ModMark: A Modular Document Markup Language

Designing and implementing a markup language utilising WebAssembly

Bachelor's thesis in Computer Science and Engineering

Eli Adelhult
Gustav Bruhn
Carl Forsinge
Axel Larsson
Hugo Mårdbrink
Jonathan Widén

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Bachelor's thesis 2023

ModMark: A Modular Document Markup Language

Designing and implementing a markup language utilising WebAssembly

Eli Adelhult
Gustav Bruhn
Carl Forsinge
Axel Larsson
Hugo Mårdbrink
Jonathan Widén

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

ModMark: A Modular Document Markup Language
Designing and implementing a markup language utilising WebAssembly
Eli Adelhult Gustav Bruhn Carl Forsinge Axel Larsson
Hugo Mårdbrink Jonathan Widén

© ELI ADELHULT, GUSTAV BRUHN, CARL FORSINGE, AXEL LARSSON, HUGO MÅRDBRINK, JONATHAN WIDÉN 2023.

Supervisor (handledare): Magnus Myreen, Department of Computer Science and Engineering
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering
Graded by teacher (rättande lärare): Nick Smallbone, Department of Computer Science and Engineering

Bachelor's Thesis 2023
Department of Computer Science and Engineering Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Photo by Raphael Schaller, licensed under the Unsplash License.

Typeset using LaTeX
Gothenburg, Sweden 2023

ModMark: A Modular Document Markup Language
Designing and implementing a markup language utilising WebAssembly

Eli Adelhult, Gustav Bruhn, Carl Forsinge, Axel Larsson,
Hugo Mårdbrink, Jonathan Widén

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

We present ModMark — a new modular document markup language that aims to provide a balance between the simplicity of Markdown and the power of LaTeX. It can be used in both a web playground and via the command-line. ModMark utilises the capabilities of WebAssembly and the WebAssembly System Interface, allowing for a extensible markup language that can be customised to fit the needs of specific use cases. ModMark allows users to create and manage custom packages that can be added to their documents to enhance functionality, while still maintaining a straightforward and intuitive syntax. Furthermore, the modularity of ModMark also enables users to write packages that add support for any output format they desire. The language is ergonomic enough to use for notes while also powerful enough to produce this bachelor's thesis.

Sammandrag

Vi presenterar ModMark – ett nytt modulärt markup-språk vars mål är att ge en balans mellan enkelheten i Markdown och kraften av LaTeX. Det kan användas både i en webb-miljö och via kommandotolken. ModMark nyttjar teknikerna WebAssembly och WebAssembly System Interface, vilket möjliggör ett utökningsbart markup-språk som kan anpassas till specifika användningsfall. ModMark gör det möjligt för användare att enkelt skapa och hantera anpassade paket som kan läggas till i deras dokument för att förbättra funktionaliteten, samtidigt som man behåller en enkel och intuitiv syntax. Dessutom möjliggör ModMarks modularitet även användare att skriva paket som lägger till stöd för det utdataformat de vill ha. Språket är tillräckligt ergonomiskt för att användas för anteckningar samtidigt som det är tillräckligt kraftfullt för att producera denna kandidatuppsats.

Keywords: Markup languages, WebAssembly, WASI

Acknowledgements

We would like to express our sincere gratitude to our supervisor, Magnus Myreen, for his support and guidance throughout the entire project. We would particularly like to acknowledge his consistent feedback, which has improved both the quality of this thesis and the technical aspects of our work.

Eli Adelhult, Gustav Bruhn, Carl Forsinge, Axel Larsson, Hugo Mårdbrink,
Jonathan Widén Gothenburg, June 2023

Contents

1

Introduction

1.1 Background

Typesetting is the art of composing characters or symbols in a way that matches the desired format and conveys the correct meaning. Historically, types were set by hand, and later on mechanically to reduce the effort required. While mechanical typesetting was a major improvement, the large machines could only cast a few characters per second while emitting significant noise and heat [1]. With the digital age, it became natural that typesetting would be done by computer systems.

1.1.1 Early digital typesetting

One of the earliest systems developed to typeset digital documents was Troff [2], [3]. It was developed at Bell Labs in the early 1970s to achieve compatibility with a new generation of computers. While there was a significant demand for such a typesetting system, especially within the Unix community, Troff had its flaws. Most notably, it was built as a layer on Nroff, which in turn was built on top of Roff, and all this complexity led to the final product being quite complicated. The famous computer scientist Donald Knuth said that "it was a fifth generation, each of which was a patch on another one. So it was time to scrap it" [4, p. 43].

Later in the 1970s, TEX [5] was created by Knuth in an attempt to improve on Troff. It was created from scratch, but shares many concepts with Troff, such as macros. TEX uses macros as low-level building blocks to generate complex code from smaller and simpler code at compilation. This design makes it possible to extend the functionality of the language with custom macros.

Neither Troff nor TEX see much usage in their pure forms today. There are instances where versions of them are used today such as GNU troff in Linux man-pages [6], but they have largely been replaced by newer alternatives.

1.1.2 Markup languages

A term that is often used when discussing typesetting is *markup*. A markup language is typically defined to be a language that is used to structure and describe the format of text. This definition includes the previously mentioned Troff and TEX but also languages such as HTML and XML. The aim of these languages is not

necessarily to be easy to use but to be precise. Over time, popularity has shifted towards more lightweight markup languages, designed to increase readability and convenience rather than focusing on maximum control of output.

A selection of popular and relevant markup languages are presented below, ordered by date of release.

LaTeX Leslie Lamport developed LaTeX (short for Lamport TeX) in the 1980s [7]. It includes many useful macros that ease the process of adding images, headings, etc. to documents without worrying about the low-level typesetting primitives. While this added a layer of convenience, the language is still quite verbose in comparison to more recent markup languages.

reStructuredText The first version of reStructuredText, or reST for short, was released in 2001 [8]. Since its release it has been adopted by the Python community for language documentation. It features syntactic sugar for formatting such as bold, italic, lists and tables. Additionally it allows users to write custom blocks in Python to extend the functionality.

AsciiDoc AsciiDoc first arrived just a year after reST [9], with a newer implementation from 2013 called Asciidoctor [10]. It is aimed towards technical documentation, and includes a variety of syntactical constructions for convenient writing much like reST. Figure 1.1 shows a small sample of both AsciiDoc and reStructuredText. The original implementation was written in Python, while Asciidoctor is implemented in Ruby (and ported to the web and the Java Virtual Machine). Customisation of the language can be done in both implementations through macros, which are regular expressions that are substituted during compilation. These do not require writing any external code. Asciidoctor also has support for extensions [11], which allows developers to use programming languages to generate output.

reStructuredText

```
#########
A heading
#########
A short paragraph.

++++++++++++++
**Bold text**
++++++++++++++
Another paragraph containing
*italic text* and a link to
`ModMark <https://modmark.org>`_
```

AsciiDoc

```
== A heading
A short paragraph.

=== **Bold text**
Another paragraph containing
_italic text_ and a link to
https://www.modmark.org[ModMark]
```

Figure 1.1: Source code for producing two headings, two paragraphs and a link in reStructuredText and AsciiDoc.

Markdown Markdown was released in 2004, just two years after the release of AsciiDoc [12]. It was developed by John Gruber and Aaron Swartz with a heavy focus on readability and being ergonomic. The original description of the language was lacking in detail and contained ambiguities, which eventually led to discrepancies between implementations. A decade later an effort was made to remedy this by redefining and clarifying the language with a more complete specification, which was released under the name CommonMark [13]. While it succeeded at what was intended, it also introduced complexity in both syntax rules and parsing.

In comparison to other markup languages, Markdown is quite limited in functionality. It revolves around convenient syntactical patterns but has little to no support for custom extensions.

Djot In 2022 John MacFarlane, who was active in the creation of CommonMark, presented Djot — a language aimed at fixing the shortcomings of Markdown and the complexities in CommonMark [14]. It expands on Markdown's original design by including elements such as tables and footnotes. Additionally, users are able to customise the language by creating custom filters.

Typst Later in the same year, Mädje [15] and Haug [16] presented a new alternative to LaTeX that they call Typst. The motivation behind creating Typst was frustration with the LaTeX macro system and lack of complex typesetting in lightweight languages like AsciiDoc. Typst has syntax for styling text, creating lists etc. but also supports many common programming language features like loops and conditionals [15]. It is essentially a way to combine code and ergonomic markup to produce documents.

1.1.3 Document converters and generators

Along with the development of markup languages, a number of tools such as document converters have emerged. Tools like these are meant to provide yet another layer of convenience for the user, and in particular improve the accessibility of markup languages.

One of the most popular document converters is Pandoc [17]. Pandoc was released in 2006 and created by John MacFarlane. It allows users to write a document in one markup language and automatically convert it to other languages. In fact, all of the previously presented languages are (to some extent) supported by the tool [14], [17].

Another tool is Aurelio Jargas' txt2tags [18], which was released in 2001. It is described as a document generator and functions slightly differently from Pandoc. Unlike Pandoc it does not support other markup languages as input. Instead, it uses minimal markup that was designed to simplify both implementation and usage. The primary features of this minimal markup are headers, tags (such as bold and italic), lists, tables and images. In terms of output, it works similarly to a document converter by converting the input to a variety of markup languages.

1.1.4 The current state of markup

Among the languages presented in Section 1.1.2, Markdown and LaTeX are the most popular. Other languages see some usage from corporations and communities. For example, AsciiDoc was used to document Git [19], and reST is still used for Python's documentation [20].

However, when it comes to personal use, Markdown and LaTeX are more prominent. LaTeX sees wide usage in academic writing, while Markdown is common in Internet forums and in programming documentation to provide simple formatting. When comparing Markdown and LaTeX, they quickly appear very different. A comparative example for LaTeX and Markdown can be found in Figure 1.2, which illustrates the difference in verbosity.

LaTeX

```
\documentclass{article}

\begin{document}

\section{A heading}
A short paragraph.

\subsection{\textbf{Bold text}}
Another paragraph containing
\textit{italic text} and a link to
\href{http://www.modmark.org}{ModMark}

\end{document}
```

Markdown

```
# A heading
A short paragraph.

## **Bold text**
Another paragraph containing
some _italic text_ and a link to
[ModMark](https://modmark.org)
```

Figure 1.2: Source code for producing two headings and two paragraphs of text in LaTeX and Markdown. However, it is worth noting that LaTeX documents rarely have an empty preamble (the section before the document begins).

LaTeX has a strong focus on expressiveness and customisation, though its complexity can make it difficult for new users to use properly. The macro system also makes editor integration and error handling less pleasant. On the other hand, Markdown's simplicity can be limiting for more demanding use cases.

While customisation in LaTeX is more accessible than that of TeX, it requires users to learn niche design patterns nonetheless. Newer languages like AsciiDoc and Djot generally allow users to write extensions in programming languages instead. However, these are often limited to the language their compiler was written in.

1.1.5 Future possibilities using WebAssembly

One can argue that these customisable markup languages would benefit from supporting more programming languages, as this would increase the reach and availability of their ecosystem. A recent technology that is capable of approaching this is WebAssembly, or WASM for short [21]. WASM is an instruction format for a virtual machine that is embedded in all modern web browsers. Many programming languages support WASM as a compilation target, including C, C++, Swift and Rust. Using a virtual machine allows for interoperability between different languages and makes it easier to distribute binaries that run close to native speed in all platforms supported by the browser.

However, WASM has also shown to be useful outside of browsers. Running WASM outside of browsers can be achieved using runtimes such as Wasmer [22] and Wasmtime [23]. These runtimes can be embedded in other applications to offer a portable way of running arbitrary code in a sandboxed environment. In addition to this, these runtimes implement WebAssembly System Interface (WASI) to provide an interface to the underlying system [24]. This enhances the capabilities of the program to allow for disk I/O, environment variables, and more.

WebAssembly presents an opportunity for markup languages to support many programming languages while still being platform independent. This is crucial not only to reach more users but also to ease development. Furthermore, this becomes even more important in ecosystems like LaTeX, where contributions from users are key to improving the language.

1.2 Aim

We aim to design and implement a declarative markup language named ModMark. The language should be suited for both note-taking and academic writing.

There is already a plethora of existing languages with the same goal in mind, but we explore a different set of trade-offs. The hope is to find a good middle ground between lightweight markup languages, such as Markdown, and more complex systems, such as LaTeX. The following bullet points summarise the three main design goals of ModMark.

- **A1. Expressive and modular**
  The language should be more expressive and extendable than Markdown and Djot by providing module expressions that offer a general syntax to extend the language with extra functionality. For instance, if a user wants to include citations, plots or figures in their document they can use this general syntax to import and use third-party implementations of these features. This is in contrast to Markdown where every feature is built-in and has concrete syntax. To put even more control in the hands of the user, ModMark also aims to support nested structures in a similar fashion to LaTeX. Continuing in the spirit of modularity, fundamental parts of ModMark's compiler should be designed to be agnostic of the output file format.

- **A2. Simple**
  The TEX family of systems (and similar systems like Typst) solve the problem of expressiveness and portability by making the document language itself powerful, even Turing complete. Our language will hopefully be easier to understand and require less programming knowledge since users will only need to interface with modules instead of writing actual programs inside of the document. Also, the syntax of ModMark should be lightweight and familiar to users of Markdown or Djot.

- **A3. Portable and developer-friendly**
  AsciiDoc, Djot and a few other languages offer similar expressive capabilities using an embedded language, compiler plugins or macros. However, this leaves package developers tied to using a specific programming language. Another option, used by Pandoc for example, is inter-opting with any program via the shell, but this risks losing portability. Our language aims at being platform independent and yet offer developers greater flexibility. This will be attempted by embedding a sandboxed WebAssembly virtual machine.

1.3 Scope

Successfully implementing ModMark includes both designing the language and developing a functioning compiler. However, we had to impose certain limitations on the project in order for it to fit within the time frame of a bachelor's thesis.

Firstly, ModMark does not directly control the resulting layout and typesetting. Instead, source code is transcompiled into another popular format, similar to Markdown's approach. We chose to focus on HTML and LATEX, because they give ModMark ample control of the final output while also being widely used and supported. This may seem to contradict the output agnosticism mentioned in Section 1.2 as a part of A1, but in practise this means designing the bundled version of the final layer to target those formats. Given the modularity of ModMark, this can easily be extended to include other formats.

There are also limitations in regards to the types of documents that are possible to produce. A main goal for the project was the ability to write this thesis in ModMark. This naturally drew focus towards elements such as figures and references. Overall the project has aimed to target web documentation and report-like documents, which covers only a fraction of the possible types of documents. Although, the modularity of the language allows for more document types.

During the course of the project there has also been a trade-off between developing the language itself and developing the surrounding tools. The tools are essential for testing and demonstrating the language, but can on the other hand steal development time. We decided to create a web page and a command-line interface relatively early in the development process. The implementations of these are sufficient for the group to perform testing and debugging, but lack polished user experience design.

1.4 Outline

The rest of the thesis is structured as follows. Chapter 2 *The ModMark Language* presents the design of the language, including an overview of the syntax and the ideas behind packages. Following that, Chapter 3 *From Concept to Implementation* describes the method and implementation used to realise our prototyped language. An evaluation of the results can be found in Chapter 4 *Evaluation*, where we discuss how well our aims were achieved. Lastly, Chapter 5 *Conclusion* discusses the current state of our language, conclusions and potential improvements as well as further reading.

2

The ModMark Language

We present ModMark — a new document markup language with a strong focus on modularity. It offers a lightweight syntax for common writing tasks such as headings, emphasised text and maths equations. Additionally, it provides a package system which allows the language to be extended with new features depending on the users needs.

Packages are independent WebAssembly programs that, amongst other things, add support for additional *modules*. Modules are a way of using a general syntax to include other elements in a document, such as images and code. They can also be used for more complicated elements that require a nested structure, such as tables and lists.

ModMark is designed to be transcompiled into other, more common, formats such as HTML or LaTeX. However, take note of the fact that the language itself is entirely agnostic of the output format. Due to the fact that the module system underpins the entire language, support for a new output format can be added by providing a new package that supports that format.

A short example of a document written in ModMark can be seen in Figure 2.1. The rest of this chapter will explain the syntax and semantics of the ModMark language, and gives an introduction to some of the included packages.

```
# A heading ①
A short paragraph.

## **Bold text** ②
Another paragraph containing
//italic text// ③ and a link to
[link ModMark] https://www.modmark.org ④
```

Figure 2.1: The same example document as in Figure 1.2 but written in ModMark. It demonstrates a few syntactical constructions such as (1) headings, (2) bold tags, (3) italic tags and (4) module expressions.

2.1 Headings

The first line of a paragraph may start with one or more number signs `#` to declare that line as a heading. The number of number signs used represents the level of the heading. If a paragraph, for example, starts with the line `## Heading`, it will become a heading at level 2. To recreate the beginning of Chapter 2 of this thesis in ModMark, one could write this:

```
# The ModMark Language
We present ModMark --- a new document markup
language with a strong focus on modularity ...

## Headings
The first line of a paragraph may be started
by one or more number signs ...
```

When compiling the document, the content of the heading is delegated to a package which declares that it may transform a heading to the desired output format. The headings will be transformed at the discretion of that package, and the limit of how many heading levels are allowed depends solely on the package. For example, the LaTeX package supports three levels of headings by outputting `\section{}`, `\subsection{}` and `\subsubsection{}`, while the HTML package supports six levels by outputting `<h1>` to `<h6>`.

2.2 Tags

Tags are used to style text which is done by encasing content between an opening tag and closing tag. This allows formatted text to be expressed with similar simplicity to how it may be expressed in Markdown. Examples of such tags are `**bold**` and `//italic//`.

Nesting tags, or using one tag within another, makes sense in some cases. For example, you may want to have some of your bold text italic as well, and the language allows for writing `**bold //bold and italic//**` to achieve such behaviour. In some other cases, it does not make as much sense. Encasing content in `$$` will output it as math, but it does not make sense that `**` inside that tag would be treated as bold text. It is much more likely that those characters would refer to a mathematical operation than to apply formatting to the encasing content. For this reason, some tags may allow nesting, meaning that their content will be checked for additional tags, while some tags do not allow for nesting. The complete set of tags built-in to ModMark is listed in Table 2.1, including their output and whether they allow for nesting.

Table 2.1: The tags built-in to ModMark

Source	Output	Allows nested tags
`**bold**`	**bold**	yes
`//italic//`	*italic*	yes
`__subscript__`	subscript	yes
`^^superscript^^`	superscript	yes
`==underlined==`	underlined	yes
`~~strikethrough~~`	~~strikethrough~~	yes
`` ``verbatim`` ```	`verbatim`	no
`Math: $$x^2+y^2$$`	Math: $x^2 + y^2$	no

Similarly to headings, tags are also delegated to a package capable of transforming those tags to the desired output format. The end result for a bold tag may be `<strong>` for HTML or `\textbf` for LaTeX. This also means that not all tags may be supported in all output languages, depending on if the package declares that it may handle that tag or not.

2.3 Smart punctuation

Smart punctuation is a feature of ModMark which replaces some characters with other characters that may be more appropriate. For example, a sequence of three dots `...` gets replaced with the ellipsis character …, and a sequence of two dashes `--` gets replaced with the endash character –. This makes it easier to write correct typographical characters.

The full list of smart punctuation features available in ModMark is the following:

- `"Double quotes"` gets replaced with “left and right double quotes”
- `'Single quotes'` gets replaced with ‘left and right single quotes’
- Two dashes `--` gets replaced with an endash –
- Three dashes `---` gets replaced with an emdash —
- Three dots `...` gets replaced with the ellipsis character …

Smart punctuation can be disabled by inserting an escape symbol (a backslash) before the characters.

2.4 Modules

Module expressions allow users to include any arbitrary element in their documents. Modules capture some input text in its raw form and then transform it to some output. The expression consists of a name, optional arguments, and a body of text for the module to capture. Modules may be used both inline from within a paragraph or by itself, spanning multiple lines.

Inline modules are written with its name within square brackets, followed by the text to be captured. The expression `[link] https://modmark.org/` is an expression for the `link` module, containing the text `https://modmark.org/`. This will, in turn, use a package supporting the `link` module to turn the expression to some output.

Modules outside of paragraphs, so-called multiline modules, are written as if they were their own paragraph, with at least one empty row preceding and following it. Here is an example of such an expression:

```
[poem]
Roses are red,
violets are blue,
unexpected '{'
on line 32.
```

Module expressions by default capture text until some boundary, which for inline modules is a space or a newline, and for multiline modules is an empty line. The text is then passed to the package which defines that module, and that package is free to arbitrarily transform the content. That means that it is possible to write many different kinds of modules. For example, a `code` module which renders code and keeps its text as-is, or a `table` module which wants to split the input text to different cells, possibly allow tags and modules within them.

To make a module expression capture more text than it would by default, a *custom delimiter* is used. When using a custom delimiter, the module expression captures text until a matching closing delimiter is found. To use a custom delimiter, it can be placed right after the closing square bracket containing the module name. Any non-alphanumeric character may be used as delimiter, and opening brackets use their corresponding closing bracket in the closing delimiter. This means that the expression `[math] x^2 + y^2` used inline would only capture `x^2`, since there is a space following it, but `[math](x^2 + y^2)` would capture the entire mathematical expression.

For inline modules, only one character may be used as delimiter, but for multiline modules, any number of characters may be used. To capture the entire code in this example, the delimiter `{[(` is used:

```
[code]{[(
fn first_elem(arr: &[u8]) -> u8 {arr[0]}

fn print_elems(arr: &[u8]) {
    arr.into_iter().for_each(|i| println!("{i}"))
}
)]}
```

There are a couple of noteworthy points in this example. First of all, since this expression uses a custom delimiter, it captures text which includes a newline. Second of all, the code itself uses a lot of brackets in combination, such as `])` and `]}`. Since

the custom delimiter can be of any length, one could always find a delimiter that does not collide with the content that is captured, and thus any content can be captured regardless of what it actually contains. Using either `([` or `{[` as delimiters would match closing delimiters inside of the captured content itself, but the combination `)]}` does not occur anywhere in the code.

In addition to the module name and captured text, modules may also take arguments passed to them. When a module is defined by a package, the package also defines if it takes any arguments and whether or not they are optional. Arguments are passed to the module inside of the square brackets, and may or may not include the name of the argument. If an argument is unnamed it works as a positional argument, and these can not appear after named arguments. If the `code` module would take an argument `lang`, both `[code rust]` and `[code lang=rust]` would pass the value `rust` as the argument `lang` to the module. Here is another larger example of a module expression with argments, this time of an image:

```
[image
    alt     = "Black and white photo of Alonzo Church"
    caption = "Alonzo Church"
    label   = "fig:alonzo"
    width   = 0.5
]
alonzo.jpg
```

2.5 Packages

A *package* is a WebAssembly program that adds additional features to your document. They add support for transforming modules, document templates, and/or tags into one or more output formats. When a user is in need of specific feature or want to add support for a new output format they can import a package that supports those transforms. Packages are imported using a special `[config]` module found once at the very top of a file. For example, here is the source code for importing a package named `prettify` and then using a module with the same name.

```
[config]
import prettify

[prettify]
Pretty text
```

It is worth noting that every package also includes a manifest that declares what transform it provides but also additional information such as descriptions, arguments, and types. This makes it possible to automatically generate documentation and provide better compilation diagnostics. It also has the potential to be used for editor tooling like completions and hover-over descriptions.

2.5.1 Standard packages

Users expect to have a sensible set of default packages that allow them to start writing documents without having to configure everything themselves — this is why ModMark comes bundled with a collection of *standard packages* that support both HTML and LaTeX output. They are ordinary packages that help with common writing tasks but can easily be replaced if a user has other needs.

This includes features such as bibliography management, images, lists and tables. An exhaustive list can be found on the projects online documentation:

https://modmark.org/#/package-docs

2.5.2 Native packages

These previously mentioned external WASM packages may do anything with the input they receive, but some features require access to the compiler itself. For this reason, *native packages* exist to provide functionality not accessible by external packages. They are native in the sense that they are built-in to the ModMark compiler itself and thus has access to the compilers internal data structures and lives outside the external sandboxed WASM environment.

Native packages only include module transformations, examples of such modules are `[block_content]` and `[inline_content]` which allows other modules to include nested structures by parsing a part of the document again. Two more examples are `[error]` and `[warning]` that are used to produce errors and warnings.

3

From Concept to Implementation

For ModMark to be practical to use, there must be a way of translating documents written in the language into other more commonly used formats such as HTML or LaTeX. This is what a compiler is used for.

To translate ModMark into other formats, we implemented a compiler using the Rust programming language. We also utilised the Wasmer WebAssembly runtime [22] to run the previously mentioned external packages.

This chapter presents the development process, technical details and relevant design considerations for the implementation of the compiler and other language related tooling. Figure 3.1 gives an overview of the compilation process which is explained in the rest of this chapter.

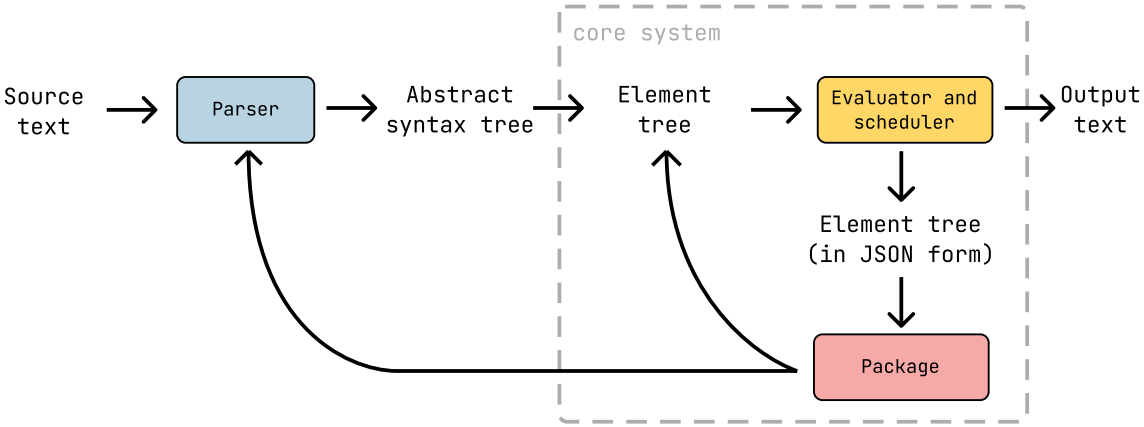

Figure 3.1: A high-level overview of the compilation pipeline showing how source text is transformed through multiple intermediate formats, iteratively evaluated by packages which finally results in some output text in another other format (such as HTML or LaTeX).

3.1 Development workflow

At the start of development three key areas of the system were identified. These three consisted of a parser for the language (Section 3.2), a core runtime for evaluating the parsed tree (Section 3.4 – 3.5), and tooling for using the language via the command-line (Section 3.6) as well as in the browser (Section 3.7). All of these systems were in

place after five weeks and we could then move to a more iterative and less restrictive workflow.

Our workflow was based on GitHub's [25] issues and pull request system. We established a set of conventions for commit messages and branch naming to avoid confusion. Whenever a bug was encountered or a new feature was discussed, an issue describing the bug or feature would be created so that there is a clear list of tasks that are available.

We held weekly meetings to prioritise and coordinate the workload, usually by assigning group members to certain issues. When an issue was completed, a pull request was opened so that other members could review the code. After being approved, the branch was rebased into the main branch. Rebasing was chosen instead of merging to get a clean commit history without merge commits.

Table 3.1 lists all subsystems developed during the project along with the total amount of code. While the number of lines of code is not always an accurate metric for measuring the size and complexity of a project, it can provide a helpful overview. The following sections will describe these systems in greater detail.

Table 3.1: Summary of the total lines of code (LOC) in the project.

Subsystem	**Language**	**Files**	**LOC**
Parser	Rust	7	2 292
Core	Rust	12	4 335
Standard packages	Rust	16	4 581
Parser tests	JSON	22	1 205
Parser test runner	Rust	119	145
Standard package tests	JSON & Toml	143	4 581
package test runner	Rust	1	414
Command-line interface	Rust	4	932
VS Code extension	YAML	1	93
Chalmers-thesis package	Rust	3	1 450
Website	TypeScript	10	1 757
	Rust	2	541
	JavaScript	1	94
Demo packages	Zig	2	188
	Go	2	280
	Swift	2	268
	AssemblyScript	1	180
	C++	1	147
Total		**351**	**23 483**

3.2 Parsing

The first step of compiling a document is parsing. Parsing is the process of transforming the original source text into an *abstract syntax tree* (AST). In practise, this process means distinguishing different parts of the source text and storing them in data structures that contain relevant information. Structuring the information in such a tree makes it easier for the succeeding compilation steps to further transform and then evaluate the document. Figure 3.2 shows an example of a simple ModMark document parsed into an AST.

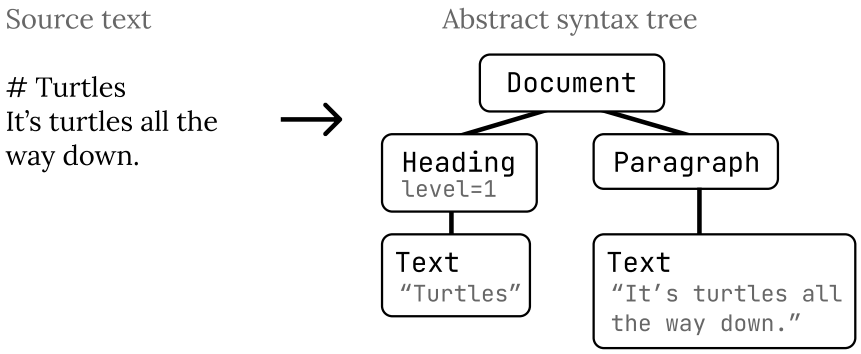

Figure 3.2: Some example source text being parsed into an AST.

The ModMark compiler uses a parsing technique called combinatory parsing. Parser combinator libraries use an embedded domain specific language approach and offer a small set of primitive parsers that can be combined into larger parsers, often in the form of ordinary functions that simply consume parts of the input [26, p. 252–254]. In this particular case the Rust library Nom [27] was used.

The parsing step in the ModMark compiler is performed by an isolated library (a *crate* in Rust terminology) that simply is imported in the main part of the system. Having the parser as a separate library also eases the potential future process of creating ModMark bindings for other programming languages. The responsibilities of the parser library include producing an AST from source text and including the typographically correct punctuation characters mentioned in Section 2.3. In addition to this, the parser retrieves information from a document configuration module and passes this to the later stages of compilation.

3.3 Element-tree conversion

Once the parser has produced an abstract syntax tree, the next step in the compilation chain is translating it into a more useful intermediate representation — what we call an *element tree*. Each constituent of the document, including paragraphs, headings, images, and other entities, is regarded as an *element*.

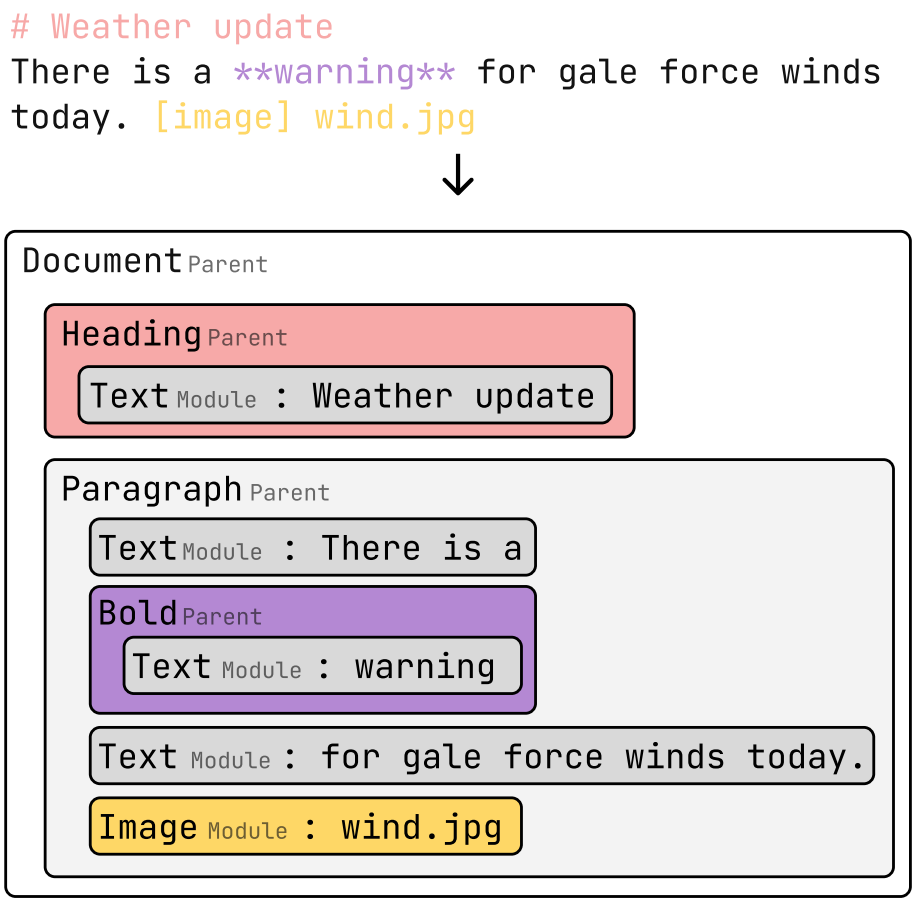

Figure 3.3: Some source text that has first been parsed into an AST and then converted into an element tree.

The translation of the AST into an element tree categorises the source text into two variants: *modules* (including all `[module]` expressions and some other things like plain text) and *parents* that may contain other elements as children (for instance, bold tags, headings and paragraphs). Unique and totally ordered IDs are also attached to each element in this process. *Raw* and *compound* element variants may also exist in the tree, but will only occur in later stages of the compilation process. See Figure 3.4 that contains the Rust code that describes the element tree and Figure 3.3 for an example conversion from a document to an element tree.

```
enum Element {
    Parent {
        name: String,
        args: HashMap<String, String>,
        children: Vec<Element>,
        id: GranularId,
    },
    Module {
        name: String,
        args: ModuleArguments,
        body: String,
        inline: bool,
        id: GranularId,
    },
    Compound(Vec<Element>),
    Raw(String),
}
```

Figure 3.4: A Rust enum (a sum type) describing the element tree.

3.4 Scheduling

After creating an element tree, it should be transformed into the desired output format. However, simply recursively evaluating each element starting from the root is not sufficient. Consider the following example of a document that includes a table of contents.

```
[table-of-contents]

# Introduction
# Method
# Discussion
```

In order to generate the final document, all headings need to be evaulated before the `[table-of-contents]` module. This is because the module itself does not have access to the rest of the document. Thus, the compiler needs to produce information about the headings that is available to the module during its execution. Furthermore, other modules may evaluate to any arbitrary set of elements, making it difficult to determine which elements should be included in the table of contents. This means that you can have a `[mystery]` module that later, when being transformed, happens to contain yet another heading that the table of contents module must take into consideration. This issue is not unique to headings either, the same problem arises in other situations where shared state is needed e.g. figures, imports, references etc.

There are multiple ways to address the issue, LaTeX for example requires the document to be compiled multiple times to resolve these so-called cross-references [28, p. 36]. The ModMark compiler solves this issue using a different approach. The compiler includes a variable system that tracks which element accesses what variables and the dependencies between them. Packages define which variables each element needs access to. Variables can then be read and written to using six modules that are native to the language; these modules are listed in Figure 3.5. Because external packages are run using WASI in a similar fashion to command-line applications, it is also possible to read the variables as if they were ordinary OS environment variables.

Set	List	Constant
`set-add`	`list-push`	`const-decl`
`set-read`	`list-read`	`const-read`

Figure 3.5: The six modules used to write and read to variables. The different types and operations they offer affect the dependencies between elements.

Using the information from the variable system, the compiler can model the problem as a directed acyclic graph (DAG), where elements are vertices and edges are drawn between elements that need to be transformed after one another. An example of such a graph can be seen in Figure 3.6. To decide in which order to transform the elements, the DAG can then simply be topologically sorted.

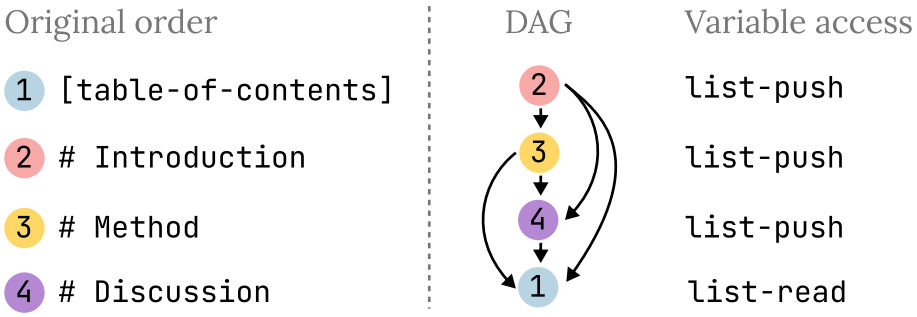

Figure 3.6: An example of a DAG generated by the scheduler. Note that all dependencies (edges) in this example are caused by list push operations to a *headings* variable. Push operations are required to be performed in the same order as they appear in the source text and this is why the headings depend on each other too.

Once sorted, the elements are then transformed and any new elements that are created in this process will be inserted in DAG and sorted once again. There are algorithms such as Pearce-Kelly [29] that are designed to efficiently maintain a topological order in a DAG with incremental insertions. However, since elements are discarded after they have been transformed the compiler can use Kahn's algorithm [30] which is simpler. The algorithm works by repeatedly removing vertices that have no incoming edges.

3.5 Element transformation

Transforming takes place during the evaluation process, where the parsed element is transformed into the chosen output format. When an element is to be transformed, as determined by the scheduler, the name of the element is looked up in a map to find what package is responsible for the transform. As previously mentioned in Section 2.5, the package that transforms the element may either be a native package or a WebAssembly program.

If the transform is provided by a native package, it means that the implementation is written in Rust and is included in the compiler itself. If the package instead is a WASM file, the compiler needs to run it in a WASM runtime. Since programs using the WebAssembly System Interface (WASI) behaves much like command-line program, ModMark supplies the element name and the output format as arguments to the program, and passes the rest of the element serialised as JSON to `stdin`.

Once the package has received information about the element, it executes the transform and generates a number of new elements as output. The elements are printed to `stdout` as a JSON array, whereupon the compiler reads the data and deserializes it. Then, the compiler proceeds to replace the original element with a *compound* element containing the data from the package. Packages can produce warnings and errors by printing to `stderr` or exiting with a non-zero exit code, very much like a conventional command-line program would.

There is a multitude of WebAssembly runtimes available for use outside of the browser. The ModMark compiler uses Wasmer, which has its own interpreter which is used in combination with a compiler back-end, Cranelift and LLVM being the most note-worthy. The ModMark compiler uses Cranelift, a Rust compiler framework often used in the WASM ecosystem, which is generally found to have faster compile-speeds than LLVM but less optimised output [31].

An advantage of Wasmer over other options is that it also can use the browser's built-in runtime if targeting the web. This reduces the binary size for embedding the compiler in a website and additionally increases performance. Also, the standard packages are bundled in the binary itself, and when targeting the web we thus need to bundle the raw WASM bytecode to let the browser compile it with its own compiler. However, when targeting a native platform, we also include the option to compile the WASM bytecode using Cranelift directly to the intermediate representation used by Wasmer at build-time instead of at run-time, drastically improving startup times (at the cost of a larger binary file size).

3.6 Command-line interface

To allow users to interact with the compiler, we built a command-line interface (CLI). The ModMark CLI is supported on all major operating systems and allow source files to be converted into an output file. A simple version of the tool was completed within the first two weeks of development.

More features has since been added to improve the user experience. Most notably, the tool supports live updating HTML previews. This is accomplished by running a file system watcher, static web server and a WebSocket server that informs the outputted HTML website refresh to itself on changes.

The CLI also offers granular control over file system access and by default packages have no access to the host machine at all. Another interesting feature that has been explored is the ability to auto-generate example packages in multiple programming languages such as Rust and C++ by running the `modmark init <LANGUAGE>` command. This is intended to ease the process of getting started with package development.

3.7 Web playground

Aside from the command-line interface there is also a web playground (seen in Figure 3.7) that allows users to easily test ModMark without having to download anything. The playground uses the same compiler as the rest of the project but is built to WebAssembly and comes with accompanying automatically generated JavaScript bindings. Two versions of the playground has been built during the project, one only using ordinary JavaScript and another more full-featured version using the JavaScript library React.

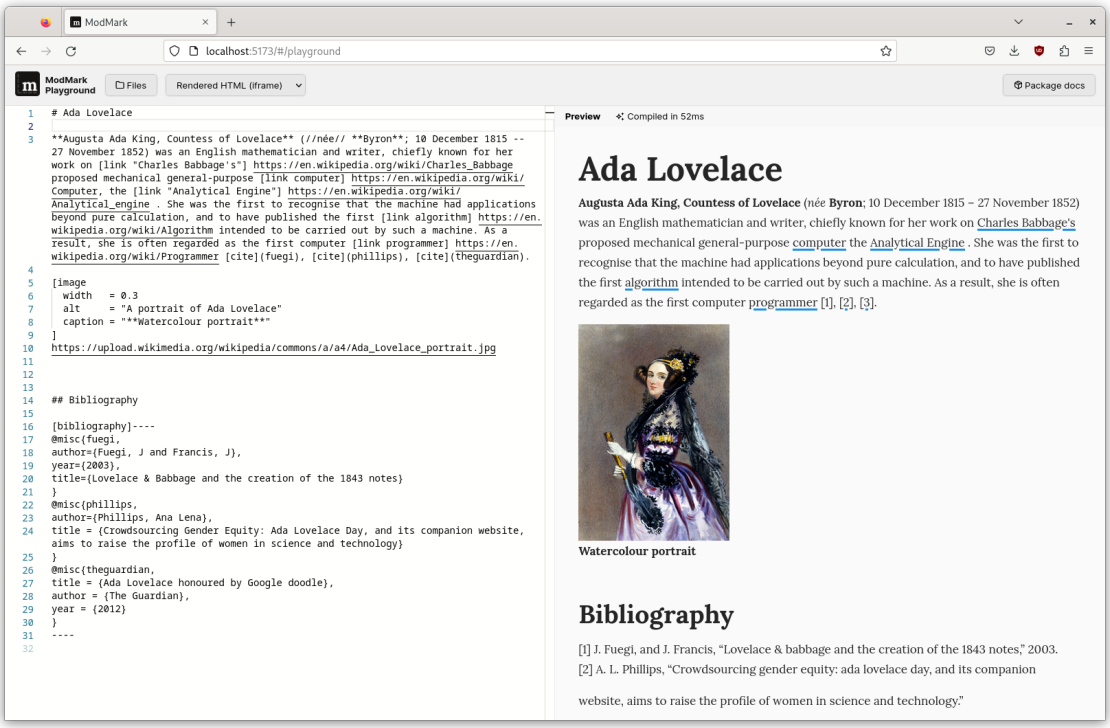

Figure 3.7: A screenshot of the web playground with a reproduction of the Wikipedia article about Ada Lovelace. This screenshot uses material from the Wikipedia article Ada Lovelace, which is released under the Creative Attribution-Share-Alike License 3.0.

The playground has many features that have been helpful when developing ModMark. The text written in the playground can of course be converted to any output like HTML and LaTeX, but there is also an option to see the AST representation of the text which can be very helpful when debugging. The playground also has support for displaying automatically generated documentation for any package, an example of this can be found in Figure 3.8. Finally, there is virtual file system with an accompanying file browser that allows users to upload files such as images to use in their document.

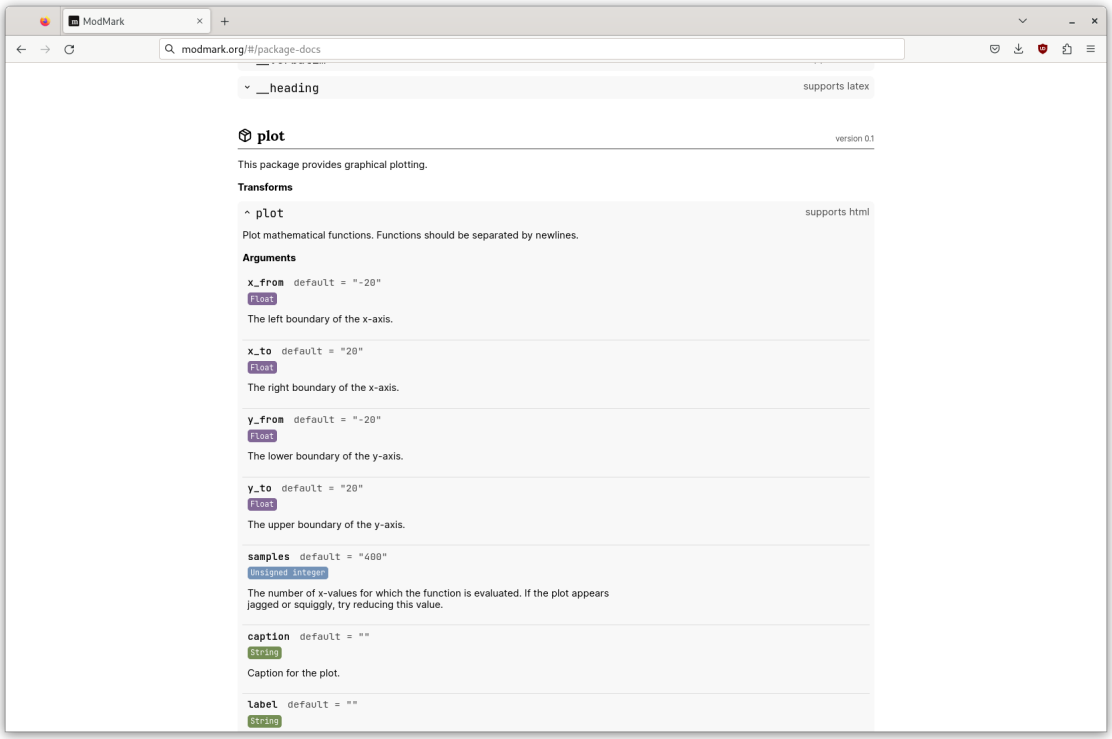

Figure 3.8: A screenshot of the documentation view in the playground.

Web-based (and sometimes real-time collaborative) tools are becoming the norm for writing and editing documents with popular examples like Google Docs and Overleaf. Even though developing a tool like this for ModMark is outside the scope of this thesis, the playground serves as an important proof of concept showing that it would be possible to do so — even without any server side code needed to evaluate and render documents.

3.8 Package manager

Modularity is a key feature of ModMark, and as such, it is crucial for users to have an easy way to manage third-party packages. In the TeX ecosystem, packages are typically obtained through CTAN [32] (short for "the comprehensive TeX archive network"). However, users are often left to download either huge distribution of all common packages or manage packages themselves [33].

We instead aim to provide a built-in mechanism for managing dependencies, where missing packages can be automatically resolved when compiling a document. To achieve this, we have included a package resolution protocol in the compiler, allowing users to provide a path to the packages they want to use. An integrated proof of concept package manager (found in both the command-line tool and on the web) then communicates with the compiler and provides any missing packages by asynchronously downloading them from the Internet and then caches them locally.

Our protocol is inspired by the Deno JavaScript runtime [34] and uses a *specifier* as prefix to the path in order to inform the package manager where to find the desired packages. See Figure 3.9 for an example.

```
[config]
import math                                  From local file
import https://modmark.org/math.wasm         From website
import catalog:math                          From catalog (canonical
                                             package registry)
import std:math                              From standard library
```

Figure 3.9: A configuration module with import statements with different paths. The specifier is highlighted in purple.

3.9 Testing, continuous integration and deployment

Continuous integration (CI) is the practise of continuously running integration tasks on code which is in development. Grady Booch [35, p. 314] describes that CI is essential for testing, which should be a continuous activity during development.

For this project, GitHub Actions were used to enable testing with CI. The project includes the following test suites:

- **Parser tests** that contains sample ModMark input and runs it though the parser, and verifies that the input is parsed correctly. We currently have 22 test cases which ensures that the parser is working correctly for those test cases, and if changes are done to the parser the tests may catch any change that breaks functionality.
- **Package tests** that contains test cases for the standard packages included in ModMark by default. Since the standard packages are standalone programs written in Rust, the packages are compiled and all the test cases are passed to them one by one. The result of the program is compared to the expected output, and if they match, the test passes. The test worker, which is the program executing the tests, allows running the packages targeting different languages and with different environment variables, simulating different variables provided by the ModMark compiler. There are currently 143 tests for the standard packages, which ensures that the packages are working correctly

and gives valid, parsable and correct output for those test cases. In addition to this, there are package tests checking the Rust toolchain configuration for each package, ensuring that the toolchain is configured such that it produces packages that can be located when building the project.

- **Package loading test** is a test which instantiates our compiler and loads the standard packages. When loading packages, the package manifests are also checked, ensuring that argument names, value types and default values are valid, and that variables and variables referencing arguments are valid.

In addition to CI, a related term is continuous deployment (CD) which is the practise of continuously release deployments of code which is in development. The continuous deployment part of this project includes building and publishing the playground, see Section 3.7. When changes to the published version of the code are made, GitHub Actions compiles the code into a web page and publishes it online. When a pull request is made or is updated, the playground is also compiled with the code contained in the pull request, and the playground is published to be able to preview the changes. During the development of this project, there have been over 650 previews deployed and over 120 deployments of the main branch.

Both the continuous integration and continuous deployment has been central in the development of this project. When a pull request is made to propose code changes, the continuous integration procedures automatically ensure that none of the test cases fail. This means that the developer writing and reviewing the proposals does not have to spend time manually testing every single input to make sure no functionality has been broken. Additionally, the continuous deployment provides a convenient way to test the changes online in the generated preview playground, rather than downloading and running the code locally.

4

Evaluation

The primary objective of the project is to design and implement a markup language that is well-suited for both note-taking and academic writing. As evidenced by the fact that the text you are currently reading has been written in the ModMark language (and transcompiled into LaTeX), significant progress towards achieving this goal has been made.

The successful compilation of this bachelor thesis demonstrates that the ModMark compiler is functioning as intended, and that the language itself is expressive and powerful. Nevertheless, it is important to acknowledge that both the design and implementation of ModMark are not without their shortcomings. The remainder of this chapter will investigate to which extent the design ideals outlined in Section 1.2 were realised.

4.1 Expressiveness (A1)

Aim A1 is to make ModMark expressive enough to bridge the gap between lightweight languages and powerful systems like LaTeX. We also stated that the language should be output agnostic. While the former is more subjective, the latter is a core property of the compiler presented in Chapter 3.

The majority of ModMark's expressiveness comes from its packages and modules. It relies on these to provide functionality that is necessary to produce the desired document. The set of packages that is included in the standard bundle of ModMark goes a long way in enabling most kinds of writing. As was presented in Section 2.5.1, they support figures, tables, and even a standalone bibliography and citation system. This is adequate for most parts of a typical academic report. However, the standard packages may not be enough when conforming to certain report standards. Sections such as cover page and title page are more cumbersome to reproduce in ModMark without specialised packages.

To write our thesis in ModMark we needed to create a *chalmers-thesis* package. It is responsible for creating LaTeX chapters and the pages that precede the table of contents. While the need for such a package can be seen as a shortcoming, it also demonstrates that the core concepts of ModMark are working. It shows that it is possible to extend the language with new functionality if needed. This is akin to the many LaTeX templates used by different scientific journals, such as the Association

for Computing Machinery [36]. Granted, the extendability of the language only becomes a positive factor once there is a healthy ecosystem of packages.

Furthermore, part of our goals towards expressiveness was to make ModMark agnostic of output format. In Chapter 3, we presented the design and implementation of a compiler that achieves this. Additionally, both the CLI and web playground were designed to be output agnostic, by allowing the user to specify the output format. Although, because HTML and LaTeX were prioritised in development they were also given additional functionality (see Section 3.6 and 3.7). In that sense, while the compiler is truly output agnostic, the full ModMark suite is not.

In Figure 4.1, three example documents in widely different domains are presented, showing that the language is extensible enough to add capabilities to be used for different purposes. Links to these documents can be found in Section 5.3.

Figure 4.1: Three example documents written in ModMark, the first one containing lecture notes, the second one being an article about the chess opening *Three Knights Game*, and the third one being this thesis. The second article uses material from the Wikipedia article Three Knights Game, which is released under the Creative Attribution-Share-Alike License 3.0.

The decision to make ModMark output agnostic has served us well in development. It eased the process of extending and fine tuning the output for both HTML and LaTeX. Because the support for these was written exclusively as packages, the process should translate well to adding support for other formats. In practise, a package that provides fundamental support for a new format is relatively small in size and complexity. Therefore, it is possible for individual users to add support for new output formats.

4.2 Simplicity and syntax (A2)

Another aim A2 in designing ModMark is simplicity. The language is intended to offer an ergonomic lightweight syntax and also be easier to understand in comparison to more full featured typesetting systems such as LaTeX and Typst.

Simplicity is highly subjective and very hard to evaluate in an objective manner. However, since ModMark aims to strike a balance between the simplicity of Markdown and the expressiveness of LaTeX we will offer a non-exhaustive comparison of syntax and features for common writing tasks in the three languages[1].

Headings Headings in ModMark are very similar to Markdown with some subtle differences. No whitespace character after the number sign is needed like in Markdown and the alternate syntax that uses equal signs or hyphens is not supported. See Figure 4.2 for a comparison.

Markdown (CommonMark)	ModMark	LaTeX
`# Heading 1` `# Heading 1 #` `Heading 1` `=========`	`# Heading 1`	`\section{Heading 1}`
`# Heading 2` `# Heading 2 #` `Heading 2` `---------`	`## Heading 2`	`\subsection{Heading 2}`

Figure 4.2: Comparing the syntax for headings in ModMark, Markdown, and LaTeX.

Styling text ModMark follows Markdown's convention of using non-alphanumerical characters as tags to surround text that should be displayed in another format such as `**bold**` or `//italic//`. See Figure 4.3 for a full comparison.

Using tags results in a shorter and less obtrusive syntax in comparison to LaTeX (with the exception of math mode). Since ModMark includes many more tags than Markdown, expressing other styling options such as superscript and math also leads to a shorter syntax since Markdown requires the use of HTML for expressing those concepts.

A potential downside of ModMark's approach is that the list of characters with special behaviour that users need to know about gets larger than for instance LaTeX where, almost, everything in text mode follows the convention of a command prefixed with a backslash. It is also possible to argue that ModMark is needlessly verbose by always requiring four characters for tags but it helps to avoid confusion

[1]Note that there are many dialects of Markdown that offer different features, such as GitHub-flavoured Markdown, John Grubers original Perl script and Pandoc Markdown. These comparisons will use the CommonMark specification [13].

or ambiguity in situations with nested tags or prefixes (`**__a__**example`), which most Markdown implementations do not support.

Markdown (CommonMark)	ModMark	LaTeX
`**Bold**` `__Bold__`	`**Bold**`	`\textbf{Bold}`
`*Italic*` `_Italic_`	`//Italic//`	`\textit{Italic}`
`` `Verbatim` ``	``` ``Verbatim`` ```	`\verb|Verbatim|`
`<u>Underline</u>`	`==Underline==`	`\underline{Underline}`
`<del>~~Strikethrough~~</del>`	`~~Strikethrough~~`	`\usepackage[normalem]{ulem}` `\sout{Strikethrough}`
`<sup>Superscript</sup>`	`^^Superscript^^`	`\textsuperscript{Superscript}`
`<sub>Subscript</sub>`	`__Subscript__`	`\textsubscript{Subscript}`
`<math` ` display="inline">` `  <mtext>` `  Math` `  </mtext>` `  <msup>` `  <mi>x</mi>` `  <mn>2</mn>` `  </msup>` `<math>`	`$$Math x^2$$`	`$Math x^2$` `\(Math x^2\)` `\begin{math}` `Math x^2` `\end{math}`

Figure 4.3: Comparing the syntax for common inline text formatting options in ModMark, Markdown, and LaTeX. Note that HTML and MathML is used for situations where Markdown does not provide any concrete syntax.

Modules Markdown has no syntax for user-provided features and only offer concrete syntax for a few more common elements such as images and links. LaTeX and ModMark on the other hand has a syntax that may be used for any arbitrary feature: `\commands[]{}` and `[modules]`. It is debatable if ModMarks modules are easier to understand than the TeX macro system. Although ModMark has some potential benefits when it comes to automatically generated documentation and editor integration.

The syntax for multi-line modules in ModMark is also shorter than the environments (`\begin{something}`) found in LaTeX. But ModMark has its own flaws, since modules always consume content verbatim and then later can choose to handle nested structures examples such as this may confuse users:

```
[center]
[code]{
Some code

This will not be part of the code block since 'center'
has no delimiter and is ended by the line break above.
}
```

4.3 Portability and package development (A3)

One of the central ideas of ModMark is portability; a document written on one computer should still produce the same result if compiled on another computer, regardless of what packages are used. A3 formulates this as a concrete requirement, and the same aim also states that the solution should offer package developers a lot of flexibility. At the core of our solution for A3 is the Wasmer WebAssembly runtime. It supports the major operating systems: Linux, Windows and macOS and also all common web browsers. There were also recent improvements to their chipset support. Version 3.2 of Wasmer added RISC-V to the list of chipsets that is supported by both Cranelift and LLVM, which now consists of x86_64, arm64, x86 and RISC-V [37]. To that extent, the portability of ModMark is sufficient for the goal that was set.

There is also the aspect of being accessible and convenient for developers who wish to extend the language, which is an essential criterion for the growth of the ModMark ecosystem. Development of the standard modules (all written in Rust) was relatively straightforward. However, Rust may not be a good benchmark for the typical experience of package development, since it works well with both WebAssembly and Wasmer and there are well-established libraries for serialisation (including JSON support).

A handful of the popular programming languages can be compiled into WebAssembly, although most do not have the same first class support that Rust does. A hurdle for all other languages is the requirement of compatibility with both Wasmer and the WebAssembly System Interface. To better evaluate this goal we have attempted to implemented packages in multiple programming languages:

- **Rust**
  13 different packages have been implemented in Rust. As previously mentioned, the developer experience using the rustc compiler and the `wasm32-wasi` target has been great. The ModMark CLI tool also supports generating new Rust packages with `modmark init rust`.
- **C**
  A template for using C and the Clang-based Wasienv toolchain [38] is available using the `modmark init c` command. However, we have not implemented any larger packages in C since the language is far from ergonomic when it comes to string manipulation and JSON.

- **C++**
  C++ also has support for the init command and can be used with the same Wasienv toolchain as C does. Additionally, a demo package called prettify has been implemented.

- **AssemblyScript**
  AssemblyScript, a sub-set of Typescript, is also supported by the ModMark init command and a demo package for using the Vigenère ciphers has been implemented. AssemblyScript has first-class WASM support but requires additional libraries for WASI support.

- **Go**
  A demo package for rendering chess boards has been implemented and can be used with the TinyGo compiler. Compiling for WASM is very straight forward but sadly TinyGo does not support the entire Go standard library.

- **Zig**
  Zig has first-class support for WASM and WASI and a demo package for "rövarspråket" has been implemented. However, since Zig is a very low-level language it might not be very well suited for developing packages in practice.

- **Swift**
  Swift also has good WASM and WASI support and a package for generating lorem ipsum text has been written. The big downside of developing packages in Swift is the large runtime which leads to larger binary sizes.

- **Haskell**
  We have also made unsuccessful attempts at implementing a package using the Glasgow Haskell Compiler, but their support is currently only a tech preview that is incompatible with Wasmer [39].

To add even more nuance to the issue of supporting multiple languages, there is also a matter of performance. If users were to develop packages in languages such as Python and JavaScript, it could slow down the ModMark compiler significantly. This is because higher-level languages need to include a larger runtime [2], introducing a large performance overhead and size. However, recent developments may remedy this issue. There is an active proposal for WebAssembly to add support for built-in garbage collection [21]. It includes a handful of features to reduce the size and increase the efficiency for use of high-level languages that use garbage collection, JavaScript and Python included.

There are some noteworthy downsides to using WebAssembly too. WASM and WASI are yet to allow functions that return more complex data types. Subsequently, the API for manipulating the element tree is at risk of being cumbersome to use in comparison to an embedded language with free access to the element tree. Although, this problem can arguably be solved by providing bindings that convert the input provided by ModMark's WASI API into an easier-to-use API suited for a particular language.

[2] CPython compiled to WebAssembly with WASI support is around 125 MB.

When comparing the portability to other languages, ModMark appears as a good option nonetheless. Systems like LaTeX and Typst solve the issue of portability by having a fully Turing complete language that lets developers write packages directly in the document language itself. Another common approach is embedding a scripting language such as Lua that is used to create extensions [14]. A notable downside to these approaches is that developers are restricted to a domain-specific language they might not be familiar with. Moreover, they do not have access to the rich ecosystem of libraries that is often present in a general-purpose programming languages and have to manually implement every feature themselves.

4.4 Accessibility

Wang, Cachola, Bragg, *et al.* [40] discuss how PDFs are not accessible to everyone and have poor support for screen readers. They present a possible solution to this problem with a system that converts PDFs to more accessible HTML documents. While this is a good option, a report written in ModMark could be converted directly to both a PDF and an accessible HTML document if there is proper package support for it.

As PDFs do not have much support for accessibility it is difficult to accommodate everyone when generating LaTeX. However, when generating HTML there are a lot of things that can be done to assist people with disabilities or other barriers. ModMark attempts to generate semantic HTML to give as much information as possible about the different parts of the documents. For example `<figure>` tags are used for images and plots, `<table>` is used for tables, and paragraphs are wrapped in `<p>` tags. The `files` package that is responsible for converting images will also give a compiler warning if no alt-text is given when converting to HTML.

5

Conclusion

This chapter summarises our achievements and discusses how we aim to extend and improve ModMark in the future.

5.1 Summary

ModMark has been successfully implemented as evident by the fact that this document is written in ModMark. The aims set for the project in Section 1.2 have been reached as outlined in Chapter 4. While the language is not perfect, the flexible and expressive package system utilising WebAssembly allows for many interesting possibilites.

The fact that the web playground and the command-line interface uses the same codebase demonstrates that WASM can be used to create complex portable applications. Embedding a WASM runtime to evaluate packages also shows that plugin and packages systems can use WASM to provide more flexibility for developers.

The process of implementing and designing ModMark has also highlighted some of the current limitations of WASM and WASI. As mentioned in Section 4.3 some of the most popular languages like Python and JavaScript do not have good support for WASM due to their large runtimes. As previously mentioned there are proposals that will hopefully alleviate this issue but it is clear that WASM and WASI are still new technologies and the ecosystem will continue to change rapidly. If this project was to be done in a few years time it is likely that there would be new approaches available.

5.2 Future work

ModMark will continue to be developed beyond the scope of this project. The compiler is released as free and open source software[1] and we welcome new contributors. Due to the modular nature of the language most missing features can be implemented in the form of third-party packages. However, there are some noteworthy topics closely tied to the language design and the compiler implementation of ModMark that could be further explored too.

[1] Licenced under the Apache Licence version 2.0.

5.2.1 Incremental compilation

A potential downside of the modular design of ModMark in combination with the fact that packages are evaluated in WebAssembly virtual machines is slow compilation speeds. In order to improve the speed at which a document is transcompiled, incremental parsing and incremental compilation techniques could be further investigated.

Incremental compilation is a technique that attempts to reuse parts of the previous output depending on how the source text was changed. It reduces the time spent by avoiding a full recompilation. In a system where all elements are independent of each other it can be achieved by simply checking which elements were changed. However, in Modmark the situation is more complex, as our modules have access to both variables and files. For instance, a module outputting images from a file may appear unchanged to the compiler but the previous output cannot be reused unless the file itself is unchanged. Similarly, the compiler may need to recompile a module with variable access if any of the variables have different values.

Incremental parsing is, in a sense, the same concept implemented at an earlier stage. While incremental compilation performs checks after the abstract syntax tree has been produced, incremental parsing may avoid producing the tree itself. Typst is one of the languages that feature both incremental parsing and compilation [16]. For incremental parsing, Typst uses an algorithm based on the so called *red-green tree* which keeps track of dependencies between elements and invalidates them as needed. For incremental compilation, it uses a cache that is queried for reusable entries. While Typst and ModMark have vastly different output formats, it is likely that similar ideas could be used to implement incremental parsing and compilation for ModMark.

5.2.2 Diagnostics

Good compiler diagnostics that help users when they encounter issues is essential for an ergonomic user experience. The benefits of unobtrusive syntax quickly disappear if the user gets stuck as soon as they make a minor mistake. In the related world of programming language research, results from a study have shown that students' learning is hindered by ineffective error messages [41]. The study also presents a timeline of efforts in enhancing error messages as well as guidelines on how they should be structured.

LaTeX, despite its popularity, produces error messages that are arguably complex and unhelpful to inexperienced users [15]. On the other hand, Typst and programming languages like Elm [42] and Rust [43] has made more recent attempts at providing helpful error messages that reference the source code and attempt to explain the error. An example of this is found in Figure 5.1.

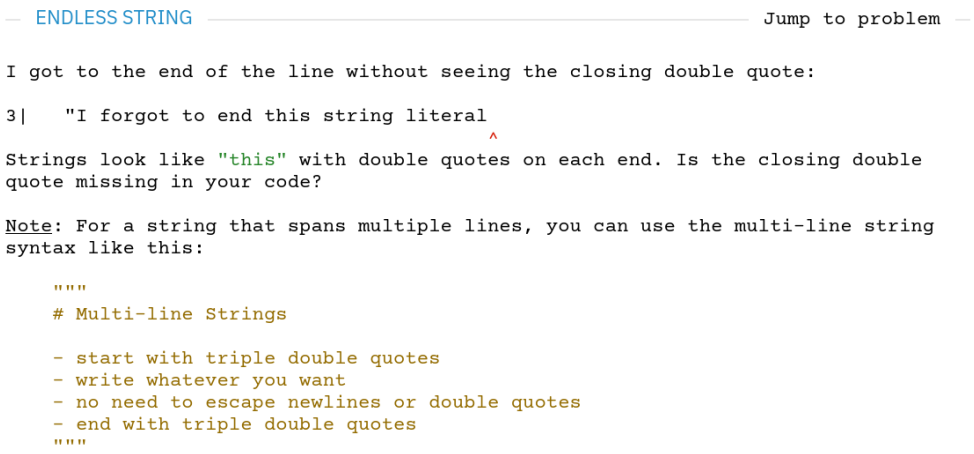

Figure 5.1: An example of an error message produced by the compiler for the Elm programming language.

We could draw inspiration from these languages in order to improve the the error messages produced by the ModMark compiler. The most significant change would be a rewrite of the parser so that each node in the AST stores a reference to the position in the source text, meaning that error messages can describe where the error occurred. Nom, the parsing library that is currently used, is somewhat ill-suited for this task so it might be appropriate to consider writing hand-written recursive-decent parser or opting for another parser combinator library such as *Chumsky* [44] which has a stronger focus on error recovery.

There are several other aspects regarding error messages in ModMark that are worth exploring further too. The standard packages included with the language were developed with helpful errors messages in mind, but there are no such guarantees for third-party packages. In addition to this, package developers need useful errors that can help them with debugging. This becomes especially significant when considering the importance of our package ecosystem. In the future, it may be ideal to author a convention for package errors and also offer a separate tool specifically for package development.

5.2.3 Editor support

Another feature that could be explored is building a WYSIWYG (*What You See Is What You Get*) editor for ModMark. More recently, editors and knowledge managements systems like Notion [45] and Microsoft Loop [46] have taken a different approach to rich text editing where a document consists of *blocks* with different kinds of content: text, schematics, interactive elements and other embedded documents. ModMark could potentially be well suited for such an interface since modules allow vastly different types of content to be mixed in a portable manner.

Apart from building a stand-alone editor for ModMark better integration for other

text editors could also be developed. We have developed a TextMate grammar that can be used to provide syntax highlighting for multiple editors and is used in a Microsoft Visual Studio Code extension. However, there is still room for improvement. For instance, integrating a server with support for the Language Server Protocol (LSP) [47] directly in the ModMark command-line tool is one possible direction.

5.3 Further reading

More information about ModMark as well as the source code for the compiler can be found at the following online resources. This thesis is also available as an HTML document.

- **Source code**
  https://github.com/modmark-org/modmark
- **This thesis as an HTML document**
  https://modmark-org.github.io/modmark-thesis/
- **ModMark website**
  https://modmark.org/
- **Web playground**
  https://modmark.org/#/playground
- **Repository for this thesis and the *chalmers-thesis package***
  https://github.com/modmark-org/modmark-thesis
- **Maths notes**, from Figure 4.1, editable in the Playground
  https://modmark.org/#/playground?gist=788fb98f4919327ab5d53ddcee00cdb9
- **Three Knights Game**, from Figure 4.1, editable in the Playground
  https://modmark.org/#/playground?gist=29b40792a008c3ce3b25a65c20a8c78b
- **Snapshot of this thesis**, from Figure 4.1, editable in the Playground
  https://modmark.org/#/playground?gist=530d97a6746d2d05c4df2af6ba7b9b1e
- **Example Swift package**
  https://github.com/CMDJojo/modmark-lorem
- **Example C++ package**
  https://github.com/hugomardbrink/modmark-prettify
- **Example Go package**
  https://github.com/forsinge/modmark-fen
- **Example Zig package**
  https://github.com/axoxelol1/modmark-robber
- **Example AssemblyScript package**
  https://github.com/bruhng/modmark-vigenere

Bibliography

[1] G. O. Walter, "Typesetting," *Scientific American*, vol. 220, no. 5, pp. 60–69, 1969, doi: 10.1038/scientificamerican0569-60.

[2] GNU, "The gnu troff manual," 2018. Accessed: Jan. 25, 2023. [Online]. Available: https://www.gnu.org/software/groff/manual/groff.html

[3] M. D. McIlroy, "A research unix reader: annotated excerpts from the programmer's manual, 1971-1986," Bell Labs, 1987. Accessed: Mar. 21, 2023. [Online]. Available: https://www.cs.dartmouth.edu/~doug/reader.pdf

[4] C. Thiele, "Knuth meets NTG members," 1996. Accessed: Jan. 25, 2023. [Online]. Available: https://www.ntg.nl/maps/16/15.pdf

[5] D. Knuth, *Computers & Typesetting, Volume A: The TEXbook*, Addison-Wesley, American Mathematical Society, 1986.

[6] M. Kerrisk, "Linux man-pages," 2023. Accessed: Mar. 7, 2023. [Online]. Available: https://www.man7.org/linux/man-pages/

[7] L. Lamport, *LATEX: A Document Preparation System*, 2nd ed., Addison-Wesley, 1994.

[8] D. Goodger, "An Introduction to reStructuredText," 2022. Accessed: Mar. 21, 2023. [Online]. Available: https://docutils.sourceforge.io/docs/ref/rst/introduction.html

[9] Eclipse Foundation, "AsciiDoc." Accessed: Jan. 23, 2023. [Online]. Available: http://asciidoc.org/

[10] Asciidoctor Project 2022, "Asciidoctor." Accessed: Mar. 6, 2023. [Online]. Available: https://asciidoctor.org/

[11] D. Allen, S. White, et al., "Asciidoctor extensions." Accessed: May 11, 2023. [Online]. Available: https://docs.asciidoctor.org/asciidoctor/latest/extensions/

[12] J. Gruber, "Markdown," 2004. Accessed: Jan. 25, 2023. [Online]. Available: https://daringfireball.net/projects/markdown

[13] J. MacFarlane, "Commonmark spec," 2021. Accessed: Mar. 21, 2023. [Online]. Available: https://spec.commonmark.org/0.30/

[14] J. MacFarlane, “Djot.” Accessed: Jan. 23, 2023. [Online]. Available: https://djot.net/

[15] L. Mädje, “Typst: a programmable markup language for typesetting,” Thesis, Technical University of Berlin, 2022. Accessed: Jan. 25, 2023. [Online]. Available: https://www.user.tu-berlin.de/laurmaedje/programmable-markup-language-for-typesetting.pdf

[16] M. E. Haug, “Fast typesetting with incremental compilation,” Thesis, Technical University of Berlin, 2022. Accessed: Apr. 8, 2023. [Online]. Available: https://www.user.tu-berlin.de/mhaug/fast-typesetting-incremental-compilation.pdf

[17] L. Phillips, “Technical writing with pandoc and panflute,” 2021. Accessed: Apr. 11, 2023. [Online]. Available: https://lee-phillips.org/panflute-gnuplot/

[18] A. Jargas, “Txt2tags.” Accessed: Apr. 11, 2023. [Online]. Available: https://txt2tags.org/

[19] “Git documentation.” Accessed: Mar. 7, 2023. [Online]. Available: https://git-scm.com/doc

[20] “Sphinx.” Accessed: Mar. 6, 2023. [Online]. Available: https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html

[21] A. Rossberg, Ed., “WebAssembly Core Specification,” W3C, Rep. 2.0, Apr. 19, 2022. Accessed: Apr. 16, 2023. [Online]. Available: https://www.w3.org/TR/wasm-core-2/

[22] Wasmer, Inc., “Wasmer.” Accessed: Jan. 23, 2023. [Online]. Available: https://wasmer.io/

[23] Bytecode Alliance, “Wasmtime.” Accessed: Jan. 23, 2023. [Online]. Available: https://docs.wasmtime.dev/

[24] “WASI: The WebAssembly System Interface.” Accessed: Jan. 23, 2023. [Online]. Available: https://wasi.dev/

[25] “GitHub.” Accessed: Mar. 3, 2023. [Online]. Available: https://github.com/

[26] S. D. Swierstra, “Combinator parsing: a short tutorial,” *Language Engineering and Rigorous Software Development: International Lernet ALFA Summer School 2008, Piriapolis, Uruguay, February 24-March 1, 2008, Revised Tutorial Lectures*, pp. 252–300, 2009.

[27] G. Couprie, “Nom, a byte oriented, streaming, zero copy, parser combinators library in rust,” in *2015 IEEE Security and Privacy Workshops*, 2015, pp. 142–148. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7163218

[28] B. Dowling, “An introduction to LaTeX,” 1996. Accessed: Apr. 26, 2023. [Online]. Available: http://kcgs.soc.srcf.net/sites/default/files/computing/LaTeX/LaTeX.pdf

[29] D. J. Pearce, and P. H. Kelly, "A dynamic topological sort algorithm for directed acyclic graphs," *Journal of Experimental Algorithmics (JEA)*, vol. 11, pp. 1–7, 2007.

[30] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[31] D. Hockley, and C. Williamson, "Benchmarking runtime scripting performance in wasmer," in *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, 2022, pp. 97–104.

[32] G. D. Greenwade, "The Comprehensive TEX Archive Network (CTAN)," *Tugboat*, vol. 14, no. 3, pp. 342–351, 1993. [Online]. Available: https://www.tug.org/TUGboat/tb14-3/tb40green.pdf

[33] Ubuntu community help wiki, "LATEX," 2013. Accessed: Apr. 27, 2023. [Online]. Available: https://help.ubuntu.com/community/LaTeX

[34] Deno Land Inc., "Deno manual." Accessed: Apr. 26, 2023. [Online]. Available: https://deno.com/manual@v1.32.5/node/npm_specifiers

[35] G. Booch, R. Maksimchuk, et al., *Object-Oriented Analysis and Design With Applications*, Addison-Wesley Professional, 2007.

[36] S. Spencer, "Preparing Your Article with LATEX," 2022. Accessed: May 15, 2023. [Online]. Available: https://authors.acm.org/proceedings/production-information/preparing-your-article-with-latex

[37] Wasmer Team, "Wasmer features." Accessed: May 15, 2023. [Online]. Available: https://docs.wasmer.io/ecosystem/wasmer/wasmer-features

[38] S. Akbary, "Wasienv: wasi development toolchain for c/c++," 2019. Accessed: May 15, 2023. [Online]. Available: https://medium.com/wasmer/wasienv-wasi-development-workflow-for-humans-1811d9a50345

[39] GHC Team, "Using the GHC WebAssembly backend," 2023. Accessed: May 4, 2023. [Online]. Available: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/wasm.html

[40] L. L. Wang, I. Cachola, et al., "Improving the accessibility of scientific documents: current state, user needs, and a system solution to enhance scientific pdf accessibility for blind and low vision users," *Arxiv Preprint*, 2021, doi: 10.48550/arXiv.2105.00076.

[41] B. A. Becker, P. Denny, et al., "Compiler error messages considered unhelpful: the landscape of text-based programming error message research," *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, pp. 177–210, 2019, doi: 10.1145/3344429.3372508.

[42] E. Czaplicki, "Compiler errors for humans," 2016. Accessed: May 9, 2023. [Online]. Available: https://elm-lang.org/news/compiler-errors-for-humans

[43] J. Turner, “Shape of errors to come,” 2016. Accessed: May 9, 2023. [Online]. Available: https://blog.rust-lang.org/2016/08/10/Shape-of-errors-to-come.html

[44] J. Barretto, “Chumsky: a parser library for humans with powerful error recovery.” [Online]. Available: https://github.com/zesterer/chumsky

[45] Notion, “What is a block?.” Accessed: May 10, 2023. [Online]. Available: https://www.notion.so/help/what-is-a-block

[46] W. McKelvey, “New Microsoft Loop app is built for modern co-creation,” 2023. Accessed: May 9, 2023. [Online]. Available: https://www.microsoft.com/en-us/microsoft-365/blog/2023/03/22/new-microsoft-loop-app-is-built-for-modern-co-creation/

[47] Microsoft, “Language Server Protocol Specification - 3.17,” 2022. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/