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 Markdown

One of the most popular markup languages at this time is Markdown. 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. 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. To achieve this simplicity however, a 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 to create a strong structured model but it still doesn't allow for very much customisability. The lack of customisability in Markdown is further enforced due to the fact that it has no programming model.

1.2 LaTeX

LaTeX in comparison to Markdown does provide 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 documents such as books and articles. 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. 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.

1.3 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, 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. 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.

1.4 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. Even though some customisability is available, no programming model is available 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. 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. The complex document formats are also hard to be used in combination with Version Control Systems.

Chapter 2

Neio

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 concretise what we want to achieve with our new markup language and how we go about doing so.

The name of the markup language designed and implemented in this thesis is `Neio, which is read as Neo`.

2.1 Goals

Taking into account the advantages and disadvantages of the solutions presented in Chapter 1, we came up with the following goals:

1. Userfriendly: easy to pick up
2. Use a modern programming paradigm
3. High customisability

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.

Neio makes use of two kinds of documents, the first kind is a Neio script file. This file contains plain text, much like a Markdown document, but it also allows for inline code to be added to it. The second kind of documents are Neio class files. These documents resemble Java classes and allow to customise anything that can be created in a Neio script file.

To allow for extensive customisability a script file is seen as a chain of method calls. Everything you see in the document is a method call.

2.2 Neio script files

2.3 Neio class files

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.

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.

2.4.2 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. 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.

2.4.3 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.

2.4.4 Code blocks

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

Implementation

3.1 Used libraries and frameworks

3.1.1 ANTLR4

3.1.2 Chameleon

3.2 Compile flow

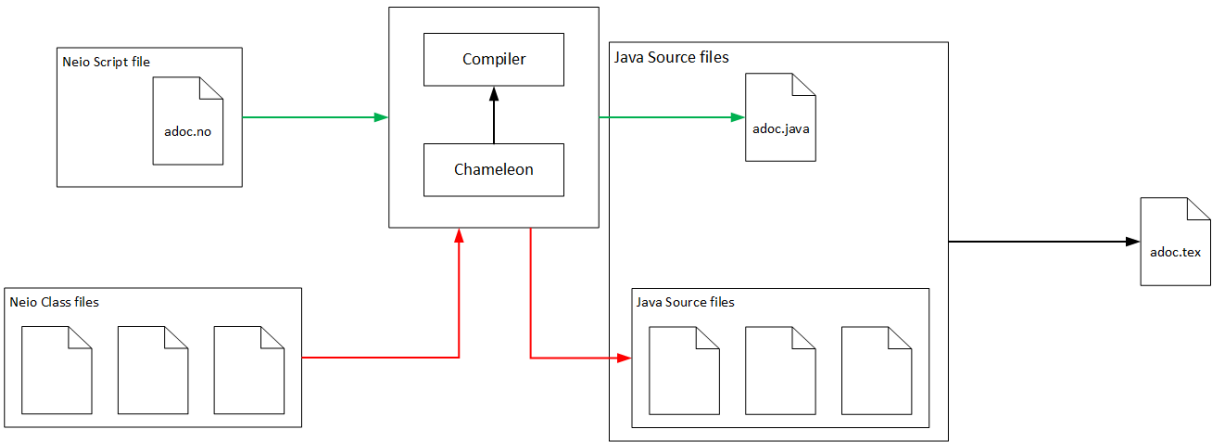

Figure 3.1: Illustration of how a Neio document is compiled. Neio class files translate to Java classes, Neio script files to a Java file with a main function. This then gets output to Tex using Neio's standard library

3.3 Translation to Java

3.3.1 Fluent Interface

3.3.2 Reflection

3.3.3 Limitations

3.3.4 Reasons for choosing Java

3.4 Outputting

Chapter 4

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.

4.1 Document types

4.1.1 Report

4.1.2 Letter

4.1.3 Book

4.1.4 Article

4.1.5 Slides

4.2 Libraries

4.2.1 TikZ

4.2.2 Beamer

4.2.3 LaTeX math and amsmath

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