Group 1 - 14: MD-to-LaTeX

Yong Zhe Rui Gabriel   Visshal Natarjan   Kaveri Priya Putti
Aurelius Bryan Yuwana

LaTeX has been the markup language of choice for many academics. However, the complexity and many commands of the markup language has led to many shying away from it and opting to work with the simpler alternative Microsoft Word. We seek to revolutionize this and allow for an easy start into using LaTeX by creating MD-to-LaTeX which allows users to parse their Markdown files into LaTeX for compilation. This tool will allow students to work in a simpler markup language while utilizing the powers of LaTeX for formatting. The parsing tool will create an abstract syntax tree that allows us to consider expansion to other formats of output, such as HTML.

1 Introduction

LaTeX has been the markup language of choice for many academics. However, the complexity and many commands of the markup language has led to many shying away from it and opting to work with the simpler alternative Microsoft Word. We seek to revolutionize this and allow for an easy start into using LaTeX by creating MD-to-LaTeX which allows users to parse their Markdown files into LaTeX for compilation. This tool will allow students to work in a simpler markup language while utilizing the powers of LaTeX for formatting.

For the project, we spent time understanding the intricacies and details surrounding the markdown syntax before diving into how we will create a tokenizer and parser for the markdown syntax. We then used our understanding to construct the state machine for creating the abstract syntax tree for any given markdown text.

We spent some time figuring out how to traverse the abstract syntax tree to then create meaningful LaTeX output. The project also implements some diagram syntax to allow for creation of simple diagrams in LaTeX such as sequence diagrams and simple acyclic graphs. These diagrams are commonly used in Computer Science and would be useful to start with.

2 Markdown Syntax

Most of the markdown syntax guidelines have been taken from Markdown Guide. This serves as a good starting point to understand the markdown syntax. Our group then selected various key components of the markdown language to implement. They are as follows:

- Headings
- Paragraphs
- Emphasis
- Blockquotes
- Lists
- Code
- Math
- Links

Submitted to:

- Images
- Mermaid diagrams

However, we have also implemented various additional conditions to ensure that we could easily parse certain components. For example, for lists, we decided for simplicity that if users wanted to generate nested lists, they would have to indent strictly with 2 spaces. This allows for ease of implementation on our end. Hopefully, in the future, we can consider removing such an opinionated limitation.

To ensure parsing accuracy, we have made it necessary to escape specific characters such as '*' and '_' since these are used in markdown to indicate bold and italicized text. This keeps parsing simpler and also makes users more conscious about their usage of such symbols in case it produces undesirable effects in the output.

Images are also currently only functional if defined at the start of a new line, ensuring that users are conscientious about utilizing images in their markdown document. This is due to the slight ambiguity between the syntax used between images (`![alt text](URL)`) and links (`[alt text](URL)`).

Additionally, we have extended our markdown parser to include support for Mermaid code blocks, which allows users to embed complex diagrams directly in their markdown files. Our parser can convert these Mermaid blocks into corresponding LaTeXdiagrams, enhancing the documentation with visual representations. Currently, we support the following types of Mermaid diagrams:

1. Sequence Diagrams
2. Pie Charts
3. Class Diagrams
4. Graph Diagrams (Acyclic)

These enhancements to markdown functionality should be noted when evaluating the parser.

3 LaTeX Syntax

Most of the markdown syntax can easily be carried over to LaTeX and have typical equivalents. The equivalent environments/elements used to translate our Markdown components to LaTeX ones are as follows.

- Headings → `\section` and corresponding headers
- Paragraphs → regular text elements
- Emphasis
  - Bold → `\textbf`
  - Italics → `\emph`
- Blockquotes → `quote` environment
- Lists
  - Numbered List → `enumerate` environment
  - Bullet List → `itemize` environment
- Code → `verbatim` environment
- Links → `\href{}{}`
- Images → `\includegraphics` with `\caption` within `figure` environment

Currently, these standards are enforced by the renderer in the code. Future works could be done to allow for more customizability in the choice of these LaTeX conversions, allowing users to input their own desired components they might want to convert into.

4 Tokenizer

Markdown is a line-based system which means that each line has a specific type and usage. For example, a header consists only of a single line and does not spread over multiple. Likewise for paragraphs, they are also characterized as ending with an empty line. This all makes for the tokenizer to make more sense as working on a line by line basis instead of word by word.

As a result, our project parses the input markdown file line by line and classifies the type of each line before passing it to the parser.

The following are the various line types:

- `LINE_EMPTY` $\rightarrow$ Denotes empty line
- `LINE_LISTITEM` $\rightarrow$ Denotes the line looks like a list item
- `LINE_HEADER` $\rightarrow$ Denotes the line is a header
- `LINE_IMAGE` $\rightarrow$ Denotes the line holds an image linke
- `LINE_CODE_DELIM` $\rightarrow$ Denotes the line is a code block delimiter
- `LINE_MATH_DELIM` $\rightarrow$ Denotes the line is a math block delimiter
- `LINE_BLOCKQUOTE` $\rightarrow$ Denotes the line is part of a blockquote
- `LINE_TEXT` $\rightarrow$ Denotes the line is plain text

The following function in the code in `src/parser/parser.c` is responsible for identifying the type of line.

```c
LineType get_line_type(const char *line, int line_length) {
  if (line_length == 0) {
    return LINE_EMPTY;
  } else if (is_header(line, line_length)) {
    return LINE_HEADER;
  } else if (is_mathblock_indicator(line, line_length)) {
    return LINE_MATH_DELIM;
  } else if (is_codeblock_indicator(line, line_length)) {
    return LINE_CODE_DELIM;
  } else if (is_image_link(line, line_length)) {
    return LINE_IMAGE;
  } else if (is_blockquote(line, line_length)) {
    return LINE_BLOCKQUOTE;
  } else if (is_list_item(line, line_length)) {
    return LINE_LISTITEM;
  } else {
    return LINE_TEXT;
  }
}
```

The various helper functions are typically functions that rely on regex to identify the various types of lines. For example, the regex for headers is `^#{1,6} .+`. Regex was chosen due to its ability to easily detect the patterns in strings without us having to come up with our own state machines to ensure the string matches our desired logic.

The various rules used to identify some of the line types are as follows

- Headers consist of one to six # symbols followed by text.
- Lists are a sequence of list items marked with bullets (`-`, `+`, or `*`) or numbers followed by a period. Nested list items will start with an even number of `' '` characters
- Code blocks are delimited by lines containing three backticks (```` ``` ````).

- Math blocks are delimited by lines containing dollar signs (`$$`).
- Blockquotes start with `'>'` and contains block elements. Which effectively mean they nest documents within them
- Mermaid diagrams are fenced code blocks with the language specified as `mermaid`.

5 Parser

After the tokenization and identification line, we parse the lines according to the markdown requirements. This was done by first coming up with a rudimentary understanding of markdown's context free grammar, following which we designed state machines for the various line types to know where to place the data for each line.

5.1 Context Free Grammar

We constructed a rough representation of the context free grammar of markdown to understand how we can parse the various lines into an abstract syntax tree (AST).

The tokens we identified are as follows:

- Non-terminal symbols
  - `D`: Document
  - `P`: Paragraph
  - `Lb`: List Block
  - `Mb`: Math Block
  - `Cb`: Code Block
  - `Bb`: Block quote Block
- Terminal symbols
  - `He`: Heading
  - `Emp`: Empty Line
  - `Tl`: Text Line
  - `Bt`: Bold Line
  - `It`: Italics Line
  - `Li`: List Item Line
  - `Md`: Math Block Delimiter
  - `Cd`: Code Block Delimiter
  - `I`: Image Line
  - `Bq`: Block Quote Line
  - `' '`: White Space

The rudimentary production rules are as follows

```
    T -> (He | Emp | Lb | I | Bb | Cb | Mb)
    D -> D T
    D -> D P T
    P -> P Tl
    F -> F (Tl | T)
    Cb -> Cd F Cd
    Mb -> Md F Md
    Lb -> Lb Li
    Lb -> Lb '  ' Lb (for nested lists, starting with 2 white spaces)
```

We realised many of these rules are left recursive and tend to depend only on the right most state. Hence, implementing the parsing logic via state machine works well. Note that the rules for parsing Bold and Emphasis text is not here and will be explained in Section 5.1.1.

Our key takeaways from this exercise is as follows:

- Document is a sequence of blocks.
- Document is made up using a left recursive rule to include more blocks
- Blocks can be a header, paragraph, list, code block, blockquote, or a mermaid diagram.
- Paragraph is a sequence of non-empty text lines and is terminated with the appearance of any line that is not text.

It is to note that we didn't develop a complete CFG for the markdown syntax but rather used it as an exercise to gain an understanding of the syntax. This means that the above production rules are not complete but simply a guideline which we used to determine our state machine in parsing the file.

5.1.1 Text Lines

For text lines, we are only concerned with identifying matching tokens in the various cases. The tokens are as follows:

- '*' and '_' and the number of consecutive characters to identify italicized and bolded text.
- '$' to identify inline math
- '‘' (backticks) to identify inline code text
- '[' as it could signify the link pattern of '[]()'

The concept for parsing these elements is that we would use somewhat of a maximal munch algorithm. Since '*' and '_' can occur from 1 to 3 times. Each time we encounter the symbol, we will try to see how many occurrences of the symbol we can get. Following which, suppose we find 2 '*', we then look forward through the rest of the line to see if we can find the matching characters to close the block. If we are unable to close the block, we will print a warning to the user since it could have been an input error. However, we will still continue to parse that section as just regular text.

For example, the following text `***bold and italics**` does not have 3 matching '*' on both sides which could be a user input error since they might want 3 or 2 on both sides as delimiters. The algorithm will then only detect the 2 matching '*' and identify the text simply as bolded (2 surrounding '*') instead of bolded and italicized(3 surrounding '*').

This logic is coded in the function `void parse_new_paragraph_line(md_node *paragraph_node);` in `src/parser/features/paragraph.c`

A similar logic is applied to find links where upon encountering '[' we attempt to find the next occurrence of '](' before parsing the link data.

Note that characters can always be escaped by using the '\' such that the character that comes after it will be parsed as is without thinking it is a special identifier.

Hence, it is advised to always escape the following characters: '*', '_', '[', ']', '(', ')' and '\' to ensure that they appear as expected.

5.2 Abstract Syntax Tree

The parsing process results in the construction of an AST, which represents the hierarchical structure of the markdown document. Each node in the AST corresponds to a markdown element, such as a paragraph, list item, or header.

The AST nodes contain metadata about their type, content, and relationships with other nodes, enabling the renderer to traverse the tree and generate corresponding LaTeX commands. Nodes are dynamically allocated and linked to form a tree structure that accurately represents the nested and sequential nature of the markdown elements.

The parser populates the AST during the line-by-line analysis of the markdown file. The AST then serves as the input for the rendering process, where each node is converted into LaTeX format based on its type and content.

5.2.1 Nodes in AST

The nodes in the AST were implemented as having children in the form of a linked list. This allowed for easy traversal of elements through the attributes like `first_child`, `last_child`, `parent`, `next` and `prev`

The data was then stored in `data` with the length of the data being in `len`. The node also contains many other metadata.

The struct of the node is shown in Appendix B

5.2.2 Structure of AST

Based on the rough grammar we have identified, we see that our AST will consist of the following nodes

- `NODE_BLOCK_QUOTE` $\rightarrow$ Node for Block Quotes
- `NODE_CODE` $\rightarrow$ Node for Inline Code
- `NODE_CODE_BLOCK` $\rightarrow$ Node for Code Blocks
- `NODE_MATH` $\rightarrow$ Node for Inline Math
- `NODE_MATH_BLOCK` $\rightarrow$ Node for Math Blocks
- `NODE_EMPH` $\rightarrow$ Node for Italicized text
- `NODE_HEADING` $\rightarrow$ Node for Headings
- `NODE_IMAGE` $\rightarrow$ Node for Images
- `NODE_LINK` $\rightarrow$ Node for Links
- `NODE_LIST` $\rightarrow$ Node for Lists
- `NODE_ITEM` $\rightarrow$ Node for List Items
- `NODE_PARAGRAPH` $\rightarrow$ Node for Paragraphs
- `NODE_ROOT` $\rightarrow$ Node for the root of the document
- `NODE_STRONG` $\rightarrow$ Node for Bolded text
- `NODE_TEXT` $\rightarrow$ Node for regular text
- `NODE_NONE` $\rightarrow$ Empty label for placeholder in case

Some key points to note are the following:

- The `NODE_ROOT` represents the entire document.
- `NODE_TEXT` will never be direct descendants of `NODE_ROOT`
- Only `NODE_TEXT` and `NODE_IMAGE` are leaf nodes
- Most of the nodes in the AST have text data stored in their children nodes via a `NODE_TEXT`

A sample Abstract Syntax Tree can be seen in Appendix C.

Note that it does not use the recursive nature of the CFG described but rather keeps nodes all on the same level. Hence, it may not truly be considered an AST but utilizes the basic concept of it to construct a tree that allows us to parse the data.

5.2.3 State Machines

For the parsing operation, we designed a state machine for the various types of lines to determine how to process the new incoming line. The state machine is not explicit since it is dependent on the last child in the AST. It simply reads the type of the current last child of the root node and depending on that node, it will execute various functions to process the new line.

This means that the tokenizer's and parser's function are linked together and there is not explicit tokenization step in our program. It simply tokenizes the line on the fly and adds the corresponding new nodes to the AST.

The state machine is responsible for determining the start and end of blocks as well as deciding when to append data to the previous node or create a new node in the AST or traverse it when necessary.

The state machine depends on a few components. They are as follows

- Current Line Type
- Previous Node's Type
- Previous Node's Mode

The state is therefore the Previous Node's Type and Mode, while the input is the Current Line Type. The output is then the set of functions to execute. The Node's Type is stored in the struct as a `type` attribute while the Mode is stored in `user_data`. A Mode was introduced to allow for easier tracking of whether a node was completed and no additional data could be added, or whether it was still allowing data to be appended. Hence, there were two modes, `MODE_APPEND` and `MODE_PROCESSED`.

A simplified understanding of the parser state machine is as explained in Table 1. The state machine is written in function `parse_line` in `src/parser/parser.c`.

For brevity, the state matching rules for `LINE_MATH_DELIM` with `NODE_MATH_BLOCK` has been omitted since it has similar behaviour to `LINE_CODE_DELIM` with `NODE_CODE_BLOCK`

Likewise, the state machine rules for `LINE_LISTITEM` with `NODE_LIST` and `LINE_BLOCKQUOTE` with `NODE_BLOCK_QUOTE` are not in the table as they are similar to the case of `LINE_TEXT` with `NODE_PARAGRAPH`.

There are some key differences to note for these line types compared to `LINE_TEXT`.

- `LINE_TEXT`
  - `LINE_TEXT` is further tokenized and parsed as necessary into `NODE_EMPH`, `NODE_STRONG`, etc. as required and appended to the corresponding `NODE_PARAGRAPH`
  - If matching tokens to create the corresponding modified text are not found, warning statements will be printed
- `LINE_LISTITEM`
  - The list delimiter is validated against existing elements and checked to see whether there are correct number of indents if it is a nested list.
  - Warning statements will be printed or the program will be exited should validation criteria not be met
  - Criteria needs to be met like having the right amount of indentation (only even number of indents) as well as ensuring nested levels do not jump depth. So users are not allowed to immediately declare nested lists without having a list to begin with.
- `LINE_BLOCKQUOTE`
  - We strip the first '>' character and as many ' ' (spaces) we see and recursively call a parse line function on the remaining text

Previous Node's Type	**Previous Node's Mode**	**Current Line**	**Function Output**
NODE_CODE_BLOCK	MODE_APPEND	Any line other than LINE_CODE_DELIM	Append line to the existing NODE_CODE_BLOCK
NODE_CODE_BLOCK	MODE_APPEND	LINE_CODE_DELIM	Change NODE_CODE_BLOCK mode to MODE_PROCESSED
Any node	Any mode	LINE_HEADING	Set previous node to MODE_PROCESSED if it exists and add new NODE_HEADING
Any node	Any mode	LINE_IMAGE	Set previous node to MODE_PROCESSED if it exists and add new NODE_IMAGE
Any node other than NODE_CODE_BLOCK	Any mode	LINE_CODE_DELIM	Set previous node to MODE_PROCESSED if it exists and add new NODE_CODE_BLOCK
NODE_CODE_BLOCK	MODE_PROCESSED	LINE_CODE_DELIM	add new NODE_CODE_BLOCK
Any Node other than NODE_PARAGRAPH	Any Mode	LINE_TEXT	Set previous node if any to MODE_PROCESSED and create a new NODE_PARAGRAPH which we append the line to in the form of a NODE_TEXT
NODE_PARAGRAPH	MODE_PROCESSED	LINE_TEXT	Create a new NODE_PARAGRAPH which we append the line to in the form of a NODE_TEXT
NODE_PARAGRAPH	MODE_APPEND	LINE_TEXT	Append the line to the existing NODE_PARAGRAPH in the form of a NODE_TEXT

Table 1: Parsing State Machine

6 Renderer

Once the parser has constructed the Abstract Syntax Tree (AST), the Renderer converts the structured data into LaTeX. The Renderer is designed to be modular, where each type of markdown syntax element is handled by a distinct rendering function. These functions translate the markdown elements, as represented in the AST, into corresponding LaTeXcommands and environments.

6.1 General Markdown Rendering

The Renderer iterates through the nodes of the AST (Abstract Syntax Tree), visiting each node based on its type and converting it to LaTeXwith the following mappings:

- **Headings:** Heading nodes, based on their level, are translated into corresponding sectioning commands in LaTeX, such as `\section`, `\subsection`, and so on.

- **Paragraphs:** Paragraph nodes are transformed into simple text blocks in LaTeX. A blank line is ensured after each paragraph to comply with LaTeXformatting.
- **Emphasis:** Text emphasis, such as bold and italics, is handled by wrapping the text with `\textbf{}` for bold and `\emph{}` for italics.
- **Blockquotes:** Blockquote nodes are rendered using the `quote` environment to represent quoted text.
- **Lists:** Unordered and ordered lists are rendered using the `itemize` and `enumerate` environments, respectively. Nested lists are correctly indented and formatted.
- **Code:** Code blocks are enclosed within the `verbatim` environment to preserve spacing and special characters. Inline code is handled using the `\texttt{}` command.
- **Math:** Mathematical expressions are handled using the `math` environment or `$` symbols for inline math, ensuring correct formatting and display of equations.
- **Links:** Link nodes are converted to clickable hyperlinks using the `\href{}{}` command, where the first bracket contains the URL and the second bracket the anchor text.
- **Images:** Image nodes are included as figures using the `\includegraphics{}` command inside a `figure` environment, optionally accompanied by captions with the `\caption{}` command.

6.1.1 Iterator

To aid in the rendering of the elements, the iterator constructed is aware of whether it is entering or exiting a node in the traversal process. This was so it could appropriately open and close the parentheses used in LaTeX. Hence, all rendering functions also have an additional input parameter `int entering` which denotes whether the traversal is entering or exiting the node. For example, if we were entering a list node, it would then know to correctly output `\begin{itemize}` and subsequently when exiting output `\end{itemize}`.

This method of traversal ensures that all elements have the correct opening and closing syntax.

6.2 Mermaid Diagrams

The syntax to generate guidelines have been taken from Mermaid as its simple markdown-like syntax complement the rest of our syntax. Moreover, the syntax of Mermaid is straightforward as each line of mermaid have a finite amount of "formats" (more on this in the following subsections). Moreover, Mermaid ensures that the type of diagram can be determined from the first line making parsing easier. For our MD-to-LaTeX program, we implemented class diagrams, graphs, pie charts and sequence diagrams.

6.2.1 Graphs

Graph diagrams from Mermaid syntax to LaTeX is executed through a two-step process: parsing of the Mermaid input to extract graph elements, followed by rendering using the TikZ package for graphical display.

Parsing and Node Detection

Initially, the parser scans the Mermaid syntax for lines indicating directed edges, specifically looking for the "->" symbol that denotes connections between nodes. This step involves:

- Identifying edge declarations, where each line with "->" is considered a potential connector between two nodes.
- Separating node identifiers found on either side of the arrow and trimming any surrounding whitespace to ensure clean, usable data.
- Ensuring uniqueness of each node by maintaining a list of previously identified nodes and only adding new ones if they haven't been listed yet.
- Recording relationships by mapping the 'from' node to the 'to' node for each detected edge, thus preserving the graph's structure for the upcoming layout computation.

Dynamic Layout Calculation This step arranges the nodes to optimize the graph's readability and visual quality:

- Each node is assigned a level based on its links, with source nodes placed at higher levels and their respective targets cascading downward.
- Nodes within the same hierarchical level are positioned equidistantly to avoid overlaps and ensure a clean visual flow of edges.
- This structured placement minimizes edge crossings and enhances the overall graphical representation, making complex diagrams easier to interpret.

Rendering Graph rendering in LaTeX is accomplished with the TikZ package, which is initiated with parameters configured for optimal graph visualization. The process involves:

- Preparing the TikZ environment by setting arrow styles and node spacing according to the graph's requirements.
- Drawing each node at its designated coordinate and connecting them with arrows to depict the relationships extracted and computed from the Mermaid syntax.

This ensures that the graphical output in the LaTeX document correctly represents the structured relationships and node hierarchies as defined in the original Mermaid input.

6.2.2 Class Diagrams

Conversion of class diagrams from Mermaid syntax to *LaTeX* is achieved through a structured, multi-step process involving parsing, data organization, layout computation, and rendering using the TikZ package. This method ensures an accurate graphical representation of class diagrams and their relationships in *LaTeX* documents.

Parsing and Data Structuring The conversion begins by parsing the Mermaid syntax to extract elements such as class names, attributes, methods, and relationships. The key components extracted are:

- **Class Details:** Names, attributes, and methods are identified, distinguishing between attribute declarations and method declarations.

- **Relationships:** Types of relationships (aggregation, composition, inheritance, association) and associated classes are cataloged, ensuring accurate linkage between classes.

This parsing phase helps in laying the foundation for structuring the data into a manageable format that facilitates subsequent layout computation.

Dynamic Layout Computation and Rendering Following the parsing process, we move to the layout computation for the class diagram, which is essential for ensuring a visually coherent output. The computation begins with the identification of nodes and edges using a graph-based approach. In this context, classes are treated as nodes and their relationships are represented as edges. This method involves accurately mapping these nodes and edges to depict the structural relationships inherent in the class diagram. Subsequently, the layout algorithm is applied to optimize the placement of nodes. This algorithm focuses on minimizing edge crossings and logically grouping related classes to enhance the diagram's readability and overall aesthetic appeal. Rendering is performed with TikZ, translating the structured data into a LaTeX diagram:

1. **TikZ Environment Setup:** The TikZ environment is prepared, setting styles for nodes and relationships to visually differentiate between them.
2. **Diagram Drawing:**
   - **Classes:** Rendered using the `\umlclass` command, detailing each class's name, attributes, and methods.
   - **Relationships:** Illustrated with commands like `\umlaggreg` and `\umlinherit`, clearly depicting the design semantics.
3. **Compilation:** The elements are compiled into a *LaTeX* document, producing a scalable vector graphic that accurately depicts the class diagram.

6.2.3 Pi Charts

Conversion of pie charts from Mermaid syntax to LaTeX is a two-stage process involving tokenizing and parsing, followed by rendering. This approach ensures a seamless transition from a user-friendly chart definition to a detailed graphical representation in a LaTeX document.

Tokenizing and Parsing In the first stage, the parser directly targets and extracts key components from the Mermaid syntax. This includes:

- The `"title"` of the pie chart, which provides a heading for the chart in the final LaTeX document.
- The labels and values for each section of the pie chart, which are crucial for constructing the visual representation. Each section is defined by labels and corresponding values enclosed within quotation marks in the Mermaid code.

This process eliminates the need for a separate tokenization step by leveraging Mermaid's structured format, allowing for a direct extraction of essential data. The parser navigates through the Mermaid syntax, identifying these elements based on specific markers and patterns, and organizes them into a structured format suitable for rendering.

Rendering The second stage involves converting the parsed data into LaTeX code that visually represents the pie chart. This is achieved by utilizing the pgf-pie LaTeX package, which specializes in creating pie charts. The rendering process involves:

- Initiating the LaTeX document with the necessary preamble and package inclusions for pie chart creation.
- Generating the LaTeX code for the pie chart based on the structured data obtained from the parsing stage. This includes setting up the pie chart environment, specifying the chart title, and defining each section with its label and value.

This two-stage process ensures that pie charts defined in Mermaid syntax can be accurately and efficiently converted into LaTeX, facilitating their inclusion in academic and scientific documents with high-quality graphical representations.

6.2.4 Sequence Diagrams

For sequence diagrams, there are three type of lines:

- `sequenceDiagram` (Type of diagram)
- `participant` *`name`* (Explicitly declare participants)
- *`name1`* `->>` *`name2`* `:` *`message`* (Create a message)

As said before, the formats of a mermaid line can only be one of these three. This means a simple line-by-line iteration is sufficient for tokenization.

The tokenization process involves storing information about all participants (both declared implicitly and explicitly), and the messages sent between them. We used our own implementation of C++'s vectors to dynamically store information into a vector of participants and a vector messages. THis means that the tokenization and parsing process is done together. These two vectors are then passed to the renderer.

The render process involves the use of a LaTeX package pgf-umlsd. The renderer then iterates through each element of the vector to generate a corresponding output. Due to capability differences in mermaid and the pgf-umlsd package, certain things are not rendered properly. For example, pgd-umlsd requires that every self-call message have a return value, but mermaid's self-call doesn't need a return value. This results in self-calls to be rendered with an additional empty arrow.

7 Code Execution

To use the 'md_to_tex' tool effectively, you need to understand its command-line options which allow you to specify input and output files and optionally enable debugging to see how the Markdown is parsed.

7.1 Command-Line Options

Here's a breakdown of the available options:

- **-i**: This is a mandatory option where you specify the path to the input Markdown file that you want to convert to LaTeX.

- **-o**: This optional parameter allows you to define the path and name of the output file where the LaTeX code will be saved. If not provided, the tool will automatically create an output file with the same name as the input file but with a '.tex' extension.
- **-d**: An optional debugging switch. When this is turned on, the tool outputs detailed information about the parsing process, including each step of the AST generation and its conversion into LaTeX format.
- **–h**: Displays help information which includes usage, options, and a brief description of what the tool does.

7.2 Running the Tool

Here is how you can run the tool in various scenarios:

- **Basic Conversion:** Convert a Markdown file named 'document.md' to LaTeX with the default output filename ('document.tex'):

  ```
  ./md_to_tex -i document.md
  ```

- **Specifying Output File:** Convert a Markdown file and specify a custom output filename:

  ```
  ./md_to_tex -i document.md -o custom_output.tex
  ```

- **Enabling Debugging:** Convert a Markdown file with debugging enabled to see detailed processing logs:

  ```
  ./md_to_tex -i document.md -d
  ```

8 Challenges

The development of the MD-to-LaTeX tool was marked by a series of challenges that was tied to the specific nature of Markdown to LaTeX conversion:

- **Complex Markdown Features:** Handling Markdown's less standard features like nested lists, code blocks, and especially the extended syntax for Mermaid diagrams required complex parsing strategies to maintain structural fidelity in the LaTeX output.
- **LaTeX Rendering Limitations:** Translating the flexible Markdown syntax into the more rigid LaTeX structures posed significant challenges, particularly in preserving the visual and functional aspects of the LaTeX diagram.
- **Mermaid Diagram Conversion:** Integrating Mermaid diagrams into LaTeX was particularly challenging as it involved interpreting graphical representations and converting them into equivalent TikZ commands.
- **Operating System-Specific Debugging:** A significant challenge arose when the feature to convert class diagrams, which was initially developed and tested on MacOS, did not function as expected on Linux. This required extensive debugging to identify and resolve.

9 Future Works

- **Custom LaTeX inputs:** We hope that users are able to define their own requirements for LaTeX output environments should they have a preference
- **Support for Tables:** Tables is a very useful feature as well to utilize and would be an ideal addition as a feature.
- **Alternative output types:** Due to the availability of a syntax tree, we can also consider developing for other output markup languages like HTML if we would like to consider.

10 Conclusion

The MD-to-LaTeX project effectively delivers a tool that converts Markdown documents to LaTeX format, addressing the need for a seamless transition between these two popular markup languages. The tool successfully integrates a robust parser that supports a wide range of Markdown elements and Mermaid diagrams, ensuring accuracy in the final output.

Challenges in handling complex Markdown syntax were overcome by developing a custom parser and employing state machines, which enhanced the tool's ability to maintain the structural integrity of source documents.

Overall, the MD-to-LaTeX project achieved its objectives by creating a practical tool that simplifies document conversion, enhances user productivity, and maintains the quality and precision required for professional documentation.

11 Contributions

- **Yong Zhe Rui Gabriel**: Wrote the parser for the markdown features and wrote the code to generate Abstract Syntax Trees as well as the iterator to be used by the renderer. Wrote the report portion for introduction, tokenizing, parsing and future improvements.
- **Kaveri Priya Putti**: Implemented the renderer for the markdown features, integrated mermaid diagrams parsing, implemented mermaid graphs and class diagrams. Wrote the report portion for rendering, graphs and class diagrams.
- **Aurelius Bryan Yuwana**: Implemented a utility container which behaves like C++'s vectors, coded mermaid sequence diagrams. Wrote the report portion for sequence mermaid diagrams.
- **Visshal Natarajan**: Implemented the mermaid pi diagram, class diagram and the CLI. Wrote the report portions for rendering, graphs, pi and class diagrams.

Appendices

Appendix A Links

Github: MD To Tex

Appendix B Node Struct

Not all attributes are used in the Node Struct but the key ones such as `next`, `prev`, `parent`, `first_child` and `last_child` are used.

```c
typedef struct md_node {
  struct md_node *next;
  struct md_node *prev;
  struct md_node *parent;
  struct md_node *first_child;
  struct md_node *last_child;

  Mode user_data;
  char *data;
  int len;

  int start_line;
  int start_column;
  int end_line;
  int end_column;
  NodeType type;
  LineType prev_line_type;
  uint16_t flags;

  int heading_level;
  char *code_language;
  ListType list_type;
  char *url;
  int url_length;
  char *title;
  int title_length;
  char *mermaid_code;
  char delimiter;
  int list_number;
} md_node;
```

Appendix C Sample AST

As an example, the following document:

```
# Heading

This is a paragraph. It can have _italics_ and **bold** words

It can also have $math$ in line

- this is a list
- item 1
  1. number 1
  2. number 2
- last item
```

```
‘‘‘
This is code
‘‘‘

![caption](image link)

> # Nested heading
>
> This is in a blockquote
```

Will have an AST as follows: The AST can be interpreted by looking at the indent levels of the nodes.

```
root node
  header node
  - level: 1
    text node
    - data: Heading
  paragraph node
    text node
    - data: This is a paragraph. It can have
    emph node
      text node
      - data: italics
    text node
    - data:  and
    strong node
      text node
      - data: bold
    text node
    - data:  words
  paragraph node
    text node
    - data: It can also have
    math node
      text node
      - data: math
    text node
    - data:  in line
  list node
    list item node
      text node
      - data: this is a list
    list item node
      text node
      - data: item 1
    list node
    list item node
      text node
      - data: number 1
    list item node
      text node
      - data: number 2
  list item node
    text node
    - data: last item
code node
  text node
  - data: This is code
image node
- url: image link
- title: caption
block quote node
  header node
  - level: 1
    text node
    - data: Nested heading
  paragraph node
    text node
    - data: This is in a blockquote
```