Assignment 1 : Report for Conversion of Markdown file to HTML file.

Ganraj Borade
2023MCS2478
August 26, 2023

1 LEXICAL ANALYSIS STAGE

So, I have started from the lexical analysis stage of compiler design. In the `lexer.l` file, I have mentioned various tokens for a markdown file.

```
textword            [a-zA-Z:\.\,'/\%\&\?]+
digits              [0-9]+
header              #|##|###|####|#####|######

%option noyywrap
```

Here, I am specifying the corresponding snippet for regular expressions, and I will be using `textword`, `digits`, and `header` subsequently.

```
[ \t]+                                  {
                                         return WHITESPACE;
                                        }

[\n]                                    {
                                         return SINGLE_NEWLINE;
                                        }

[\n][\n]+                               {
                                         return NEWLINE;
                                        }

{header}                                {
                                         yylval.string=(char* )malloc(yyleng);
                                         strcpy(yylval.string,yytext);
                                         return HEADER;
                                        }

{textword}                              {
                                         yylval.string=(char* )malloc(yyleng);
                                         strcpy(yylval.string,yytext);
                                         return TEXTWORD;
                                        }

{digits}                                {
                                         yylval.string=(char* )malloc(yyleng);
                                         strcpy(yylval.string,yytext);
                                         return INT;
                                        }
```

I have specified the following tokens: `WHITESPACE`, `SINGLE_NEWLINE`, `NEWLINE`, `HEADER`, `TEXTWORD`, and `INT`.

I have separately specified `SINGLE_NEWLINE` and `NEWLINE` to maintain consistency in the code generation phase. For example, in paragraph generation, a single newline will not change the paragraph, but if there are more than one consecutive single newlines, then the paragraph will change. A clearer understanding will be achieved by examining the `bison.y` file.

```
^{digits}[.]                            {
                                         yylval.string=(char* )malloc(yyleng);
                                         strcpy(yylval.string,yytext);
                                         return ORDERED_LIST;
                                        }

[_*]                                    {
                                         return ITALIC;
                                        }

                                        {
[*]{2}                                   return BOLD;
                                        }

\[                                      {
                                         return URL1;
                                        }

!\[                                     {
                                         return IMG1;
                                        }

\]\(                                    {
                                         return IMG2_URL2;
                                        }

=                                       {
                                         return IMG3;
                                        }

@                                       {
                                         return IMG4;
                                        }
```

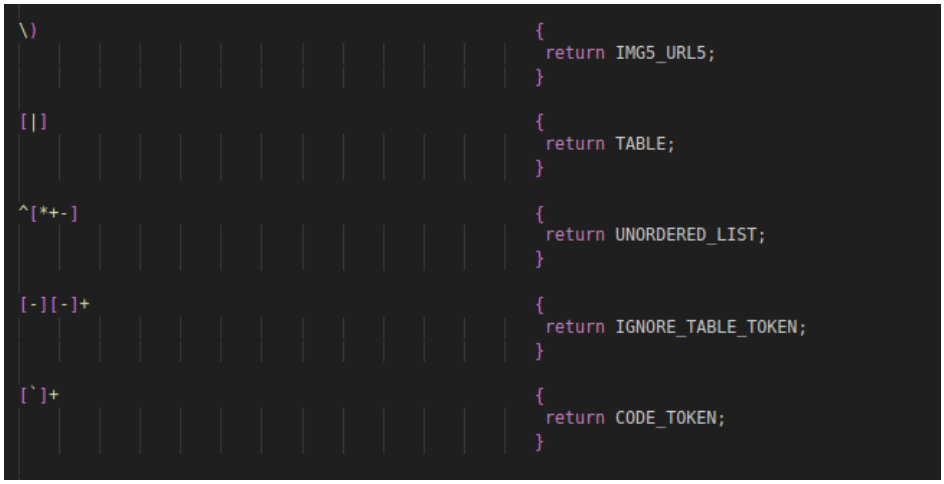
The `lexer.l` file or Flex operates in a top-to-bottom manner. If it encounters the longest matching token early on, it will recognize that token.

The `lexer.l` file, implemented using Flex, processes input in a top-to-bottom sequential manner. It gives precedence to the longest matching token encountered early in the process.

In the context of this file, an evolution has occurred: initially, the pattern `{digits}` was assigned to the `INT` token. However, a new pattern `^digits[.]` has been introduced to identify the `ORDERED_LIST` token. This distinction arises from the observation that lines beginning with `1.`, `2.`, or `3.` signify an ordered list in Markdown. To discriminate between standalone integers and those prefixed by a dot, this specific token was crafted. Flex's preference for longer matches ensures that it captures the `ORDERED_LIST` token for cases like `3.`, rather than categorizing it merely as an `INT` and then recognizing `.` as the other token.

Complementing this, additional tokens cater to Markdown's formatting. For instance, patterns denoted by `-` or `*` at the start are identified as `ITALIC` text. Similarly, a pattern of `**` is recognized as `BOLD` formatting.

Beyond these, the lexer also handles tokens for Markdown's URL and image syntax.



The `IGNORE_TABLE_TOKEN` is used in this context when encountering `|-----|-----|` in the Markdown. It serves to ignore this pattern and modify the row types in the table to `<tbody>`.

2 LEXICAL ANALYSIS STAGE

```
%union
{
      struct AST *ast;
      char *string;
}

%token NEWLINE SINGLE_NEWLINE WHITESPACE
%token HEADER TEXTWORD INT
%token ITALIC BOLD BOLD_ITALICS
%token URL1 IMG1 IMG2_URL2 IMG3 IMG4 IMG5_URL5
%token UNORDERED_LIST ORDERED_LIST
%token TABLE IGNORE_TABLE_TOKEN
%token CODE_TOKEN

%{
/**
 * @brief Here we are defining the types of our both terminals and non_terminals.
 *
 */
%}

%start begin
%type <ast> NEWLINE
%type <ast> header
%type <ast> markdown sub_content content lines
%type <ast> items word_format text textword image url
%type <ast> unordered_list ordered_list
%type <ast> table table_para table_gen
%type <ast> code_text
%type <string> HEADER TEXTWORD INT IGNORE_TABLE_TOKEN
```

Here in above snippet, I am mentioning tokens to be used in this `bison.y` file from lexer.l file . I am also defining a union which consist of data type of both terminals and non terminals used in this grammer. `start` is my start non-terminal symbol.

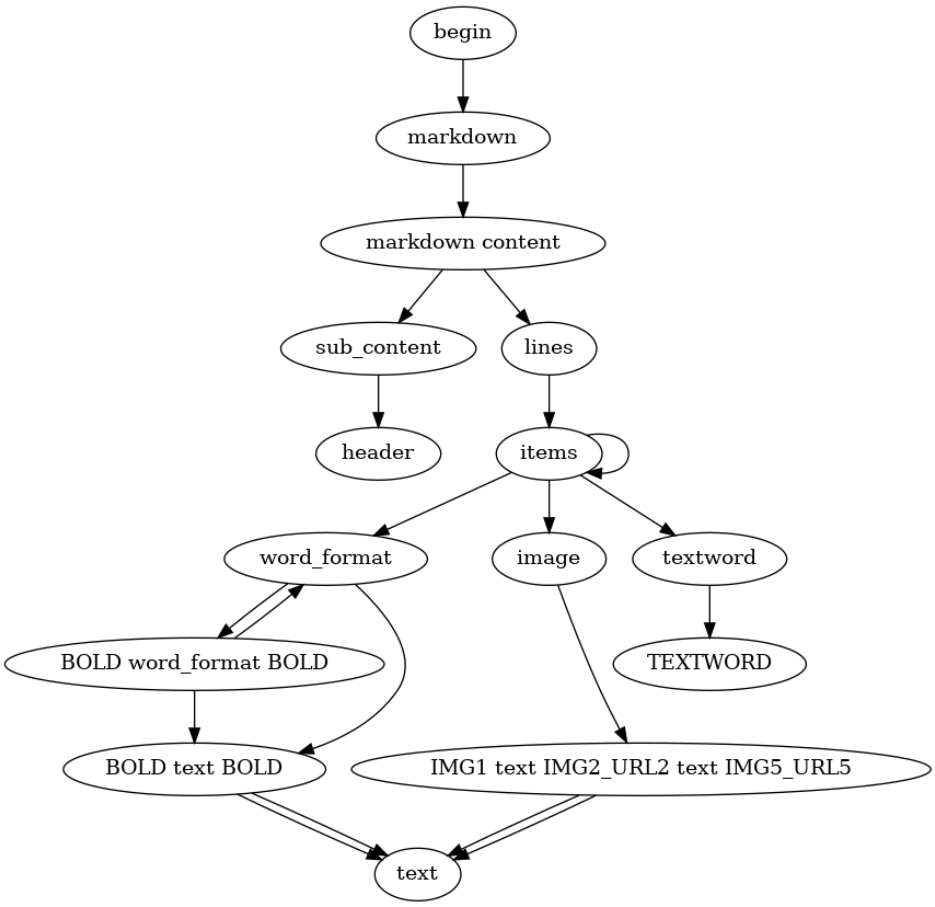
I am writing below only the grammer rules which I have written in the `bison.y` file.

```
        begin : markdown
markdown : markdown content
         | markdown SINGLE_NEWLINE content
         | markdown NEWLINE content
         |
      content : WHITESPACE
         | sub_content SINGLE_NEWLINE
         | sub_content NEWLINE
  sub_content : header
         | lines
         | unordered_list
         | ordered_list
         | table_gen
    table_gen : table_para TABLE
table_para : table_para table
         | table
```

table : TABLE WHITESPACE text WHITESPACE
| TABLE WHITESPACE IGNORE_TABLE_TOKEN WHITESPACE
header : HEADER WHITESPACE lines
lines : lines WHITESPACE items
| lines items
| items
| lines WHITESPACE SINGLE_NEWLINE
unordered_list : UNORDERED_LIST WHITESPACE lines
ordered_list : ORDERED_LIST WHITESPACE lines
items : word_format
| image
| textword
| url
| code_text
word_format : BOLD word_format BOLD
| BOLD text BOLD
| ITALIC word_format ITALIC
| ITALIC text ITALIC
| BOLD_ITALICS word_format BOLD_ITALICS
| BOLD_ITALICS text BOLD_ITALICS
code_text : CODE_TOKEN text CODE_TOKEN
url : URL1 text IMG2_URL2 text IMG5_URL5
image : IMG1 text IMG2_URL2 text IMG5_URL5
text : text WHITESPACE textword
| text textword
| textword
textword : TEXTWORD
| INT

In the yellow content above, I have written the left recursive grammar. I have made sure that it does not cause ambiguity as it is not giving me any Shift-Reduce (SR) or Reduce-Reduce (RR) conflicts. By default, Bison uses Look-Ahead Left-Associative Rightmost (LALR(1)) grammar, which has the main limitation of only looking ahead one token to make parsing decisions. This limitation means that we cannot completely convert Markdown to HTML for all possible inputs of a Markdown file.

Here is how the Syntax Tree will look like :



CHALLENGES FACED

- **Paragraph Change:** If there is only one newline change between two lines, then the paragraph must remain the same. However, in the grammar that I defined, I had to change the paragraph for every newline. To maintain the same paragraph, I need to look at two lookahead tokens:
  - One is the `SINGLE_NEWLINE` token.
  - The other is a `TEXTWORD` token or, in general, any token not equal to `SINGLE_NEWLINE`.

  Therefore, here we need two lookaheads to make parsing decisions. As a result, it might not be possible to maintain the same paragraph for a single line change. Although I have written code snippet in the main.c file which will solve this problem (the code snippet of that is given below).

```
replaceSubstring(htmlContent, "</p><p>", " ");
replaceSubstring(htmlContent, "</ol><ol>", "");
replaceSubstring(htmlContent, "</tr><tr>", "");
replaceSubstring(htmlContent, "<th><th>", "<th>");
replaceSubstring(htmlContent, "</th></th>", "</th>");
replaceSubstring(htmlContent, "<td><td>", "<td>");
replaceSubstring(htmlContent, "</td></td>", "</td>");
replaceSubstring(htmlContent, "</table><table>", "");
```

  Here, we are replacing the strings mentioned with equivalent correct strings.

- **Ordered List and Table Generation:** Similar issues arise with the ordered list and table generation phases as we can see in the above figure too.

- **Added Production Rule:** I have also added the following production rule:

```
content : WHITESPACE
```

  This rule will essentially ignore all the leading whitespaces at the beginning of a newline, as specified in the semantic conversion, i.e., the code generation phase. The problem here is that it captures as many whitespaces as possible; ideally, it should not do this. However, I believe I did not have many options other than to keep it.

- **Images:** Regarding images, if we try to add `iitd-logo.jpg`, then it will not parse the HTML element properly because, according to my grammar, the text does not contain the character '-'.

- **CODE SNIPPET (This was not mandatory in the assignment):**

  Regarding code in markdown, ‘‘‘ are used for representing codes. My grammar has the constraint that in the `text` inside the starting ‘‘‘ and ending ’’’, there should not be a single ‘ (single backtick).

3 CODE GENERATION PHASE

In the `bison.y` file, we can observe the corresponding semantic conversions as well. I have created the `functions.c` file in which all the functions are defined.

The code snippet

$$\$\$ = \text{newAST}('P', \$1, \$2)$$

creates a node in the Abstract Syntax Tree (AST) with the tag 'P'. The left subtree of the node is represented by the symbol $1, and the right subtree is represented by the symbol $2. This structure adheres to the specification outlined in the 'helper.h' file. The specific rules governing $1 and $2 depend on the subsequent grammar description.

Similarly, the code

$$\$\$ = \text{paragraph\_Generator}(\text{evaluateAST}(\$1))$$

first evaluates the AST represented by the symbol $1. The result obtained from this evaluation is then used as input for the paragraph_Generator function. This function appends a paragraph to the existing structure. You can find the implementation details of this operation in the 'functions.c' file.

Similarly, following a similar approach, we can analyze all other functions within the codebase. It's worth noting that I have included crucial comments throughout the code to enhance its readability and provide insights into the implementation details.

In essence, the parsing strategy employed here is a bottom-up approach. The recursive grammar progresses from the top of the structure to the bottom during evaluation. However, the actual evaluation of these structures takes place in a bottom-to-top fashion. Finally, as we reach the topmost node of the structure, applying semantic conversion to that node yields the complete HTML document.

4 UNIT TESTING

In the context of my work, I have created a dedicated `test_directory`. This directory serves as a structured repository containing individual markdown files, each corresponding to a specific feature under examination. Simultaneously, the directory accommodates the resulting output files saved in the `.html` format. The primary goal of this setup is to conduct a thorough evaluation, meticulously comparing the outcomes produced by the code execution with the pre-existing output files. This process allows for a comprehensive assessment of the congruence between the newly generated results and the established benchmarks.