HyloDoc

Automated Code Documentation Generation for the Hylo Language

CSE2000: Software Project

Markas Aišparas, Alex Cazacu, Eviatar Hadasi, Ambrush Tóth, Thomas van Weert

Delft University of Technology

TU Delft hylo

HyloDoc

Automated Code Documentation Generation for the Hylo Language

Instructor:	Regina Hoffmann
Teaching Assistant:	Ruben Backx
Coach:	Alex Voulimeneas
Project Duration:	May, 2024 - June, 2024
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science

Cover:	A photograph of a spider web, symbolizing the intricate interconnections within documentation. Source: Pixabay.
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld, and Thomas van Weert

Preface

This report was written by a team of five computer science students from Delft University of Technology during our Year 2 software project. Over nine weeks, we developed a documentation extraction tool tailored for the Hylo programming language. The project allowed us to practice the skills and tools we learned in the program, such as effective teamwork and technical writing, in a real-world setting.

After learning about Hylo from conference talks and podcasts and being fascinated by its novel approach to memory safety and efficiency, we reached out to the Hylo Community to see if we could contribute to the Hylo ecosystem with a project that we could complete in a quarter. After brainstorming possible ideas with Dimitri Racordon, we formulated a project proposal that was subsequently accepted by the university and ultimately completed successfully.

We would like to thank Dimitri Racordon, the founder of Hylo, for her dedicated research and contributions to the language and its compiler. Also, thanks to Lucian Teodorescu and Dave Abrahams for their feedback. Special thanks to Ruben Backx for his assistance and thorough feedback and to Alex Voulimeneas for sharing his experiences and tips at the start of the project.

While writing this report, we assumed the reader to have basic programming knowledge and be familiar with the MarkDown syntax.

Readers particularly interested in the design of the syntax and the tool will find that in Chapter 4, Product Design. Readers more interested in the implementation details and performance considerations of compilers will find Chapter 7, Implementation insightful.

Markas Aišparas, Alex Cazacu, Eviatar Hadasi, Ambrush Tóth, Thomas van Weert
Delft, August 2024

This text was written within the course CSE2000 Software Project of the faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology.

Total word count: 14814

Statement on the use of generative AI:
In writing this text, we used generative AI to improve the text's grammar, style, and spelling.
See Appendix D for details.

Summary

The HyloDoc report details the development of an automated documentation generation tool for the Hylo programming language targeted for memory safety, efficiency and simplicity. Initially, Hylo lacked a standardized system for creating and maintaining documentation, and the standard library's documentation was unavailable on the website.

HyloDoc was developed to address this issue by automating the extraction of documentation from the codebase and generating a comprehensive, user-friendly documentation website. The tool comprises two main components: documentation extraction and website generation. The extraction component scans the source code for comments and relevant information and stores it in a documentation database. The website generation component processes this data into HTML pages to form a complete and navigable website.

The development process began with a detailed problem analysis, identifying the lack of automated documentation in Hylo and the challenges it posed for stakeholders. The requirements were gathered and prioritized using the MoSCoW method. A Markdown-based syntax was developed for writing documentation, ensuring ease of use and consistency. Ethical considerations such as accessibility and inclusivity were integral to the design.

Implementation involved creating a robust architecture for HyloDoc, covering the processes of documentation extraction and website generation. Rigorous testing, including unit and integration tests, ensured functionality and reliability. A user study provided valuable feedback, leading to refinements that enhanced the tool's usability. Performance has been improved to ensure a seamless and scalable experience.

HyloDoc significantly improves the documentation process for Hylo, making it easier to maintain and more accessible to users. The tool helps to keep documentation up-to-date with code changes, eliminating manual effort and errors. The generated website offers a clear and structured format, enhancing the user experience. This advancement supports the growth and adoption of the Hylo language, providing a reliable solution for maintaining high-quality documentation.

Contents

Preface i

Summary ii

1 Introduction 1

2 Problem Analysis of Effective Documentation 2
- 2.1 Problem Statement . . . 2
- 2.2 Stakeholders . . . 2
- 2.3 Use Cases . . . 3
- 2.4 Common Practices of Documentation . . . 4
- 2.5 Comparison of Alternatives . . . 4

3 Product Requirements 5
- 3.1 Functional Requirements . . . 5
- 3.2 Non-Functional Requirements . . . 6

4 Product Design 7
- 4.1 The Documentation Syntax and Conventions . . . 7
  - 4.1.1 Documentation Organization . . . 7
  - 4.1.2 Syntax of API Documentation . . . 8
  - 4.1.3 Special Sections in API Documentation . . . 10
  - 4.1.4 Syntax of links . . . 11
- 4.2 The Generated Website . . . 12
- 4.3 The HyloDoc Toolset . . . 12

5 Ethical Considerations about Inclusivity, the User Study, and a Political Controversy 14
- 5.1 Accessibility of Web Content . . . 14
- 5.2 Ethical Considerations for the User Study . . . 14
- 5.3 A Controversial Library . . . 15

6 Development Methodology 16
- 6.1 Work structure and communication . . . 16
- 6.2 Version Control and Progress tracking . . . 16
- 6.3 Continuous Integration . . . 17

7 Product Implementation 18
- 7.1 Product Architecture . . . 18
  - 7.1.1 Documentation Extraction . . . 19
  - 7.1.2 Documentation Database . . . 20
  - 7.1.3 Website Generation . . . 21

8 Testing 22
- 8.1 Unit Testing . . . 22
- 8.2 Integration Testing . . . 22
- 8.3 Confidence in Correctness . . . 23
- 8.4 Error Handling Strategies . . . 23

9 User-centered evaluation 25
- 9.1 User Evaluation . . . 25
- 9.2 User Evaluation Results . . . 25
  - 9.2.1 The navigation tree . . . 26
  - 9.2.2 Other suggestions . . . 26

10 Performance Analysis and Optimisation **28**
10.1 Performance Improvements in Website Generation 28
10.2 Comprehensive Lighthouse Analysis and Optimisation 31

11 Conclusion **34**

References **35**

A Requirements **36**

B Syntax Reference of Documentation **39**

C Details of API Syntax Design **45**

D The Use of Generative AI Tools for the Project **51**

E Syntax of Links **53**

F Design of the Generated Website **57**

G Performance Measurements **61**

H HREC Checklist **62**

I Questionnaire for Potential Developers **64**

J Contributor Questionnaire **74**

K Consent Form **84**

L Chrome Lighthouse Report After the Optimisations **88**

M Distribution of labour **93**

N TW Feedback **94**

1 Introduction

Effective documentation is crucial for software engineering. Without it, understanding the functionality of the software can be challenging, especially when helpful comments are lacking. Industry best practices advocate for documenting software as you write it [1], and so various technologies have been developed. Those technologies are used both to standardise and display the comments within code editors, and to generate a searchable reference website. Hylo, a new programming language still in development, lacked a conventional syntax for writing documentation and did not have public documentation for its standard library.

This report details the development of HyloDoc, a documentation generation tool for Hylo designed to fill this gap and help understand how Hylo functions. HyloDoc leverages the existing code processing tool and code comments to automatically generate a comprehensive documentation website. By analysing existing solutions in other languages and reviewing related scientific literature, a design was made for suitable syntax and tool architecture.

The report is structured as follows: Chapter 2 analyses the problem, including stakeholders, use cases, and research. Chapter 3 outlines the requirements. Chapter 4 discusses the product design, covering the documentation syntax and the website. Chapter 5 addresses ethical considerations. Chapter 6 explains the development methodology. Chapter 7 details the implementation, including architecture, documentation database, extraction processes and website generation. Chapter 8 covers the testing methodology. Chapter 9 presents a user-study conducted for the website. Finally, Chapter 11 concludes the report, including recommendations for future development.

2

Problem Analysis of Effective Documentation

"A Problem well stated is half solved." (quote usually attributed to Charles Kettering, who was head of research at General Motors from 1920 to 1947). This chapter clarifies the problem the project is trying to solve and the basic premises required to understand and derive the product requirements in Chapter 3. The chapter starts off by introducing what is the problem (2.1) and who are the stakeholders (2.2). After this, the use cases for the product are discussed (2.3), followed by some common practices (2.4) related to the problem that should be adhered to. Lastly, some alternatives and their takeaways are analyzed (2.5).

2.1. Problem Statement

Hylo was missing an infrastructure for creating documentation from code comments. The language lacked a standardized documentation syntax, a framework for organizing documentation, and a way to present this documentation in the form of a website or other accessible format.

Due to this, the utilization of the Application Programming Interfaces (APIs) written in Hylo presented a significant challenge. The learning process was heavily reliant on the availability and quality of various forms of documentation, including code comments and other related materials.

2.2. Stakeholders

The tool and the syntax it requires have several different user groups with varying needs. Those groups are the creators and contributors of Hylo; as well as any present or future developers who would like to employ the Hylo language in their projects.

Requirement Elicitation

The requirements were derived from various sources, including meeting notes, interviews, and discussions with *Dimi Racordon* (a programming languages expert researcher and the lead developer of the Hylo compiler), *Dave Abrahams* (Hylo contributor, lead developer of Swift's standard library) and the general Hylo Slack Community.

The following is a list of each of the different stakeholders and their requirements:

- **Maintainers of the Hylo standard library** should be able to document APIs and any type of named symbol using a consistent, easy-to-read and write format. It should be possible to refer to code symbols easily from the documentation.
- **External developers** have at least the same expectations and needs from the syntax and the tooling as the contributors of the standard library. However, the latter group is usually intrinsically

motivated to complete a steeper learning curve, and they can live with more usability issues without abandoning the language. That is not the case for external contributors. If they find the syntax unintuitive or the tooling hard to utilize, they can easily lose interest and choose another programming language or not follow the established syntax in their codebase. Therefore, syntax features must be logical, intuitive (or at least familiar), and easy to discover. It should be hard to make mistakes - everything should parse just as expected, without visually ambiguous rules. The tooling should be easy to set up and configure.

- **Regular Hylo developers who are just viewing the documentation website** can be classified based on the purpose of their interaction [2]:
  - **Information Seekers** are people who are looking for a specific piece of information. They need to be able to find what they are seeking fast: either by an internet search or using the navigation features of our website. The presented content needs to have structure, good hierarchy, and consistency.
  - **Information Stumblers** are people who want to get an overview of a certain topic, just having a rough idea of their goal. They need to be able to familiarize themselves with concepts quickly without looking at too many details. Introduction pages, tutorials, and conceptual documentation can be very useful to them. People who are just learning about the language for the first time most likely fall into this category and have similar requirements for accessing the website.

2.3. Use Cases

Two distinct use cases exist for using the tool gathered from the stakeholders. The first is a more general use case of the tool applicable to most users. The second is a use case that is more project-specific but still something the tool should be able to do.

Documenting Libraries

The ability to access comprehensive, structured, and reliable reference documentation for both the standard library and external libraries is crucial for the adoption and usability of a language. Users typically expect to find official documentation when searching for information about a library's type, function, or API online. The absence of such resources can become a barrier, potentially leading to a loss of initial excitement when learning a new language or library. Therefore, library authors (including the authors of the standard library) want to generate reference documentation websites for their projects.

Inline tooltip information in an IDE is particularly important, as it brings this information as close to the user as possible, enhancing the learning experience. When Hylo gets a language server, this functionality can be implemented using our tooling as a library.

Generating Older Versions of the Documentation

Hylo's philosophy encourages users to always use the latest version of the language, removing barriers to upgrading. Therefore, the tool will only need to deploy the latest version of the documentation to the public website after each successful merge to the compiler's repository. However, users will have the option to go back to any particular commit, download the sources, and generate the documentation for it.

2.4. Common Practices of Documentation

The syntax of a programming language alone is not enough as documentation, as the location of documentation also matters. This is especially true when a project grows in size. This is part of the reason why Google migrated all their documentation into its version control system. By doing so, they could identify the person responsible for a piece of code and its documentation [2], so asking for reviews and further information became easy. In the Glasgow Haskell Compiler, they introduced Notes authored inside code comments, which could be easily referred to from anywhere in the code by name. It is also possible to see the usages of the note, so when updating it, one can easily check whether they broke assumptions somewhere else.

Over the last decades, it became a norm to document library APIs inside the source code, using code comments or annotations near the symbol they are describing. However, this is not yet true for regular, conceptual documentation articles and notes related to the code base: people often use online collaboration tools such as Confluence or GitHub / GitLab Wiki, which come with a strong disconnect from the code base: it is impossible to refer to code entities from the documentation, and related documentation articles from the source code without the references becoming obsolete over time.

2.5. Comparison of Alternatives

There are many available products similar in functionality that allow programmers and other users to generate documentation. The problem with most of these tools is they rely heavily on the language they are meant for, to be able to access the needed information to generate the documentation. Doxygen is an exception to this: it supports many languages, implementing custom parsers for each of them. However, because of its large code base with a fairly complex architecture, and the limitations of its design related to documentation articles, it was deemed as too generalized solution. Instead, a specialized solution was developed by learning from previous technologies. Below are two key takeaways from DocC, the documentation generation tool for the Swift programming language.

- **Formatting:** DocC allows developers to write their documentation using a variant of Markdown, which supports a variety of rich text elements, making it highly readable and easy to write. JavaDoc uses HTML tags within comment blocks, which can be a bit more cumbersome. Doxygen supports both Markdown and HTML, providing flexibility to the developers.
- **Design:** The website generated by DocC offers a clean, modern design with features like tree-based navigation and a search bar, enhancing usability. JavaDoc also generates a well-structured website, but its design can be considered more utilitarian. The design of Elm's website is clean and minimalistic, but it still supports search and basic navigation features.

The conclusion from looking at these alternatives is for the constraints set upon the project by the client, an already existing product cannot be adapted to solve the lack of documentation tooling for Hylo. By defining who will be needing such tooling and in what way they would use it, a set can be distilled of functional and non-functional requirements for the project, which will be presented in Chapter 3.

3

Product Requirements

Establishing the requirements and priorities is a fundamental step before commencing the implementation of any project. For this project, the MoSCoW model [3] was employed to structure the requirements, categorizing them into four priority levels: Must, Should, Could, and Won't. The MoSCoW model [3] was chosen for its simplicity and flexibility, allowing it to prioritize effectively without the constraints of fixed durations or deadlines. This chapter outlines the major functional (3.1) and non-functional (3.2) requirements that drove the project, providing a foundation for the subsequent design and development chapters.

3.1. Functional Requirements

Must-Have Requirements

The essential functionalities that the tool must provide are as follows:

- **Comprehensive API Information Extraction:** The tool must extract API information from all named symbols in a Hylo source file, including traits, type-aliases, product types, variable bindings, initializers, functions, methods (and method bundles), subscripts (and subscript bundles), properties, operators, associated types, and associated values. Comprehensive extraction is crucial for establishing usable documentation.
- **Documentation Comments Extraction:** The tool must extract API documentation comments from the recorded symbols where present. This technique is well-established for ensuring maintainability and ease of editing.
- **Website Generation with Hyperlinks and Inherited Documentation:** The tool must generate a basic website to display the extracted documentation, including hyperlinks for symbols and inherited documentation for conforming product types and child traits. A web interface is crucial for users to easily access and navigate the documentation.
- **Scope-Aware Name Resolution:** The tool must handle references to any Hylo symbol with scope-aware name resolution. Correctly resolving names and references ensures that documentation links are accurate yet concise.
- **Method Bundle Refinement and Article Documentation:** The tool must allow extending the documentation of a method bundle for each inner declaration and support writing documentation in markdown articles.
- **Command-Line Interface (CLI):** The tool must provide a command-line interface for user interaction. A CLI ensures that users can interact with the tool in a flexible and scriptable manner, facilitating automation and integration into existing workflows such as GitHub Actions.

Should-Have Requirements

These functionalities are highly desirable and would significantly enhance the tool:

- **Enhanced Navigation and Validation:** The website should have tree navigation and a symbol search feature, and the tool should check for broken references to symbols and linked files before generating the website. These features improve user experience and maintain the integrity and reliability of the documentation.

3.2. Non-Functional Requirements

The project must also meet several non-functional requirements to ensure quality and maintainability:

- **Integration with Hylo Compiler:** The tool needs to use the Hylo compiler's frontend as a library. This integration is crucial for accurate extraction and analysis of Hylo code.
- **Implementation in Swift:** The tool must be written in Swift. Integrating the Hylo compiler's frontend with a different language would cause complications, such as the need for a bridging layer, which would waste development time and compromise Swift's type-safety. Additionally, Swift aligns with the expertise of the Hylo contributors, who will be responsible for operating and maintaining the tool.
- **Accessibility and Performance:** The documentation must be accessible without the use of JavaScript, and the website must achieve a Google Chrome Lighthouse score of at least 90% (preferably 95%) in all categories, including performance, accessibility, search engine optimization (SEO), and best practices. Ensuring both accessibility and high performance is essential to support the productivity of developers and reach a wider audience.

For the new documentation syntax, we must meet the following constraints:

- **Rich Text and Familiarity:** It should support rich text in descriptions, including code examples, bold and italics text formatting, images, and links to URLs, documentation articles, and code symbols. Learning new syntax for documentation can overwhelm many people, so it should be as familiar as possible.
- **Low Confusability and Readability:** If we reuse some markup language but modify its syntax or requirements for it to parse in unexpected ways, it might cause confusion. Mistakes should be visually apparent and easily detectable. The source code of the documentation should be readable and well-structured.
- **Feature Discoverability:** The syntax should make it easy to discover language features through auto-complete (e.g., at certain trigger characters).

These requirements provide a clear roadmap for developing a tool that effectively extracts and displays API documentation for Hylo source files. Adhering to these prioritized requirements ensures that the tool meets essential functionality, usability, and performance standards.

The requirements were derived from various sources, including meeting notes, interviews, and discussions, as described in 2.2. This collaborative approach ensured that the requirements are comprehensive and aligned with user needs.

For a detailed list of all requirements, including their implementation status, please refer to appendix A. This list offers a complete overview of the project's scope and progress.

With the requirements now clearly defined, the next step is to translate these needs into a concrete product design. The following chapter will delve into the design considerations and decisions that shaped the development of the tool, ensuring it meets the outlined requirements.

4

Product Design

Using the requirements presented in chapter 3, we designed a syntax and conventions for authoring documentation and a tool that generates a searchable and easily accessible website. The following sections introduce the design of this syntax (4.1), the generated website (4.2), and finally the documentation extractor tool (4.3).

4.1. The Documentation Syntax and Conventions

4.1.1. Documentation Organization

The flexibility of a documentation system plays a vital role in achieving a maintainable documentation base, which is the key reason why Google migrated all types of documentation into a single system [2].

In HyloDoc, documentation can be authored in three forms (as shown in Figure 4.1):

- **API documentation** can be embedded inside `.hylo` source files, in comments above the symbol they describe.
- **Standalone articles** can be written in Markdown format in `.hylodoc` files. Their use cases include tutorials, landing pages, getting-started guides, conceptual documentation and wiki pages.
- **Supplementary content**, such as images, PDFs, and scripts, can be attached as files in the repository and linked or embedded from documentation content.

It's important to keep API documentation close to its declaration to prevent it from becoming outdated or out of sync with the code. However, we cannot put everything in the source files because long explanations and examples make the source code harder to navigate and edit. This can be seen when looking at Swift's standard library code base.[1]

To solve this issue, HyloDoc uses a similar approach to the Glasgow Haskell Compiler's code base, introduced in the Architecture of Open Source Applications [4]. They made it possible to create notes in code comments and seamlessly refer between them, searching for usages. Much like code, refactoring and exploration of documentation became much simpler thanks to tooling, which has contributed to its successful maintenance for over three decades.

In HyloDoc, notes are represented as standalone *article files*, which can be written with `*.hylodoc` extension in MarkDown format. This design ensures that there are no name conflicts and that the title can be changed freely without needing to refactor all links. allows developers and technical writers to use existing authoring tools such as the built-in markdown editors of IDEs or external tools. In this case, they can use plain markdown; no special tags will be considered in the titles.

A source file as a whole can also be documented using a *file-level* comment block. This is useful when a file has multiple top-level declarations, and there is a common theme between them that needs

[1] `https://github.com/apple/swift/blob/main/stdlib/public/core/Collection.swift`

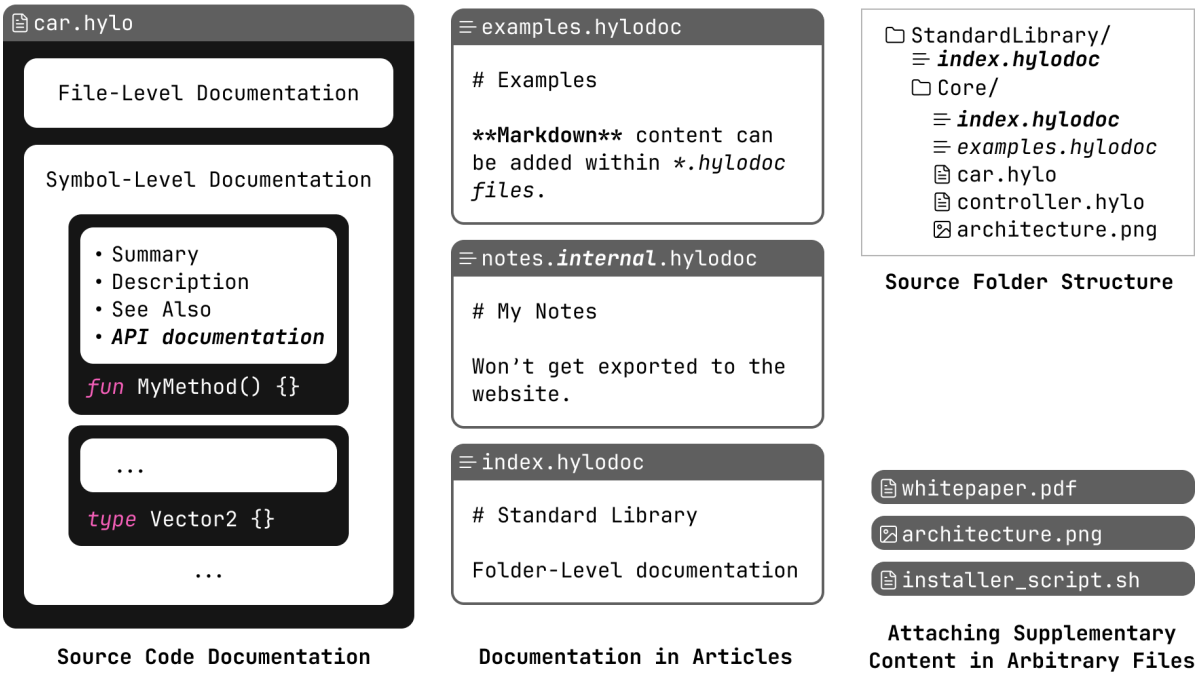


Figure 4.1: Overview of Documentation Organization: Documentation can be created as articles or within source files. Within a source file, it can describe individual symbols or provide documentation for the entire file.

to be documented.[2]

Documenting folders is done by creating `index.hylodoc` files in them. This can be used as a landing page by providing a brief introduction for the contents of the folder or module, giving a quick overview for information *stumblers*.

Internal articles can be created by writing them inside `*.internal.hylodoc` files. The articles are only meant to be read by the developers of the product or library, thus their files are ignored when generating the website. Internal documentation can be written inside source files by documenting non-public symbols since only public ones are exported on the website. The `# Internal File-Level` annotation can be used to mark file-level documentation internal - this can be useful if there are no public declarations in the file, but it should still be documented internally.

4.1.2. Syntax of API Documentation

The team explored a wide variety of possibilities for encoding the structured information about symbols into comments and compared different comment styles whose informal design process can be seen in Appendix D.

API reference documentation can be embedded inside source files in the form of contiguous blocks of line comments with 3 leading slashes and 1 space as shown in Figure 4.2. The content can be authored as a markdown text, but a standard format and interpretation of it were introduced, which allows reliable tooling to be built for it.

The syntax is fundamentally similar to Swift DocC's convention: the first short paragraph defines the summary, then comes a possibly multi-paragraph rich-text description, and finally, zero or more optional metadata sections depending on the type of documented symbol in question. These sections are identified by being first-level headings, which here are reserved for the special sections. These might appear before any named code entity declaration (except function- or local-scope variable bindings): *trait, type-alias, product type, function, subscript, initializer, deinitializer, property, and variable binding declarations.*

[2]Operators in the Hylo Standard Library: `https://github.com/hylo-lang/hylo/blob/main/StandardLibrary/Sources/Core/Operators.hylo`

```
/// First paragraph is the summary.
///
/// Markdown __formatting__ is *supported* in
/// the description.
///
/// It can also be multiple paragraphs long.
///
/// # Parameters:
///   - width:  The width of the rectangle.
///   - height: The height of the rectangle.
///
/// # Preconditons:
///   - width < height - the rectangle should be
///     longer vertically than horizontally
///   - width and height must be positive
///
/// # Returns:
///   - The calculated area if the width is even
///   - 0 otherwise
///
/// or inline:
/// # Returns: the area of the rectangle
/// # Parameter width:  Width of the rectangle
/// # Parameter height: Height of the rectangle
fun area(width: Int, height: Int) -> Int {
    return width * height;
}
```

Figure 4.2: Example of the API Documentation Syntax

```
/// First paragraph is the summary.
///
/// - Parametrs:                          // <- This should have been an error
///   - width:  The width of the rectangle.
///   - height: The height of the rectangle.
func area(width: Int, height: Int) -> Int {
    return width * height;
}
```

Figure 4.3: Swift's Documentation Syntax

A notable change from Swift's documentation syntax is the use of first-level headings (`#`) for introducing special sections. In Swift, regular Markdown lists are used for this purpose (see Figure 4.3). There are multiple reasons for this diversion:

- **Semantic meaning:** *Parameters, Returns, Preconditions,* etc. are titles of special sections, so it was logical to reflect this in the syntax.
- **Less ambiguity:** Using first-level headers is forbidden so that slight mistypes are caught as errors - special section names are special, and it is useful to dedicate a separate syntactic element to them. In contrast, list elements cannot be forbidden, and if a typo is made in *Parameters*, it would be parsed as a markdown list - completely different from the user's intention.
- **Dedicated trigger for auto-complete:** Using a `#` can trigger autocomplete to suggest the names of relevant special sections, making features easily discoverable. Although this could be applied to lists, it would be cumbersome to get autocomplete suggestions for every list item instead of just when using a specific trigger character.

Introducing headings in the descriptions is generally discouraged: when some headings seem necessary for structuring the content, users have most likely already written about something broader than just the API. It might be a good sign that they should refactor some part of the description into its own article, allowing to keep the source code readable.

Nonetheless, sometimes it might be reasonable to add more content, as traits and product types may include many children. In these cases, it is possible to use second and lower-level headings (`##, ###, ...`) inside the content or add **section marker blocks** with titles and descriptions in between code entities, thus grouping related declarations, which is illustrated in Figure 4.4.

4.1.3. Special Sections in API Documentation

The set of available special sections differs per language construct.

Function-like constructs such as function, subscript, initializer, etc. have a `# Parameters:` or zero or more `# Parameter <name>:` sections. This freedom to represent plural sections with multiple singular sections allows for flexibility. Generally, if there are more than 2 parameters, it is better to use the plural version, as it provides better hierarchy and thus readability for people editing the code.

`# Returns:` and `# Projects:` are similar, but the first applies to functions, while the latter applies to subscripts. They allow to insert and provide arbitrary markdown content until the next special section or the end of the comment block. It is recommended to keep it to one short paragraph of text or a structured list of cases with different returned or projected values paired with their corresponding condition.

`# Throws:` applies to any language construct that has an executing body (function, subscript, property, initializer, deinitializer declarations) and is responsible for describing why the execution might fail. The content of this section is similarly flexible as that of `# Returns:`.

`# Precondition(s):` can be added to any construct having input parameters (function, subscript, property and initializer declarations) to document the preconditions. In contrast, `# Postcondition(s):` can be added to any construct outputting some information - may that be through returning, setting a `set` or `inout` parameter or modifying the object itself.

```
public type WordDocument {
  /// # Section: Initializers

  public init(fileName: String) { ... }

  /// # Section: Editing the Content
  /// This is the summary of the section
  ///
  /// This is the longer description of
  /// the section. Only the title is mandatory.

  public fun rotate(range: Range, by: Int) { ... }
  public fun duplicate(page: Int) { ... }
  public fun remove(page: Int) { ... }

  /// # Section: Exporting

  public fun exportToPdf() -> PdfDocument { ...}
  public fun exportToHTML() -> HTMLDocument { ... }
}
```

Figure 4.4: Section Marker Blocks can group and categorize related declarations within a lexical scope.

Invariants are important to document for data structures and language constructs having mutable state: mutable variable binding, property, product type, and trait declarations can use `# Invariants: <list>` or multiple `# Invariant:` sections for this purpose.

Complexity can be documented in many different ways depending on what is important to show for a given algorithm, so the syntax provides flexibility in the `# Complexity:` section to write a short description inline or a list.

`# See Also: <link> | <list of links>` serves the purpose of embedding supplementary documentation within a documentation entity. In certain instances, adequately documenting a symbol necessitates a lengthy description. However, including such content within the documentation string would result in inflated code files, making them cumbersome to navigate. By crafting comprehensive documentation in `*.hylodoc` articles, the `See Also` section facilitates linking and presenting them as expandable sections on the website while maintaining the conciseness of documentation in source files.

4.1.4. Syntax of links

When designing the syntax of links, the following entities needed to be considered as potential link targets:

- web URLs, email and phone addresses
- local files and folders within the documented folder's root:
  - `.swift` source files
  - `.hylodoc` articles
  - other local assets (that are directly copied to the output for downloading)
- Hylo code entities:
  - within the current scope, based on name lookup
  - within modules for which documentation is exported but are not exported to the scope currently being documented in the source file itself.

The team analyzed the linking syntax in Rust, Swift and explored the tooling support for local file references which is detailed in Appendix E. The main challenge was to design a syntax without ambiguities between local files and code entities, but we eventually arrived at a suitable design.

URLs

HyloDoc supports autolinks and full Markdown links for arbitrary URLs (except ones with the file:// protocol). It does not support implicit conversion of URLs into links, as this is not part of the CommonMark standard, and they can be cumbersome to parse.

Local File References

HyloDoc only supports the full link syntax and the image syntax for local file references, as autolinks would introduce ambiguity with HTML tags. Only relative paths are supported since, depending on the context, / might refer to different places, e.g., the root of the module or the repository. Furthermore, relative file references work perfectly in the markdown editors of IDEs such as IntelliJ and VSCode, so developers can benefit from existing tooling without Hylo needing to invest in an editor plugin.

The local assets within the documented module's directory are copied to the website's output directory. References to `*.hylodoc` articles, folders and source files are resolved to their corresponding pages, while other assets resolve to download links when the user navigates to them.

Hylo Symbol References

HyloDoc supports the double backticks syntax for code entity references, just like Swift DocC does. This is a concise syntax, and it is clear that it refers to a code entity without ambiguity with URLs.

```
``MyType.myFunction(x:y:_:)``
```

The developer might want to refer to a symbol that is not imported in the current source file but is still part of the set of modules for which the tool extracted documentation. The following syntax allows referring to entities from other modules:

```
``@OtherModule.MyType.myFunction(x:y:_:)``
```

4.2. The Generated Website

To design the website for our documentation generator, we utilized Figma, a powerful collaborative interface design tool. This allowed us to create a visually appealing and user-friendly layout that effectively presents the generated documentation. The designs created in Figma served as a blueprint for implementing the website, ensuring consistency and clarity in the final product. For a detailed view of these designs, please refer to Appendix F, which contains the complete set of Figma mockups and design specifications.

The resulting website shares many similarities with the output generated by Swift DocC, a well-established documentation tool made by Apple Inc. Both have a hierarchical sidebar navigation with filtering options. The reader can search for any file or symbol and open the corresponding documentation page. A page displays the given entity's *title*, *summary* and *description*, and the *path* of the symbol within the project, with each of the path parts being linked to the corresponding pages of modules or file.

The page of source files and symbols displays the declarations that are direct children of the given entity, together with their summary. The page of a folder displays the child folders, source files and articles along with their corresponding summary and the contents of the `index.hylodoc` article if present within the given folder.

Generally, all code entity names displayed on the website are links to their corresponding documentation page and show their summary when the pointer hovers over them.

4.3. The HyloDoc Toolset

The HyloDoc Toolset is composed of two main components: **the documentation extractor** is responsible for extracting all documentation-related metadata from the source code, while the **website generator** uses that information and presents it in the form of a website.

The documentation extractor and the website generator are designed to work as libraries. They just happened to be invoked by the command line interface of HyloDoc. This ensures that other tools can be easily built on top of them. A language server for Hylo may use the documentation extractor library and present documentation in the developer's editor rather than a website. It would also be possible to write a tool that performs various analytics on the extracted documentation, checks the accuracy of documentation with the implementations using large language models, or simply see how many fractions of our code entities have documentation.

5

Ethical Considerations about Inclusivity, the User Study, and a Political Controversy

This chapter elaborates on the ethical implications of the tool, the potential issues it presents, and how those issues were addressed. The chapter first examines inclusivity considerations for the website, followed by ethical considerations for the user study. Finally, the chapter explains the decision to avoid a library containing controversy.

5.1. Accessibility of Web Content

Software development can be done even by those with disabilities, making them potential users of the website generated by the tool. Two primary disability types that pose a great challenge for that use are visual, such as impaired vision and colour blindness, and cognitive, such as ADHD and autism [5]. Here are some issues these users might encounter and the chosen approach to mitigating them.

Visually impaired users often rely on screen readers to transcribe website content for them. Some attributes the screen reader uses are an image's alternative text and the link's title. We made sure that all images have suitable alternative text and all links have a meaningful title.

Another visual disability is colour blindness, which prevents a person from distinguishing between colours (e.g., seeing red and green as the same colour). The website's part that relies on colour for its functionality is the code examples, where different colours help differentiate between syntactic elements to make the structure more readable. The colour palette for the code examples is colour-blindness friendly, as checked by Google Lighthouse.

An aspect of cognitive disabilities that can make it difficult to use the website is motor issues. Moving the mouse can be challenging, causing those users to prefer keyboard-based navigation. The website has a simple layout that allows for navigation using basic keys, like the arrow keys for scrolling the page, 'Tab' for circling between links and input elements, and 'Return' for selecting them.

Other than assisting with movement between elements on the page, the website's simple layout also helps prevent confusion. Each page includes a navigation tree and clickable content (e.g., a function in a product type's page is also a link to the function page). This structure is a major help for those with cognitive disabilities, as the frustration arising from a bad structure can damage their experience of the website.

5.2. Ethical Considerations for the User Study

Before performing the user evaluations detailed in chapter 9 it was necessary to assess the potential risk that may arise for participants engaging in the evaluation and experiments. The TU Delft Human

Research Ethics Checklist (HREC) [6] was used to evaluate the participants' risks; this checklist is found in Appendix H. Only low risks were found that were associated with participating in the evaluations, the only risk worth mentioning is the potential privacy concerns as namea were recorded. All users participated of their free will and could withdraw from the evaluation at any point. Participants were informed of the risks and asked to complete a consent form before answering the questions. This consent form is found at the beginning of the questionnaires and is available in Appendix K.

5.3. A Controversial Library

During our project, we encountered an ethical issue regarding the use of a specific HTML-parsing library. SwiftSoup[1], found to be the most common and intuitive option, was initially selected. However, the library's GitHub page displays a banner with controversial claims about the Israeli-Palestinian conflict, linking to the Boycott, Divestment, and Sanctions (BDS) [2].

An objection was raised within our team about using this library. The argument was that the banner advocates for exclusion and discrimination, which goes beyond acceptable political expression. It was explained that the banner's claims are manipulative and divisive, promoting a movement commonly accused of spreading misinformation, antisemitism, and discriminatory policies against individuals based on ethnicity and nationality.

After discussing the issue and understanding each other's perspectives, we decided to avoid potential conflicts and chose another library that met our requirements, despite being slightly harder to use. To address the issue, we submitted a pull request[3] to SwiftSoup, expressing our concerns and suggesting the removal of the political banner. We emphasized that while supporting peace is important, separating political controversy from professional tools is crucial to maintaining unity and focus within development teams. We recommended removing all political references from the repository, potentially replacing them with a non-controversial statement promoting peace without causing division.

The author of SwiftSoup responded positively to our pull request. They acknowledged our concerns and expressed their openness to modifying or removing the banner, provided that changes are well-detailed and justified by accredited sources. This response demonstrates a willingness to engage in constructive dialogue and shows that our team's effort to address the ethical issue was well-received.

In summary, while the core intent behind the banner was positive, its presence in a professional tool was deemed inappropriate, leading us to choose an alternative solution and advocate for a more neutral approach in the library's presentation. The positive response from the library author highlights the impact of our ethical considerations and collaborative approach.

HyloDoc has ethical implications regarding inclusivity, social safety, and moral implications for using external tools supporting controversial movements. Considering those implications while acknowledging the limited timeframe, the code is designed both to accommodate the current goals and allow its expansion in the future with additional functionalities. After considering what functionality the code should have, comes the decision on how to achieve it. The following chapter elaborates on the development methodology established to ensure deadlines are met, and goals for the implementation can be achieved.

[1] `https://github.com/scinfu/SwiftSoup`
[2] `https://en.wikipedia.org/wiki/Boycott,_Divestment_and_Sanctions#Criticism`, accessed on 2024-06-21
[3] Pull request: `https://github.com/scinfu/SwiftSoup/pull/270`

6

Development Methodology

The purpose of this chapter is to discuss the methodology used by the development team to work together during the creation of project. During development, a hybrid methodology of Waterfall and Agile was adopted, called the Water-Scrum-Fall model, which best fitted the needs of the team and client.

6.1. Work structure and communication

Water-Scrum-Fall model is a hybrid Agile method where the software development activities are divided into three distinct phases: Water, Scrum, and Fall. Water represents the upfront activities such as defining requirements and planning. Scrum is the middle phase of the process where the software is actually built. On the other hand, Fall controls the release of software release frequency in stages. [7].

The aforementioned representation for Water has been followed in this project's case, but a few changes have been made to the Scrum phase. Each team member has a different circadian rhythm and activities. Therefore, there have been daily, asynchronous stand-ups where each team member has written all of the items one would go through in a typical Scrum stand-up. This updated the rest of the team on progress in day-sized pieces, which enabled some flexibility in spotting and rectifying shortcomings before they have a more serious impact. Additionally, it enforced a degree of accountability for each team member. Each member's stand-up went in a personal channel on Discord, on a server specifically created for internal communication regarding this project. Furthermore, there was only one actual release, namely the deadline for the project. Therefore, there was no need to consider anything related to the release cycle outside of the Scrum sprints. For the Fall part of the project, the work has been structured similarly to Waterfall, where progress is linear, requiring the completion of sequential stages before moving to the next. Several stages have been defined for the project, namely MVP[1], Extend MVP and Polishing & More advanced features.

We have been having weekly meetings with the client to discuss progress and challenges. Also, a weekly in-person meeting has been held every Thursday to consolidate project updates and brainstorm issues. Since the entire team was based in Delft, the commute was short and didn't hinder work progress, minimizing non-productive overhead due to transportation time. Additionally, unscheduled meetings have often been held between groups of team members working on related issues or to workshop ideas.

6.2. Version Control and Progress tracking

The work was carried out on GitLab[2], a version control platform that hosted the entire codebase. Furthermore, extensive use has been made of this platform's different features to enhance the development process. Requirements were broken down into issues and then filed into milestones depending

[1] Minimum Viable Product
[2] https://about.gitlab.com/

on weight or on the features that required this issue to be completed by specific deadlines. When starting work on a new issue, a branch was created for it, and code changes were usually committed in manageable sizes[3]. After this work was finished, a Merge Request was created to apply the changes to the codebase. Every MR had to be thoroughly reviewed by at least one other team member, and the pipeline had to pass. The pipeline consisted of several automated tasks and checks that are detailed in the subsequent section. Furthermore, the version management strategy was chosen to be *rebasing*, as it allowed flexible cherry-picking and easy synchronization of commits from the upstream repository.

6.3. Continuous Integration

A Continuous Integration pipeline has been created to ensure the project builds and the tests pass on an independently set-up machine, simplifying the reviewer's work. To accomplish these goals, the following automated tasks have been created that run anytime a code change is submitted:

1. **Formatting check**: The first job uses the SwiftSyntax library to check and enforce proper code formatting by matching it to a provided set of rules. If improperly formatted code is submitted, the pipeline fails, and the merge request is automatically denied.
2. **Coverage check**: The number of lines of code a test runs through during execution can be retrieved and, when pooled together with results from the rest of the test suite, can produce a figure, a percentage of the code covered by tests of the whole repository. A high coverage does not guarantee that a program is well tested, but it is a great place to start. The second task builds the project and runs all of the tests in the source code, creating a coverage report which displays code lines covered by tests and a total coverage figure. If the total coverage is less than 85%, the execution of this task fails, and the code change is automatically rejected.

At the beginning of the project, building and testing were separated into multiple independent tasks, one building the project and three running tests for their respective modules. However, by its nature, the command that generates the coverage report also builds the project and runs all the tests. Therefore, these other tasks were removed not only to make the pipeline run faster but also to save on storage used by the repository.

This CI pipeline and GitLab configuration gave adequate information for reviewers, including relative changes in test coverage and line-by-line test coverage results that could showcase if an error path was not covered by tests. When the reviewers saw such regressions, they usually commented on the code asking for better coverage except before deadlines for presentations and the beginning of user study, when thorough automated testing was postponed into separate issues (replaced by manual, experimentation-based testing during development), in favour of completing vital features on time that we could get feedback on. After such cases, it was our highest priority task to implement the remaining tests.

The following chapter discusses the implementation of the documentation compiler in detail, including the necessary design choices and their implications, and further technical considerations discovered during the project.

[3]Except one large-scale refactoring when many parts of the system had to be changed at once to improve the architecture and separation of concerns regarding URL-resolution

7

Product Implementation

The purpose of this chapter is to clarify the architecture of the product. The architecture consists of two phases, joined by a common layer. The chapter first goes over the general architecture in section 7.1, Product Architecture. section 7.1.1, Documentation Extraction dives deeper into the first phase, followed by section 7.1.2, Documentation Database , which clarifies the aforementioned common layer between the two phases. Finally section 7.1.3, Website Generation explains the second phase.

7.1. Product Architecture

This section aims to explain the architecture of the product and how the three different components work together. A visual representation of the connections between the components and their responsibilities can be found in figure 7.1, Architecture Components. .

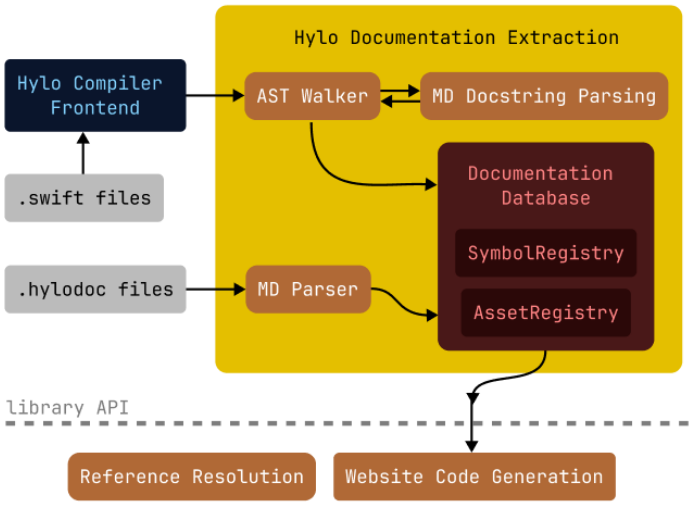


Figure 7.1: Architecture Components

The first component is the Documentation Extraction, this component is responsible for loading all the different files that might have documentation or are otherwise involved in the process. This includes using the Hylo Compiler Frontend to parse the source files into an Abstract Syntax Tree. During the documentation extraction the tool goes over all the files and symbol declarations present in the directory that it needs to document. From this it extracts all information that might be relevant for generating the website. This includes, but is not limited to:

- Articles
- Symbol Declarations
- Other files such as PDF's or images.

This information then gets inserted into the Documentation Database which is responsible for storing the information gathered during documentation extraction and allowing retrieval during the website generation phase.

After all the information has been inserted into the Documentation Database the tool steps into the Website Generation phase. This phase goes over every file in the Documentation Database and the symbol declarations in the Abstract Syntax Tree that the tool allows documentation for. It makes sure that for every element it resolves a URL and renders a page for the website.

Having introduced the different components, the next three sections will go more into detail for each of them. Starting with section 7.1.1, Documentation Extraction .

7.1.1. Documentation Extraction

The Documentation Extraction phase is crucial for gathering and processing all necessary files that will form the basis of the generated website. This phase systematically scans through various types of files, including source files, article files, and other external files such as images and PDFs.

Source Files

The primary focus during the extraction from source files is to collect relevant documentation embedded within the code. This is accomplished through a multi-step process:

1. **Lexical Analysis**: The tool employs a lexer to traverse the source files. The lexer's primary role is to skip over code tokens, thereby ignoring the actual code.
2. **Comment Detection**: Once code tokens are bypassed, the remaining elements, specifically comments, are detected. These comments are parsed using a modified Markdown (MD) parser tailored to the extended markdown syntax that we use described in chapter 4, Product Design .
3. **Symbol Comments**: If a comment is identified as a symbol comment, it notes the location of the upcoming symbol in the source file. This precise location tracking is essential for linking the symbol's documentation correctly.
4. **Database Insertion**: The parsed comments, along with the associated symbol information, are then inserted into the Documentation Database. This database serves as a repository for all extracted information, facilitating easy retrieval in later stages.

Article Files

For article files, the extraction process is more straightforward since these files do not contain code. Their contents are parsed directly using the same modified MD parser used for source files, after which their title is extracted by looking for a level 1 heading.

External Files

External files, including images and PDFs, are handled differently. These files do not require parsing. Instead, they are directly added to the Documentation Database recording their URLs, and they are just directly copied into the target directory at website generation time.

Folders

The tool records the folder structure into the asset database by recording individual folders as assets, and saving the list of their child asset ids.

With the documentation extracted and organized in the Documentation Database, the tool is well-prepared to move into the next phase. Section 7.1.2, Documentation Database , will delve deeper into the structure and functionality of this crucial component.

7.1.2. Documentation Database

Positioned as an intermediary layer between the documentation extraction and the website generation, the documentation database is a structured storage system designed to handle and organize documentation entities efficiently. The database captures and stores various entities such as symbols, modules and other assets, enabling quick and easy retrieval for the subsequent website generation.

Structure

The main components of the Documentation Database are:

1. **Asset Database**: Holds information about different kinds of assets in its sub-stores:
   - source files (along with file-level documentation
   - articles
   - folders
   - any other files that are not `*.hylo` or `*.hylodoc`.
2. **Symbol Documentation Database**: Holds the extracted symbol-level documentation in its sub-stores, organized by the specific symbol declaration types (e.g. product type, function, subscript, trait declarations)
3. **Module Database**: Contains information about modules we have extracted documentation for.

Advantages

The advantages of the Documentation Database structure are:

1. **Efficiency and Performance**: Most stores implement the same entity store, which is optimized for fast lookups and insertions that are performed in constant time. Furthermore, all entities are stored compactly in contiguous memory locations without using additional indirection due to a type-erased structure. See Andrew Kelly's talk on Data Oriented Design [8].
2. **Type safety**: The structure of the database and its use of various features of the Swift programming language, especially conformance, guarantees that the IDs which are associated with specific entity types are strictly associated with their corresponding documentations, minimizing the risk of passing incorrect IDs.
3. **Malleability and Control**: The blocks of documentation that are identified when traversing the source files are parsed by an external library from their original Markdown format into the library's own data structure. However, by further processing this data into the Documentation Database, a higher amount of control over the underlying data can be obtained. As a result, further validation can be performed, such as checking that the user has only written parameter documentation for parameters that actually exist for that specific symbol. Additionally, the data be more easily modified compared to using the library's data structure directly.
4. **Easy and Organized Access**: By filing documentation blocks by their types into the database, it greatly facilitates website generation. For example, a method's documentation may contain information about its parameters or what kinds of exceptions it throws, while an operator might just list an invariant. As these two symbols have greatly different information associated to them, the website generation can directly tap into the database and recognize what kind of layout it should create for a symbol based on its type.
5. **Modularity, Extensibility and Flexibility**: Different types of documentation (symbols, assets, modules) are managed by distinct stores and the design is highly generic, allowing easy extension for new types of documentation entities without significant changes to the existing infrastructure. This approach enhances maintainability and future development.

The Documentation Database is a robust and efficient system for managing and retrieving documentation entities. Its design focuses on performance, type safety, and modularity, making it a powerful intermediary layer in the documentation generation process. After the comments have been extracted and correctly filed into the database, the website generation can begin.

7.1.3. Website Generation

The last phase is Website Generation, this phase is responsible for processing the incoming information stored in the Documentation Database and the Abstract Syntax Tree from the Hylo Compiler. It traverses all the files and symbol declarations present in those two data-structures and generates a website based on this information.

This phase consists of the following four steps:

1. **URL Resolution**: The first step is to traverses all the files and symbols and for each of them keep track of the parent elements during traversal. This information is then used to generate a URL for every element. The website generation phase uses this to refer between elements and to later decide where information should be stored relative to an output directory.
2. **Page Rendering**: The second step requires repeating the traversal process, but this time, a page is rendered for each element. These rendered pages collectively form the content of the website.
3. **Symbol Rendering**: When rendering a documentation page for a particular symbol, the symbol itself and all its members (e.g., methods of a product type) need their definitions rendered. The URL resolver from the first step is used to provide links to each type. Some types (e.g., tuples, buffers, generic clauses) consist of other types and require recursively calling the render function, as illustrated in 7.2. Once the symbols are rendered and inserted into the page, the information is ready for export.
4. **Exporting Documentation**: In the final step, all the rendered pages are exported to their respective destinations as the tool traverses them. This step concludes by copying the necessary assets for the website's style and interactive elements and creating an index page for all the modules being documented by the tool.

```
fun toTuple() → (Float64, Float64) { sink }
```

Extracts the tuple of components out of the vector.

Figure 7.2: Rendering of a method symbol requiring recursive rendering of the tuple output type, which consists of two Float64 types.

After these four steps, the Website Generation phase is done, and the tool successfully generates documentation based on the modules provided at the start. The documentation is now ready to be viewed locally or displayed on a web-server to share with others. The next crucial step is verifying that the implementation works as expected, detailed in chapter 8, Testing .

8 Testing

Testing is a critical component of the software development process, ensuring the reliability, functionality, and performance of the HyloDoc. This chapter elaborates on the chosen testing strategies employed in this project to provide a suitable approach to verifying the correctness and robustness of the system within a limited time frame. The chapter clarifies the use of unit testing, dependency injection, and integration testing. Lastly, it expands on a novel approach to creating testable errors that was developed throughout this project.

8.1. Unit Testing

Unit testing is the cornerstone of the testing strategy, focusing on verifying the correctness of individual components in isolation. Each function and module within the project is rigorously tested to ensure desired behaviour under various conditions and inputs. Dependency Injection is extensively used throughout the testing suite. This allows the decoupling of components and the injection of mock or stub dependencies during testing, simulating scenarios without relying on actual implementation. The XCTest framework is used to write and execute unit tests.

8.2. Integration Testing

While unit testing increases confidence in the correctness of individual components, integration testing verifies that different components of the project work together seamlessly. This level of testing ensures the integrated system meets specified requirements and behaves as expected in real-world scenarios. Integration tests cover the interactions between all of the modules of the project: the comment extraction, the documentation database, and website generation. These tests simulate the complete workflows used in real-world use cases, e.g. extracting comments from source code, parsing those comments into the Documentation Database, and generating a navigable website but they often only test some fraction of the pipeline by relying on simple test doubles or hand-written input data.

For compiler projects, hand-writing input data can be overly challenging and verbose, as the system needs to rely on many layers of abstractions to construct the needed data structures such as the abstract syntax tree or the documentation database. For many of our tests, we did not differentiate between traditional unit and integration tests but we tried to limit the scope of the used dependencies that were already unit-tested and trusted with great confidence.

Small-scoped website generation tests only relied on the Hylo frontend to generate the abstract syntax tree (often, from an in-memory synthesized source file) while the documentation database was manually propagated with test data. Large-scoped tests run the full pipeline with a sample Hylo module written in a sibling directory near the test's source file so that relevant text features are easily located.

To facilitate the writing of expressive and relatively concise test cases, custom assertions and convenience initializer extensions to commonly used data structures were developed such as `AST.fromSingleSourceFile()` and `AST.simplyResolveDeclaration<T: Decl>(by name: String)`. In addition, we used an HTML

parsing library for testing the website's content by asserting that certain text or regular expressions were matched inside certain elements.

8.3. Confidence in Correctness

The code base, being written in Swift, makes many classes of mistakes either impossible or unlikely to happen, as usually the types are architected in a way that they cannot encode invalid or partially formed values, except during their construction. Mutability is oftentimes not needed, as data structures are created once, recorded into the database, and left there indefinitely until they get destroyed. After the documentation parsing and the recording of documentation entities have been finished by the asset scanner and the source file documenter, the documentation database is passed immutably down the pipeline, and its well-formedness is already guaranteed.

In other cases, when we had to interact with libraries and customize their functionality in a non-API-breaking manner such as the custom rendering of a MarkDown block, we needed to rely on hacks and reference semantics to be able to pass in extra input to the rendering, or adding a way to signal errors from a non-throwing overridden function. In these cases, we needed to ensure the preconditions of certain functions, which are all properly documented and asserted at runtime, so that failing to meet them would make the tests fail, or at least crash the whole test suite with the diagnostics.

Fuzz testing is used by running the program on the standard library and a set of simple custom libraries tailored for exercising many behaviours of the system are essentially property-based tests whose properties are encoded in the assertions within the source code instead of tests, providing the team great confidence about a class of errors in this form.

While URL resolution has many unit tests and high coverage, it proved to be an error-prone part of the system, sometimes making links redirect to a page that was not found. A future recommendation for extending the test suite would be to write a browser automation script in Python or JavaScript that visits every link on every page and checks if it resolves to a page or results in a *page not found* error.

8.4. Error Handling Strategies

The development of this project involved exploring various methods for error handling and diagnostics reporting to provide informative and helpful error messages to users, while also being able test the error conditions. The existing diagnostic system from the Hylo project had some limitations, particularly in testing scenarios.

The Hylo diagnostics

The Hylo repository made use of a diagnostic class and a diagnostic set which was passed around, collecting errors, warnings and notes during execution. While this design allowed for centralized diagnostic management, it posed several challenges:

- **String Matching for Tests**: Verifying specific diagnostics in tests required matching on message strings. This method was cumbersome and prone to errors, as it relied on exact string matches that could be affected by trivial changes in wording or formatting.
- **Limited Type Safety**: The reliance on message strings for diagnostics meant there was little type safety, making it harder to ensure that the correct diagnostics were being checked and handled appropriately.

Result types

Another approach considered was using Result types for error handling. Result types can encapsulate both success and failure values, providing a functional-style approach to error handling. When abstracting away from the dependencies' possible error cases using associated types on their protocols, the compiler can ensure that all error cases are handled correctly when we are pattern matching on a particular result type, which ensures that a class of problems are caught at compile-time, and provides a transparent way of inspecting returned errors exhaustively without relying on type erasure and casting, which might always fail if the functionality changes its implementation.

However, this method also had its drawbacks:

- **Verbose Syntax**: While Result types provide the best type safety compared to other methods, they often result in verbose syntax, especially when having to write the type declaration of a result of a function that is chaining multiple operations that can fail. This verbosity can reduce code readability and increase maintenance overhead.
- **Limited Diagnostic Information**: Similar to try-catch, Result types are typically used to represent binary outcomes (success or failure) and do not inherently support the rich diagnostic information required. Extending Result types to carry detailed diagnostics would require additional boilerplate and custom structures.
- **Testing Overhead**: Using Result types would still necessitate extensive testing frameworks to check for specific errors or diagnostics, potentially involving custom pattern matching or unwrapping logic. This overhead overcomplicates the testing process further.

Traditional Try-Catch Error Handling

An alternative considered was using traditional try-catch blocks for error handling. In Swift, the exception system is implemented as a `Result<T, any Error>`. However, it provides significantly different developer experience and features. Its type-erased error case does not provide any guarantee about the exact set of possible types thrown in the function, so ensuring completeness when pattern matching on the error type is not possible at compile-time. However, it provides great syntactic sugar for control flow, interoperability with optionals, and pattern matching on error cases based on their type or protocol.

The greatest limitation of throwing errors everywhere was that it made diagnostic collection and error recovery control flow look highly verbose at the call site of throwing functions.

Error Macros and Diagnostic Protocol

To address these limitations, a more robust and type-safe approach was introduced by defining a diagnostic protocol. This protocol enabled the creation of specific error types that conformed to it, simplifying the process of testing for specific diagnostics. It contains the required properties and methods for a diagnostic, providing a standardized way to define and handle any error types. This significantly improved testability. In tests, specific diagnostics could now be checked by matching against the specific type rather than relying on string matching.

While this protocol improved type safety and testability, it introduced a significant amount of repetitive, “boilerplate” code. Each new diagnostic type required the implementation of this protocol which would have been repetitive and error-prone.

To streamline this process, a recently introduced Swift feature was used: macros. A macro is a metaprogramming tool allowing code generation at compile time. Macros can inject code into a program, reducing the need for boilerplate code. Therefore, the “@Diagnostify” macro was created so that only the error signature needs to be added to a file, and the macro automatically adds the required conformance to the diagnostic protocol.

Over time, our experimentation with errors, result types, and diagnostics, learning about their advantages and drawbacks, and focusing on testability have led to significant improvements in how we handle and test diagnostics within the project. This has also deepened our understanding of the different values provided by language constructs. The introduction of a diagnostic protocol and the use of macros for boilerplate reduction enhanced the codebase’s robustness and maintainability, ultimately contributing to a more efficient development process.

As a result of the comprehensive testing strategy employed during the project’s development, encompassing unit testing, dependency injection, and integration testing, the reliability and robustness of the Hylo Documentation Compiler were consistently ensured throughout the project. By verifying individual components, their interactions, and real-world workflows, a high level of confidence in the correctness and performance of the software is maintained. The mentioned testing techniques gave a high degree of correctness of the inner workings of the project, but an external check was also required to ensure the quality of the project itself. For that, a user-centred evaluation was conducted and will be presented in the next chapter.

9

User-centered evaluation

The following chapter discusses the user testing that was conducted for this project. Section 9.1 discusses the methodology used and section 9.2 addresses the aggregated results.

9.1. User Evaluation

To evaluate the system design and the project's usability, we created two questionnaires for the main stakeholder groups: potential users of the Hylo language and contributors to the Hylo repository. Users from both of these groups were asked to participate in the evaluation in order to get feedback directly from the target user groups.

The evaluation consisted of post-task questions, in which participants were asked to perform specific actions using the Hylo Documentation Compiler on their own devices and then to answer questions about the design using a Google Forms questionnaire. Many of the questions asked to evaluate using Likert scales how easy it is to find or use specific functionalities. This questionnaire resembles a Single-Ease Questions (SEQ) [9] approach, which is beneficial as it is easy to assess which particular functionality of the design has the most issues since questions directly follow each task.

These questions help assess if the envisioned design is usable and intuitive enough that users can use the project's features to the intended extent without requiring help. Likert scales were used to make the results simple to understand and aggregate over all participants. A few open-ended questions are also included to allow participants to share any final thoughts and ideas related to their experience. The following are two summaries of the tasks each group was asked to complete and provide feedback on:

1. **Potential users of the Hylo language**: For the questionnaire for the potential developers, a website was provided, generated using the tool from some source files to which the users did not have access. Then, they were asked to find information about different symbols without having access to the search bar. This task was designed to evaluate how easily a user can access information. By turning off the search bar, the users were forced to use the navigation sidebar and the links between files in order to navigate the website. The next task provided the users with the documentation generated from the Hylo standard library and asked them to find symbols in this significantly more populated website. This questionnaire is available in Appendix I.
2. **Hylo contributors**: In addition to the same research question for the potential developers (how easy is it to find information), the additional tasks for the Hylo contributors were also focused on finding out how usable the new documentation syntax is. This user group was provided with some source code files but was asked to generate the website themselves. Then, they were asked to document some trivial code using the new syntax. The questionnaire is available in Appendix J.

9.2. User Evaluation Results

Overall, the responses were mainly positive. The contributors and the potential developers found no significant problems with the usability of all facets of the project. All participants found the design and

overall layout of the generated website enjoyable and intuitive. For each question in the user study, an average score was calculated shown in Figure 9.1. The the average score overall was 4.54, indicating high user agreement with statements about the ease of use of the project. The results of the contributor study also validate the project's accessible design, from the CLI to the new documentation syntax.

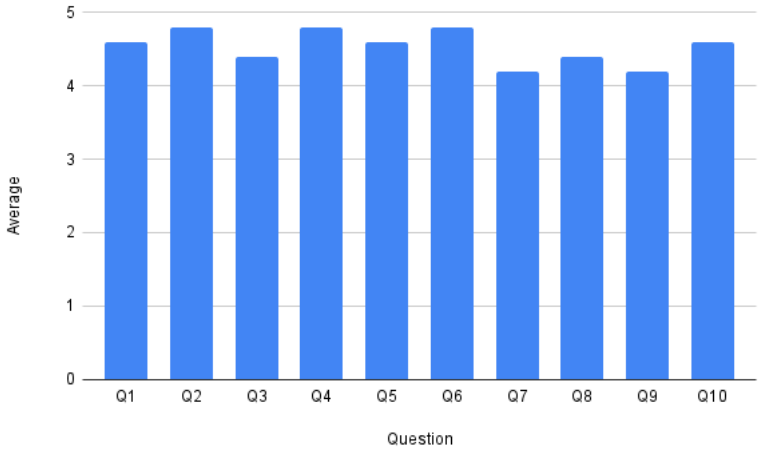


Figure 9.1: Average score for every question

9.2.1. The navigation tree

Both groups of participants had an issue with the navigation tree side panel. Initially, the panel would always start fully expanded, and users could collapse items. However, the problem was that the expanded or collapsed state of the tree would not be saved when the page was reloaded, so everything would expand again when navigating. This caused frustration, especially when using the standard library, which had a lot of items in the navigation tree. To fix this, we changed the default setting to have the navigation tree collapsed, and we made sure that its state was saved when navigating between pages. You can see the before and after comparison in Figure 9.2.

Our client pointed out issues with the accessibility of the tree navigation. It would be ideal if users could navigate between tree items using the up/down arrow keys, collapsing elements using the left-/right arrow keys or the space bar, and visiting a link using Enter. Implementing this functionality and reviewing the experience of navigating the website with only a keyboard would significantly enhance the website's accessibility and user experience.

9.2.2. Other suggestions

Through the open questions, participants submitted other suggestions for this product that could not be further investigated or implemented in the project's timeframe. Below is a list of the most relevant ones:

1. Despite the great contrast ration between text and the background, adding the option to switch between dark and light modes would make the website more readable in different lighting conditions.
2. **Simplifying the error messages** can streamline user interactions with the tool, making it easier to identify and resolve issues without being overwhelmed by information.
3. **Turning off the ligature of operators** would ensure that each character is displayed clearly and distinctly, making it easier for developers to see what they need to type exactly to replicate the code.
4. Make the navigation tree side panel resizeable and hidable.

We created GitLab Issues for these suggestions and leave them as future recommendations for development.

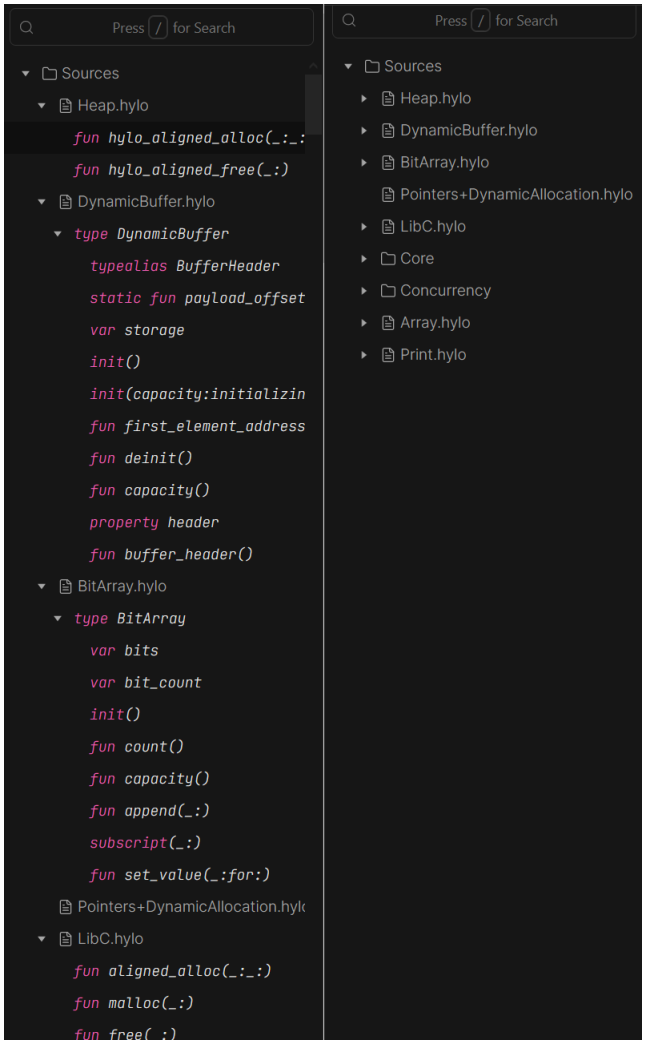


Figure 9.2: Default state of navigation panel before (left) and after (right) implementing feedback.

The feedback from participants in this user test has been generally positive, demonstrating that the project is easy to use and has an intuitive design. In the next section, we will discuss the optimization activities that were carried out before the beginning of the user study. These activities are related to the performance of the documentation extractor tool and the website. They aimed to enable evaluators to provide feedback on issues that could not be detected through automated tools.

10

Performance Analysis and Optimisation

We understand that developer tools' performance directly impacts how productive their users can be. That's why we prioritised performance from the start when designing the tool's architecture. Trying to improve performance after the fact can be difficult if the architecture is not right to begin with. Despite our initial focus on performance, we encountered several performance issues when testing the tool with Hylo's standard library. By the end of our software project, we addressed all of these performance issues and provided recommendations for future improvements.

This chapter will cover the performance improvements we implemented in website generation, including our analysis methodology, the introduction of template caching, and the optimisation of navigation rendering. Additionally, we will present a comprehensive website performance evaluation using Lighthouse, outlining the initial metrics, the optimisation strategies employed, and the final results. We will conclude with a summary of our findings and future recommendations.

10.1. Performance Improvements in Website Generation

When we first ran the tool on the standard library, generating the website took over two minutes. This was unacceptable, as adding multiple minutes to a repository's pipeline for documentation generation is not worth the cost for library maintainers. In addition, the output directory contained 85MB of HTML, which we needed to investigate whether it was reasonable. .

Performance Analysis of the Website Generation

Our investigation began with an examination of why the tool generated 85MB of data. The folder structure appeared correct, containing 462 files, each with a size of 170KB. This size was attributed to the tool outputting the navigation tree structure to all HTML files to ensure functionality for javascript-disabled browsers and search engine crawlers. Each entry in the navigation tree consisted of multiple HTML elements to achieve the desired layout. Additionally, we observed that the tool produced unminified HTML with comments and extra whitespace. After running an HTML minifier, we determined that file sizes could be reduced by approximately 25-35%, resulting in 110-130KB files.

Following a discussion with stakeholders in the Hylo community, we concluded that the 170KB HTML files represented a reasonable trade-off to meet the website's performance and accessibility goals. The loading time for these files remained significantly lower than that of custom fonts and images on the website. Furthermore, hosting and deploying the 85MB of data posed no issues, as we could deploy the website on Fireship in approximately 10 seconds.

To see if the problem is inherently slow because of slow IO operations, we performed some isolated tests to see how long it takes to output 85MB of small files. We wrote a program that constructed 170KB strings and saved them into new files on disk. The program completed in 3-5 seconds on a laptop, so

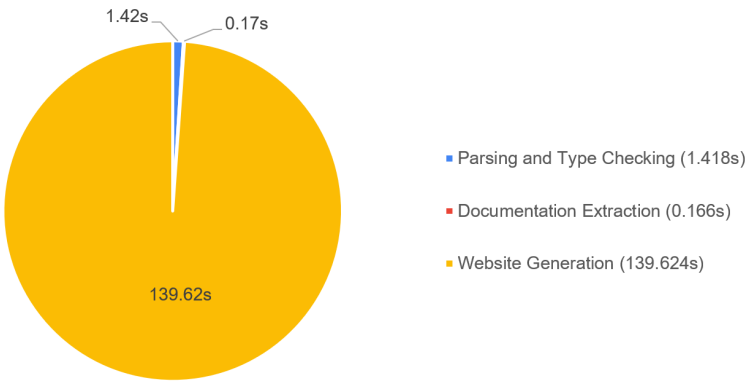


Figure 10.1: Initial distribution of time taken by different phases of the pipeline when running HyloDoc on the standard library

we knew that if we were IO-bound, we should not have needed 25x the time to complete.

To isolate the problem, we recorded time stamps to see how long each pipeline phase takes. We subsequently identified website generation as the bottleneck, as shown in Figure 10.1. After running the program with a performance profiler, we identified that the main performance bottleneck is Stencil and its rendering of the HTML templates.

Introduction of Stencil Template Caching

Stencil, the library used by HyloDoc to render HTML templates, works by loading a template from the disk, parsing its content, and filling it with data from the provided generation context. This strategy is reasonable for small websites, but when generating hundreds of pages, with potentially recursive template inclusions for the navigation tree rendering, it is highly wasteful to reload the contents of the templates from disk again and reparse them for every include directive and render calls throughout the program's run.

Someone has already made a pull request[1] to fix this issue in Stencil by introducing template caching, but it has not been merged because it included API-breaking changes, as it made the rendering functions mutating. We updated our dependency to use this fork, and the rendering performance improved from an average of 139 seconds to an average of 102 seconds, **which was a 26% improvement**. We also gave feedback on the GitHub pull request for mitigating the API breakage by utilising a shared, global cache with reference semantics and we hope that this pull request will be merged by the maintainer. The last change to Stencil was committed more than 10 months ago, so we are not concerned about the fork getting outdated but more about the library's long-term maintenance and health.

Reusing Rendered Navigation Between Pages

The website rendering was still unacceptably slow, so we started to think about optimising the slowest part: generating the tree navigation for all pages. Although the navigation tree's content is the same across different pages, there were some limitations why we rerendered it for every page. Firstly, we wanted to render relative URLs so that the links would work regardless of which subdirectory they were deployed in, and they would also work locally without a web server. Secondly, we wanted to highlight the active tree item of the tree, which required adding a CSS class to one of the items.

These benefits would be great, so we evaluated the costs of migrating from using Stencil and constructing the website using native Swift function calls or a DSL tailored for constructing HTML structures. We made an isolated test to see how well those approaches would perform, and we found that generating 85MB of content without Stencil's interpreted nature takes 1-2 seconds, depending on how we structure the content. Dropping Stencil would also ensure that we would have type safety and a more robust generation instead of the current dynamic typing approach introduced by HTML templates, which is error-prone, especially when refactoring code.

[1] `https://github.com/stencilproject/Stencil/pull/345`

While dropping the Stencil would have solved the performance problems in the cleanest way, it also had some drawbacks. Firstly, we were developing the website's design implementation in a separate npm-based project that used the Vite build system for making small CSS and JavaScript bundles and to have hot-reloading support during development. Maintaining the templates in Swift would have made updating the design a slightly more cumbersome process, as it would require the templates to be translated from HTML into Swift. Secondly, we were already 8 weeks into the project, and changing the HTML rendering framework would have been a major refactoring effort.

Instead, we addressed the issues that prevented us from caching the navigation tree for all pages. Highlighting the active tree item is not an essential feature for JS-disabled contexts, so we solved the highlighting issue in JavaScript by adding the necessary class to the HTML element dynamically based on the current page's URL.

The second issue was that we wanted to be able to deploy the website with any given base URL, such as `example.com/documentation/`, and we wanted to be able to view the website locally. We addressed both of these by adding a `--base-url` CLI argument to the tool (with a default value of /). This way, absolute URLs could be generated, so the same links would work regardless of how deeply their page is nested. In addition, we added a CLI flag `--preview` that started a web server, serving the website after the generation has completed.

These measures allowed us to cache the rendered tree navigation, and the website generation could be drastically sped up to an average of 2.7 seconds, making the whole pipeline's run, including parsing, type checking and documentation extraction, take only 4.5 seconds on average, as illustrated by Figure 10.2 and Figure 10.3.

Completing the documentation extraction for the whole Hylo Standard Library in 0.16s means the tool is suitable for low-latency applications and integration into Hylo's language server. As a future recommendation for improvement, we would advise to replace Stencil with the Plot library, as that could greatly improve performance.

For benchmarking the tool, we performed 5 runs compiled in release mode after each added optimisation. Refer to tables in Appendix G, Performance Measurements for the detailed results.

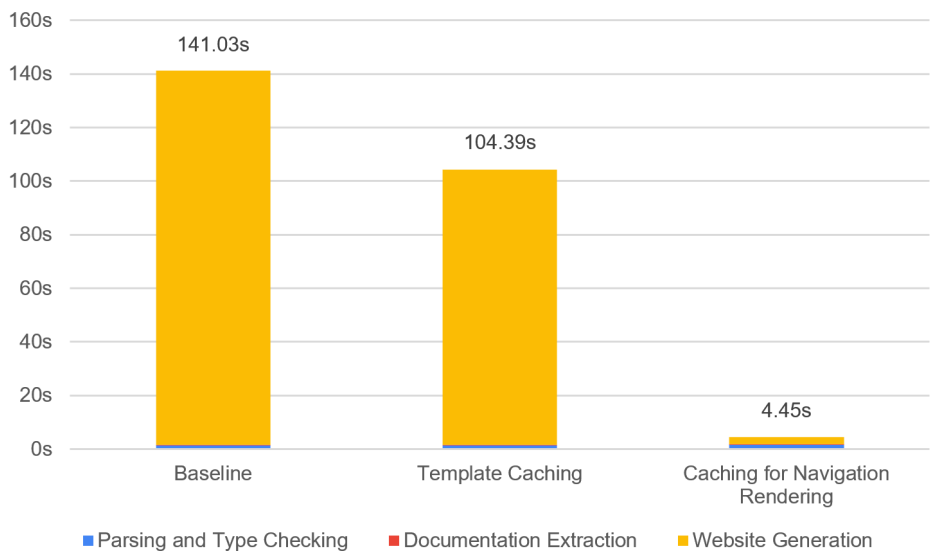


Figure 10.2: Performance Comparison between different Optimisations

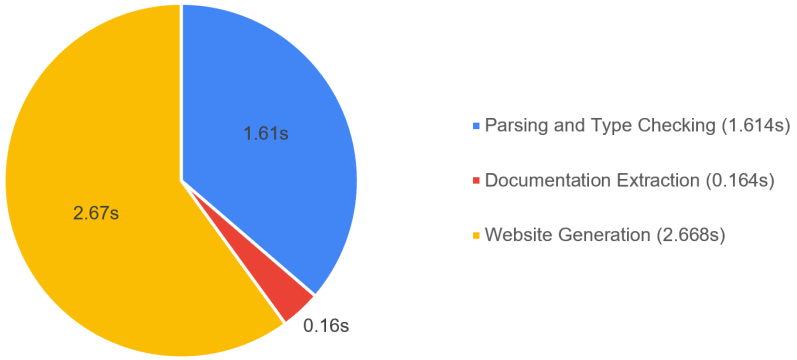


Figure 10.3: Final distribution of time taken by different phases of the pipeline when running HyloDoc on the standard library.

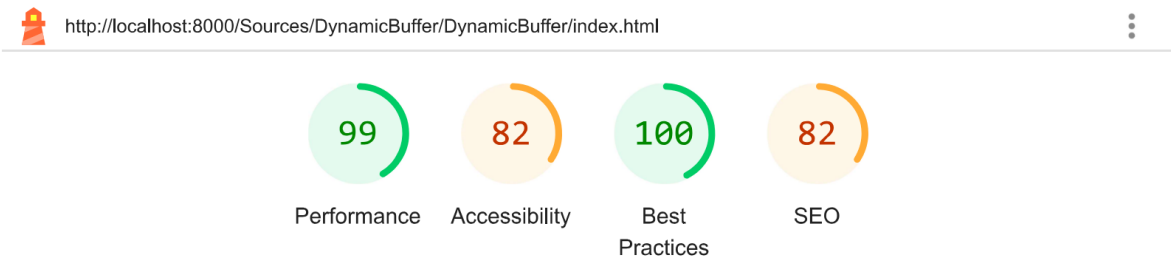


Figure 10.4: Initial Lighthouse measurements of HyloDoc

10.2. Comprehensive Lighthouse Analysis and Optimisation

The performance and accessibility of the generated website is important for a good user experience and the effectiveness of the tool. Fortunately, static site generation can eliminate the possibility of many issues, since most of the work to construct the website is done at deployment time but we still had some room to improve.

To assess the performance and accessibility of the generated content, we used the Chrome Lighthouse tool in Desktop mode. The website's layout is not responsive enough to be suitable for mobile users, and after discussing this with our client, we concluded that the website's main use case is during work, in which context the developers are using a desktop computer, therefore optimising for mobile is out of scope for this project.

Initial Lighthouse Analysis

The website's initial lighthouse scores before optimisations were already promising, and comparable with those of Swift's DocC. Lighthouse gave actionable feedback on what the website is missing, and where it should be improved.

Strategies for Optimising Lighthouse Performance

Website Performance

The website received a performance score of 99%. The key areas for improvement were eliminating render-blocking resources and avoiding large layout shifts. We also got a remark on having many HTML elements in the DOM (3500 elements). We have applied this feedback by learning more about how the browser loads the content, giving explicit sizes to images, and deploying the website on a real server with text compression enabled instead of using a local development server.

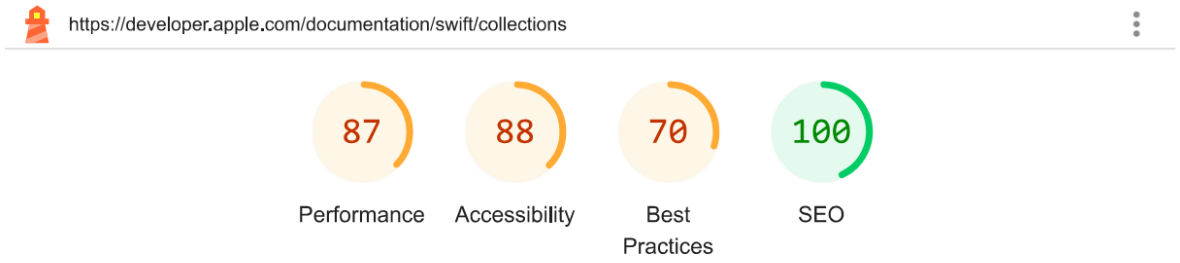


Figure 10.5: Lighthouse measurements of Swift's DocC

Website Accessibility

The accessibility received a score of 82%, and the feedback showed that some images do not have an `alt` attribute and that form elements do not have associated labels. We needed to add an alt tag to the website's logo and aria-label attributes to the tree navigation checkboxes that control the entries' collapsing state. The tool has passed audits related to the legibility of text, e.g. having a suitable contrast ratio between the background and the content.

However, many of the accessibility issues are subjective or hard to test in an automated fashion. That is why manual testing is crucial when evaluating the accessibility of the website. During our user evaluation, detailed in chapter 9, we received additional feedback regarding the keyboard navigation on the sidebar. We can get more feedback after the completion of the project when we release Hylo's Standard Library documentation website to the public.

Best Practices

From the best practices category, we only received the feedback that we should ensure the content security policy is effective against XSS attacks. Even though we do not have any server-side state that users can influence, mitigating the possibilities for XSS (cross-site scripting) attacks can be a useful consideration, as library maintainers might not notice particular changes to the documentation made by external contributors, and noticing XSS attacks is not as straightforward as scanning for the code for included JavaScript or URLs.

The security policy is controlled by the web server in the HTTP response and cannot be influenced by the generated HTML. The initial test was performed on a local HTTP server with a misconfigured content security policy, but the issue was resolved when we deployed it on Firebase.

Search Engine Optimization

The website did not have meta description tags and the check about images not containing alt tags applied to search engine optimization as well. Because of these factors, we received a score of 82%.

To fix the first issue, the tool now generates a roughly 155-character-long summary of the displayed page's content. When an authored summary is not present in the case of an article or folder page, the tool takes the first paragraph's beginning as the summary. When even that is not present, the tool outputs a general description based on the type of the page and its location within the module, which is considered bad practice as being overly general, but there were no better solutions available.

Final Lighthouse Report

After applying the feedback we received from Lighthouse, along with some input based on manual testing, we achieved a perfect lighthouse score, having 100% in all categories as shown in Figure 10.6, making the website ready to be released to the public. The final Lighthouse report can be found in Appendix L.

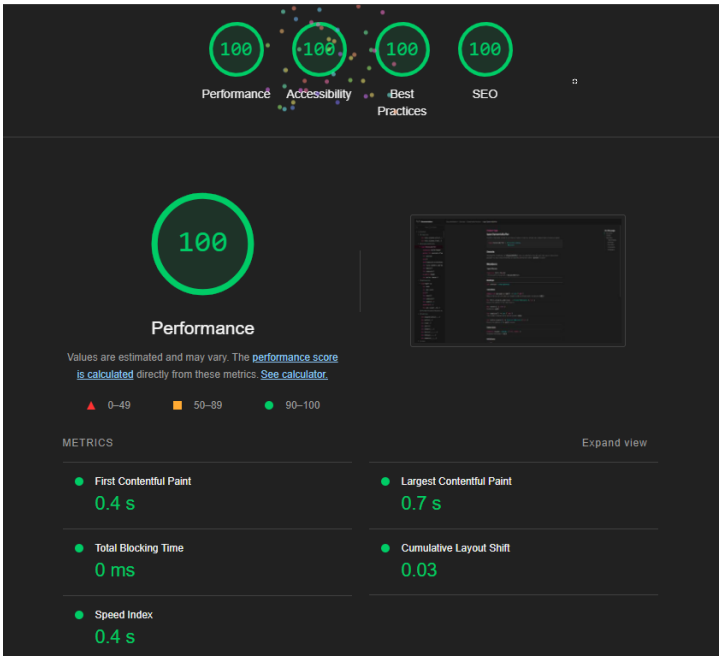


Figure 10.6: Final Lighthouse Report

11 Conclusion

Adequate documentation is essential for software engineering, providing clarity and understanding of software functionalities. The development of HyloDoc, a documentation generation tool for the Hylo programming language, addresses the need for automated and comprehensive documentation.

HyloDoc achieves its objectives by automating the extraction of documentation from a given Hylo codebase and generating a user-friendly website. This tool comprises two primary components: documentation extraction and website generation. The documentation extraction process involves scanning the source code for comments and relevant information stored in the documentation database. The website generation component processes this data into HTML pages to create a comprehensive and navigable website.

The development process began with a thorough problem analysis, identifying the lack of automated documentation in Hylo and its impact on developers and users. Requirements were gathered and prioritised using the MoSCoW method, focusing on essential features. A markdown-based syntax was developed to ensure the documentation was readable and concise. Ethical considerations, such as accessibility and inclusivity, were integral to the design process.

Implementation involved creating a robust architecture for HyloDoc, covering the processes of documentation extraction, database management, and website generation. In-depth testing, including unit and integration tests, gives confidence in the tool's functionality and reliability. A user study provided valuable feedback, leading to refinements that improved usability. Performance optimisation efforts addressed initial issues, enhancing the tool's efficiency and scalability.

HyloDoc successfully meets all proposed mandatory requirements. It extracts Application Programming Interface (API) information from the code, along with developer-written comments, generating an easy-to-access and navigate website. References to symbols are handled with scope-aware resolution, and inheritance is recorded. Folders can be documented using Markdown files, and the tool can be operated using a command-line interface (CLI). Additionally, many optional requirements were met, including the addition of a navigation tree and search feature on the website, and the resulting product's quality and usability exceeded the initial expectations.

Several possible enhancements are recommended to improve and extend HyloDoc in the future, and many others will be collected once the Hylo repository adopts the tool. Some recommendations concern the user experience and interactive features of the website, like keyboard-based navigation, while others would provide greater expressive power to authors, such as documenting built-in types or adding executable code examples to pages. Incorporating these suggestions will allow HyloDoc to become one of the best open-source documentation generation tools.

The realisation of HyloDoc significantly improves the documentation process for Hylo, making the language easier to maintain and more accessible to users. This advancement is expected to increase the adoption and growth of Hylo within the programming community by providing a reliable solution for maintaining high-quality documentation.

References

[1] Robert C. Martin. “Clean Code: A Handbook of Agile Software Craftsmanship”. In: O’Reilly Media, 2008. Chap. 4. Comments, pp. 53–75.

[2] T. Winters, T. Manshreck, and H. Wright. “Software engineering at Google: lessons learned from programming over time”. In: O’Reilly Media, 2020. Chap. 10. Documentation, pp. 185–206.

[3] Wikipedia. *MoSCoW method*. 2024. URL: https://en.wikipedia.org/wiki/MoSCoW_method (visited on 04/26/2024).

[4] S. Marlow and S. Peyton-Jones. “The Architecture of Open Source Applications”. In: ed. by Amy Brown and Greg Wilson. Vol. 2. Lulu.com, null edition, 2012. Chap. The Glasgow Haskell Compiler. ISBN: 978-1105571817. URL: https://aosabook.org/en/v2/ghc.html.

[5] web.dev. *What is digital accessibility, and why does it matter?* 2024. URL: https://web.dev/learn/accessibility/why?continue=https%3A%2F%2Fweb.dev%2Flearn%2Faccessibility%23article-https%3A%2F%2Fweb.dev%2Flearn%2Faccessibility%2Fwhy (visited on 05/20/2024).

[6] TU Delft. *Human Research Ethics*. 2024. URL: https://www.tudelft.nl/en/about-tu-delft/strategy/integrity-policy/human-research-ethics (visited on 06/17/2024).

[7] S. Rahim et al. “ScrumFall: A Hybrid Software Process Model”. In: *International Journal of Information Technology and Computer Science(IJITCS)* 10 (2018), pp. 41–48. DOI: 10.5815/ijitcs.2018.12.06.

[8] Handmade Cities. *Andrew Kelley - Practical DOD*. 2021. URL: https://vimeo.com/649009599 (visited on 06/20/2024).

[9] Page Laubheimer. *Beyond the NPS: Measuring Perceived Usability with the SUS, NASA-TLX, and the Single Ease Question After Tasks and Usability Tests*. 2024. URL: https://www.nngroup.com/articles/measuring-perceived-usability/ (visited on 06/17/2024).

A

Requirements

This appendix includes the full list of all functional (must, should, could, won't) and non-functional requirements constructed for this project, according to the MoSCoW model [3]. Additionaly for each functional requirement a completion status is included indicating whether or not the feature was implemented fulfilling the requirement.

Must

These are essential requirements critical for project success and are non-negotiable. (See A.1)

Table A.1: All must-have requirements along side their completion status

Requirement	Status
The Tool extracts API information from all named symbols in a Hylo source file (except function- and local scope variables): trait, type-alias, product type, variable binding, initializer, function, method (and method bundle), subscript (and subscript bundle), property, operator, associated type and associated value.[1]	Completed
The Tool extracts API documentation comments from the recorded symbols where present.	Completed
The Tool generates a (basic) website that displays the extracted documentation of the symbols.	Completed
The Tool handles references to any Hylo symbol with scope-level name resolution.	Completed
The Tool generates hyperlinks for symbols displayed on the website, pointing to their respective documentation page.	Completed
The Tool records inherited documentation from a parent trait for conforming product types and traits.	Completed
The Tool allows refining the documentation of a method bundle for each inner declaration.	Completed
The Tool allows the writing of documentation for a module as a markdown file.	Completed
The Tool exposes a command-line interface.	Completed

[1] We initially also mentioned deinitializer and namespace symbols in the project plan, however they we're deemed obsolete as we learned Hylo's feature set throughout the project.

Should

Important but not vital; enhances functionality, deferrable if necessary. (See A.2)

Table A.2: All should-have requirements along side their completion status

Requirement	Status
The website has tree navigation between symbols and articles	Completed
The website has a search feature for symbols	Completed
A user can add to the website documentation that does not belong to a particular code entity using markdown files	Completed
The Tool checks for broken references to symbols and linked files.	Completed
The Tool does not export documentation that is not publicly visible from outside the module (those documentation should only serve as notes for the implementers and are not exposed to the public).	Completed
The Tool does not export markdown articles that are named *.internal.md	Completed
The Tool ensures that special sections follow the conventional order.	Completed
The Tool checks if a public reference exists to a file, if it is marked as internal or if it is a symbol not accessible from outside its module. (It detects dangling references before generating the website.)	Completed
The Tool resolves local file references and makes the assets available on the website.	Completed

Could

Desirable but non-essential; included if time and resources allow. (See A.3)

Table A.3: All could-have requirements alongside their completion status

Requirement	Status
The Website has titled sections grouping symbols within a trait/type	Completed
The tool allows the creation of titled sections grouping multiple symbols.	Not Completed
Code entities pages on the website have a ”go-to source” feature (e.g. button), which redirects to the implementation of the entity in the codebase	Not Completed
Pages on the website with code examples have a feature to open the examples in Compiler Explorer	Not Completed
On the website, code examples can have supplementary code. The supplementary code only appears when trying to open the example in Compiler Explorer	Not Completed
The Tool generates the latest version of the documentation in a CI/CD pipeline	Not Completed
The Tool automatically deploys the website in the CI/CD pipeline	Not Completed
The Tool uses a configuration file to specify a list of folders and files to search/exclude for articles	Not Completed
The Website supports switching between a dark and a light theme	Not Completed
The Tool exposes useful library functions that allow querying the collected information about documentation.	Not Completed

Won't

Least critical or out of scope; not planned for this release. [2]

- The Tool supports configuring the documentation root for dependencies to allow the generation of links to external symbols.
- The tool has documentation quality metrics in the CI, just like test coverage metrics. These analytics might be useful, e.g. % of parameters documented, % of method documented.
- The Tool can provide statistics on the project and feedback on the documentation's completeness by pinpointing files or specific symbols that need more work.
- The Tool can provide feedback about the documentation's quality (including accuracy and freshness) by utilizing a large language model.

Non-Functional Requirements

- The Tool needs to use the Hylo compiler's frontend as a library.
- The Tool needs to be written in Swift.
- References to symbols must look like references in Hylo code.
- The CI/CD Pipeline must only generate documentation for the latest commit
- The documentation must be accessible without JavaScript enabled (although certain features like navigation and search might not work without it)
- The Google Chrome lighthouse score of the website must be at least 90% in all categories, preferably 95%

[2] We did not consider implementing them in the software project scope, but we kept these in mind when designing and implementing the system so that future work on these would be possible.

B

Syntax Reference of Documentation

This appendix contains an informal syntax specification used for documentation. It includes the syntax already implemented into the tool, so it is a great reference for users and future contributors to see what the tool is capable of. The reference is also available in our repository. [1]

[1] https://gitlab.ewi.tudelft.nl/cse2000-software-project/2023-2024/cluster-e/04d/automated-documentation-generation-tool/-/blob/main/SyntaxReference.md

Documentation Syntax Reference

Overview

The Hylo Documentation Compiler converts Markdown-based text into rich documentation for Hylo modules. The tool extracts documentation comments from two places: standalone Markdown articles and comment blocks embedded inside `.hylo` source files.

Embedded documentation

The documentation comment block

Consecutive lines starting with 3 slashes (`///`) constitute a comment block:

```
/// This is a valid comment line.
/// This is also valid and it's part of the same block.

/// This line is part of another comment block.
```

It is required to place a space after the 3 slashes unless it is an empty line. This space will be stripped away from the actual content.

The supported Markdown syntax

Markdown syntax is supported in documentation comments according to the CommonMark specification with some differences:

- Level 1 headings are reserved for declaring special sections (`# Parameters:`) and the file-level modifier (`# File-level`).
- Inline code snippets surrounded by double backticks are reserved for Hylo code references (``` ``MyType.myFunction(x:,y:)`` ```) that are resolved and checked in the scope of the symbol they appear in.

Example:

```
/// I *like* **markdown** _a lot_
///  - It is very readable
///  - It is `easy to write`
///
///     This becomes a code block because of the 4 leading spaces.
```

The first paragraph of the block will always be parsed as that block's Summary. All other Markdown that follows until the next level one heading (in the case that special sections are present) or the end of the block is considered as that block's Description.

```
/// This is the summary
///
/// This is the description
///   - This is also part of the description
/// ## So is this
```

Special Sections

Special sections are signaled by level one headings and document a specific part of a symbol. They make use of predefined styles and layouts in the website generation. A special section will always start with a level one heading with the first word being that section's title. Depending on what the block is associated with only some special sections are available. A special section's content can either be

- a list

  ```
  /// # Preconditons:
  ///  - width //< height – the rectangle should be
  ///    longer vertically than horizontally
  ///  - width and height must be positive
  ```

- or a one-line paragraph

  ```
  /// # Preconditon: width //< height – the rectangle should be longer vertically than horizontally
  /// # Precondition: width and height must be positive
  ```

Note that the less than (`<`) character has been escaped.

Parameter/Generic sections

All special sections titles are followed by a colon `:`, except Parameter and Generic Parameter sections in their inline(singular) form. In their list/plural form, i.e. `# Parameters:` and `# Generics:` they still have the colon, but in their inline forms they do not: `# Parameter` and `# Generic`. This is because these inline sections have a special syntax where the title is followed by a space, then the name of the parameter, then a colon `:`, and then the parameter description:

```
/// # Parameter myParameter1: Example parameter description
/// # Parameter myParameter2: Example parameter description
```

And the list form:

```
/// # Parameters:
///   - myParameter1: Example parameter description
///   - myParameter2: Example parameter description
```

The parameters are validated. A parameter or generic parameter can only be documented if is present in that symbol's declaration.

Scope of Embedded Documentation

Documentation comment blocks can have two different scopes:

- A **symbol-level** documentation comment annotates the symbol declaration it preceeds, and it needs to be placed before a symbol declaration, possibly separated by spaces.
  Example:

  ```
  /// Checks a necessary condition for making forward progress.
  ///
  /// # Parameters:
  ///   - condition: The condition to test.
  ///   - message: A description of the error
  public fun precondition(_ condition: Bool, _ message: String) {
    ...
  }
  ```

- A **file-level** documentation comment describes the source file as a whole, and they are identified by starting with a `# File-Level` heading.
  Example:

  ```
  /// # File-level:
  ///
  /// The first paragraph is the summary
  ///
  /// This is the description
  ///
  /// # See also:
  ///   - [this other file](MyModule/article.hylodoc)
  ///   - ``MyModule.myFunction(x:y:_:)``
  ```

Symbol Documentation

A documentation comment block preceding a symbol declaration is a symbol-level documentation, unless marked file-Level by a heading.

The first paragraph is the summary, then the rest of the content until the first section is the description. Example:

```
/// This is the summary
///
/// This is the description.
///
/// This is still the description.
/// # See Also:
///  - This is part of the first section.
trait A { ... }
```

Section Types:

1. `# See Also:` - inline, list
   The See Also section is used to provide references to related symbols, articles, or external resources.
   Available for: all declarations.
   Example:

   ```
   /// # See Also:
   ///   - ``MyModule.relatedFunction(_:)``
   ///   - [External Resource](https://example.com/related-topic)
   ```

2. `# Invariant:` - inline, `# Invariants:` - list
   The Invariant section is used to document invariants or conditions that must hold true for the associated type or binding. Available for: **Product Types, Traits, Binding**.

```
/// # Invariant: `startIndex < endIndex`.
/// # Invariant:
///   - The `capacity` must be greater than `count`.
///   - The earth must be kept spinning
```

3. `# Parameter` - inline, `# Parameters:` - list
   The Parameter section is used to document the parameters of a function, method, subscript, or initializer.
   Available for: Function, Method declaration, Method Bundle implementation, Subscript declaration.
   Example:

```
/// # Parameter label: A brief description of the `label` parameter.
/// # Parameters:
///   - label: A brief description of the `label` parameter.
///   - value: A description of the `value` parameter.
```

4. `# Generic` - inline, `# Generics:` - list
   The Generic(s) section is used to document the generic parameters of a declaration.
   Available for: function, method declaration, initializer and product type.
   Example:

```
/// # Generic T: The type of elements in the array.
/// # Generics:
///   - T: The type of elements in the array.
///   - U: The type of the transform function's argument.
```

5. `# Throws:` - inline, list
   The Throws section is used to document the errors that a function, method, subscript, or initializer can throw.
   Available for: function, method declaration, method Bundle implementation, subscript declaration, subscript bundle implementation.
   Example:

```
/// # Throws: An `InvalidArgumentError` if the provided argument is invalid.
```

6. `# Precondition:` - inline, `# Preconditions:` - list
   The Precondition section is used to document the preconditions that must be met before a function, method, subscript, or initializer is called.
   Available for: function, method declaration, method bundle implementation, subscript declaration, subscript bundle implementation, initializer.
   Example:

```
/// # Precondition: The `index` must be within the bounds of the array.
/// # Preconditions:
///   - The `start` index must be less than or equal to the `end` index.
///   - The `capacity` must be greater than or equal to the requested `newCapacity`.
```

7. `# Postcondition:` - inline, `# Postconditions:` - list
   The Postcondition section is used to document the conditions that must hold true after a function, method, subscript, or initializer has executed.
   Available for: Function, Method declaration, Method Bundle implementation, Subscript declaration, Subscript bundle implementation, Initializer.
   Example:

```
/// # Postcondition: `array.count == 0`
/// # Postconditions:
///   - The `capacity` is greater than or equal to the `count`.
///   - The `isEmpty` property returns `true` if the `count` is zero.
```

8. `# Returns:` - inline, list
   The Returns section is used to document the return value of a function or method.
   Available for: Function, Method declaration, Method Bundle implementation.
   Example:

```
/// # Returns: The sum of all elements in the array.

/// # Returns:
///  - true on success
///  - false on failure
```

9. `# Yields:` - inline, list
   The Yields section is used to document the values yielded by a subscript or computed property.
   Available for: subscript/property declaration, subscript/property bundle implementation.
   Example:

```
/// # Yields: The element at the specified `index`.
```

Supported Symbol Types and Their Available Sections

Different special sections are available for different declaration types.

1. Product type
   - `# See Also:` (inline, list)
   - `# Invariant:` (inline), `# Invariants:` (list)
   - `# Generic:` (inline), `# Generics:` (list)
2. Trait
   - `# See Also:` (inline, list)
   - `# Invariant:` (inline), `# Invariants:` (list)
3. Type alias
   - `# See Also:` (inline, list)
4. Associated type
   - `# See Also:` (inline, list)
5. Associated value
   - `# See Also:` (inline, list)
6. Binding
   - `# See Also:` (inline, list)
   - `# Invariant:` (inline), `# Invariants:` (list)
7. Operator
   - `# See Also:` (inline, list)
8. Function
   - `# See Also:` (inline, list)
   - `# Parameter` (inline), `# Parameters:` (list)
   - `# Generic` (inline), `# Generics:` (list)
   - `# Throws:` (inline, list)
   - `# Precondition:` (inline), `# Preconditions:` (list)
   - `# Postcondition:` (inline), `# Postconditions:` (list)
   - `# Returns:` (inline, list)
9. Method declaration
   - `# See Also:` (inline, list)
   - `# Parameter` (inline), `# Parameters:` (list)
   - `# Generic` (inline), `# Generics:` (list)
   - `# Throws:` (inline, list)
   - `# Precondition:` (inline), `# Preconditions:` (list)
   - `# Postcondition:` (inline), `# Postconditions:` (list)
   - `# Returns:` (inline, list)
10. Method bundle implementation
    - `# See Also:` (inline, list)
    - `# Throws:` (inline, list)
    - `# Precondition:` (inline), `# Preconditions:` (list)
    - `# Postcondition:` (inline), `# Postconditions:` (list)
    - `# Returns:` (inline, list)
11. Subscript/property declaration
    - `# See Also:` (inline, list)
    - `# Parameter` (inline), `# Parameters:` (list)
    - `# Generic` (inline), `# Generics:` (list)
    - `# Throws:` (inline, list)
    - `# Precondition:` (inline), `# Preconditions:` (list)
    - `# Postcondition:` (inline), `# Postconditions:` (list)
    - `# Yields:` (inline, list)
12. Subscript/property bundle implementation
    - `# See Also:` (inline, list)
    - `# Throws:` (inline, list)
    - `# Precondition:` (inline), `# Preconditions:` (list)
    - `# Postcondition:` (inline), `# Postconditions:` (list)
    - `# Yields:` (inline, list)
13. Initializer
    - `# See Also:` (inline, list)
    - `# Parameter` (inline), `# Parameters:` (list)

- `# Generic` (inline), `# Generics:` (list)
- `# Throws:` (inline, list)
- `# Precondition:` (inline), `# Preconditions:` (list)
- `# Postcondition:` (inline), `# Postconditions:` (list)

Articles

Markdown articles can be written in `.hylodoc` files.

A **level 1 heading** may only appear at the beginning of the article, and, if present, is interpreted as the main title of the article, and removed from the content.

Hylodoc references are supported in articles, and they are resolved in the scope of the module in which they appear in.

Here's the section restructured in a more reference manual style:

Syntax of Links

URLs

The following Markdown syntax for URLs is supported:

- Autolinks: `<https://example.com>`
- Full Markdown links: `[title](url)`
- Images: `![title](https://example.com/logo.png)`

Implicit links are not supported.

Local File References

Usage:

- Full Markdown links: `[title](relative/path/to/file.ext)`
- Images: `![alt text](path/to/image.png "title")`

Only relative paths are supported. These can refer to the following:

- Articles (`.hylodoc` files)
- Folders
- Source code (`.hylo` files)
- Other assets (PDF, images, etc.)

`.hylodoc` files, folders, and source files are resolved to their corresponding documentation pages. Other assets are copied to the output directory for download, preserving their relative tree structure.

Examples:

```
[Article](./article.hylodoc)
[Article](../src/article.hylodoc)
![Image](../public/logo.png "Logo")
[Whitepaper](../public/MVS.pdf)
[Source File](main.hylo)
[Folder](../)
[Folder](./)
```

Symbol References

Usage: ``` ``MyModule.myFunction(x:y:_:)`` ```

Parameter labels can be omitted if the reference is unambiguous: ``` ``MyModule.myFunction`` ```

> Note: Currently, only the latter syntax is implemented, and it is a fatal failure if the overload set contains more than 1 declaration.

C

Details of API Syntax Design

This appendix is an informal discussion of the design process behind choosing the design for the API documentation syntax for Hylo. First, we will explore how we should embed documentation in the source code, then we follow with designing the specific internal syntax of the comment's content.

Comparison Between Different Outer Syntaxes

First, we will discuss ways to Include the documentation syntax within source code. We can already assume that the content will be styled as MarkDown, as it is the most readable and well-known for authoring simple, structured and styled content.

```
/// This is the summary.
///
/// Markdown __formatting__ is *supported* for richer
/// [richer](https://google.com/?search=rich)
/// content editing.
///
/// Here is a nice unordered list:
///  * item 1
///  * item 2
///
/// @param width  The width of the rectangle. Must be a **positive** number.
/// @param height The height of the rectangle.
/// @returns The calculated area.
public fun myMethod(width: Int, height: Int) -> Int {
    return width * height;
}
```

Figure C.1: Swift and Rust convention for documentation comments: line comments with preceding ///.

Figure C.2 shows that mixing markdown and this docstring shell type would be error-prone because of the preceding stars. The problem is that those preceding stars are also optional, or multiple consecutive ones could be placed in the beginning, and they all would be, even though only historically – we could make an exception, but because of the familiarity, nobody would assume something different, which leads to confusion.

What if the comments did not include any stars on the left? We can see this version in Figure C.3. It appears to lack cohesion within the documentation comment block; the whitespace between paragraphs of the documentation entirely separates some content from others, and while visually scanning with our eyes towards the top, the top of the comment can only be noticed when the text's colour changes to a different colour, or notice the little `/**` token.

We are convinced that adding leading characters before the documentation is helpful, but they should

```
/**
 * This is the summary
 *
 * Markdown __formatting__ is *supported* for  [richer](https://google.com/?search=rich)
 * content editing.
 *
 * Here is a nice unordered list:
 *  * item 1
 *  * item 2
 **oopsie this was intended to be a list item but now got concatenated to previous
 *list item.
 *
 * *I want this to be italic*
 **I also want this to be italic*
 *
 ** ...But I want to ignore multiple stars in the beginning...
 **
 **
 * @param width  The width of the rectangle. Must be a **positive** number.
 * @param height The height of the rectangle.
 * @returns The calculated area.
 */
public fun calculateArea(width: Int, height: Int) -> Int {
    return width * height;
}
```

Figure C.2: Java and TypeScript convention for documentation comments: multiline comments with preceding stars.

```
/**
 Calculates the area of a rectangle.

 Markdown __formatting__ is *supported* for
[richer](https://google.com/?search=rich)
 content editing.

 Here is a nice unordered list:
  * item 1
  * item 2

 @param width  The width of the rectangle. Must be a **positive** number.
 @param height The height of the rectangle.
 @returns The calculated area.
*/
public fun calculateArea(width: Int, height: Int) -> Int {
    return width * height;
}
```

Figure C.3: When the documentation comment doesn't include stars on the left at all

not be $*$-s. Pythonic comments look great when mixed with MarkDown content, but there is one problem: `\#`-es are already reserved for compiler directives such as `\#filepath` or `\# #if os(macOS)`.

Assembly-style comments, using `;` as the comment-beginning character, look a bit weird at the beginning of lines, although this might be because of familiarity bias.

Overall, the Swift/Rust-style comment looks the best while maintaining the language's clear and unambiguous syntax. Next, we will compare the different alternatives for the syntax within the documentation content.

Comparison Between Inner Comment Syntaxes

Swift's documentation style looks very nice and can provide a great hierarchy as shown in Figure C.6. However, there is one problem regarding the error-proneness of the special section titles: if mistyped,

```
# This is the summary.
#
# Markdown __formatting__ is *supported* for
# [richer](https://google.com/?search=rich)
# content editing.
#
# Here is a nice unordered list:
#  * item 1
#  * item 2
#
# @param width  The width of the rectangle. Must be a **positive** number.
# @param height The height of the rectangle.
# @returns The calculated area.
public fun calculateArea(width: Int, height: Int) -> Int {
    return width * height;
}
```

Figure C.4: When using Pythonic comments

```
; This is the summary
;
; Markdown __formatting__ is *supported* for
; [richer](https://google.com/?search=rich)
; content editing.
;
; Here is a nice unordered list:
;  * item 1
;  * item 2
;
; @param width  The width of the rectangle. Must be a **positive** number.
; @param height The height of the rectangle.
; @returns The calculated area.
public fun calculateArea(width: Int, height: Int) -> Int {
    return width * height;
}
```

Figure C.5: Assembly-style comments

```
/// This is the summary.
///
/// This is the description.
///
/// It can also be multiple paragraphs long.
///
/// - Parameters:
///   - width:  The width of the rectangle. Must be a **positive** number.
///   - height: The height of the rectangle.
///
/// - Requires:
///   - width < height – the rectangle should be longer vertically than horizontally
///   - width and height must be positive
///
/// - Returns: The calculated area.
public fun myMethod(width: Int, height: Int) -> Int {
    return width * height;
}
```

Figure C.6: Swift-style documentation comments: The first paragraph is the summary, followed by a description and structured API documentation.

they are interpreted as a regular markdown list and considered part of the description without emitting a warning.

```
///
/// Markdown __formatting__ is supported for [richer](https://google.com/?search=rich)
/// content editing. This is the description.
///
/// It can also be multiple paragraphs long.
///
/// - Parameter width:  The width of the rectangle. Must be a **positive** number.
/// - Parameter height: The height of the rectangle.
/// - Requires: width < height – the rectangle should be longer vertically
/// - Requires: width and height must be positive
/// - Returns: The calculated area.
public fun myMethod(width: Int, height: Int) -> Int {
    return width * height;
}
```

Figure C.7: Swift-style documentation comments using the inline variant.

Swift also supports the writing of inline item descriptions, so that no hierarchy needs to be introduced when we want to spare vertical space as shown in Figure C.7. This looks a bit unorganized and more bloated, mainly because we include the full Parameter keyword instead of Java's @param width syntax (here, the information vs bloat rate is lower).

```
/// First paragraph is the summary.
///
/// This is the description.
/// Lorem ipsum dolor sit amet.
///
/// - Parameter width:  The width of the rectangle. Must be a **positive** number.
/// - Parameter height: The height of the rectangle.
///
/// - Requires: width < height – the rectangle should be longer vertically
/// - Requires: width and height must be positive
///
/// - Returns: The calculated area.
public fun myMethod(width: Int, height: Int) -> Int {
    return width * height;
}
```

Figure C.8: Swift-like inline but separated sections

We can try to add back structure by separating them with an empty line, as show in Figure C.8. This layout looks much more organized now, but we still consider the hierarchical version more likeable, as it looks like sections had a title.

At this point, we realized that MarkDown already had a well-established syntax for titles, so we tried applying it to the syntax's design:

The syntax illustrated in Figure C.9 has the same syntax essentially as Swift, except that section titles are replaced by level 1 MarkDown headings, denoted by #-es. This looks great and allows verifying whether the user has typed a correct heading label if we reserve the level 1 heading for declaring special sections exclusively.

Lastly, we considered some other syntaxes but found nothing more usable and readable than the previously introduced syntax. These are introduced in Figure C.10

```
/// First paragraph is the summary
///
/// Markdown __formatting__ is *supported* for richer
/// content editing. This is the description.
///
/// It can also be multiple paragraphs long.
///
/// # Parameters:
///   - width:  The width of the rectangle.
///   - height: The height of the rectangle.
///
/// # Returns:
///   - The calculated area if width is even
///   - 0 otherwise (here we can also include markdown)
///
/// or inline:
/// # Returns: the area of the rectangle
/// # Parameter width:  The width of the _rectangle_.
/// # Parameter height: The height of the _rectangle_.
public fun myMethod(width: Int, height: Int) -> Int {
    return width * height;
}
```

Figure C.9: Swift's syntax except section titles are denoted by #-es.

```
/// This is the summary
///
/// This is the long description
///
/// .params:
///     .x: the description of x
///     .y: description of y
/// .returns: the sum of x and y.
fun a(x: Int, y: Int) -> Int {
    return x + y
}


/// This is the summary
///
/// Markdown __formatting__ is *supported* for richer
/// [richer](https://google.com/?search=rich)
/// content editing.
///
/// It can also be multiple paragraphs long.
///
/// @param     width  The width of the rectangle. Must be a **positive** number.
/// @param     height The height of the rectangle.
/// @requires  width < height, width and height > 0
/// @returns   The calculated area if width is even, 0 otherwise
public fun myMethod(width: Int, height: Int) -> Int {
    return width * height;
}


@doc(
    "This is the summary",
    """This is the long description""",
    .params((
        x: "the description of x",
        y: "description of y"
    )),
    .returns("the sum of x and y")
)
fun a(x: Int, y: Int) -> Int {
    return x + y
}
```

Figure C.10: Miscaleneaus syntax considerations

D

The Use of Generative AI Tools for the Project

Technical Writing

Generative AI tools played a helpful role in improving the quality of our technical documentation throughout the development of our documentation extractor tool for Hylo. These tools were used for various purposes, including:

Improving Flow: AI-assisted writing tools helped ensure that our documentation was logically structured and easy to follow. The AI suggested more coherent ways to present information, making the documentation more accessible to users.

Eliminating Redundancy and Verbosity: The AI tools identified and removed repetitive and verbose sections, streamlining the text to maintain clarity and conciseness. This was important in keeping our documentation concise and user-friendly.

Correcting Grammar Mistakes: Automated grammar-checking features were utilized to identify and correct grammatical errors, ensuring that the documentation maintained a high standard of language accuracy. We used the premium version of Grammarly for this purpose.

Reviewing and Getting Feedback on Reports: AI tools provided useful feedback on our reports, highlighting areas for improvement and suggesting enhancements. This iterative process improved the overall quality of our written materials.

We generally used prompts in the following form:

```
<Short introduction of the context of the report and the project>
<Inroduction of the task>
```
<The input text, written in a MarkDown code fragment that GPTs
                                        are highly trained on.>
```
Please give a detailed feedback.
```

where `<Introduction of the task>` varied between different use cases, such as “Please review the following paragraph/section and give feedback on flow and grammar:“, “Please review the following paragraph and eliminate the redundancy related to <...>:“

Programming

Generative AI tools also proved beneficial in the coding aspects of our project. Their capabilities were employed in several key areas:

Copilot Autocompletion: AI-powered code autocompletion tools like GitHub Copilot enhanced our productivity. These tools assisted in writing code more efficiently by predicting and suggesting code completions. This was particularly useful during long refactorings, where the AI recognized patterns and provided relevant code suggestions, thereby reducing the manual effort required.

Help Debugging Code: Debugging is a critical phase in software development, and AI tools were useful in this area. By analyzing our code, these tools identified potential issues and provided suggestions for fixes, which helped us resolve bugs more quickly and effectively.

E

Syntax of Links

Syntax of References

Introduction

Documentation in Hylo code, and in .hylodoc files is written in markdown. We would like to support local file refernces and references to code entities in markdown (similar to as if they were in code). This document describes different alternatives that were considered when designing the syntax.

We need to account for the following use cases in links:

- Reference a local file within the project by relative path
- Reference a local file within the project by an absolute project path
- A web link (http://, https://, ftp://, mailto: etc.)
- Reference to an arbitrary named Hylo code entity

General Link Syntax in Markdown

There are 3 types of links in markdown:

- Implicit links: `https://something.com`
- Autolinks: `<https://something.com>`
- Full links: `[Link Text](https://something.com "title")`

The first option (implicit links) is not in the CommonMark standard but many tools support it (Github, IntelliJ, VSCode). Often, it only works for links starting with a common protocol like `http://`, `https://`, `ftp://` but not for `mailto:` or `tel:`.

The second option is included in the standard and also works for other protocols. `https://`, `tel:`, `mailto:`, etc.) It is required to include the protocol in the link (except for email addresses) because otherwise it might be interpreted as an html tag: `<localhost>`.

The third option is the most common and most versatile. It allows for custom link text and title, and is not limited to requiring the protocol.

Syntax Design for the Special Cases

Local file references and code symbol references need special care when incorporating them into the existing MarkDown syntax. We need to make them expressible easily and unambiguously.

Local File References

Full markdown links:

- `[Link Text](./module.md)` - file in the current directory
- `[Link Text](../module.md)` - file in the parent directory
- `[Link Text](/module.md)` - path starting from the root module
- `[Link Text](~/module.md)` - path starting from the root directory (alternative syntax, not common)
- `[Link Text](module.md)` - file in the current directory (without the `./` prefix)
  - There is possibility of ambiguity here, as `~` is a valid file/folder name in windows. If we want to refer to folder or file named `~`, we can use `./~`, thus keeping the `~` prefix reserved for referring to the root module.

Github uses this syntax exclusively for referring to local files. They use the `/` prefix to refer to the repository root but advise to use relative links generally, as absolute links might not work well in clones of the repository or when the scope of the project became bigger (e.g. we made our project a subfolder inside another folder).

Autolinks:

- `<module.md>`
- `<./module.md>`
- `<../module.md>`
- `</module.md>`
- `<~/module.md>`

Unfortunately, local files and folders can have names like `strong`, `i`, which are also HTML tags which would become ambiguous: `<strong>` `</i>`. For this reason, most MarkDown parsers do not support this syntax for local files. There are multiple possible solutions to this problem:

- **In case of ambiguity, interpret it as an html tag.** The problem is that we cannot be sure what tags exist within the context of evaluating the markdown, the user might even define their own custom elements. Furthermore, custom elements are allowed to include `.` in their names, and conversely, file names are allowed to not include an extension.
- **Restrict or disallow the use of HTML elements.** Markdown is a small language and it cannot be used to express the needs of every user. Therefore, providing an escape hatch for users to express their needs in HTML is a useful feature that we should not sacrifice.
- **Require `./`, `~/` or `../` prefixes for local file references.** There are no such tags in HTML, so this would eliminate the ambiguity. The resulting references would look like this: `<./module.md>`, `<../module.md>`, `<~/module.md>`. This is an intuitive syntax, and it is clear from looking at the reference that it is referring to a file.
- Don't allow shorthand for referencing local files. This would leave us with the full link syntax for referring to local files: `[Link Text](./module.md)`. This is the most unambiguous syntax, but it is also the most verbose. The upside is that local file references would work well with existing tooling, and it is less work for us to implement.

Symbol References

Referring to code entities is a common use case in documentation. We would like to provide a way to refer to these entities unambiguously, and respecting the scoping rules of the language (resolving the reference from the parent scope relative to where the documentation is written).

We assume that when the documentation is outside any top-level code entity (e.g. within a file-level documentation comment or in a `.hylodoc` file), we are in the scope of the parent module.

Here are some examples of how we could refer to code entities:

```
MyModule.myFunction(x:y:_:)
ParentModule.MyModule.myFunction(x:y:_:)
myFunction(x:y:_:)
MyModule
ParentModule.MyModule
```

It will be possible to refer to any declaration

Writing the Code Entity Name Inside the URL

Rust uses the following syntax:

```
](std::collections::HashMap::new)
```

Rust has the advantage that it does not have function overloading and its entity paths are separated by `::`.

Hylo paths contain `.` and `:` characters, which are also used in URLs. This makes it more difficult to distinguish between a URL and a code entity reference.

```
](MyModule.myFunction(x:y:_:))
```

Custom Protocol

Writing a protocol name for referencing Hylo code entities is not the nicest solution, but it is certainly unambiguous. This would work for both autolinks and full links.

```
<hylo:myMethod(_:_:)>
](hylo:myMethod(_:_:))
```

Double Backticks

This is what Docc uses. It is a concise syntax, and it is clear that it is referring to a code entity, without ambiguity with URLs.

```
``MyModule.myFunction(x:y:_:)``
```

Final Design

URLs

We support autolinks and full Markdown links for arbitrary URLs (except ones with the `file://` protocol). We will not support implicit links, as they are not part of the CommonMark standard.

Local File References

We will only support the full link syntax and the image syntax for local file references. Only relative paths are supported, since depending on the context, `/` might refer to different places, e.g. root of module, root of repository. Furthermore, relative file references work perfectly in the markdown editors of both IntelliJ and VSCode, so we don't have to invest in new editor support.

One can refer to arbitary assets within the documented modules. The assets will be copied to the website's output. References to hylodoc files, folders and source files will be resolved to their corresponding pages, while other assets are simply downloaded when the user navigates to them.

Examples:

```
[Article](./article.hylodoc)
[Article](article.hylodoc)
[Article](../src/article.hylodoc)
![Image](../public/logo.png "alternative text")
[Whitepaper](../public/MVS.pdf)
[source file](main.hylo)
[folder](../)
[folder](./)
[folder](.)
```

Symbol References

We will use the double backticks syntax for code entity references, just like Swift DocC does. This is a concise syntax, and it is clear that it is referring to a code entity, without ambiguity with URLs.

```
``MyModule.myFunction(x:y:_:)``
```

F

Design of the Generated Website

hylo Documentation

Hylo › Standard Library › Geometry › Vector2.hylo

Getting Started Language Tour GitHub

Press / for Search

- Standard Library
  - Concurrency
  - Core
    - Numbers
    - Bool
    - Assert
    - Collection
    - Comparable
      - *trait Comparable*
      - *fun min(_:_:)*
      - *fun max(_:_:)*
      - *typealias Result*
    - Comparisons
    - Collection Examples
    - RegExp

Product Type

type Vector2

in namespace Hylo.Lorem.Ipsum

Represents a 2D vector as a pair of two floating point numbers.

```
type Vector2 : Deinitializable,
               TupleConvertible
```

Declaration

Details

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

```
let original = Vector2(x: 2.0, y: 3.0)
let other = original.mult(by: 2)
let myString = "Hello World!"
print(other) // other = Vector2(4, 6)
```

Members

Stored Properties

`var x: Float64`
First component of the vector.

`var y: Float64`
Second component of the vector.

Initializers

`init(x: Float64, y: Float64)`
Initializes the vector with two components.

Subscripts

`property length: Float64 { let }`
The length of the vector computed by sqrt(x ^ 2 + y ^ 2)

`subscript(_ n: Int): Int { let, inout, set }`
Yields the n-th component of the vector where n ∈ { 0, 1 }

- `let` Yields the n-th component of the vector using bit-manipulation magic
- `inout` Lets you mutate n-th element in-place

`property manhattan: Float64 { let }`
The sum of the components: x + y

Methods

`fun multiplied(by scale: Float64) → Vector2 { let }`
Returns the vector multiplied by scale, without mutating the original vector.

`fun toTuple() → (Float64, Float64) { sink }`
Extracts the tuple of components out of the vector.

`fun multiple() → (Float64, Float64) { sink, inout }`
Extracts the tuple of components out of the vector.

`static fun multiple() → (Float64, Float64)`
Extracts the tuple of components out of the vector.

Conditionally present if `Self.Element : Deinitializable`

Static Variables

`static let null: Self = .init(0, 0)`
A vector with length 0.

Types

`typealias Element = Float64`
Contained type of the collection.

`type SubDeclaration`
This is a product type declaration within this one.

Conformances

`conformance Vector2 : Deinitializable where Self.Element : Deinitializable`
The type can be destructed.

See Also

- Common Geometry Operations on Vector2
- Geometry.Render(vec:)
  Draws a given vector on the screen.

On this page

- Summary
- Details
- Invariants
- Members
  - Stored Properties
  - Initializers
  - Subscripts
  - Methods
  - Static Variables
  - Types
- See Also

Figure F.1: Design of a Product Type's page

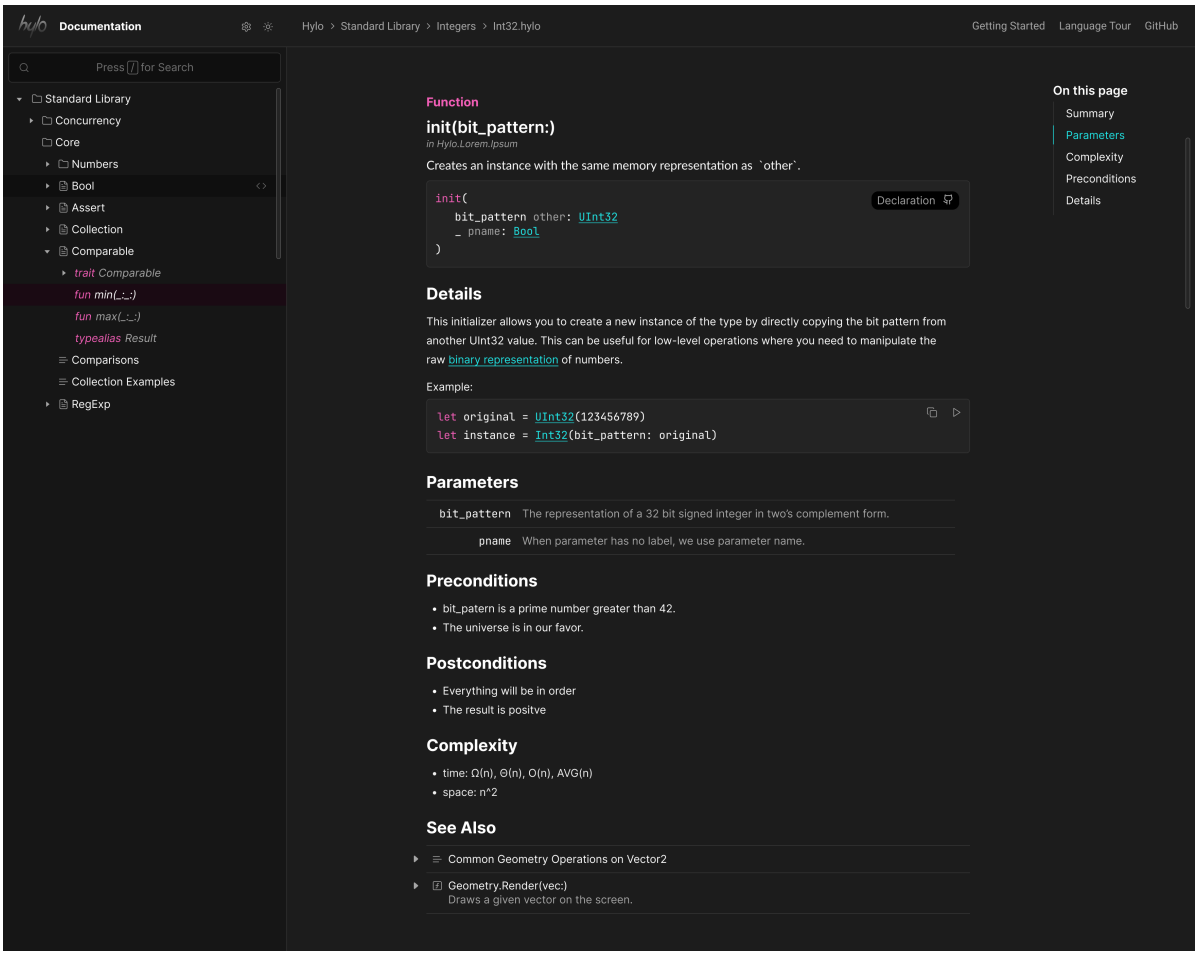


Figure F.2: Design of an Initializer's page

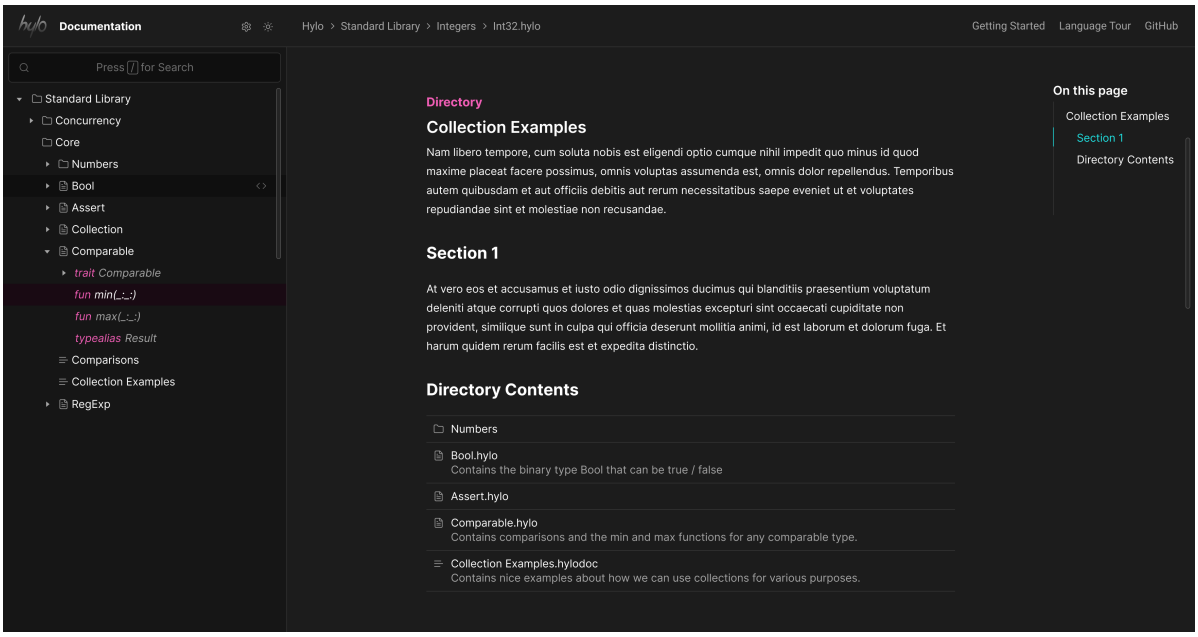


Figure F.3: Design of a Folder's page

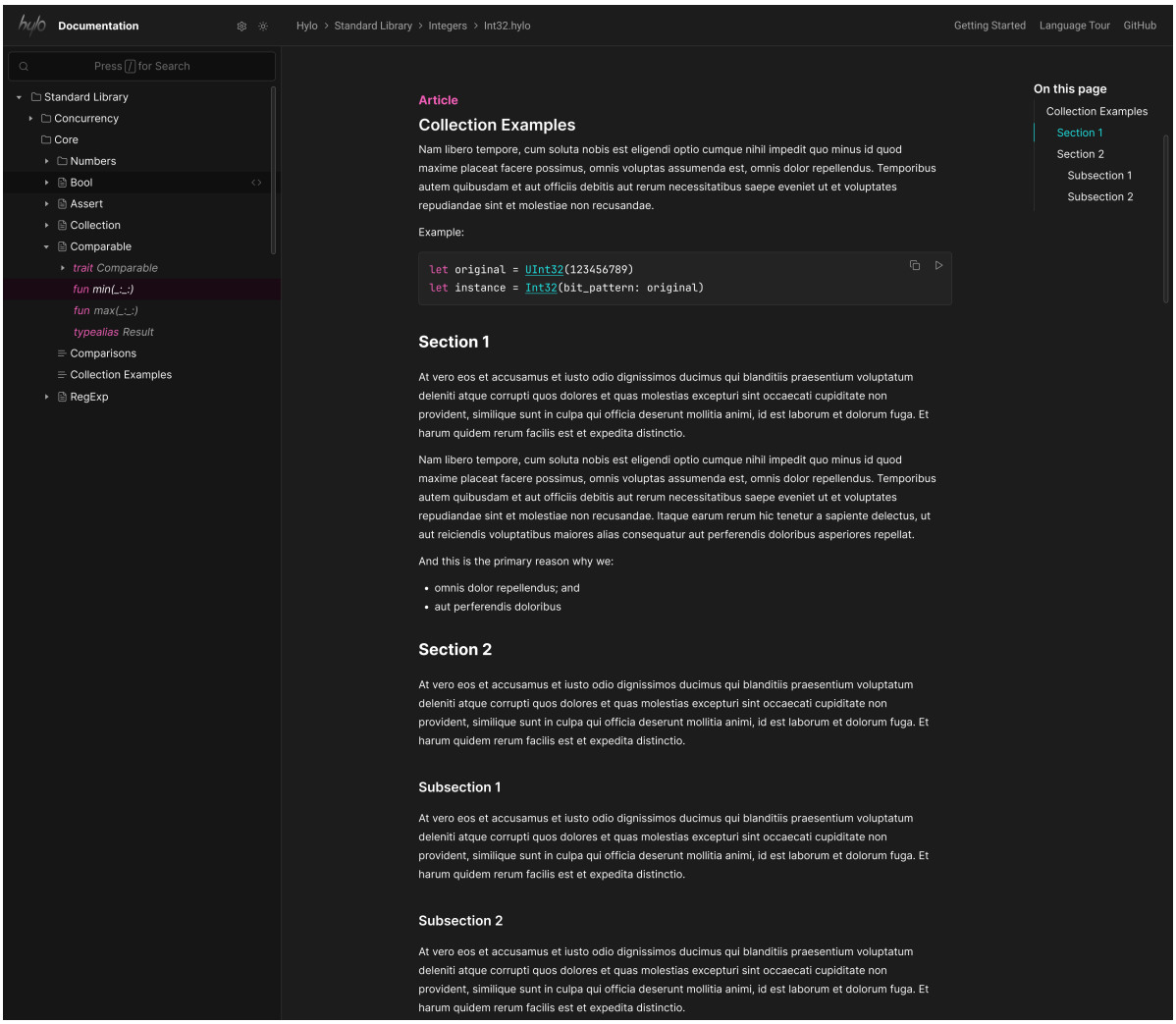


Figure F.4: Design of an Article's page

G

Performance Measurements

The following tables contain a series of performance measurements for three distinct states of the tool during optimization presented in chapter 10, Performance Analysis and Optimisation. These measurements have been performed by compiling the tool in release mode and running them on the Hylo Standard Library five consecutive times.

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Typed Program	1.34	1.37	1.52	1.45	1.41	1.418
Documentation Extraction	0.16	0.18	0.16	0.16	0.17	0.166
Website Generation	138.69	139.98	139.43	140.07	139.95	139.624
Full Pipeline	140.19	140.64	141.12	141.68	141.35	141.032

Table G.1: Baseline Measurements

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Typed Program	1.33	1.34	1.34	1.37	1.49	1.374
Documentation Extraction	0.17	0.16	0.16	0.16	0.16	0.162
Website Generation	103.58	102.34	102.87	102.81	102.67	102.854
Full Pipeline	105.08	103.84	104.36	104.34	104.32	104.388

Table G.2: Template Caching Measurements

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Typed Program	1.76	1.39	1.66	1.65	1.61	1.614
Documentation Extraction	0.16	0.16	0.16	0.16	0.18	0.164
Website Generation	2.83	2.98	2.56	2.35	2.62	2.668
Full Pipeline	4.75	4.53	4.38	4.17	4.41	4.448

Table G.3: Reduction of Navigation Rendering Measurements

H

HREC Checklist

			If YES please complete the Risk Assessment and Mitigation Plan columns below.	
ISSUE	**Yes**	**No**	**RISK ASSESSMENT**	**MITIGATION PLAN**
B: Location				
4. Will the research take place in a country or countries, other than the Netherlands?		X		
6. Will the research take place in a place/region or of higher risk – including known dangerous locations (in any country) or locations with non-democratic regimes?		X		
C: Participants				
7. Will the study involve participants who **may** be vulnerable and possibly (legally) unable to give informed consent? (e.g., children below the legal age for giving consent, people with learning difficulties, people living in care or nursing homes,).		X		
8. Will the study involve participants who **may** be vulnerable under specific circumstances and in specific contexts, such as victims and witnesses of violence, including domestic violence; sex workers; members of minority groups, refugees, irregular migrants or dissidents?		X		
9. Are the participants, outside the context of the research, in a dependent or subordinate position to the investigator (such as own children, own students or employees of either TU Delft and/or a collaborating partner organisation)? *It is essential that you safeguard against possible adverse consequences of this situation (such as allowing a student's failure to participate to your satisfaction to affect your evaluation of their coursework).*		X		
10. Is there a high possibility of re-identification for your participants? (e.g., do they have a very specialist job of which there are only a small number in a given country, are they members of a small community, or employees from a partner company collaborating in the research? Or are they one of only a handful of (expert) participants in the study?		X		
D: Recruiting Participants				
11. Will your participants be recruited through your own, professional, channels such as conference attendance lists, or through specific network/s such as self-help groups	X		Participants may feel obligated to participate as they are personally asked by someone they know. Their connection with the researcher may cause biased answers.	Will ensure that participants understand that their participation is voluntary.
13. Will you be recruiting your participants through a crowd-sourcing service, and/or involve a third party data-gathering service, such as a survey platform?		X		
14. Will you be offering any financial, or other, remuneration to participants, and might this induce or bias participation?		X		
E: Subject Matter *Research related to medical questions/health may require special attention. See also the website of the CCMO before contacting the HREC.*				
16. Will drugs, placebos, or other substances (e.g., drinks, foods, food or drink constituents, dietary supplements) be administered to the study participants? *If yes see here to determine whether medical ethical approval is required*		X		
18. Does the study risk causing psychological stress or anxiety beyond that normally encountered by the participants in their life outside research?		X		
19. Will the study involve discussion of personal sensitive data which could put participants at increased legal, financial, reputational, security or other risk? (e.g., financial data, location data, data relating to children or other vulnerable groups) *Definitions of sensitive personal data, and special cases are provided on the TUD Privacy Team website.*		X		
20. Will the study involve disclosing commercially or professionally sensitive, or confidential information? (e.g., relating to decision-making processes or business strategies which might, for example, be of interest to competitors)		X		
23. Does your research involve observing illegal activities or data processed or provided by authorities responsible for preventing, investigating, detecting or prosecuting criminal offences *If so please confirm that your work has been discussed with the appropriate legal advisors and approved by your Department/Faculty.*		X		
F: Research Methods				
24. Will it be necessary for participants to take part in the study without their knowledge and consent at the time? (e.g., covert observation of people in non-public places).		X		
25. Will the study involve actively deceiving the participants? (For example, will participants be deliberately falsely informed, will information be withheld from them or will they be misled in such a way that they are likely to object or show unease when debriefed about the study).		X		
26. Is pain or more than mild discomfort likely to result from the study? And/or, could your research activity cause an accident involving (non-) participants?		X		
27. Will the experiment involve the use of devices that are not 'CE' certified? *Only, if 'yes': continue with the following questions:*		X		
• Was the device built in-house?				
• Was it inspected by a safety expert at TU Delft? *If yes, please provide a signed **device report***				
• If it was not built in-house and not CE-certified, was it inspected by some other, qualified authority in safety and approved? *If yes, please provide records of the inspection*				
28. Will your research involve face-to-face encounters with your participants and if so how will you assess and address Covid considerations?		X		
29. Will your research involve **either:** a) "big data", combined datasets, new data-gathering or new data-merging techniques which might lead to re-identification of your participants **and/or** b) artificial intelligence or algorithm training where, for example biased datasets could lead to biased outcomes?		X		
G: Data Processing and Privacy				
30. Will the research involve collecting, processing and/or storing any directly identifiable PII (Personally Identifiable Information) including name or email address that will be used for administrative purposes only? (eg: obtaining Informed Consent or disbursing remuneration)	X		We will ask for consent.	We will document the participant's consent and ensure they are aware they can revoke their consent. Data will be stored securely.
31. Will the research involve collecting, processing and/or storing any directly or indirectly identifiable PIRD (Personally Identifiable Research Data) including videos, pictures, IP address, gender, age etc	X		The consent form will ask for the user's name.	This data will be anonymous, the participant may also choose not to answer.
32. Will this research involve collecting data from the internet, social media and/or publicly available datasets which have been originally contributed by human participants		X		
33. Will your research findings be published in one or more forms in the public domain, as e.g., Masters thesis, journal publication, conference presentation or wider public dissemination?		X		
34. Will your research data be archived for re-use and/or teaching in an open, private or semi-open archive?		X		

I

Questionnaire for Potential Developers

Setting up the task(s)

1. Download the following example hylo module.
2. Set up the project.
3. Extract the contents of the archive to a convenient location.
4. Check that the project is working by issuing a no-argument command.
5. Notify the organizers about the result.

First task

Generate the documentation website for the given `ExampleModule`.

Answer the following questions:

10. How much do you agree with the following:
"I found it easy to understand what arguments the CLI can receive."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

11. How much do you agree with the following:
"The CLI performs as I expected."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

12. Something I would change about the CLI is...

Task 2:

Start a webserver from the generated website files.
For example, if you have Python installed, run:
`
python3 -m http.server 8080 -d <generatedWebsitePath>
`

Access the website in your preferred browser.
If you used the previous command, the url is `http://localhost:8080/`

Without using the search bar, find out the preconditions for method `functionLarge`.

Questionnaire.

13. How much do you agree with the following:
"The navigation side panel gives me a good idea on how this codebase is structured."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

14. How much do you agree with the following:
"It was easy to navigate to the `functionLarge` method"

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

15. How much do you agree with the following:
"It was easy to find out what documentation was available for the functionLarge method."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

16. How much do you agree with the following:
"It was easy to find the preconditions for `functionLarge.`"

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

17. Should the navigation side panel be collapsible when you read a symbol's documentation?

Mark only one oval.

- ◯ True
- ◯ False

18. Please write any further feedback about your experience in this task.

Task 3

There is more information available about Module B in an article. Find the article that gives more information about Module B.

Questionnaire

19. How much do you agree with the following:
"The icons in the navigation side panel gave me a good idea of which pages in the website represent articles."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

20. How much do you agree with the following:
"It was easy to find the article I was looking for."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

Task 4

In the `ExampleModule`, there is a ModuleC folder which contains a file with one type declaration.

1. Write some short documentation about the type and the included method.
2. Create an article about this method and write some text in it.
3. Re-generate the documentation website (and restart the webserver).
4. Access the new website in your preferred browser.
5. Find the new documentation entries that you just created.

Questionnaire

21. How much do you agree with the following:
"I find the documentation syntax intuitive."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

22. How much do you agree with the following:
"I find it easy to document code using this tool."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

23. Please write any further feedback about your experience in this task.

Task 5

In the type that you just documented, write a new Special Section for the method, that is not allowed.

Try to re-generate the website.

Questionnaire

24. How much do you agree with the following:
"The errors give a clear understanding on what happened."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

25. How much do you agree with the following:
"It was easy to find out what sections are allowed and which are not."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

26. Please write any further feedback about your experience and in this task.

Final feedback

Try out the docs for the standard library: https://hylodoc.web.app/Sources/index.html

27. Something I would change about the visual design of this website is...

28. Something I would change about the functionality of this website is...

29. Something I would change about the syntax is...

30. Any other thoughts?

31. As a contributor to the Hylo language, would you use this project?

Mark only one oval.

- Yes
- No

This content is neither created nor endorsed by Google.

Google Forms

J

Contributor Questionnaire

Setting up the task(s)

1. Download the following example hylo module.
2. Set up the project.
3. Extract the contents of the archive to a convenient location.
4. Check that the project is working by issuing a no-argument command.
5. Notify the organizers about the result.

First task

Generate the documentation website for the given `ExampleModule`.

Answer the following questions:

10. How much do you agree with the following:
"I found it easy to understand what arguments the CLI can receive."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

11. How much do you agree with the following:
"The CLI performs as I expected."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

12. Something I would change about the CLI is...

Task 2:

Start a webserver from the generated website files.
For example, if you have Python installed, run:
`
python3 -m http.server 8080 -d <generatedWebsitePath>
`

Access the website in your preferred browser.
If you used the previous command, the url is `http://localhost:8080/`

Without using the search bar, find out the preconditions for method `functionLarge`.

Questionnaire.

13. How much do you agree with the following:
"The navigation side panel gives me a good idea on how this codebase is structured."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

14. How much do you agree with the following:
"It was easy to navigate to the `functionLarge` method"

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

15. How much do you agree with the following:
"It was easy to find out what documentation was available for the functionLarge method."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

16. How much do you agree with the following:
"It was easy to find the preconditions for `functionLarge.`"

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

17. Should the navigation side panel be collapsible when you read a symbol's documentation?

Mark only one oval.

- ◯ True
- ◯ False

18. Please write any further feedback about your experience in this task.

Task 3

There is more information available about Module B in an article. Find the article that gives more information about Module B.

Questionnaire

19. How much do you agree with the following:
"The icons in the navigation side panel gave me a good idea of which pages in the website represent articles."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

20. How much do you agree with the following:
"It was easy to find the article I was looking for."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

Task 4

In the `ExampleModule`, there is a ModuleC folder which contains a file with one type declaration.

1. Write some short documentation about the type and the included method.
2. Create an article about this method and write some text in it.
3. Re-generate the documentation website (and restart the webserver).
4. Access the new website in your preferred browser.
5. Find the new documentation entries that you just created.

Questionnaire

21. How much do you agree with the following:
"I find the documentation syntax intuitive."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

22. How much do you agree with the following:
"I find it easy to document code using this tool."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

23. Please write any further feedback about your experience in this task.

Task 5

In the type that you just documented, write a new Special Section for the method, that is not allowed.

Try to re-generate the website.

Questionnaire

24. How much do you agree with the following:
"The errors give a clear understanding on what happened."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

25. How much do you agree with the following:
"It was easy to find out what sections are allowed and which are not."

Mark only one oval.

	1	2	3	4	5	
Stro	◯	◯	◯	◯	◯	Strongly agree

26. Please write any further feedback about your experience and in this task.

Final feedback

Try out the docs for the standard library: https://hylodoc.web.app/Sources/index.html

27. Something I would change about the visual design of this website is...

28. Something I would change about the functionality of this website is...

29. Something I would change about the syntax is...

30. Any other thoughts?

31. As a contributor to the Hylo language, would you use this project?

Mark only one oval.

- Yes
- No

This content is neither created nor endorsed by Google.

Google Forms

K

Consent Form

HyloDocCompiler Developer Study

You are invited to participate in a user testing study evaluating the the Hylo Documentation Compiler. This study is performed by Software Project Group 4D from the TU Delft Computer Science & Engineering Bachelor's study. Our project is to design a documentation syntax for Hylo language, and implement a tool that can extract documentation from code and generate an easy-to-navigate website. The tool can be used to generate documentation from the Hylo standard library, as well as other Hylo libraries.

The purpose of this study is to evaluate the design and usability of our prototype, and it will take you approximately 10 minutes to complete. You will be asked to fill out a consent form, then interact with the prototype on your own device, followed by an online questionnaire about the design and usability.

Your name will be collected for administrative purposes, as we require the consent form. The only additional personal data we collect is whether you are a contributor to the Hylo repository. This data will be anonymized and results will be presented in an aggregated fashion to analyse the participants' experience and feedback. The collected data will be deleted at the end of this course, by 1st of July, 2024, and only the aggregated results will be kept for our report.

Your participation in this study is entirely voluntary and you can withdraw at any time.

You are free to omit any questions.

To the best of our ability your answers in this study will remain confidential and anonymized.

Contact a.c.cazacu@student.tudelft.nl for any questions.

* Indicates required question

Untitled section

1. I understand the above. *

Mark only one oval.

- Yes
- No

Consent Form

2. Name *

3. I have read and understood the study information dated 2024-06-18. In case I needed, I have been able to ask questions about the study and my questions have been answered to my satisfaction. *

Mark only one oval.

- Yes
- No

4. I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time, without having to give a reason. *

Mark only one oval.

- Yes
- No

5. I understand that taking part in this study involves starting an website on my personal machine and answering an online questionnaire regarding the design and the user experience of the prototype. *

Mark only one oval.

- Yes
- No

6. I understand that the information I provide will be used for aggregating statistics regarding the design and usability of the prototype. *

Mark only one oval.

- Yes
- No

7. I understand that the information collected about me will be anonymized and stored securely through Google Forms. *

Mark only one oval.

- Yes
- No

8. I agree that my information can be quoted in research outputs in an anonymized fashion. *

Mark only one oval.

- Yes
- No

9. I understand that the collected data will be kept until the end of the course, and deleted by the start of July 2024 except the aggregated results which will be presented in the project's report. *

Mark only one oval.

- Yes
- No

L

Chrome Lighthouse Report After the Optimisations

https://hylodoc.web.app/Sources/DynamicBuffer/DynamicBuffer/index.html

Performance	Accessibility	Best Practices	SEO
100	100	100	100

100

Performance

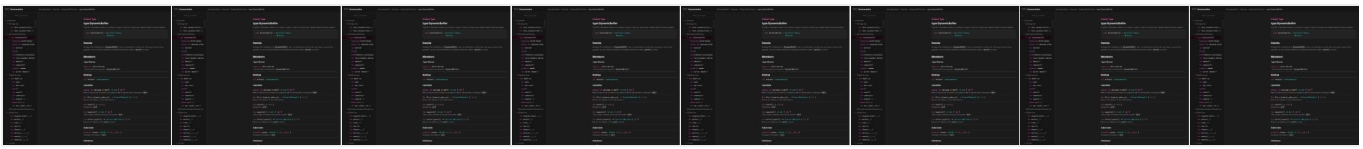
Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

▲ 0–49 50–89 90–100

METRICS Expand view

First Contentful Paint 0.4 s	Largest Contentful Paint 0.7 s
Total Blocking Time 0 ms	Cumulative Layout Shift 0.03
Speed Index 0.4 s	

View Treemap



Show audits relevant to: All FCP LCP TBT CLS

DIAGNOSTICS

- Serve static assets with an efficient cache policy — 8 resources found
- Avoid an excessive DOM size — 3,504 elements
- Avoid large layout shifts — 3 layout shifts found
- Avoid non-composited animations — 1 animated element found
- Initial server response time was short — Root document took 20 ms
- Avoids enormous network payloads — Total size was 202 KiB
- Avoid chaining critical requests — 2 chains found
- JavaScript execution time — 0.0 s
- Minimizes main-thread work — 0.3 s
- Minimize third-party usage — Third-party code blocked the main thread for 0 ms
- Largest Contentful Paint element — 730 ms

More information about the performance of your application. These numbers don't directly affect the Performance score.

PASSED AUDITS (27) Show

100

Accessibility

These checks highlight opportunities to improve the accessibility of your web app. Automatic detection can only detect a subset of issues and does not guarantee the accessibility of your web app, so manual testing is also encouraged.

ADDITIONAL ITEMS TO MANUALLY CHECK (10) Show

These items address areas which an automated testing tool cannot cover. Learn more in our guide on conducting an accessibility review.

PASSED AUDITS (20) Show

NOT APPLICABLE (37) Show

Best Practices

TRUST AND SAFETY

Ensure CSP is effective against XSS attacks

PASSED AUDITS (14) Show

NOT APPLICABLE (3) Show

SEO

These checks ensure that your page is following basic search engine optimization advice. There are many additional factors Lighthouse does not score here that may affect your search ranking, including performance on Core Web Vitals. Learn more about Google Search Essentials.

ADDITIONAL ITEMS TO MANUALLY CHECK (1) Show

Run these additional validators on your site to check additional SEO best practices.

PASSED AUDITS (8) Show

NOT APPLICABLE (2) Show

- Captured at Jun 21, 2024, 11:45 PM GMT+2
- Initial page load
- Emulated Desktop with Lighthouse 12.0.0
- Custom throttling
- Single page session
- Using Chromium 126.0.0.0 with devtools

Generated by **Lighthouse** 12.0.0 | File an issue

M

Distribution of labour

We tracked individual time spent working on the project using Google Sheets. The total accumulated times can be seen in Figure M.1

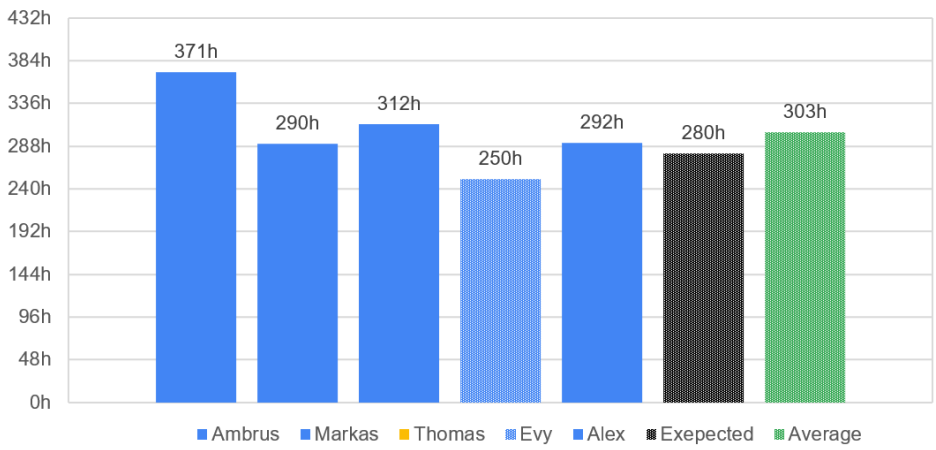


Figure M.1: Team members' time contribution, including writing code and the report and participating in meetings.

N

TW Feedback

- We removed unnecessary first-person pronouns from the report and rewrote it to be more formal
- Cited the most recent papers
- Improved the ethics sections regarding the controversial library, added a URL to the library, and added multiple sides to the story from the team's perspective
- Removed the syntagm "the following chapter"