COSC 455

Programming Languages: Design and Implementation

Fall 2025

**Project #1 – LOLCODE Markdown Language Translation**

*Any good software engineer will tell you that a compiler and an interpreter are interchangeable.*

*-- Tim Berners-Lee*

**Directions:** This project has two phases that will result in a small compiler/interpreter. Please read through the entire project description before starting it.

**Goals:** The intention of this project is to use and apply the concepts of programming languages discussed in class (and in the textbook ☺) to design and develop a relatively complex interpreter/compiler. In doing so, you should develop a better understanding in the design of a high-level programming language and the need for studying the concepts behind programming languages. Additionally, this project will familiarize you with several software development tools (VS Code, Rust, GitLab, ANTLR, etc.) and gain experience in *modern* software engineering and development.

This project will give you development experience in the following areas of programming languages that were covered in class:

* Writing BNF/EBNF grammar rules (Chapter 2)
* Lexical analysis
* Syntactic analysis and parsing
* Semantics
* Static-scope analysis

Further, the implementation of these concepts will give you further experience in essential programming concepts including:

* Program design and development
* Function invocation
* Error/exception handling
* Recursion

**Description:** As described in class, the basic idea of a compiler is to take a high-level language and convert it to a low-level language that can be executed on a computer. In this project, you will design and develop a compiler/interpreter that translates my version of a LOLCODE Markdown language (see <http://en.wikipedia.org/wiki/Markdown>, <http://daringfireball.net/projects/markdown/syntax>). Briefly, a Markdown language allows for easy-to-read annotations within text that is then automatically converted to valid, well-formed HTML. However, unlike a traditional Markdown language, our LOLCODE Markdown language was inspired by its esoteric programming language relative (see <http://progopedia.com/language/lolcode/>) and “created under the influence of a meme lolcat”. As an extension to typical Markdown languages, our language will provide for statically scoped variables to be defined and used throughout the LOLCODE Markdown document.

Specifically, our LOLCODE Markdown language will support the following commands (bold is used to emphasize the syntax and differentiate it from the text):

* **#HAI** … **#KTHXBYE**

The #HAI…#KTHXBYE annotations denote the beginning and ending of a valid source file in our LOLCODE Markdown language. All valid source files *must* start with #HAI and end with #KTHXBYE (i.e., there cannot be any text before or after). Between theses annotations, all other annotations (or none at all) may occur except for a repetition of the #HAI or #KTHXBYE annotations. In HTML, these annotations correspond with the <html> and </html> tags, respectively.

* **#OBTW** … **#TLDR**

The #OBTW…#TLDR annotations denote the beginning and ending of a comment in our LOLCODE Markdown language. The comment annotations are optional in any LOLCODE Markdown source file and may occur immediately after any legal annotation. Within the comment annotation, *only* plain text is possible (i.e., no other annotations) and may span several lines. In HTML, these annotations correspond with the <!-- and --> tags, respectively.

* **#MAEK HEAD** … **#OIC**

The head annotations are a container for any head elements, in our LOLCODE Markdown language only the #GIMMEH TITLE annotation is allowed. The head tag is *not* required in a LOLCODE Markdown file, but if it is present, it must be immediately following the #HAI annotation unless there are comments between the #HAI annotation and the #MAEK HEAD annotation. In HTML, these annotations correspond with the <head> and </head> tags, respectively.

* **#GIMMEH TITLE** … **#MKAY**

The title annotations denote the title of the resulting html page that shows up in the browser’s toolbar. Within these annotations, *only* plain text is possible (i.e., no other annotations). Title annotations *must* occur within #MAEK HEAD annotations. In HTML, these annotations correspond with the <title> and </title> tags, respectively.

* **#MAEK PARAGRAF** … **#OIC**

The paragraph annotations denote the beginning and ending of a paragraph within a LOLCODE Markdown source file. Within these annotations, the bold, italics, list, item, sounds and video (described below) annotations are allowed but not required (note that you cannot have a #MAEK PARAGRAF annotation within another #MAEK PARAGRAF annotation). In HTML, these annotations correspond with the <p> and </p> tags, respectively.

* **# GIMMEH BOLD** … **#MKAY**

The bold annotations denote the beginning and ending of text within the LOLCODE Markdown source file that is in a bold font. Within these annotations, *only* plain text is possible (i.e., no other annotations). Bold annotations *do not* have to occur within paragraph annotations, they may occur on their own. In HTML, these annotations correspond with the <b> and </b> tags, respectively.

* **# GIMMEH ITALICS** … **#MKAY**

The italics annotations denote the beginning and ending of text within the LOLCODE Markdown source file that is in an italic font. Within these annotations, *only* plain text is possible (i.e., no other annotations). Italics annotations *do not* have to occur within paragraph annotations, they may occur on their own. In HTML, these annotations correspond with the <i> and </i> tags, respectively.

* **#MAEK LIST** … **#OIC**

The list annotations denote the beginning and ending of a bulleted list within the LOLCODE Markdown source file. This annotation must be immediately followed by the #GIMMEH ITEM annotation. In HTML, these annotations correspond with the <ul> and </ul> tags, respectively.

* **#GIMMEH ITEM** … **#MKAY**

The item annotations denote the beginning and ending of a list item within the LOLCODE Markdown source file. All list item annotations *must* occur within a #MAEK LIST annotation block. Within these annotations, the bold and italics annotations are allowed but not required (i.e., it can just be plain text). In HTML, these annotations correspond with the <li> and </li> tags, respectively.

* **#GIMMEH NEWLINE**

The #NEWLINE annotation, within the LOLCODE Markdown source file, may appear anywhere within a LOLCODE source document outside of the head and title annotations. In HTML, this annotation corresponds with the <br> tag.

* **#GIMMEH SOUNDZ** *address* **#MKAY**

The audio annotations denote an audio element (see http://www.w3schools.com/html/html5\_audio.asp) within the Markdown source file. The #GIMMEH SOUNDZ annotations *must* contain some text (denoted by *address* above) giving the address of the MP3 file to link to. For example, the following Markdown annotation:

#GIMMEH SOUNDZ http://www.televisiontunes.com/themesongs/The%20Simpsons.mp3 #MKAY

would correspond in HTML to:

<audio controls>

<source src="http://www.televisiontunes.com/themesongs/The%20Simpsons.mp3">

</audio>

For this annotation, you do not need to validate the address – you may assume whatever address provided is valid. For simplicity, we will only use MP3 encoded files.

* **#GIMMEH VIDZ** *address* **#MKAY**

The video annotations denote a YouTube video element (captured in an iframe tag) within the LOLCODE Markdown source file. The video annotations *must* contain some text (denoted by *address* above) giving the address of the YouTube file to link to. For example, the following LOLCODE Markdown annotation:

#GIMMEH VIDZ http://www.youtube.com/embed/zoO0s1ukcqQ #MKAY

would correspond in HTML to:

<iframe src="http://www.youtube.com/embed/zoO0s1ukcqQ"/>

For this annotation, you do not need to validate the address – you may assume whatever address provided is valid. For simplicity, we will only use YouTube links.

In addition to these LOLCODE Markdown annotations, our language will also include the capability to define and use statically-scoped variables, defined as follows:

* **#I HAZ** *variable name* **#IT IZ** *value* **#MKAY**

This annotation structure denotes the beginning and ending of a variable definition within a LOLCODE Markdown source file. The #I HAZ … #MKAY annotations contain a variable name following the #I HAZ annotation, that is some text (denoted by *variable name* above) giving the name of the variable (i.e., a single word containing no spaces) and a #IT IZ annotation that must be followed by some text (denoted by *value* above) giving the value of the variable. The #I HAZ annotation may occur within any other annotation block but, if it occurs, it must be the very first annotation to occur within that block (i.e., immediately following the GIMMEH or MAEK of another annotation). The scope of the variable definition starts after the #I HAZ tag in the block and continues to the end of its immediate enclosing block.

* **#LEMME SEE** *variable name* **#MKAY**

This annotation denotes the beginning and ending of the use of a variable within the LOLCODE Markdown source file. The #LEMME SEE … #MKAY annotations *must* contain only text (denoted by *variable name* above) noting the variable value to use. Again, the variable name must only contain text and is a single word (i.e., no spaces). The #LEMME SEE annotation may occur within *any* other annotation block.

***Note that all annotations are not case sensitive (i.e., #HAI and #hai both are legal).***

Finally, in our LOLCODE Markdown documents, you may assume that whenever there is text (both in text and the cases when an address is provided) possible, the following are the only allowed characters:

* Upper and lower-case letters: A .. Z; a .. z
* Numbers: 0 .. 9
* Punctuation: commas (i.e., ‘,’), periods (i.e., ‘.’), quotes (i.e., ‘”’), colons (i.e., ‘:’), question marks (i.e., ‘?’), exclamation points (i.e., ‘!’), percent sign (i.e., ‘%’) and forward slashes (i.e., ‘/’)
* Special characters: newline, tabs

Except for these characters, you may assume no other character is possible in the text and your grammar does not need to account for them (i.e., the “#” character will only be used to denote one of our Markdown annotations and will not be found in the text).

**Examples:** This section presents some basic examples of our LOLCODE Markdown language and its “compiled” HTML code (indented for readability).

The LOLCODE Markdown source code

#HAI

#OBTW This is a LOLCODE Markdown source file #TLDR

#MAEK HEAD

#GIMMEH TITLE The Simpsons #MKAY

#OIC

#MAEK PARAGRAF

The Simpsons! #GIMMEH NEWLINE

#GIMMEH SOUNDZ

http://www.televisiontunes.com/themesongs/The%20Simpsons.mp3

#MKAY

#GIMMEH NEWLINE

The members of the #GIMMEH BOLD Simpson #MKAY family are:

#MAEK LIST

#GIMMEH ITEM Homer Simpson #MKAY

#GIMMEH ITEM Marge Simpson #MKAY

#GIMMEH ITEM Bart Simpson #MKAY

#GIMMEH ITEM Lisa Simpson #MKAY

#GIMMEH ITEM Maggie Simpson #MKAY

#OIC

#GIMMEH NEWLINE

Lets watch now: #GIMMEH NEWLINE

#GIMMEH VIDZ http://www.youtube.com/embed/zoO0s1ukcqQ #MKAY

#OIC

#KTHXBYE

would compile to the HTML code

<html>

<!-- This is a LOLCODE Markdown source file -->

<head>

<title> The Simpsons </title>

</head>

<p> The Simpsons! <br>

<audio controls>

<source src="http://www.televisiontunes.com/themesongs/The%20Simpsons.mp3">

</audio> <br>

The members of the <b> Simpson</b> family are:

<ul>

<li> Homer Simpson</li>

<li> Marge Simpson</li>

<li> Bart Simpson</li>

<li> Lisa Simpson</li>

<li> Marge Simpson</li>

</ul> <br>

Lets watch now: <br>

<iframe src="http://www.youtube.com/embed/zoO0s1ukcqQ"/>

</p>

</html>

Note that your code does not need to preserve the spacing and tabs as shown above.

Using a definition of a variable in our LOLCODE Markdown language, we could provide the source code:

#HAI

#I HAZ lastname #IT IZ Simpson #MKAY

#MAEK PARAGRAF

The members of the #GIMMEH BOLD #LEMME SEE lastname #MKAY #MKAY are:

#MAEK LIST

#GIMMEH ITEM Homer # LEMME SEE lastname #MKAY #MKAY

#GIMMEH ITEM Marge # LEMME SEE lastname #MKAY #MKAY

# GIMMEH ITEM Bart # LEMME SEE lastname #MKAY #MKAY

# GIMMEH ITEM Lisa # LEMME SEE lastname #MKAY #MKAY

# GIMMEH ITEM Maggie # LEMME SEE lastname #MKAY #MKAY

#OIC

#OIC

#KTHXBYE

would compile to the HTML5 code

<html>

<p> The members of the <b> Simpson</b> family are:

<ul>

<li> Homer Simpson</li>

<li> Marge Simpson</li>

<li> Bart Simpson</li>

<li> Lisa Simpson</li>

<li> Marge Simpson</li>

</ul>

</p>

</html>

Your compiler should take the value of the *lastname* variable and replace it in the compiled code whenever the variable is used for its statically determined scope. That is, the definition of the *lastname* variable in this example is essentially global since it is defined immediately following the #HAI block. To fully illustrate the scoping, consider the following example with two defined variables:

#HAI

#I HAZ myname #IT IZ Josh #MKAY

Hi, my name is # LEMME SEE myname #MKAY .

#MAEK PARAGRAF

#I HAZ myname #IT IZ Jon #MKAY

Inside the paragraph block, my name is # LEMME SEE myname #MKAY

#OIC

Now my name is # LEMME SEE myname #MKAY again.

#KTHXBYE

should correctly compile into HTML as

<html>

Hi, my name is Josh.

<p> Inside the paragraph block, my name is Jon. </p>

Now, my name is Josh again.

</html>

The scoping used here is the same as you are used to in most programming languages. If a variable is used without first being defined, this should be an error (i.e., a static semantic error).

**Details and Deliverables:**  I *strongly* suggest using the standard software engineering approach for developing the compiler for our LOLCODE Markdown language. You will be required to meet two milestones during this project.

**Phase 1: Grammar Design** (15 points) **Deadline**: October 10, 2025, 11:59pm (Blackboard)

Write and submit a BNF grammar (plain BNF, not ANTLR and no use of EBNF notations like +, \*, etc.) for our LOLCODE Markdown language. Your grammar should be parsable using a recursive-descent parser (as described in class and illustrated in Labs #1-5) using a one token lookahead (i.e., like Lab 5). Note that this grammar is strictly to be written in BNF, not EBNF! *Additionally*, develop and submit the ANTLR-based grammar definition for this language. The submission of this phase should be two files (a text/Word file of the BNF and a .g ANTLR grammar file submitted to Blackboard.

**Phase 2: Implementation** (50 points) **Deadline:** November 3, 2025, 11:59pm (Blackboard/GitHub)

To complete this project successfully, you will need to implement a lexical analyzer, a syntax analyzer and a small semantic analyzer. I have provided a Compiler Rust trait provided in this document to get started. To do so, I suggest the following steps:

Task 1. Implement a ***character-by-character*** lexical analyzer that partitions the lexemes of a source file in our LOLCODE Markdown language into tokens. To do this, you must use the Lexical Analyzer Rust trait provided in this document. You may assume that troublesome HTML characters, such as “<”, “>” and “&”, do not appear in any of the text in the source file. You may also assume that the “#” character only appears prior to one of our LOLCODE Markdown annotations (e.g., #HAI). Any lexical errors encountered (e.g., #HEY) should be reported as output to the console with as much error information as possible (i.e., similar to what a Java/Rust/whatever compiler provides). Your compiler may terminate and exit after the first error is encountered. If an error is encountered, no output file should be created.

Task 2. Implement a ***recursive-decent parser*** (i.e., syntax analyzer) that builds an abstract syntax (i.e., parse) tree, conceptually. To do this, you must use one of the two Syntax Analyzer Rust trait provided in this document. The implementation of the abstract syntax tree may best be done using a stack(s) or an array list/vector(s). Any syntax errors encountered should be reported as output to the console with as much error information as possible (i.e., similar to what the Java/Rust/whatever compiler provides). Your compiler may terminate and exit after the first error is encountered. If an error is encountered, no output file should be created.

Task 3. Implement the ***static scope variable resolution*** for our LOLCODE Markdownlanguage as described above. Any static semantic errors encountered (i.e., a variable being used before it is defined) should be reported as output to the console with as much error information as possible (i.e., similar to what the Java/Rust/whatever compiler provides). Your compiler may exit after the first error is encountered. If an error is encountered, no output file should be created.

Task 4. Implement the semantic analyzer that takes the abstract syntax tree data structure and translates it to a lower-level language – HTML in our case. That is, it should produce an output file with the same name as the input file and a .html extension in which contains the *compiled* html tags. In addition, if it compiles correctly, it should open a Chrome web browser on a Mac or Windows OS rendering this output file. [Umm…GPT it and modify/incorporate into your developed code as necessary](https://chatgpt.com/share/68dc121d-6700-8008-ab13-03798d002ee3).

**Phase 3: Execution**

The final program should take *only* one command-line argument provided by the user: an input file name in our LOLCODE Markdownlanguage. We will require and use the convention that *all* LOLCODE Markdown source files in our language will have an *lol* extension (i.e., any file that does not have a lol extension should not be accepted by the compiler). The compiler should then generate an “executable” output file (saved in the same directory as the input file) with the same name but an *html* extension and should automatically open the Google Chrome browser be viewable in the browser (see Phase 2, Task 4).

The final project executable, *lolcompiler*, should be built as a single file executable, named lolcompiler that takes a single runtime argument, the input .lol file to “compile” and can run at the command line / terminal similar to the following:

> lolcompiler.exe input.lol

A guide on how to do this within VS Code can be [here](https://chatgpt.com/share/68dc1146-d3c0-8008-ba4e-07ff14df9bf5) (thanks GPT!). In your final submission, you will need to specify if you built and tested this for a Windows OS with Google Chrome, or, even better (☺) the MacOS with Google Chrome. ***Any projects not following this format will not be graded.***

**Phase 4: Testing**

A significant portion of the grading is dedicated to correctly processing my test case input files. Thus, you are responsible for ensuring that your compiler behaves as expected through your own input/output testing. A key component of testing is to come up with test input files. The test cases should exercise the major paths through your code. For example, you should have tests that use each of the constructs in our LOLCODE Markdown language, including various kinds of blocks (paragraphs, lists, etc.). In addition, you should have test cases that check scope lookup for variables, and maybe even some tests with illegal input just to make sure you are recognizing only correctly formed programs. If your program passes each of these tests then you will know that your program is likely to be able to handle at least simple programs with each of the constructs.

I strongly suggest you test your code at the end of each task in Phase 2. That is, make sure that your lexical analyzer is correctly working (i.e., providing only the next, valid token and recognizing illegal tokens). Only after you are 100% convinced that it is working correctly that you should move on to Task 2. If you wait until after implementing all four tasks, the bugs that you will likely have will be *much* harder to find (i.e, is it in the lexical analyzer, the syntax analyzer or the semantic analyzer?).

I will make available the test cases I will use to grade your project on October 29, 2025 so that you can test your code against my test cases.

**Phase 5: Documentation**

Your code must be well commented – this includes documenting each class, method and complex parts of the code. The code comments ***must* *demonstrate YOUR understanding*** of the code.

**EXTRA CREDIT Phase 6: rustdoc Documentation**

For proper, public documentation, students that choose to use rustdoc ([umm…GPT it](https://chatgpt.com/share/68db475e-b60c-8008-b0f2-9a0d4da6a2f5)) and generate the html files will receive up to 5 points extra credit (this is easy, do it! ☺).

**Version Control:** All students will be required to maintain and update their source code on GitHub, with ***frequent*** commits and informative commit comments reflecting the code changes made. You will need to add me as a collaborator/member to your project so that I can access your code. This must be done on the GitHub website through the repository Settings > Collaborators and add me (cosc455dehlinger@gmail.com) as a Developer so that I can access your project remotely – this is the only way that I will be able to help you with technical questions and/or grade it

**Project Timeline:** To summarize, the timeline for the project is as follows:

09/30 Project assigned

10/10 BNF & ANTLR grammar deadline (Blackboard)

10/11 BNF Grammar solution provided

10/29 Grading input test cases provided

11/03 Implementation deadline (Blackboard / GitHub)

No project submissions after the deadline will be accepted.

**Grading:** Phase 2 of your project will be graded as follows: 20 points allocated for code development following tasks 1-6 and configuration management use; 20 points for handling my input test cases (which will likely include variations of the above test cases) and for Rust assert\_eq unit tests I will apply to the required, trait functions; 5 points for repository quality (i.e., commits, commit comments, etc.), and, 5 points for quality of your code (i.e., design, structure, comments, etc.). As mentioned in Phase 6, up to 5 points extra credit will be awarded to those students that use rustdoc and generate the html files for it.

**Submission:** You will submit the following in a single zipped file via Blackboard:

* A plain text file, named *readme.txt*, that has the address of your GitHub repository with this project’s source code, ensuring that cosc455dehlinger has been added as a developer and that the code is pushed/updated to GitHub. The readme file should also specify if the executable targets the Windows or Mac operating system and Google Chrome web browser. Finally, the readme should contain 2-3 sentences on if/how AI/LLMs were used.
* All BNF and ANTLR documents (i.e., Phase 1) in a directory named *design*. In addition, if any AI/LLM was used, provide a file (or files) with the AI/LLM transcript text, or permalinks to these interactions.
* All developed Rust source code (i.e., Phases 2 and 5) in a directory named *src* (as backup to your repository).
* An executable file (i.e., Phase 3) of your developed compiler in a directory named *bin*.
* A directory of the test case files (i.e., Phase 4) you used for input in a directory named *test*. Ideally, this should include more than just the test files I will provide for grading basis.
* Any documentation developed/generated (i.e., Phase 6) in a directory named *docs*.

**AI/LLM Use:** While AI/LLM (i.e., [*vibe coding*](https://en.wikipedia.org/wiki/Vibe_coding)) usage is allowed (and somewhat encouraged as that is real life and likely the future) for the design and implementation of this project, it should be well within your abilities to complete without its use. Further, despite any vibe coding done, the final project submission ***must strictly***:

1. Meet the project implementation requirements (e.g., character-by-character lexical analysis, recursive descent parsing syntax analysis, implementing the provided traits, etc.).
2. Contain code comments/documentation that ***you*** supply(not GPT generated comments) to demonstrate your full understanding.
3. Stay within the BNF, ANTLR, and Rust features we have seen in class/lab.
4. Be fully understood by the student should you have to answer any follow up questions in a lab, project, exam, or other situation.
5. Contain the AI/LLM transcripts or permalinks used to aid in design/development in the *design* directory of the final submission.

***Any submitted projects not following this format will not be graded.***

[](https://www.youtube.com/watch?v=NkmJcgbXkJA)

**Compiler trait:**

/// Trait for a simple lolcompiler front-end.

/// Errors should cause immediate exit inside the implementation.

pub trait Compiler {

/// Begin the compilation process (entry point).

fn compile(&mut self, source: &str);

/// Get the next token from the lexical analyzer.

fn next\_token(&mut self) -> String;

/// Run the syntax analyzer starting from <lolcode>.

fn parse(&mut self);

/// Get the current token being processed.

fn current\_token(&self) -> String;

/// Set the current token (typically used internally).

fn set\_current\_token(&mut self, tok: String);

}

**Lexical Analysis trait:**

/// Trait for a simple lexical analyzer.

/// Implements a character-by-character analysis

/// from a state machine design.

pub trait LexicalAnalyzer {

/// Return the next character from the input.

/// If input is exhausted, should terminate the program.

fn get\_char(&mut self) -> char;

/// Add a character to the current potential token.

fn add\_char(&mut self, c: char);

/// Lookup a potential token to determine if it is valid.

/// Returns true if a valid token/lexeme, false otherwise.

fn lookup(&self, s: &str) -> bool;

}

**Syntax Analysis trait (2 options):**

/// OPTION 1 - Trait for a recursive descent Syntax Analyzer

/// over Vec<String>. Each function parses a nonterminal in

/// the grammar. On error: exit immediately.

pub trait SyntaxAnalyzer {

fn parse\_lolcode(&mut self);

fn parse\_head(&mut self);

fn parse\_title(&mut self);

fn parse\_comment(&mut self);

fn parse\_body(&mut self);

fn parse\_paragraph(&mut self);

fn parse\_inner\_paragraph(&mut self);

fn parse\_inner\_text(&mut self);

fn parse\_variable\_define(&mut self);

fn parse\_variable\_use(&mut self);

fn parse\_bold(&mut self);

fn parse\_italics(&mut self);

fn parse\_list(&mut self);

fn parse\_list\_items(&mut self);

fn parse\_inner\_list(&mut self);

fn parse\_audio(&mut self);

fn parse\_video(&mut self);

fn parse\_newline(&mut self);

fn parse\_text(&mut self);

}

/// OPTION 2 - Trait for a recursive descent Syntax Analyzer

/// over Vec<String>. Each function parses a nonterminal in

/// the grammar. On error: return Err(message), on success: Ok(()).

pub trait SyntaxAnalyzer {

fn parse\_lolcode(&mut self) -> Result<(), String>;

fn parse\_head(&mut self) -> Result<(), String>;

fn parse\_title(&mut self) -> Result<(), String>;

fn parse\_comment(&mut self) -> Result<(), String>;

fn parse\_body(&mut self) -> Result<(), String>;

fn parse\_paragraph(&mut self) -> Result<(), String>;

fn parse\_inner\_paragraph(&mut self) -> Result<(), String>;

fn parse\_inner\_text(&mut self) -> Result<(), String>;

fn parse\_variable\_define(&mut self) -> Result<(), String>;

fn parse\_variable\_use(&mut self) -> Result<(), String>;

fn parse\_bold(&mut self) -> Result<(), String>;

fn parse\_italics(&mut self) -> Result<(), String>;

fn parse\_list(&mut self) -> Result<(), String>;

fn parse\_list\_items(&mut self) -> Result<(), String>;

fn parse\_inner\_list(&mut self) -> Result<(), String>;

fn parse\_audio(&mut self) -> Result<(), String>;

fn parse\_video(&mut self) -> Result<(), String>;

fn parse\_newline(&mut self) -> Result<(), String>;

fn parse\_text(&mut self) -> Result<(), String>;

}