COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this project report **"Markdown Compiler"** is the bonafide work of **"Maddu Sai Naga Veera Bhadra Rao(RA2011026010306),Nunna Sai Gowtham(RA2011026010325),Syed Aman Kaif Ali(RA2011026010314)"** of III Year/VI Sem B.tech(CSE) who carried out the mini project work under my supervision for the course 18CSC304J- Compiler Design in SRM Institute of Science and Technology during the academic year 2022-2023(Even sem).

SIGNATURE
Dr.A.Maheshwari
Assistant Professor
Department of Computational Intelligence

SIGNATURE
Dr. R Annie Uthra
HEAD OF THE DEPARTMENT
Department of Computational Intelligence

MARKDOWN COMPILER

A MINI PROJECT REPORT

Submitted by

MADDU SAI NAGA VEERA BHADRA RAO (RA2011026010306)
NUNNA SAI GOWTHAM (RA2011026010325)
SYED AMAN KAIF ALI(RA2011026010314)

Under the guidance of

Dr.A.Maheshwari

(Assistant Professor, Department of Computational Intelligence)

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

MAY 2023

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this project report **"Markdown Compiler"** is the bonafide work of **"Maddu Sai Naga Veera Bhadra Rao(RA2011026010306),Nunna Sai Gowtham(RA2011026010325),Syed Aman Kaif Ali(RA2011026010314)"** of III Year/VI Sem B.tech(CSE) who carried out the mini project work under my supervision for the course 18CSC304J- Compiler Design in SRM Institute of Science and Technology during the academic year 2022-2023(Even sem).

SIGNATURE
Dr.A.Maheshwari
Assistant Professor
Department of Computational Intelligence

SIGNATURE
Dr. R Annie Uthra
HEAD OF THE DEPARTMENT
Department of Computational Intelligence

ABSTRACT

Markdown is a presentational markup language that is designed for readability. It is also very context sensitive, which makes it difficult to highlight correctly using the syntax highlighting mechanisms available in existing text editor programs. In this report we discuss the implementation of a Markdown syntax highlighter that is able to handle this context sensitivity by adapting the parser of an existing Markdown compiler. We evaluate five compilers for qualities we would like to see in our highlighter (correctness, embeddability, portability, efficiency), as well as for how easy they are to modify, and pick peg-markdown as the one to take the parser from. We then adapt its parser for syntax highlighting purposes and integrate it with the Cocoa GUI application framework. The end result is a portable syntax highlighting parser for Markdown, written in C, as well as Objective-C classes for using it in Cocoa applications

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

A compiler is just a black box which translates input in a given language to output in another language. The input and output languages can be anything. To keep things simple, this project deals with the construction of a simple compiler which translates a tiny subset of markdown to HTML. Our compiler will mimic the most common compiler structure out there, and we'll boil it down to the very core of it. Our compiler will consist of three steps. The first step is transforming the input markdown string into a list of tokens. Next, we take those tokens and pass them into a parser. That parser will give us a tree data-structure representing our tokens organized in a certain way.

Markdown is an easy-to-use markup language that is used with plain text to add formatting elements (headings, bulleted lists, URLs) to plain text without the use of a formal text editor or the use of HTML tags. Markdown is device agnostic and displays the writing format consistently across device types. However, Markdown compilers often cache memory for long periods of time making them resource intensive. They need to be installed separately and can't be run directly in the CLI or directly on servers.

Marked is a low-level markdown compiler for parsing markdown without caching or blocking for long periods of time. It is built for speed as it is light-weight while implementing all markdown features from the supported flavors & specifications. It is available as a command line interface (CLI) and runs in client- or server-side JavaScript projects.

1.2 Problem Statement

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of the source program, and feeds its output to the next phase of the compiler.

The analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate code which is also referred as Assembly Language Code.

Marked is a low-level compiler for parsing markdown without caching or blocking for long periods of time built for speed. It is light-weight while implementing all markdown features from the supported flavors & specifications and works in a browser, on a server, or from a command line interface (CLI).

1.3 Software Requirements Specifications

- Node.js: Only current and LTS Node.js versions are supported. End of life Node.js versions may become incompatible with Marked at any point in time.
- Browser: All browsers except IE11 are supported.
- Marked offers advanced configurations and extensibility as well. By supporting the above Markdown flavors, it's possible that Marked can help you use other flavors as well; however, these are not actively supported by the community.

CHAPTER 2

LITERATURE SURVEY

2.1 Existing Systems

Compilers that are used for transforming a document from one format to another generally have two distinct parts: a front-end that interprets the input and transforms it into some kind of an intermediate form, and a back-end that generates the output based on this intermediate representation (programming language compilers — as opposed to Markdown compilers — may also include multiple optimization phases). The front-end does its job by performing lexical analysis, or scanning, and syntactic analysis, or parsing. The job performed by the back-end may also be called code generation. Scanning and parsing can be performed during the same pass through the input instead of in completely separate steps, so for the sake of brevity in this report we will use the term “parser” to refer to the whole front-end, including the lexical analyzer. [1, pp. 4–8] In this section we will evaluate some parsers in existing Markdown compilers. We will take the quality attributes from our non-functional requirements as the basis for this evaluation, the idea being that if the parser exhibits these qualities, then the syntax highlighter based on it would exhibit them as well. In addition we also want to consider modifiability because we certainly want to be able to easily extract the parser from the compiler and modify it for our purposes.

2.2 Proposed System

- Support Standard Markdown / CommonMark and GFM (GitHub Flavored Markdown);
- Full-featured: Real-time Preview, Image (cross-domain) upload, Preformatted text/Code blocks/Tables insert, Code fold, Search replace, Read only, Themes, Multi-languages, L18n, HTML entities, Code syntax highlighting.;
- Markdown Extras : Support ToC (Table of Contents), Emoji, Task lists, @Links;
- Compatible with all major browsers (IE8+), compatible Zepto.js and iPad;
- Support decode & filter of the HTML tags & attributes;
- Support TeX (LaTeX expressions, Based on KaTeX), Flowchart and Sequence Diagram of Markdown extended syntax;
- Support AMD/CMD (Require.js & Sea.js) Module Loader, and Custom/define editor plugins;

CHAPTER 3

SYSTEM ARCHITECTURE

3.1 Design Principles

Marked tends to favour following the SOLID set of software design and development principles; mainly the single responsibility and open/closed principles:

- **Single responsibility:** Marked, and the components of Marked, have the single responsibility of converting Markdown strings into HTML.
- **Open/closed:** Marked favour giving developers the means to easily extend the library and its components over changing Marked's behaviour through configuration options.

3.2 Architecture

1. Input Layer: This layer takes user input in the form of Markdown files, either from the command line or a web interface.
2. Parsing Layer: This layer parses the Markdown files into an Abstract Syntax Tree (AST). The AST represents the structure of the Markdown document in a machine-readable format.
3. Transformation Layer: This layer transforms the AST into an intermediate representation that can be easily rendered to different output formats. For example, the intermediate representation could be HTML or LaTeX.
4. Rendering Layer: This layer takes the intermediate representation and converts it into the desired output format, such as PDF, HTML, or plain text.
5. Output Layer: This layer presents the output to the user, either by writing it to a file or displaying it on the screen.

CHAPTER 4

MODULES AND FUNCTIONALITIES

4.1 The Parse Function

```
import { marked } from 'marked';
marked.parse(markdownString [,options] [,callback])
```

Argument	Type	Notes
markdownString	`string`	String of markdown source to be compiled.
options	`object`	Hash of options. Can also use `marked.setOptions`.
callback	`function`	Called when `markdownString` has been parsed. Can be used as second argument if no `options` present.

4.2 Inline Markdown

You can parse inline markdown by running markdown through marked.parseInline.

```
const blockHtml = marked.parse('**strong** _em_');
console.log(blockHtml); // '<p><strong>strong</strong> <em>em</em></p>'

const inlineHtml = marked.parseInline('**strong** _em_');
console.log(inlineHtml); // '<strong>strong</strong> <em>em</em>'
```

4.3 Asynchronous Highlighting

Unlike highlight.js the pygmentize.js library uses asynchronous highlighting. This example demonstrates that marked is agnostic when it comes to the highlighter you use.In both examples, code is a string representing the section of code to pass to the highlighter. In this example, lang is a string informing the highlighter what programming language to use for the code and callback is the function the asynchronous highlighter will call once complete.

```
marked.setOptions({
  highlight: function(code, lang, callback) {
    require('pygmentize-bundled') ({ lang: lang, format: 'html' }, code, function (err, result) {
      callback(err, result.toString());
    });
  }
});

marked.parse(markdownString, (err, html) => {
  console.log(html);
});
```

4.4 The Marked Pipeline

Before building your custom extensions, it is important to understand the components that Marked uses to translate from Markdown to HTML:

1. The user supplies Marked with an input string to be translated.
2. The lexer feeds segments of the input text string into each tokenizer, and from their output, generates a series of tokens in a nested tree structure.
3. Each tokenizer receives a segment of Markdown text and, if it matches a particular pattern, generates a token object containing any relevant information.
4. The walkTokens function will traverse every token in the tree and perform any final adjustments to the token contents.
5. The parser traverses the token tree and feeds each token into the appropriate renderer, and concatenates their outputs into the final HTML result.
6. Each renderer receives a token and manipulates its contents to generate a segment of HTML.

Marked provides methods for directly overriding the renderer and tokenizer for any existing token type, as well as inserting additional custom renderer and tokenizer functions to handle entirely custom syntax.

4.5 Renderer

The renderer defines the HTML output of a given token. If you supply a renderer in the options object passed to marked.use(), any functions in the object will override the default handling of that token type.

Calling marked.use() to override the same function multiple times will give priority to the version that was assigned *last*. Overriding functions can return false to fall back to the previous override in the sequence, or resume default behaviour if all overrides return false. Returning any other value (including nothing) will prevent fallback behaviour.

Example: Overriding output of the default heading token by adding an embedded anchor tag like on GitHub.

```
// Create reference instance
import { marked } from 'marked';

// Override function
const renderer = {
  heading(text, level) {
    const escapedText = text.toLowerCase().replace(/[^\w]+/g, '-');

    return `
            <h${level}>
              <a name="${escapedText}" class="anchor" href="#${escapedText}">
                <span class="header-link"></span>
              </a>
              ${text}
            </h${level}>`;
  }
};

marked.use({ renderer });

// Run marked
console.log(marked.parse('# heading+'));
```

Output:

```
<h1>
  <a name="heading-" class="anchor" href="#heading-">
    <span class="header-link"></span>
  </a>
  heading+
</h1>
```

4.6 The Tokenizer

The tokenizer defines how to turn markdown text into tokens. If you supply a tokenizer object to the Marked options, it will be merged with the built-in tokenizer and any functions inside will override the default handling of that token type.

Calling marked.use() to override the same function multiple times will give priority to the version that was assigned *last*. Overriding functions can return false to fall back to the previous override in the sequence, or resume default behaviour if all overrides return false. Returning any other value (including nothing) will prevent fallback behaviour.

Example: Overriding default codespan tokenizer to include LaTeX.

```
// Create reference instance
import { marked } from 'marked';

// Override function
const tokenizer = {
  codespan(src) {
    const match = src.match(/^\$+([^\$\n]+?)\$+/);
    if (match) {
      return {
        type: 'codespan',
        raw: match[0],
        text: match[1].trim()
      };
    }

    // return false to use original codespan tokenizer
    return false;
  }
};

marked.use({ tokenizer });

// Run marked
console.log(marked.parse('$ latex code $\n\n` other code `'));
```

Output:

```
<p><code>latex code</code></p>
<p><code>other code</code></p>
```

4.7 Walk Tokens

The walkTokens function gets called with every token. Child tokens are called before moving on to sibling tokens. Each token is passed by reference so updates are persisted when passed to the parser. When async mode is enabled, the return value is awaited. Otherwise the return value is ignored.

marked.use() can be called multiple times with different walkTokens functions. Each function will be called in order, starting with the function that was assigned *last*.

Example: Overriding heading tokens to start at h2.

```
import { marked } from 'marked';

// Override function
const walkTokens = (token) => {
  if (token.type === 'heading') {
    token.depth += 1;
  }
};

marked.use({ walkTokens });

// Run marked
console.log(marked.parse('# heading 2\n\n## heading 3'));
```

Output:

```
<h2 id="heading-2">heading 2</h2>
<h3 id="heading-3">heading 3</h3>
```

CHAPTER 5

CODING AND TESTING

5.1 Parser

```
import { Renderer } from './Renderer.js';
import { TextRenderer } from './TextRenderer.js';
import { Slugger } from './Slugger.js';
import { defaults } from './defaults.js';
import {
  unescape
} from './helpers.js';

/**
 * Parsing & Compiling
 */
export class Parser {
  constructor(options) {
    this.options = options || defaults;
    this.options.renderer = this.options.renderer || new Renderer();
    this.renderer = this.options.renderer;
    this.renderer.options = this.options;
    this.textRenderer = new TextRenderer();
    this.slugger = new Slugger();
  }

  /**
   * Static Parse Method
   */
  static parse(tokens, options) {
    const parser = new Parser(options);
    return parser.parse(tokens);
  }

  /**
   * Static Parse Inline Method
   */
  static parseInline(tokens, options) {
    const parser = new Parser(options);
    return parser.parseInline(tokens);
  }

  /**
   * Parse Loop
   */
  parse(tokens, top = true) {
    let out = '',
      i,
      j,
      k,
      l2,
```

```
    l3,
    row,
    cell,
    header,
    body,
    token,
    ordered,
    start,
    loose,
    itemBody,
    item,
    checked,
    task,
    checkbox,
    ret;

  const l = tokens.length;
  for (i = 0; i < l; i++) {
    token = tokens[i];

    // Run any renderer extensions
    if (this.options.extensions && this.options.extensions.renderers && this.options.extensions.renderers[token.type]) {
      ret = this.options.extensions.renderers[token.type].call({ parser: this }, token);
      if (ret !== false || !['space', 'hr', 'heading', 'code', 'table', 'blockquote', 'list', 'html', 'paragraph', 'text'].includes(token.type)) {
        out += ret || '';
        continue;
      }
    }

    switch (token.type) {
      case 'space': {
        continue;
      }
      case 'hr': {
        out += this.renderer.hr();
        continue;
      }
      case 'heading': {
        out += this.renderer.heading(
          this.parseInline(token.tokens),
          token.depth,
          unescape(this.parseInline(token.tokens, this.textRenderer)),
          this.slugger);
        continue;
      }
      case 'code': {
```

5.2 Renderer

```
import { defaults } from './defaults.js';
import {
  cleanUrl,
  escape
} from './helpers.js';

/**
 * Renderer
 */
export class Renderer {
  constructor(options) {
    this.options = options || defaults;
  }

  code(code, infostring, escaped) {
    const lang = (infostring || '').match(/\S*/)[0];
    if (this.options.highlight) {
      const out = this.options.highlight(code, lang);
      if (out != null && out !== code) {
        escaped = true;
        code = out;
      }
    }

    code = code.replace(/\n$/, '') + '\n';

    if (!lang) {
      return '<pre><code>'
        + (escaped ? code : escape(code, true))
        + '</code></pre>\n';
    }

    return '<pre><code class="'
      + this.options.langPrefix
      + escape(lang)
      + '">'
      + (escaped ? code : escape(code, true))
      + '</code></pre>\n';
  }

  /**
   * @param {string} quote
   */
  blockquote(quote) {
    return `<blockquote>\n${quote}</blockquote>\n`;
  }
```

```
48    html(html) {
49      return html;
50    }
51
52    /**
53     * @param {string} text
54     * @param {string} level
55     * @param {string} raw
56     * @param {any} slugger
57     */
58    heading(text, level, raw, slugger) {
59      if (this.options.headerIds) {
60        const id = this.options.headerPrefix + slugger.slug(raw);
61        return `<h${level} id="${id}">${text}</h${level}>\n`;
62      }
63
64      // ignore IDs
65      return `<h${level}>${text}</h${level}>\n`;
66    }
67
68    hr() {
69      return this.options.xhtml ? '<hr/>\n' : '<hr>\n';
70    }
71
72    list(body, ordered, start) {
73      const type = ordered ? 'ol' : 'ul',
74        startatt = (ordered && start !== 1) ? (' start="' + start + '"') : '';
75      return '<' + type + startatt + '>\n' + body + '</' + type + '>\n';
76    }
77
78    /**
79     * @param {string} text
80     */
81    listitem(text) {
82      return `<li>${text}</li>\n`;
83    }
84
85    checkbox(checked) {
86      return '<input '
87        + (checked ? 'checked="" ' : '')
88        + 'disabled="" type="checkbox"'
89        + (this.options.xhtml ? ' /' : '')
90        + '> ';
91    }
92
```

5.3 Tokenizer

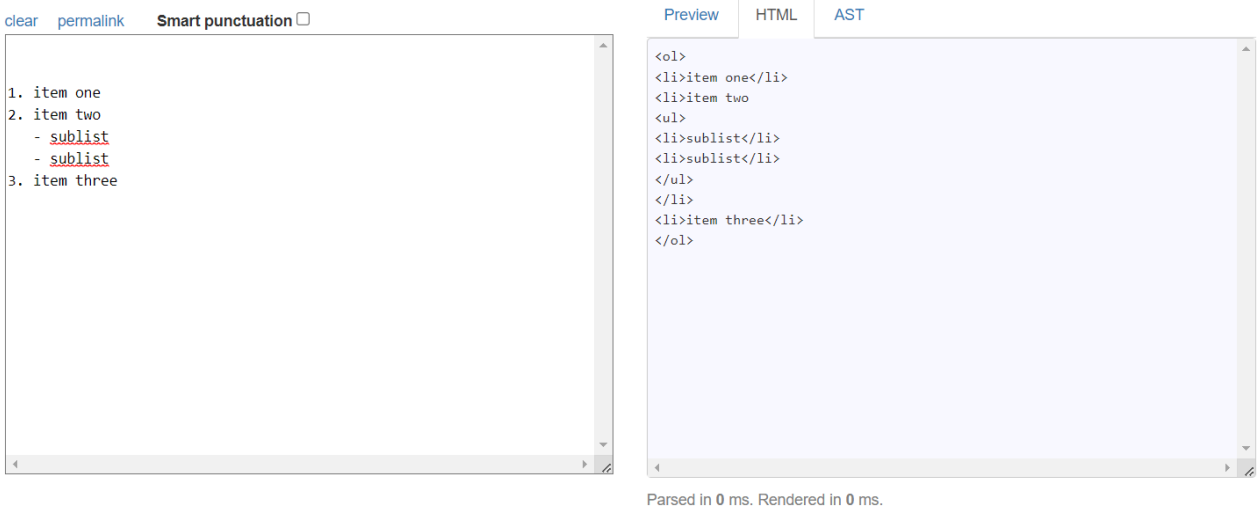
```
import { defaults } from './defaults.js';
import {
  rtrim,
  splitCells,
  escape,
  findClosingBracket
} from './helpers.js';

function outputLink(cap, link, raw, lexer) {
  const href = link.href;
  const title = link.title ? escape(link.title) : null;
  const text = cap[1].replace(/\\([\[\]])/g, '$1');

  if (cap[0].charAt(0) !== '!') {
    lexer.state.inLink = true;
    const token = {
      type: 'link',
      raw,
      href,
      title,
      text,
      tokens: lexer.inlineTokens(text)
    };
    lexer.state.inLink = false;
    return token;
  }
  return {
    type: 'image',
    raw,
    href,
    title,
    text: escape(text)
  };
}

function indentCodeCompensation(raw, text) {
  const matchIndentToCode = raw.match(/^(\s+)(?:```)/);

  if (matchIndentToCode === null) {
    return text;
  }

  const indentToCode = matchIndentToCode[1];

  return text
    .split('\n')
```

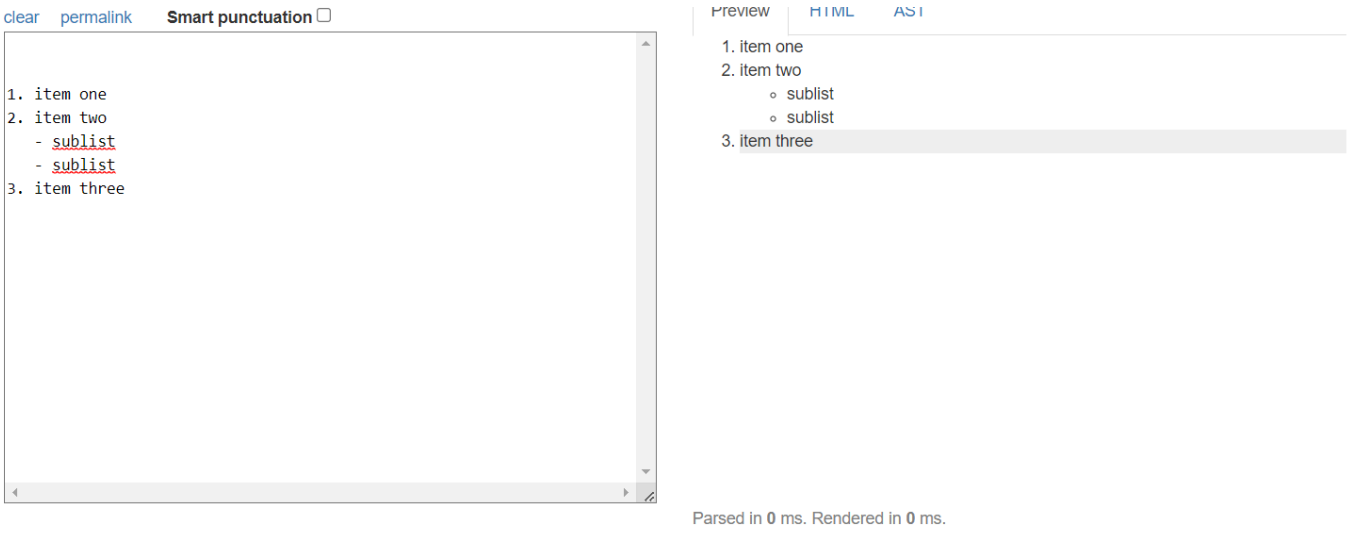
5.4 Lexer

```
import { Tokenizer } from './Tokenizer.js';
import { defaults } from './defaults.js';
import { block, inline } from './rules.js';
import { repeatString } from './helpers.js';

/**
 * smartypants text replacement
 * @param {string} text
 */
function smartypants(text) {
  return text
    // em-dashes
    .replace(/---/g, '\u2014')
    // en-dashes
    .replace(/--/g, '\u2013')
    // opening singles
    .replace(/(^|[-\u2014/(\[{"\s])'/g, '$1\u2018')
    // closing singles & apostrophes
    .replace(/'/g, '\u2019')
    // opening doubles
    .replace(/(^|[-\u2014/(\[{\u2018\s])"/g, '$1\u201c')
    // closing doubles
    .replace(/"/g, '\u201d')
    // ellipses
    .replace(/\.{3}/g, '\u2026');
}

/**
 * mangle email addresses
 * @param {string} text
 */
function mangle(text) {
  let out = '',
    i,
    ch;
```

```
  const l = text.length;
  for (i = 0; i < l; i++) {
    ch = text.charCodeAt(i);
    if (Math.random() > 0.5) {
      ch = 'x' + ch.toString(16);
    }
    out += '&#' + ch + ';';
  }

  return out;
}

/**
 * Block Lexer
 */
export class Lexer {
  constructor(options) {
    this.tokens = [];
    this.tokens.links = Object.create(null);
    this.options = options || defaults;
    this.options.tokenizer = this.options.tokenizer || new Tokenizer();
    this.tokenizer = this.options.tokenizer;
    this.tokenizer.options = this.options;
    this.tokenizer.lexer = this;
    this.inlineQueue = [];
    this.state = {
      inLink: false,
      inRawBlock: false,
      top: true
    };

    const rules = {
      block: block.normal,
      inline: inline.normal
    };

    if (this.options.pedantic) {
      rules.block = block.pedantic;
      rules.inline = inline.pedantic;
    } else if (this.options.gfm) {
      rules.block = block.gfm;
      if (this.options.breaks) {
        rules.inline = inline.breaks;
      } else {
        rules.inline = inline.gfm;
      }
```

5.5 Output Screenshot





CHAPTER 6

CONCLUSION AND FUTURE ENHANCEMENT

Markdown compilers are widely used for translating plain Markdown text into formatted text, yet they suffer from performance bugs that cause performance degradation and resource exhaustion. Currently, there is little knowledge and understanding about these performance bugs in the wild. In this work, we first conduct a comprehensive study of known performance bugs in Markdown compilers. We identify that the ways Markdown compilers handle the language's context-sensitive features are the dominant root cause of performance bugs.

In this project, we develop a low-level compiler for parsing markdown without caching or blocking for long periods of timelight-weight while implementing all markdown features from the supported flavours & specifications. It works in a browser, on a server, or from a command line interface (CLI) and is built for speed.

Currently, Marked does not sanitize the output HTML. A sanitize library, like DOMPurify (recommended), sanitize-html or insane is required on the output HTML. However, there is scope to incorporate this feature on future versions of Marked.

REFERENCES

[1] Adapting a Markdown Compiler’s Parser forSyntax Highlighting by Ali Rantakari dated June 3, 2011

[2] R Markdown as a dynamic interface for teachingby Kristine L. Grayson, Angela K. Hilliker, Joanna R. Wares

[3] P. Li, Y. Liu and W. Meng, "Understanding and Detecting Performance Bugs in Markdown Compilers," 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 2021, pp. 892-904, doi: 10.1109/ASE51524.2021.9678611.

[4] J. D. Velasquez, "bcc: A suite of Tools for Introducing Compiler Construction Techniques in the Classroom," in IEEE Latin America Transactions, vol. 16, no. 12, pp. 2941-2946, December 2018, doi: 10.1109/TLA.2018.8804260.

[5] R. P. Salas, "Reusable Learning Objects: An Agile Approach," 2020 IEEE Frontiers in Education Conference (FIE), Uppsala, Sweden, 2020, pp. 1-6, doi: 10.1109/FIE44824.2020.9273947.