Contents

Chapter 1

Introduction

In this thesis we will be introducing a new markup language that tries to improve upon a few others, namely LaTeX and Markdown, whilst still holding on to their advantages. Before we get into the details of this new language it is necessary to introduce some of the currently most used markup languages and word processors. This introduction will provide the context from which the necessity of this thesis was sparked.

1.1 State of the art

1.1.1 Markdown

One of the most popular markup languages at this point in time is Markdown [2]. It is introduced by the designers of the language as follows:

```
Markdown is a text-to-HTML conversion tool for web writers. Markdown allows you
to write using an easy-to-read, easy-to-write plain text format, then convert it
to structurally valid XHTML (or HTML).
```

Markdown's goal is to be easy to write and read as a plain text document and as such is actually based on plain text emails. Due to this, it attracts a lot of people's attention as it is very easy to create a new simple document, such as a short report. It achieves this by introducing a small and simple syntax [3]. The syntax they chose also feels very natural. For example, to create a

title you could underline it with - or = and to create an enumeration, you can use *'s instead of bullet points.

By using such a simple syntax compile times for the documents are very short and IDE support is wide. Even without the help of an IDE, Markdown is very readable as it was designed to be. Another advantage of this simple syntax is that it almost entirely removes the possibility of syntax errors to occur.

Due to it's simple nature and limited syntax Markdown can be easily translated into other formats such as PDF or HTML. A tool to convert Markdown to HTML is offered by the designers of the language itself, as stated above. This further increases the appeal of the language, as it allows to very easily create elegant looking documents in a variety of formats.

As Markdown files are just plain text files they are robust to file corruption and easy to recover from said corruption. A corrupted byte would show easily and it would likely not destroy the layout of the document as there are so few reserved symbols being used in a Markdown document.

Due to the fact that Markdown uses plain text, it is cross-platform and because it has such a simple syntax, compilers are available for all major operating systems. As a matter of fact, the Perl script offered by the Markdown designers, is cross-platform. Markdown also natively allows for UTF-8 characters to be used in the text thus not requiring weird escape sequences to insert certain symbols. A last advantage of plain text is that it is well suited for Version Control Systems (VCS) such as Git and Mercury.

To illustrate the simplicity of Markdown an example document is shown below:

Listing 1.1: document.md

```
# Chapter 1
*Markdown* allows you to write simple
    documents very fast.

## Chapter 2
Markdown is:

* Simple
* Easily readable
* Easily writable
```

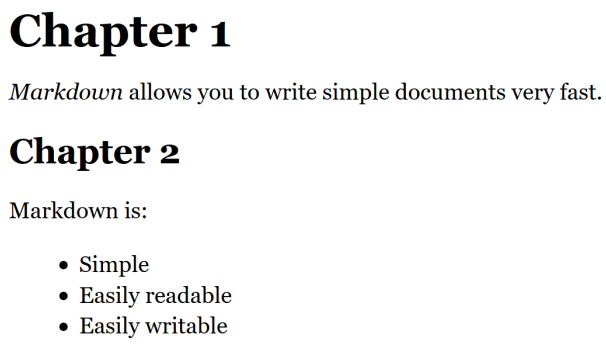


Figure 1.1: The document rendered as HTML by the official markdown conversion tool

To allow for some customisability, Markdown allows you to use inline HTML. In case Markdown does not support what you want to do, HTML is what you use. Tables are not a part of default Markdown (though there are variations that have syntax for them, like Github Markdown), so an example of how to create one using HTML is shown below:

```
Below you can find an HTML table
<table border="1">
    <tr>
        <td>
            Element 1
        </td>
        <td>
            Element 2
         </td>
     </tr>
</table>
```

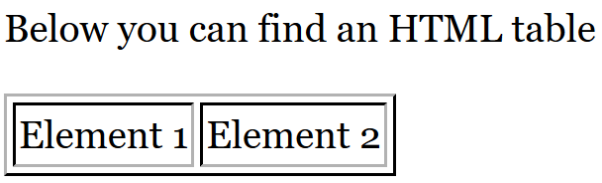


Figure 1.2: The table document rendered as HTML by the official markdown conversion tool

1.1.2 LaTeX

LaTeX provides a, Turing complete, programming model and due to this allows for rich customisability. It also provides a packaging system that allows for libraries, called packages, to be created using this programming model. Over the course of the past 38 years (the initial TeX release was in 1978) a lot of packages have been created for all kinds of different functionality. This is one of the reasons why LaTeX is often used for long and complex documents such as books, articles, scientific papers and syllabi.

Like Markdown lat uses a plain text format that is quite robust to file corruptions and is well suited for VCS. It is however less robust against file corruption as the syntax is a lot more

extensive and thus file corruption could lead to your document not compiling any more a lot easier. It also doesn't use UTF-8 making it harder to type more exotic characters such as . Later iterations of TeX, such as XeTex and LuaTex do support UTF-8 though.

In LaTeX the structure and the layout of your document are split by design. LaTeX forces you to create a structured document by requiring you to use `sections, paragraphs,...`. When using a WYSIWYG word processor for example, usually there is a way to structure your documents, using `styles, headers,...` but it is not forced on the user. As a result, in LaTeX consistency is enforced throughout the entire document without too much effort, while in aforementioned WYSIWYG word processors this is often forgotten and less convenient to use. In a WYSIWG word processor, the user thus has to make sure his document is consistent. Once documents get longer, this gets very hard to manage and to spot the inconsistencies.

Next to this customisability, LaTeX's typography is also very sophisticated. It has support for kerning and ligatures and uses an advanced line-breaking algorithm, the Knuth-Plass line-breaking algorithm [6]. The algorithm sees a paragraph as a whole instead of using a more naive approach and seeing each line individually. Kerning places letters closer together or further away depending on character combinations. A ligature is when multiple characters are joined into one glyph e.g fi vs. `fi`.

Another reason why LaTeX is widely used due to its good scientific support. LaTeX and its libraries offer good support for mathematics as well as other scientific areas, e.g. you can create complex good looking graphs using LaTeX. Due to its high customisability, LaTeX can handle about any kind of document you could think of, from sheet music and graphs, to letters and books. It should be noted though that almost all of this functionality is offered through libraries such as PGF/TikZ [7].

LaTeX also works together with BibTeX, a reference management software that is very robust and makes certain all of your references are consistent. It also allows you to create the references outside of your main document, decreasing the amount of clutter in the document. Whilst on the topic of including other files in your document, in LaTeX you can split up your document into multiple smaller documents, allowing for easier management. A common use of this function is to write every chapter, e.g. in a book, in a separate document and than all the chapters are imported into the main document. This allows to easily remove or switch out different parts of

a document.

Another very simple feature that LaTeX has, is that is allows you to use vector graphics or PDF as images in your document. This is something not possible in WYSIWIG word processors or lightweight markup languages such as Markdown.

LaTeX is free and open source, which is a huge point of attraction for the open source community. As a result of this free and open source model, combined with its popularity and good scientific support, LaTeX has been integrated in many other applications. A few examples are:

- MediaWiki [4]
- Stack Exchange
- JMathTex [1]

1.1.3 Pandoc

Pandoc [5] is a document converter, but in reality it can do much more than just convert a document from one format to another. Notably, it allows you to write inline LaTeX inside of a Markdown document allowing for a much better experience than just using one of them. However, Pandoc does not really have a programming model, outside of the LaTeX one, it works with so called filters. When a document is issued for conversion, Pandoc will transform the document into an Abstract Syntax Tree. This AST can then be transformed using filters, the AST enters a filter, is transformed and is then past on to the next filter. Filters can be written in a multitude of languages such as Haskell, Python, Perl,... and the filters to be used are passed as a command line argument to the conversion command.

1.1.4 Word processors

Modern word processors such as Microsoft Word and Pages are so called WYSIWYG editors. As you immediately, without compiling, see what your document will look like it is a very popular tool. The fact that you immediately see what you are writing also significantly decreases the complexity of the tool and it does not scare away less technical people the way LaTeX does.

There is also no possibility to use commands as in LaTeX or shorthand syntax as in Markdown, instead buttons in the GUI offer all of the functionality. Structure is not forced onto the user which offers less restrictions allowing the user to do what he wants. It is also very easy to change a font type or change the font size of a certain part of the text. Another feature that is loved in the WYSIWIG community is that you can easily add an image and drag and drop it around.

1.2 Problem statement

We can see that these solutions all have there strong points but there are also quite a few gripes. We will discuss the most common gripes for every solution underneath and afterwards introduce the problem statement.

1.2.1 Markdown

To achieve Markdown's simplicity, a hefty price has to be paid. All of the syntax elements have been hard coded into the language meaning there is no room for customisability. The only way to customise Markdown is by embedding inline HTML. HTML allows users to create a solid structured model but it still does not allow for very much customisability. While Markdown is meant to be easily readable and writeable, HTML is not and using it thus decreases readability and cleanliness of your document significantly. Once you use HTML in an Markdown document, you also require a certain technical proficiency for the source document to be read or edited. HTML also does away with the robustness of Markdown's plain text format, as having a tag being corrupted could lead to very big changes in the rendered file and might not even allow for compilation anymore. HTML also reintroduces the possibility of syntax errors and thus slows down the document creation.

The lack of customisability in Markdown is further enforced due to the fact that it has no programming model. It is one thing that the syntactic shorthands can not be reused, but there is also no other way, next to HTML to create an entirely new entity.

1.2.2 LaTeX

Even though LaTeX offers customisability through a programming model, it is not perfect. The programming model is hard to use and is not easy to read. Due to this programming model it also looks a lot more complex than Markdown, scaring away a lot of potential users. There are a few ways to counter this complex look though, for one there is LyX. This is a GUI around LaTeX offering some of the most common WYSIWG features, while still utilizing LaTeX's strength. Even though the complexity is now hidden under a GUI, it is still there and error messages are as bad as they were before, as is discussed beneath.

Having a cumbersome programming model, LaTeX makes it easy to create syntax errors. The robustness of a plain text document is also diminished, as your document will probably not even compile once an error occurs in one of the documents. Something that is also very common in LaTeX is that the shown error messages do not offer correct information and are often unclear. See the excerpt below for an example of misleading error messages.

```
documentclass{article}
begin{document}
begin{equation}
$a$
end{equation}
end{document}
```

```
line 4: Display math should end with $$
    .<to be read again>a $a
line 5: You can't use `\eqno' in math
    mode.\endequation ->\eqno\hbox {\
    @eqnnum }$$\@ignoretrue \end{
    equation}
line 6: Missing $ inserted.<inserted
    text>$ \end{equation}
```

What it is actually trying to tell you is that you are not allowed to use $ inside of an equation, but that is not clear from the error messages.

The programming model used in LaTeX does not use a static type system, which means that we have to compile the document to be able to see if we passed a wrong type to a macro at some point in the document. It also does not work the way we are used to in imperative and functional languages. It does not have functions that receive parameters and that then call more functions. Instead it has macro's and uses macroexpansion. As an example of how this translates into a real world example, a function the implementation of **foreach** in PGF/TikZ is shown below.

```
\def\pgffor@foreach{%
\pgffor@atbeginforeach%
\let\pgffor@assign@before@code=\pgfutil@empty%
\let\pgffor@assign@after@code=\pgfutil@empty%
\let\pgffor@assign@once@code=\pgfutil@empty%
\let\pgffor@remember@code=\pgfutil@empty%
\let\pgffor@remember@once@code=\pgfutil@empty%
```

```
\pgffor@alphabeticsequencefalse%
\pgffor@contextfalse%
%
\let\pgffor@var=\pgfutil@empty
%
\pgffor@vars%
}
```

Not only is this nearly unreadable and very complicated for such a common concept in programming, it also uses a ton of other functions defined in PGF/TikZ just to be able to define a foreach loop.

Luckily the application of a function is somewhat easier as shown below. The macro, called `loopfunction`, loops from 1 to a parameter #1 and prints out a section in every iteration.

```
\def\loopfunction#1{%
  \foreach \index in {1, ..., #1} {%
    \section*{Section \index}%
  }
}
```

1.2.3 Pandoc

Transforming an AST afterwards is not always as easy as doing so inline, and it also hides what really happens to the document, which could cause confusion.

1.2.4 Word processors

A first point to note is that the most popular word processors such as Microsoft Word and Pages are not free, but instead cost a significant amount. There are free variants, such as LibreOffice and OpenOffice, but the compatibility between these word processors is not perfect. The free alternatives usually also have less of a focus on the UI, making them look somewhat less sleek.

Another problem with word processors is that their format often changes over the years. This means that your document might not show up the same, or at all, a few years later than when you wrote it.

Even though some customisability is available, no programming model is available and real customisability is far to be found. Due to their lack of a programming model, word processors do not allow for even the simplest of computations such as `The result is x + y`.

It is also quite easy to create inconsistencies using word processors as noted above in Subsection 1.1.2. Next to creating inconsistencies, as structure is not enforced on you, small changes could have big side effects on the document. E.g. you placed an image exactly like you wanted, you then later on add a newline somewhere higher up, reflowing the text underneath it an shifting the precisely placed image around. These changes are not always immediately clear, which further contributes to the creation of inconsistencies in the document.

Lastly, word processors often use a proprietary or complex format for there documents, which is not only bigger than a plain text document, but also a lot more prune to corruption. A single bit flip could potentially irreversibly destroy your document. These complex document formats are also hard to be used in combination with VCS.

1.3 Proposed solution: Neio

To counter the problems occurring in the state of the art solutions, we bring forth a new markup language called `Neio` (read as neo). To be able to solve some of the aforementioned problems and still make use of all of their advantages, the following goals were devised for the Neio markup language:

1. User-friendliness: has to be easy to get started with the language
2. Has to use a modern programming paradigm
3. Has to be highly customisable

The first goal supplies us with a simple language that allows to very easily create simple documents. The use of a modern programming paradigm allows us to customise the document and execute easy and complex computations easily in effect achieving the third goal.

To achieve this we chose to use a syntax like Markdown to write our documents in as it is the easiest to read and write. For the programming paradigm we chose for something much like Java as Java is very well known, and working with macro expansion's or filters is not ideal. Lastly, we choose to translate to LaTeX as it has been successively used for decades and has proven by now that it can deliver very clean looking documents. Later on more back ends, such as HTML, could be implemented, but that is outside of scope of this thesis.

1.3.1 Target group

Neio targets anyone that is currently using Markdown but just wants to be able to customise their document a bit more. As we will see later, in Neio it is easy to for example create two images and put them next to each other, something that is not possible in markdown.

It can also be used by people without technical expertise if templates are provided. E.g. given a template for a letter, even a non-technical person is able to quickly write down the contents of the letter and fill in the template variables.

Neio also target package developers and programmers in general as it offers a modern programming model that is a lot well known and much easier than TeX's system. This includes full-time developers as well as students or professors.

Chapter 2

Design of the Neio markup language

In the previous chapter, Chapter 1, the state of the art concerning document creation has been discussed and we notice that some improvement is certainly possible. All of the document creation tools discussed above have their advantages and disadvantages. In this chapter we will present the Neio language and we will explain how our decisions were reached and how they were affected by the state of the art solutions.

2.1 Neio document

To illustrate some of the basic concepts we will now present an example. The typical files that a user will write are called Neio documents. Neio documents are heavily based on Markdown as Markdown is so easy to read and write. Since this document is what most of our consumers will be seeing, this was a very important property. Since our consumers will mostly write Neio documents, we are also able to use the widespread knowledge of Markdown to increase the size of our audience and allow users to instantly transition to Neio. The following example is one of the most basic documents you can write using Neio.

```
[Document]

# Chapter 1
This is some text in the first Paragraph
    .
```

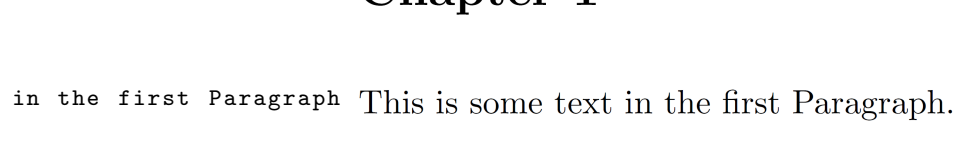


Figure 2.1: The rendered document

Even if you are not familiar with Markdown, you can probably immediately tell what this document represents. There is however a very significant difference with Markdown. In markdown the syntax is hard-coded into the language, a # will always represent a chapter for example. In Neio, on the other hand, the language has no knowledge whatsoever of what # means. In fact, it does not know what anything in a Neio document means. It does not know about these things because everything is a method call, chained together to form an object model that represents the document. The reason for doing so is simple, we want to be able to customise our documents! For example, when you are creating a slide show you might not want # to create a new chapter, instead it would be better and clearer if it would just create a new slide.

Knowing this we can now represent the document as a chain of method calls as follows:

```
new Document()
    .#("Chapter 1")
    .text("This is some text in the first Paragraph");
```

Now is also a good time to explain a small syntactical difference with Markdown, which you might have already noticed. Every Neio document has to begin with a so-called document class. This is a concept borrowed from LaTeX, and just as in LaTeX it tells us what kind of document the user is trying to build. We say that a Neio document has to start with a document class, but this is not completely correct. You are allowed to add single- or multi-line Java style comments before the document, or anywhere else in the document for a matter of fact. The following is thus also a valid document.

```
// A single-line comment
[Document]
/*
 * Multi-line comments are also available!
 */
```

To further understand this example we need to introduce the second kind of files available in the Neio markup language, Class files.

2.2 Class files

Class files are very similar to class files in Java. They are the thriving force behind the Neio documents, they define all the object and methods that are used in a Neio document. As means of an introduction we will have a look at the Document class that was used as a document class in the previous section.

Listing 2.1: Document.no

```
1  namespace neio.stdlib;
2
3  import neio.lang.Text;
4
5  /**
6   * Represents the most basic of Documents
7   */
8  class Document extends TextContainer;
9
10 private Packages packages;
11 private Bibtex bibtex;
12
13 /**
14  * Creates a Document and adds common LaTeX packages to it
15  */
16 Document() {
17     packages = new Packages();
18     packages.add("a4wide")
19     .add("graphicx")
20     .add("amsmath")
21     .add("float");
22
23     bibtex = null;
24 }
25
26 /**
27  * Creates a Chapter and adds it to this
28  *
29  * @param title The title to use for the new Chapter
30  * @return        The newly created Chapter
31  */
32 Chapter #(Text title) {
33     Chapter chapter = new Chapter(title, 1);
34     addContent(chapter);
35
36     return chapter;
37 }
38
39 /**
40  * Returns a list of Package's
41  *
42  * @return a list of used LaTeX packages
43  */
44 Packages packages() {
45     return packages;
46 }
47
48 /**
49  * Adds a LaTeX package
50  *
```

```
51  * @param pkg The package to add
52  * @return        The list of currently included packages
53  */
54 Packages addPackage(String pkg) {
55     return packages.add(pkg);
56 }
57
58 /**
59  * Sets a BibTeX file for this Document
60  *
61  * @param The name of the BibTeX file
62  */
63 void addBibtex(String name) {
64     this.bibtex = new Bibtex(name);
65 }
66
67 /**
68  * A proxy method for {@code BibTeX}'s {@code cite}
69  * Returns a Cite that is linked to {@code key}
70  *
71  * @param key The key to access the citation
72  * @return        The Cite corresponding to {@code key}
73  */
74 Citation cite(String key) {
75     if (bibtex != null) {
76         return bibtex.cite(key);
77     } else {
78         return null;
79     }
80 }
81
82 /**
83  * Creates the LaTeX preamble
84  *
85  * @return The LaTeX preamble
86  */
87 String header() {
88     StringBuilder tex = new StringBuilder("\\documentclass{article}\n");
89     for (int i = 0; i < packages.size(); i = i + 1) {
90         tex.append(packages.get(i).toTex()).append("\n");
91     }
92
93     tex.append("\\setlength{\\parindent}{0em}\n")
94          .append("\\setlength{\\parskip}{1em}\n");
95
96      return tex.toString();
97 }
98
99 /**
100  * Creates a TeX representation of this object and its children
101  *
102  * @return The TeX representation of this object and its children
103  */
104 String toTex() {
105     StringBuilder result = new StringBuilder(header()).append("\\begin{document
        }\n")
106     .append(super.toTex());
107     if (bibtex != null) {
108         result.append(bibtex.toTex());
109     }
110
111     result.append("\n\\end{document}\n");
112
113     return result.toString();
114 }
```

Except for some variations on the syntax, such as using `namespace` instead of `package`, and the lack of access level modifiers, this is valid Java code. The access level modifiers are automatically set to `private` for members and `public` for methods. They do not have to be written explicitly as the default value is usually what the developer wants and it makes the class file just a little clearer and more concise.

Again though, I stand corrected, this would be valid Java code if not for the # method. In Neio class files, you are allowed to use symbols as a method name. This is a feature that is necessary to allow a Neio document to be represented as a chain of method calls as otherwise you would have to actually write out a document as series of method calls, or you would not be able to use symbols as syntactic shorthand. All the symbols you can use for method names at this time are TODO. To keep the language simple and to maximize reusability (as well as making the parsing somewhat easier) some functionality from the Java language was dropped. It is for example not possible to create anonymous classes as this goes against the concept of reusability. It is also not possible to create multiple classes in a single file.

2.3 Newlines

Neio is newline sensitive, in the sense that newlines too are methods. The superclass `TextContainer` of `Document` and `Chapter` defines the following method.

```
1 /**
2  * Handles newlines
3  *
4  * @return returns a new ContainerNLHandler
5  */
6 ContainerNLHandler newline() {
7     return new ContainerNLHandler(this);
8 }
```

The `ContainerNLHandler` will then further handle the flow of the document. The code for `ContainerNLHandler` and `NLHandler` are added below.

Listing 2.2: ContainerNLHandler.no

```
1 namespace neio.stdlib;
2
3 import neio.lang.Content;
4 import neio.lang.Text;
5
6 /**
```

```
 * Handles newlines that follow a TextContainer
 */
class ContainerNLHandler extends NLHandler;

/**
 * Creates a ContainerNLHandler and sets the TextContainer that created it as
     parent
 */
ContainerNLHandler(TextContainer parent) {
    super(parent);
}

/**
 * Returns this
 *
 * @return this
 */
ContainerNLHandler newline() {
    return this;
}
```

Listing 2.3: NLHandler.no

```
namespace neio.stdlib;

import neio.lang.Content;
import neio.lang.Text;

/**
 * The default newline handler
 */
class NLHandler;

Content parent;

/**
 * Creates a new NLHandler and keeps a reference to the object that created it
 */
NLHandler(Content parent) {
    this.parent = parent;
}

/**
 * Returns the object that created this
 *
 * @return The object that created this
 */
Content parent() {
    return parent;
}

/**
 * Creates a Paragraph and adds it to the parent.
 *
 * @param text The text for the Paragraph
 * @return        The newly created Paragraph
 */
Paragraph text(Text text) {
    new Paragraph(text) par;
    parent().addContent(par);
    return par;
}

/**
```

```
42  * Returns this
43  *
44  * @return this
45  */
46 NLHandler newline() {
47     return this;
48 }
```

Our earlier example now translates to the following:

```
1 new Document()
2     .newline()
3     .newline()
4     .#("Chapter 1")
5     .newline()
6     .text("This is some text in the first Paragraph");
```

We needed to do this because a newline in a plain text document has meaning, depending on the context it can mean something else entirely. For example, a single newline in a paragraph should append the next sentence to the paragraph. But on the other hand when we separate two text blocks with an empty line (thus two newlines), the blocks of text are seen as two different paragraphs.

It should also be noted that any text written in a Neio document gets translated to the `text` call as seen in the example above. An implementation of the text method is seen `NLHandler`. The reason for this will become clear later on.

We will see more advanced examples using newline in Chapter 3 that will further expose its benefits. To avoid clutter, the `newline` method will not be shown in the further examples unless it is needed to understand the example.

2.4 Language features

2.4.1 Context types

As said before, everything is a method call, but to be able to just chain any method to any other method, regular methods do not suffice. We want to be able to call methods of different class files whilst constructing our document, but we do not want to specify what object we're calling the method on. This is something that should be clear from the current structure of the file, a previously created object should have a method with the correct signature.

To be able to call the right methods, we need to introduce something called ContextTypes. A ContextType combines an object, the actual type, with its context. We will illustrate this using an example.

```
1 [Document]
```

As we saw before this translates to the new call `new Document()`. This is actually a ContextType of the newly created `Document` and `null` as there is nothing else yet. We'll call this ContextType `ct1`.

Lets add a chapter to it, when there are two elements, there should be some kind of context.

```
1 [Document]
2
3 #Chapter 1
```

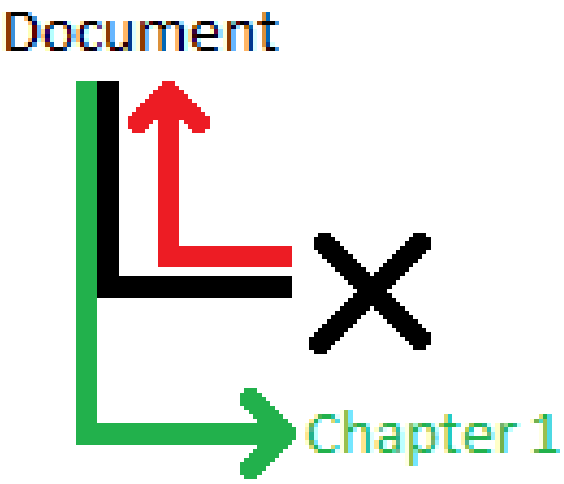
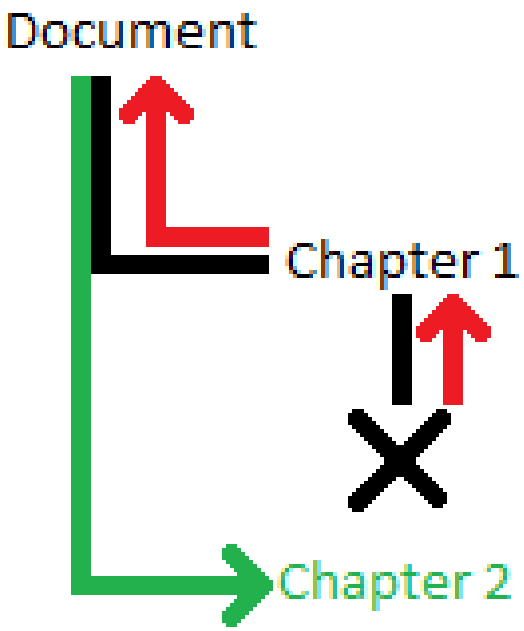


Figure 2.2: As `Document` contains the # method, we call # on `Document` and a `Chapter` is added to the `Document` , as represented by the green arrow.

After this call our document looks as follows `new Document().#("Chapter 1")`. By adding a second element we created are able to see a ContextType in action. To know where to find the # method, we will recursively iterate over our context. First we check the actual type of `ct1, if we could not find the method there, we recursively proceed to check the context of ct1`. As you can see above in the source code of `Document`, there is indeed a method # in there, so there is no need to look any further. The current document translates to ContextType `ct2, which contains the newly created Chapter and has ct1` as context.

We continue by adding one more Chapter.

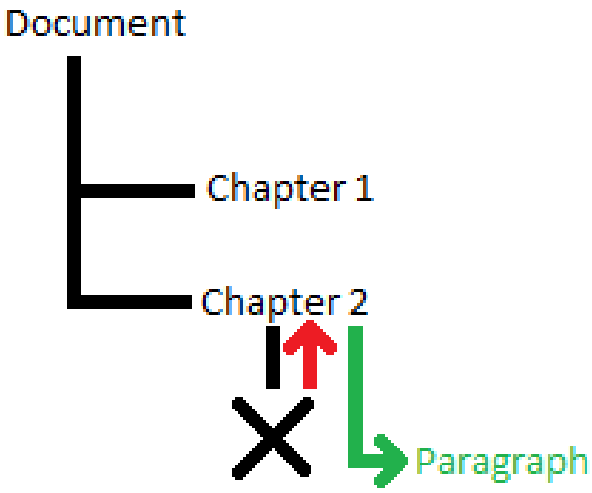
```
1 [Document]
2
3 #Chapter 2
```



Now when we look for the # method in `ct2` we first have to check Chapter, but Chapter does not have this method. We thus recursively check the context of `ct2, which is ct1`. `ct1`'s actual type is a Document, of which we know that it has a # method. As we have found the method we were looking for, # will be called on the Document in `ct1`. The ContextType that is created for this call is `ct3` that has the newly created Chapter as the actual type, but that has **`ct1`** as context. This is because we have had to skip over `Chapter 1, which shows us that ct2` is not useful anymore for the continued creation of the document.

Lastly we add a paragraph to the document.

```
1 [Document]
2
3 This is a paragraph.
```



We check the actual type of `ct3` and immediately have a matching method, we thus call the method on `Chapter 2`. The newly created ContextType `ct4, contains the Paragraph as actual type and ct3` as context.

The following structure is what we are left with.

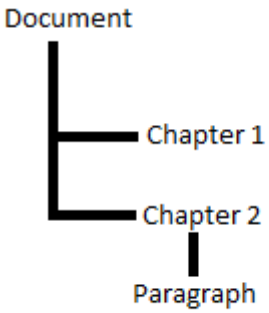


Figure 2.3: The final structure built in the example

2.4.2 Nested methods

Documents often have recursive elements such as sections or enumerations, as in Neio, everything is a method every one of these levels would have to be defined as a separate method. That means creating a method for #, ##, ###,... . This is of course very cumbersome, even impossible at times, and we would like to only define it once as the behaviour should be mostly the same no

matter how many symbols we use. Usually only one property, such as font size, is effected by the number of symbols used in such a method call. This is why nested methods were invented. A method can be annotated with a `nested` modifier which means that this method implicitly takes an extra argument, an `Integer` that reflects the depth of this recursive method. The depth of this recursive method is just the number of times the symbol has been used. If we have a look at the `Chapter` class, we see that it has a nested method #.

Listing 2.4: Chapter.no

```
1  namespace neio.stdlib;
2
3  import neio.lang.Content;
4  import neio.lang.Referable;
5  import neio.lang.Reference;
6  import neio.lang.Text;
7
8  /**
9   * Represents a chapter or a section
10  */
11 class Chapter extends TextContainer implements Referable;
12
13 protected Text title;
14 protected String marker;
15 protected Integer level;
16 protected String texName;
17
18 /**
19  * Creates a new Chapter
20  *
21  * @param title  The title of the Chapter
22  * @param level The level of nesting of the Chapter
23  */
24 Chapter(Text title, Integer level) {
25     this.title = title;
26     this.level = level;
27     this.texName = "section*";
28 }
29
30 /**
31  * Creates a new Chapter and adds it to this or
32  * or to the nearest other TextContainer above this if
33  * the level of the new Chapter and this are is lower or equal
34  * to the level of this.
35  * This is a nested call thus only ##+ will match
36  *
37  * @param title  The title of the new Chapter
38  * @param level The level of nesting of the new Chapter
39  *
40  */
41 nested Chapter #(Text title, Integer level) {
42     if (level <= this.level) {
43         Chapter c = nearestAncestor(Chapter.class);
44         if (c != null) {
45             return c.hash(title, level);
46         } else {
47             return nearestAncestor(Document.class).hash(title);
48         }
49     }
50     Chapter chapter = new Chapter(title, level);
```

```
51      addContent(chapter);
52
53      return chapter;
54  }
55
56
57  /**
58   * Returns the title
59   *
60   * @return Title
61   */
62  Text title() {
63      return title;
64  }
65
66  /**
67   * Creater a TeX representation of this and its children
68   *
69   * @return The TeX representation
70   */
71  String toTex() {
72      return toTex(level) + super.toTex();
73  }
74
75  /**
76   * Returns the TeX implementation of this Chapter and adds labels
77   * it to be able to reference to this Chapter later on.
78   *
79   * @param lvl The level of nesting of this Chapter
80   * @return      A TeX string representing this
81   */
82  String toTex(Integer lvl) {
83      String result = "\\" + subs(lvl) + texName + "{" + title.toTex() + "}\n";
84      return result + "\\label{" + hashCode() + "}";
85  }
86
87  /**
88   * Returns a string containing the right amount of "sub"'s to create a valid
89   * LaTeX section.
90   *
91   * @param lvl The level of nesting of this Chapter
92   * @return      The string containing all the needed "sub"'s
93   */
94  String subs(Integer lvl) {
95      String subs = "";
96      for (int i = 1; i < lvl; i = i + 1) {
97          subs = subs + "sub";
98      }
99
100     return subs;
101 }
102
103 /**
104  * Creates a reference to this Chapter and adds a correct prefix to it
105  * such as Chapter, Section or Subsection.
106  *
107  * @return A Reference to this Chapter
108  */
109 Reference ref() {
110     String prefix = "";
111     if (level < 2) {
112         prefix = "Chapter";
113     } else if (level == 2) {
114          prefix = "Section";
115     } else {
116         prefix = "Subsection";
```

```
117     }
118
119     return new Reference(prefix, "" + hashCode());
120 }
```

If we would now call `chapter.##("Chapter .1.1")` for example, it would translate to the following call: `chapter.#("Chapter 1.1", 2)`.

An important thing to note is that the regex matching a nested function is `..+` where `.` is the symbol that is being used as a method name. This means that the method name that contains a single symbol should be defined elsewhere. This has been done because the first level is usually special. In case of an Enumerate for example, when we create the first EnumerateItem, an Enumerate has to be added. Usually the object created by this method should also be added in a different place than the other objects.

2.4.3 Surround methods

A method could also be annotated with a `surround` modifier. This means that the method name surrounds a piece of text, this can be used to for example create bold text by surrounding it with stars. This is needed to be able to easily annotate text and it also looks very natural.

2.4.4 Code blocks

We've seen that we can use code in Neio class files and customise our document this way. But this doesn't allow us to do everything we have to. We don't want to create a new kind of documentclass every time we want to do something special. It's also not possible to add anything to a documentclass that does not exist. For this reason, it is possible to add code blocks in a Neio document. The three different kinds of code blocks are explained below.

Scoped code

Non scoped code

Inline code

2.5 Considerations

2.5.1 Static typing

2.5.2 Security

2.6 Reuse of current possibilities

2.6.1 Binding to LaTeX

Chapter 3

Supported document types and libraries

This chapter will go into some more details about what document types can be handled by Neio, and how to specifically build these kind of documents. We will also discuss which libraries have been recreated in Neio to allow for a wide use of the language.

3.1 Document types

3.1.1 Report

3.1.2 Letter

3.1.3 Book

A prime subject of writing a book is this document itself. The entirety of this book has been created using Neio script.

3.1.4 Article

3.1.5 Slides

3.2 Libraries

3.2.1 TikZ

3.2.2 Beamer

3.2.3 LaTeX math and amsmath

3.2.4 LaTeX tables

3.2.5 Lilypond

Chapter 4

Implementation

4.1 Used libraries and frameworks

4.1.1 ANTLR4

4.1.2 Chameleon

4.2 Compile flow

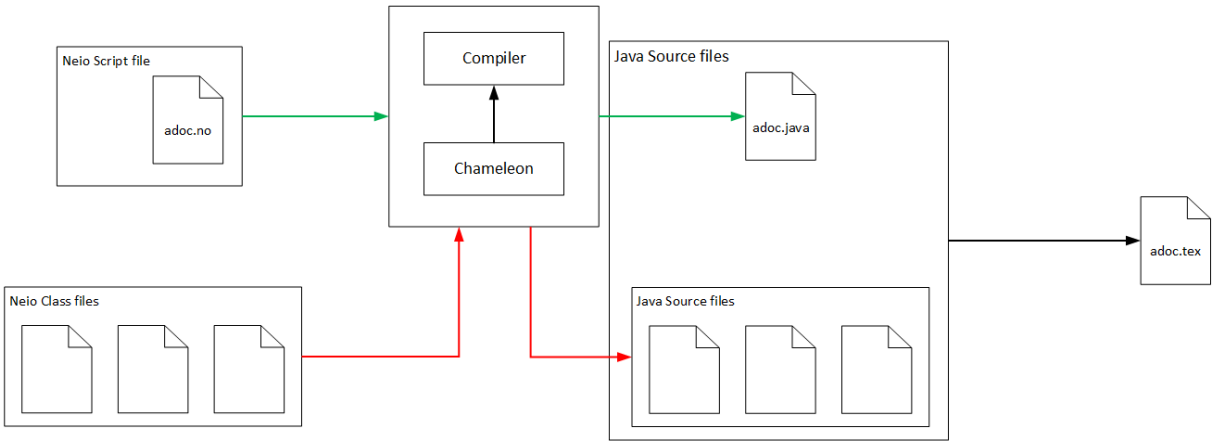


Figure 4.1: Illustration of how a Neio document is compiled. Neio class files translate to Java classes, Neio script files to a Java file with a main function. This then gets output to Tex using Neio's standard library

In Figure 4.1 the flow that a source document goes through before reaching a rendered state is depicted.

4.3 Translation

4.3.1 Escaping

4.3.2 Fluent Interface

4.3.3 Reflection

4.3.4 Limitations

4.3.5 Reasons for choosing Java

4.3.6 Automatic Text conversion

4.4 Outputting

Chapter 5

Future work

5.1 Static typing of the language

5.2 Package support

5.3 Implementation of packages

5.4 Compiler improvement

5.4.1 Correct stages

5.4.2 Other front- and backends

5.5 Tool improvement

5.5.1 Syntax highlighting

5.5.2 Auto completion

Bibliography

[1] Jmathtex. `http://jmathtex.sourceforge.net/`. Accessed: 08-05-2016.

[2] Markdown. `https://daringfireball.net/projects/markdown/`. Accessed: 07-05-2016.

[3] Markdown syntax. `https://daringfireball.net/projects/markdown/syntax`. Accessed: 08-05-2016.

[4] Mediawiki latex extension. `https://www.mediawiki.org/wiki/Extension:Math`. Accessed: 08-05-2016.

[5] Pandoc. `http://pandoc.org/`. Accessed: 08-05-2016.

[6] Donald E. Knuth and Michael F. Plass. Breaking Paragraphs into Lines. *Software - Practice and Experience*, 11:1119–1184, 1981.

[7] Till Tantau. *The TikZ and PGF Packages.*