</h2>
...
    </div>
  </div>
</div>
```

Everything is fine *except* for the header level: the heading "Inner Section" is of level 2, but it should be level 4!

This happens because the inner section is evaluated *before* the `nest_section` macro: after all, we ask for it ourselves when we call the `value` method inside the macro definition. When the value is expanded, there are no outer sections *yet*.

To avoid this unwanted behavior, we can use the `raw_value` method instead, that returns the first parameter converted back to a Glyph code string.

Tip To be on the safe side, always use `raw_*` methods when interpreting.

To access raw values, use the following methods:

- `raw_parameter` (or `raw_param`): Returns the raw parameter value of the parameter specified by number.
- `raw_value`: Returns the first raw parameter value (i.e. like `raw_parameter(0)`).
- `raw_attribute` (or `raw_attr`): Returns the attribute value of the attribute specified by name.

4.4 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.

4.5 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
    anc = href.gsub(/^#/, '').to_sym
    bmk = bookmark? anc
    if bmk then
      title ||= bmk[:title]
    else
      plac = placeholder do |document|
        macro_error "Bookmark '#{anc}' does not exist" unless document.bookmarks[anc]
        document.bookmarks[anc][: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.

4.6 Using Validators

If you need to make sure that a macro is used properly, consider using validators. These methods can be used anywhere within the macro code to check whether certain conditions are met or not. Some default validators are provided to check the number of parameters of a macro, and they are actually used in some system macros.

If you want to create your own validators, you can call the generic `validate` method which takes the message to display in case of error, a Hash of options and a block containing the validation to perform.

Validating macro placement

You can, of course, create your own validators to check whether a macro is used within another. While this may seem a good idea to enforce constraints into the way documents are created, it has one major drawback: if you define a macro with such validation, you're effectively limiting its usage, so for example you won't be able to use within snippets or other custom macros.

Suppose, for example, that the box macro is only allowed within a `section` macro. This means that, for example:

- the macro cannot be used within `chapter` or `appendix` macros.
- the macro cannot be used in snippets

Even if you consider all the possibilities within the scope of the default macros provided with Glyph, this would also make the `box` macro unusable within custom macros.

4.7 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.

4.7.1 Rewriting

While the `interpret` method is useful to evaluate Glyph code in a macro while performing other actions (storing a bookmark, checking for the presence of an anchor, etc.), in some cases it may not be necessary. If you simply want your macro to be converted into existing Glyph macro without performing any action excepting parameter substitution, you can just use the rewrite: macro within youy Glyph document

Consider the following macro definition:

```
macro :issue do
  interpret %{
    tr[
      td[.=>[http://github.com/h3rald/glyph/issues/closed#issue/#{param[0]}|##{param(0)}]]
      td[txt[#{param(1)}]]
    ]
  }
end
```

The `issue` macro is only rewriting existing Glyph code around the two parameters provided. In this case, it is possible to do exactly the same thing using the rewrite: macro:

```
rewrite:[issue|
  tr[
    td[.=>[http://github.com/h3rald/glyph/issues/closed#issue/{{0}}|#{{0}}]]
    td[txt[{{1}}]]
  ]
]
```

Within the rewrite: macro, it is possible to use a special syntax to call the `raw_attr` or `raw_param` methods:
`{{`*parameter_number* or *attribute_name*`}}`

4.8 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::Parser`
- `Glyph::SyntaxNode`
- `Glyph::Interpreter`
- `Glyph::Document`
- `Glyph::Macro`

Chapter V – 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 or you're embedding Ruby code using the ruby macro.

> **Tip** As a general rule, more diagnostic information is provided if Glyph is run in debug mode.

5.1 Generic Errors

Error Message	Description
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.

5.2 Parsing Errors

Error Message	Description
Macro delimiter '*delimiter*' not escaped	Returned in case of unescaped `[` or `]`.
macro_name[...] - A macro cannot start with '@' or a digit.	Returned if an invalid macro name was specified.
Macro '*macro_name*' not closed	Returned if a macro lacks its end delimiter `=]`.
Escaping macro '*macro_name*' not closed	Returned if an escaping macro lacks its end delimiter `=]`.
Attribute *@attribute_name* not closed	Returned if a macro attribute lacks its end delimiter `]`.
Attributes cannot be nested	Returned if a macro attribute was found immediately within another attribute.
Cannot nest escaping macro '*macro_name_1*' within escaping macro '*macro_name_2*'	Returned if an escaping macro contains another escaping macro.
Parameter delimiter '|' not allowed here	Returned if a parameter delimiter is outside a macro or inside an attribute.

5.3 Command Errors

Error Message	Description
Source file '*source_file*' does not exist	Returned if Glyph is running in lite mode and the specified source file was not found.
Source and destination file are the same	Returned if Glyph is running in lite mode and you specified the same source and destination files.
DirectoryWatcher is not available. Install it with: gem install directory_watcher	Returned if auto regeneration is enabled but the `directory_watcher` gem in not installed.
Document cannot be finalized due to previous errors	Returned if one or more errors occurred in the document prevented finalization.
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.
Cannot reset '*setting_name*' setting (system use only).	Returned by the config command when attempting to override a setting in the `system.*` namespace.

5.4 Macro Errors

The following errors are displayed in the form:

message
  source: *macro_source*
  path: *macro_path*

macro_value

Where:

- *message* is the error message.
- *macro_source* is the file or snippet where the error occurred.

- *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.
- *macro_value* is the value of the macro (shown only if `Glyph` is running in debug mode).

Error Message	Description
Macro '*name*' takes up to *x* parameter(s) (*y* given)	Returned if the macro was called with extra parameters.
Macro '*name*' takes at least *x* parameter(s) (*y* given)	Returned if the macro was called with fewer parameters than expected.
Macro '*name*' takes exactly *x* parameter(s) (*y* given)	Returned if the macro was called with a different number of parameters than.
Macro not available when compiling a single file.	Returned by the include macro if used in lite mode.
Filter macro '*extension*' not available	Returned by a filter macro if filters.by_file_extension setting is set to @true@, but the extension was not recognized.
Invalid regular expression: *regexp*	Returned by the match macro if an invalid regular expression was supplied.
Macro '*name*' takes no parameters (*x* given)	Returned if the macro was called with parameters but none are requested.
No highlighter installed. Please run: gem install coderay	Returned by the highlight macro if no highlighters are installed.
CodeRay highlighter not installed. Please run: gem install coderay	Returned by the highlight macro if filters.highlighter setting is set to @coderay@ but Coderay is not installed.
UltraViolet highlighter not installed. Please run: gem install ultraviolet	Returned by the highlight macro if filters.highlighter setting is set to @ultraviolet@ but Ultraviolet is not installed.
Mutual Inclusion in parameter/attribute(*name*): '*value*'	Returned if a catch-22 situation occurs with macro inclusion, for example if the value 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 ID 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 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 install RedCloth	Returned by the textile macro if the RedCloth gem is not installed.
No MarkDown converter installed. Please run: gem install 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 image macro or the figure 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 section macro if the anchor ID supplied as attribute 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.
Invalid XML element: '*element_name*'	An invalid XML element name was supplied to the `|xml|` system macro (see Other HTML Elements).
Invalid XML element: '*element_name*'	An invalid XML attribute name was supplied to the `|xml|` system macro (see Other HTML Elements).
Macro '*macro_name*' cannot be used in safe mode	Returned if a forbidden macro was used in safe mode (see Modes).
Cannot reset 'system.*setting_name*' setting (system use only).	Returned by the config: when attempting to override a setting in the `system.*` namespace.
Macro '*macro_name*' cannot be defined by itself	Returned by the rewrite: if it contains a macro with the same name.

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 Parameters

Parameter	Description
source	The source glyph file to compile *(Optional)*.
destination	The destination file *(Optional)*.

A.3.2 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` (`--auto`)	If specified, enable auto regeneration (requires the directory_watcher gem to be installed).

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 `outline`

Display an outline of the current document.

A.7.1 Options

Option	Description
`-l` (`--limit`)	Only display headers until the specified level.
`-i` (`--ids`)	Display section IDs.
`-f` (`--files`)	Display file names.
`-t` (`--titles`)	Display section titles.

Examples:

```
glyph outline -it -l 1
glyph outline -l 2
glyph outline -f
```

A.8 `todo`

Prints all the todo items saved using the todo macro.

Example: `glyph todo`

Appendix B – Macro Reference

B.1 Core Macros

B.1.1 `alias`

Creates a macro alias.

Example: `alias[s|section]`

B.1.1.1 Parameters

Parameter	Description
0	The name of the alias.
1	The name of an existing macro.

B.1.2 `and`

Conditional `and` operator, to be used with the condition macro.

Example: `?[and[true|false]|This is never displayed.]`

B.1.2.1 Parameters

Parameter	Description
0	The first expression to test
1	The second expression to test

B.1.3 `comment`

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

Aliases: `--`

Example: `--[=>[#link|This link will not be evaluated]]`

B.1.3.1 Parameters

Parameter	Description
0	The contents to comment out

B.1.3.2 Remarks

Macros are not expanded within comments.

B.1.4 `condition`

Tests a conditional expression. For more information, see Conditional Macros.

Aliases: `?`

B.1.4.1 Parameters

Parameter	Description
0	The condition to test
1	The contents to expand if the condition is satisfied.

B.1.4.2 Remarks

For examples see any of the following:

- and macro
- or macro
- not macro
- match macro
- eq macro

B.1.5 `config`

Returns the value of a configuration setting.

Aliases: `$`

Example: `$[document.author]`

B.1.5.1 Parameters

Parameter	Description
0	The full name of a configuration setting.

B.1.6 `config:`

Sets the value of a configuration setting.

Aliases: `$:`

Example: `$:[document.draft|true]`

B.1.6.1 Parameters

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

B.1.6.2 Remarks

This macro cannot be used in safe mode.

B.1.7 `eq`

Conditional equality operator, to be used with the condition macro.

Example: `?[eq[$[document.draft]|true]|This is displayed only in draft documents.]`

B.1.7.1 Parameters

Parameter	Description
0	The first expression to test
1	The second expression to test

B.1.8 `escape`

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

Aliases: `.`

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

B.1.8.1 Parameters

Parameter	Description
0	The contents to escape.

B.1.9 `include`

Evaluates to the contents of a text file stored in the `text/` directory referenced by its relative path. 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: `include[frontmatter/introduction]`

B.1.9.1 Parameters

Parameter	Description
0	The file to include.

B.1.9.2 Remarks

- This macro cannot be used in safe mode.
- `.glyph` is assumed if no file extension is specified.
- This macro can also be used to include `.rb` ruby files within the `lib` directory. File contents are evaluated in the context of the Glyph module.

B.1.10 `match`

Checks a string against a regular expression.

Example: `?[match[Hello!|/^hell/i]|This is always displayed]`

B.1.10.1 Parameters

Parameter	Description
0	The string to check.
1	The regular expression to match against the string.
2	The contents to expand if the string matches.

B.1.10.2 Remarks

This macro must be used with the condition macro.

B.1.11 `macro:`

Defines a macro.

Aliases: `%:`

Example: `%:[test|"test: #{value}"]`

B.1.11.1 Parameters

Parameter	Description
0	The name of the new macro.
1	The macro definition (Ruby code).

B.1.11.2 Remarks

- This macro cannot be used in safe mode.
- The new macro can only be used *after* its declaration.

B.1.12 `not`

Conditional `not` operator, to be used with the condition macro.

Example: `?[not[false]|This is always displayed.]`

B.1.12.1 Parameters

Parameter	Description
0	The expression to negate

B.1.13 `or`

Conditional `or` operator, to be used with the condition macro.

Example: `?[or[true|false]|This is always displayed.]`

B.1.13.1 Parameters

Parameter	Description
0	The first expression to test
1	The second expression to test

B.1.14 `rewrite:`

Defines a new macro by rewriting (for more information, see Rewriting)

Aliases: `rw:`

Example

```
rw:[release|
  section[
    @title[Release {{0}}]
    {{1}}
  ]
]
```

B.1.14.1 Parameters

Parameter	Description
0	The name of the new macro.
0	The macro definition (Glyph code).

B.1.14.2 Remarks

- The new macro can only be used *after* its declaration.
- This macro cannot be used in safe mode.

B.1.15 `ruby`

Evaluates its value as Ruby code within the context of the Glyph module.

Aliases: `%`

Examples:

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

B.1.15.1 Parameters

Parameter	Description
0	The Ruby code to evaluate.

B.1.15.2 Remarks

This macro cannot be used in safe mode.

B.1.16 `snippet`

Returns the value of a snippet.

Aliases: `&`

Example: `&[glang]`

B.1.16.1 Parameters

Parameter	Description
0	The ID of the snippet to retrieve.

B.1.17 `snippet:`

Defines a snippet.

Aliases: `&:`

Example: `&:[test|This is a test]`

B.1.17.1 Parameters

Parameter	Description
0	The ID of the new snippet.
1	The contents of the new snippet.

B.1.17.2 Remarks

The new snippet can only be used *after* its declaration.

B.2 Block Macros

B.2.1 `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.2.1.1 Parameters

Parameter	Description
0	The box title.
1	The box text.

B.2.2 `codeblock`

Used to render a block of code within `<pre>` and `<code>` tags.

Example

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

B.2.2.1 Parameters

Parameter	Description
0	The code to be formatted.

B.2.2.2 Remarks

For code highlighting, see the highlight macro.

B.2.3 `figure`

Includes an image in the document, with an optional caption (see Images and Figures).

Example

```
figure[
  graph.png|Monthly Pageviews
  @width[90%]
]
```

B.2.3.1 Parameters

Parameter	Description
0	The name of the image file (relative to the `images/` folder).
1	The image caption *(optional)*.

B.2.3.2 Attributes

Attribute	Description
*	Any attribute supported by the img tag.

B.2.4 `image`

Includes an image in the document

Example

```
img[
  holidays/landscape.jpg
  @class[photo]
        @style[border: 1px solid black;]
]
```

B.2.4.1 Parameters

Parameter	Description
0	The name of the image file (relative to the `images/` folder).

B.2.4.2 Attributes

Attribute	Description
*	Any attribute supported by the img tag.

B.2.5 `note`

Creates a note `div` containing the value.

Aliases: `important, caution, tip`

Example: `note[This is a note.]`

B.2.5.1 Parameters

Parameter	Description
0	The text of the note.

B.2.6 `pubdate`

Evaluates to a date string (in the format: *current_month current_year*; i.e. *%B %Y*), within a `<div>` tag.

Example: `pubdate[]`

B.2.7 `revision`

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

Example: `revision[]`

B.2.8 `subtitle`

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

Example: `subtitle[]`

B.2.9 `title`

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

Example: `title[]`

B.3 Inline Macros

B.3.1 `anchor`

Creates a named anchor (or bookmark).

Aliases: `bookmark, #`

Example: `#[test|Test Bookmark]`

B.3.1.1 Parameters

Parameter	Description
0	The identifier of the bookmark
1	The contents of the bookmark *(optional)*

B.3.2 `draftcomment`

If the document.draft setting is set to `true`, displays a draft comment within the document.

Aliases: `dc`

Example: `dc[This is printed only in draft documents.]`

B.3.2.1 Parameters

Parameter	Description
0	The text of the comment.

B.3.3 `fmi`

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

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

B.3.3.1 Parameters

Parameter	Description
0	The object that needs additional explanation.
0	A valid bookmark within the document.

B.3.4 `link`

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

Aliases: `=>`

Example: `=>[http://www.h3rald.com|H3RALD.com]`

B.3.4.1 Parameters

Parameter	Description
0	A valid bookmark within the document or an URL.
1	The text of the link *(optional)*.

B.3.5 `todo`

Saves the value as a TODO item, which can be printed using the todo command and included in the document if the document.draft setting is set to `true`.

Aliases: `!`

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

B.4 Filter Macros

B.4.1 `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.

Aliases: `md`

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

B.4.1.1 Parameters

Parameter	Description
0	The Markdown text to filter.

B.4.2 `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` or a `.txt` extension.

Aliases: `txt`

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

B.4.2.1 Parameters

Parameter	Description
0	The Textile text to filter.

B.4.3 `highlight`

Highlights a piece of source code (second parameter) according to the specified language (first parameter). for more information on code highligting, see Source Code.

Example

```
highlight[ruby|
  def hello
    puts "Hello World"
  end
]
```

B.5 Structure Macros

B.5.1 `article`

Used to create a simple article. By default, it includes the following macros:

- `document`
  - `head`
    - `style[default.css]`
  - `body`
    - `halftitlepage`
      - `title`
      - `pubdate`
      - `subtitle`
      - `author`

B.5.1.1 Parameters

Parameter	Description
0	The article contents.

B.5.1.2 Attributes

Attribute	Description
pre-title	Contents to include before the title macro.
post-title	Contents to include after the title macro.
head	Contents to include instead of the default `head` macro.
pubdate	Contents to include instead of the default `pubdate` macro.
halftitlepage	Contents to include instead of the default `halftitlepage` macro.

B.5.2 book

Used to create a book. By default, it includes the following macros:

- `document`
    - `head`
        - `style[default.css]`
    - `body`
        - `titlepage`
            - `title`
            - `pubdate`
            - `subtitle`
            - `revision`
            - `author`

B.5.2.1 Parameters

Parameter	Description
0	The article contents.

B.5.2.2 Attributes

Attribute	Description
pre-title	Contents to include before the title macro.
post-title	Contents to include after the title macro.
head	Contents to include instead of the default `head` macro.
pubdate	Contents to include instead of the default `pubdate` macro.
titlepage	Contents to include instead of the default `titlepage` macro.
frontmatter	Contents to include within a `frontmatter` macro.
bodymatter	Contents to include within a `bodymatter` macro.
backmatter	Contents to include within a `backmatter` macro.

B.5.3 document

Creates an `<html>` tag and a DOCTYPE declaration. Called internally by the book macro and the article macro.

B.5.3.1 Parameters

Parameter	Description
0	The document contents.

B.5.4 head

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

B.5.4.1 Parameters

Parameter	Description
0	The head contents.

B.5.5 `section`

Creates a section (`<div>` tag).

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

Example

```
section[
	@title[Test Section]
	@id[test]
...
]
```

B.5.5.1 Parameters

Parameter	Description
0	The text of the section

B.5.5.2 Attributes

Attribute	Description
title	The title of the section *(optional)*
id	The ID of the section *(optional)*
notoc	If not blank, the header will not appear in the Table of Contents. *(optional)*

B.5.6 `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.5.6.1 Parameters

Parameter	Description
0	The stylesheet file to embed.

B.5.7 `toc`

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

Example: `toc[1]`

B.5.7.1 Parameters

Parameter	Description
0	The maximum header depth of the TOC *(optional)*.

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.

Name	Description	Default (YAML)
`document.author`	The author of the document.	`""`
`document.draft`	If set to `true`, the document is considered a draft, so draft comments and todo items will be displayed.	`false`
`document.filename`	The name of the output file.	`""`
`document.output`	The format of the output file. It can be set to any value stored in the document.output_targets setting.	`html`
`document.output_targets`	An `Array` containing all the possible output formats. This setting should not be changed by the user.	`nil`
`document.revision`	The document's revision.	`""`
`document.source`	The main source file to compile. It can be also be overridden by calling the compile command with the `-s` option.	`document.glyph`
`document.subtitle`	The subtitle of the document, displayed using the subtitle macro.	`""`
`document.title`	The title of the document, displayed using the title macro.	`""`

C.2 filters.*

Name	Description	Default (YAML)
`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).	`true`
`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.	`bluecloth`
`filters.coderay.*`	Some Coderay-specific options.	`nil`
`filters.highlighter`	The current highlighter to use. It can be set to `coderay` or `ultraviolet`	`coderay`
`filters.redcloth.restrictions`	An `Array` containing restrictions applied to RedCloth, used by the textile macro (see RedCloth Documentation for more information).	`[]`
`filters.target`	The output target for filters. It can be set to `html` (for RedCloth and MarkDown) or `latex` (RedCloth-only).	`html`
`filters.ultraviolet.line_numbers`	Whether the Ultraviolet highlighter should display line numbers or not.	`true`
`filters.ultraviolet.theme`	The theme used by the Ultraviolet highlighter.	`lazy`

C.3 language.*

Name	Description	Default (YAML)
`language.set`	Determines which macro set will be loaded. It can be set to: ▪ glyph – Loads core, filter, xml macros plus all macros necessary for the document.output setting. ▪ xml – Loads core and xml macros. ▪ core – Loads core macros only.	`glyph`
`language.options.xml_blacklist`	The XML tags listed here cannot be generated using Glyph code.	`- applet - base - basefont - embed - frame - frameset - iframe - isindex - meta - noframes - noscript - object - param - title`
`language.options.xml_fallback`	If set to true, any unknown macro name will considered an XML element (see Other HTML Elements).	`true`

C.4 tools.*

Name	Description	Default (YAML)
`tools.pdf_generator`	The external program used to generate PDF files. It can only be set to `prince`.	`prince`

Appendix D – Changelog

D.1 v0.3.0 – June 13th 2010

D.1.1 27 Features Implemented

ID	Description
#39	A new outline command is available to display the document outline.
#110	It is now possible to use Glyph language to produce arbitrary XML code.
#111	System settings are now stored within a `system.*` namespace and cannot be changed via the config: macro or the config command.
#116	It is now possible to use named attributes within Glyph macros.
#119	A new parser was implemented from scratch to improve performance. Treetop gem no longer required.
#121	Some macros have been removed in favor of XML fallback, others have been updated.
#123	The SyntaxNode class has been specialized to differentiate between macros, attributes, parameters, text and escapes.
#124	Implemented new article macro and book macro.
#126	A new rewrite: macro has been implemented to create simple macros using just Glyph code.
#127	A new alias macro has been implemented to create macro aliases.
#128	A blacklist for XML tags has been exposed via the language.options.xml_blacklist setting.
#129	The include macro can now be used in lite mode, it can evaluate ruby files and requires relative paths.
#130	A new "safe mode" has been implemented to explicitly forbid certain potentially unsafe macros.

D.1.2 7 Bugs Fixed

ID	Description
#109	Performance has been dramatically improved by implementing a parser from scratch (see #119)
#122	Macro encoding/decoding no longer necessary due to the new parser (see #119)
#125	Warning messages have been streamlined.

D.2 v0.2.0 – May 9th 2010

D.2.1 23 Features Implemented

ID	Description
#62	A new highlight macro is available to highlight source code (CodeRay or UltraViolet required).
#76	It is now possible to use Glyph programmatically via the new `Glyph#filter` and `Glyph#compile` methods.
#87	It is now possible to define snippets inside a Glyph source file using the snippet: macro.
#88	It is now possible to change configuration settings inside a Glyph source file using the config: macro (Jabbslad).
#89	It is now possible to compile a single Glyph source file without creating a Glyph project.
#92	6 new macros have been defined to allow conditional processing (condition macro, eq macro, not macro, and macro, or macro, match macro)
#94	It is now possible to add *validators* to macros, for example to check the number of parameters they take.
#97	The compile command command can now take an extra `--auto` switch to trigger document auto-regeneration whenever a source file is changed (koraktor).
#99	Added a `document.draft` setting. If set to `true`, comments and TODOs are rendered in output files.
#100	Glyph CSS files are no longer copied to new projects, but they can be referenced as if they were (see also #93).
#108	It is now possible to define Glyph macros within Glyph source files using the macro: macro.

D.2.2 17 Bugs Fixed

ID	Description
#86	Warning and error messages have been updated, and it is now possible to show additional debug information. Additionally, syntax errors are now handled before the document is processed.
#93	Default css files were not copied when creating a new project. The issue has been resolved by allowing the style macro to reference Glyph's system styles (see also #100).
#95	The config command did not save data to YAML configuration files. This has been fixed ensuring that internal configuration overrides are not saved to the YAML file too.
#98	Glyph is now fully compatible with Ruby 1.9.1 and JRuby 1.4.0.
#101	Additional tests have been developed to improve Textile support. There should no longer be errors when using textile block elements inside Glyph macros.
#103	Fixed a bug that caused test failures when deleting the test project directory.
#104	Nested Glyph macros calling `Macro#interpret` no longer ignore escape delimiters.
#107	Added the possibility to encode (using the `encode` macro) and decode (using the `decode` macro) macros so that they can be interpreted later.

D.3 v0.1.0 – April 8th 2010

Initial release.