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. 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 time is Markdown. 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. Due to this, it attracts a lot of people as it is very easy to create a new simple document. It achieves this by introducing a small and simple syntax. By using such a simple syntax compile times are 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. 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 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. 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 suite for Version Control Systems (VCS).

To illustrate the simplicity of Markdown we'll take a look at an example document:

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

## Chapter 2
Markdown is:
* Simple
* Easily readable
* Easily writable
```

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.

1.1.2 LaTeX

LaTeX provides a, Turing complete, programming model and due to this allows for rich customisability. This is one of the reasons why LaTeX is often used for long and complex documents such as books and articles.

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.

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. In a WYSIWG word processor, the user has to make sure his document is consistent. Once documents get longer, this gets very hard to manage and to spot.

Next to this customisability, LaTeX's PDF rendering is also very sophisticated. Another reason why LaTeX is widely used due to its good scientific support. LaTeX offers 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 TikZ.

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
- Stack Exchange

1.1.3 Pandoc

Pandoc 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. Filters can be written in a multitude of languages such as Haskell, Python, Perl,... and have to be passed as an 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.

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 this 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 is thus decreases readability and cleanliness of your document significantly. 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 slowing down the document creation.

The lack of customisability in Markdown is further enforced due to the fact that it has no programming model.

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. It also looks a lot more complex than Markdown, scaring away a lot of potential users. Lastly, 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 that we passed a wrong type at some point in the document.

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. 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.

1.3.1 Target group

Chapter 2

Design of the Neio markup language

In the previous chapter 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 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 get started with the language. The following example is one of the most basic documents you can write using Neio.

```
1 [Document]
2
3 # Chapter 1
4 This is some text in the first Paragraph.
```

Even if you are not familiar with Markdown than you probably immediately know what this doc-

ument 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:

```
1 new Document()
2     .#("Chapter 1")
3     .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.

```
1 // A single-line comment
2 [Document]
3 /*
4  * Multi-line comments are also available!
5  */
```

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.

```
1  namespace neio.stdlib;
2
3  import neio.lang.Text;
4
5  class Document extends TextContainer;
6
7  private Packages packages;
8
9  Document() {
10     packages = new Packages();
11     packages.add("a4wide")
12     .add("graphicx")
13     .add("amsmath")
14     .add("float");
15 }
16
17 Chapter #(Text title) {
18     Chapter chapter = new Chapter(title, 1);
19     addContent(chapter);
20
21     return chapter;
22 }
23
24 Packages packages() {
25     return packages;
26 }
27
28 Packages addPackage(String pkg) {
29     return packages.add(pkg);
30 }
31
32 String header() {
33     StringBuilder tex = new StringBuilder("\\documentclass{article}\n");
34     for (int i = 0; i < packages.size(); i = i + 1) {
35         tex.append(packages.get(i).toTex()).append("\n");
36     }
37
38     tex.append("\\setlength{\\parindent}{0em}\n")
```

```
39          .append("\\setlength{\\parskip}{1em}\n");
40
41      return tex.toString();
42  }
43
44  String toTex() {
45      return header() + "\\begin{document}\n" + super.toTex() + "\n\\end{document
            }\n";
46  }
```

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. Again, 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. This includes adding unnecessary symbols such as brackets to pass parameters, quotes to indicate strings and so on. All the symbols you can use for method names can be found in . 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.

2.3 Language features

2.3.1 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.

To be able to call the right methods, we introduced ContextTypes. Any time an instance of an object is returned form such a method we wrap it into a ContextType, together with the object created in the previous method call. This first object is the object that was returned from the

most recent method call, we call it the actual type. The second object is the object that was returned in the last but one method call, it is called the context type as it represents the context of the document at that point in time. When we call a method on a ContextType, we first check if the actual type has the method we're trying to call. If it does than we stop searching, if not we recursively check the context type.

2.3.2 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. This is of course very cumbersome and we would like to only define it once. 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.

2.3.3 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.

2.3.4 Code blocks

Scoped code

Non scoped code

Inline code

2.4 Considerations

2.4.1 Static typing

2.4.2 Security

2.5 Reuse of current possibilities

2.5.1 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

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

Chapter 4

Implementation

4.1 Used libraries and frameworks

4.1.1 ANTLR4

4.1.2 Chameleon

4.2 Compile flow

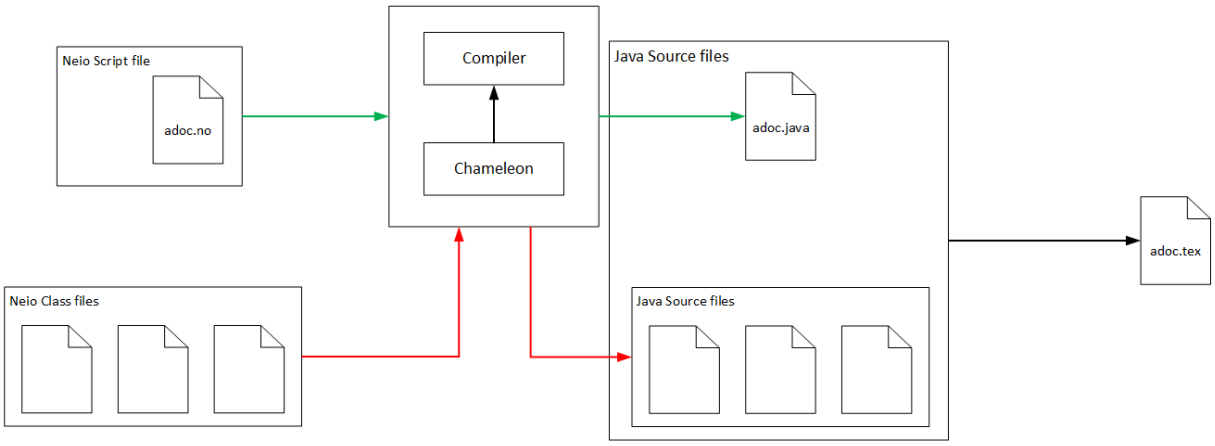

Figure 4.1: Illustration of how a Neio document is compiled. Neio class files translate to Java classes, Neio script files to a Java file with a main function. This then gets output to Tex using Neio's standard library

4.3 Translation to Java

4.3.1 Fluent Interface

4.3.2 Reflection

4.3.3 Limitations

4.3.4 Reasons for choosing Java

4.3.5 Automatic Text conversion

4.4 Outputting

Chapter 5

Future work

5.1 Static typing of the language

5.2 Tool improvement

5.2.1 Syntax highlighting

5.2.2 Auto completion