Markdown
To
HTML

SUBMITTED BY

Sourabh Tiwari
(2023MCS2487)

SUBMITTED TO

PROF. SMRUTI RANJAN SARANGI

PROJECT REPORT FOR

COP 701 SOFTWARE SYSTEMS LABORATORY

Department of Computer Science and Engineering

Indian Institute of Technology, Delhi

August 2023

INTRODUCTION

The assignment is to convert a Markdown document to an equivalent HTML document. In pursuance of this objective, I have written a Markdown to HTML parser from scratch.

This assignment covers the use of Flex, Bison and C.

This report briefly describes what I have done, my experiences related to it, results, and limitations.

Lexical Analyzer and Parser

Flex (Fast Lexical Analyzer Generator) and Bison (GNU Bison) are powerful tools used for generating lexical analyzers (lexers) and parsers, respectively. Lexers tokenize input text, breaking it into meaningful chunks, while parsers analyze the structure of the text according to predefined grammar rule

- **Flex**: Flex generates the lexer based on regular expressions and corresponding actions. It recognizes patterns in the input text and produces tokens that are then passed to the parser.
- **Bison**: Bison generates the parser based on a context-free grammar designed using production rules. It analyzes the sequence of tokens and constructs a syntax tree based on the defined grammar.

Lexer File:

- Lexer Directives:
  - The '%{ …. %}' section is used for including header files, such as '<stdio.h>', '<stdlib.h>', and "test.tab.h". The latter is created when the Bison file is compiled.
  - '%option noyywrap' directive disables the default behavior of wrapping input.
- Regular Expressions and Token Definitions:
  - Regular expressions like num, alpha, alphanum, space, punc, etc are defined using regular expression patterns.
  - These patterns are used to define tokens like HEADING1, BOLD1, ITALICS1, etc, which correspond to various elements of a markdown document.
- lexer Rules:
  - lexer rules are defined using regular expressions and actions. For instance, when a line matches the pattern for a heading, the lexer returns the corresponding token.

- Similarly, actions associated with tokens include tasks like printing HTML tags or storing token text (yylval.str=strdup(yytext);).

Lexical Analyzer:

I have made my lexical file in such a way that it can handle the following tags of Markdown

- Headings: It can recognize all the Markdown Headings #, ##,.....,###### with the tokens HEADING1, HEADING2…..HEADING6 .
- Lists: Tokens such as ULS1, ULS2, ULS3, OLS can recognize the start of the Ordered and Unordered list.
- Formatting: BOLD1, BOLD2, ITALICS1, ITALICS2, BOLDITALICS2_OPEN, BOLDITALICS2_CLOSE etc can handle the formatting part.
- Hyperlinks: For the Hyperlinks URL tag can extract urls from the input file and tokens such as LB, RB, RSQRB, LSQRB are used in the parser.
- Tables: For tables PIPE and SEPARATOR are used in the parser.
- Images:EXCLAIM, LB, RB, RSQRB are used in the parser to extract the image.

Parser File:

- Parser Directives:
  - The '%{ ….%}' section includes necessary header files ans external function declaration, such as' yylex()' and 'yyerror()'.
  - It also defines a '%union' structure to pass additional information between lexer and parser.
- Token Declarations:
  - Tokens from the lexer are declared using `%token` directives, including `HEADING1`, `BOLD1`, `TEXT`, `URL`, and others.
- Start Symbols and Nonterminals:
  - '%start' program designated the starting symbol for the grammar.
  - '%type<str>' annotations are used to specify the type of values returned by certain nonterminals.
- Grammar Productions:
  - Productions define grammar rules. They consist of non-terminals and terminals (tokens).
  - Productions correspond to markdown elements like headings, lists, bold

and italics, hyperlinks, and more.
    - Actions associated with production generate HTML tags and handle special formatting.
- Controls and Flags:
    - Control flags (`olflag`, `ulflag`, `linkflag`, `tableflag`, `paraflag`) manage the state of various elements like ordered lists, unordered lists, links, tables, paragraphs, etc.
    - These flags are used to ensure proper HTML output and formatting.
- Semantic Actions:
    - Actions within the production generate HTML code or perform specific operations based on the recognized grammar structure.

- Error Handling:
    - The `yyerror()` function is defined to handle parsing errors. It outputs an error message when parsing encounters issues.
- Main Function:
    - The ‘main()’ function initializes parsing using ‘yyparse()’.

Features

My assignment can successfully convert the following types of Markdown elements to HTML:

- Headings
- Ordered List
- Unordered List
- Bold
- Italics
- Bold and italics
- Paragraph
- Linebreak
- Images
- Hyperlinks
- Tables

Extra features

I also implemented some extra features such as:

- Strikethrough
- Horizontal lines
- Block quotes

Conversion Process

1. The user provides a Markdown file as input to the converter.
2. The Flex Lexer scans the input Markdown text, recognizing various tokens based on defined regular expressions.
3. The tokens are then passed to the Bison parser, which constructs a syntax tree according to predefined production rules.
4. Semantic actions embedded in the parser rules generate HTML code based on the recognized Markdown elements.
5. The resulting HTML code is written to an output HTML file.

Challenges that I faced in the assignment

- The first time I tried to create a regex expression for Text, it used to swallow up some of my other tokens, so that token didn't serve any purpose in the parser.
- Creating a grammar for my lexer was by far the biggest challenge in this assignment, since it is essential for the lexer and parser to work together.
- After finishing the grammar for the rest of the elements, I went on to the table, which required some patience and some good logic to implement.
- The last challenge I faced was for the paragraph and line break.

Limitations

- The biggest limitation of my project is that I haven't implemented AST, instead I used printf statements in the semantic action.
- There is a redundancy of ' ' in the final output file.
- If a new line occurs in the list it's not able to handle it.

Improvements that can be done

The improvement that can be done is to implement AST in my assignment and remove my other limitations from the assignment.